# Trabalho 2 Versão A3

Miguel Andrade (201709051)
*Faculdade de Engenharia*
*Universidade do Porto*
Porto, Portugal
up201709051@fe.up.pt

Diana Sandoval (201811491)
*Faculdade de Engenharia*
*Universidade do Porto*
Cidade do México, México
up201811491@fe.up.pt

***Abstract*—O jogo implementado foi o Mancala. Um jogo simples que consiste em semear e colher sementes. Começaremos por abordar a implementação do jogo, falando depois no algoritmo minimax e a versão com os cortes alpha e beta. Nos teste decorridos no houve surpresa ao verificar que o algoritmo minimax alpha-beta apresentou os melhores resultados em termos de eficiência no que toca ao tempo decorrido. Verificamos também que o algoritmo implementado conseguiu superar bastante bem os jogos com um agente aleatório como adversário.**

***Index Terms*—minimax, alpha, beta, mancala, inteligência artificial**

## I. Introdução

Neste segundo projeto o objetivo é implementar um jogo para dois jogadores, onde exista a possibilidade de haver dois jogadores humanos a jogar entre si, um jogador humano e um computador e dois computadores a jogar entre si. O jogo implementado foi o mancala. Como o foco do projeto é a parte referente aos agentes inteligentes, vamos abordar variações do algoritmo minimax assim como diferentes heurísticas.

## II. Descrição do Problema

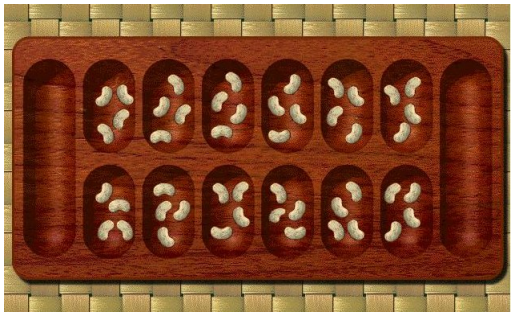

Fig. 1. Jogo Mancala

O Mancala é essencialmente um jogo onde os jogadores semeiam e colhem sementes [1]. O objetivo do jogo é colecionar mais sementes que o adversário. O tabuleiro de jogo é constituído por duas filas de 6 buracos cada uma, e na extremidades encontra-se a mancala (depósito das sementes) onde se coleciona as sementes de cada jogador.

O jogo começa com 4 sementes em cada buraco. Um jogador começa por selecionar um dos seus 6 buracos. Ao selecionar um, este coleciona todas as sementes desse buraco e começa a colocar uma semente em cada um dos buracos seguintes, no sentido contrário ao ponteiros do relógio, até não haver mais sementes na sua mão.

Se o jogador encontrar o seu depósito, este continua a jogar, se encontrar o depósito do jogador adversário, então salta para o buraco seguinte. Caso seja a última semente a ser colocada no depósito do próprio jogador, então este tem direito jogar mais uma vez. Se a última semente for colocado num buraco vazio do lado do próprio jogador, então este pode colecionar as todas as sementes no buraco oposto, incluindo a própria semente colocada. O jogo termina quando todos os 6 buracos de um dos lados ficam vazios. Caso um dos jogadores ainda tenha peças do seu lado quando o jogo termina, então este coleciona-as e junta-as ao seu depósito. Quem tiver mais sementes colecionadas é o vencedor.

## III. Formulação do Problema

### A. Representação do Estado

O Estado do jogo é representado pelo próprio tabuleiro, sendo que os depósitos dos jogadores são os fatores para decedir quem ganha. De forma a poder representar o tabuleiro de jogo, foi usado uma lista de tamanho 14, sendo que os 7 primeiros índices dizem respeito ao primeiro jogador e os restante ao segundo jogador. Os índices 6 e 13 são usados para representar os depósitos do primeiro e segundo jogador respetivamente. Define-se ”primeiro jogador” o jogador que joga nas casas ”inferiores” do tabuleiro e o ”segundo jogador” o jogador que controla as casas ”superiores”. A cada casa é atribuído um valor que corresponde ao número de sementes guardadas nessa casa.

```
------12----11----10----09----08----07-----
|  | |00|  |01|  |00|  |00|  |00|  |00| |  |
|10|                                    |05|
|  | |00|  |00|  |06|  |06|  |11|  |09| |  |
------01----02----03----04----05----06-----
```

Fig. 2. Tabuleiro de Jogo

### B. Operadores

Cada casa jogável é um operador, sendo que as pré-condições são:

1) A casa escolhida não pode estar vazia.
2) A casa escolhida pertencer à área do atual jogador.

### C. Estado Inicial

Todas as casas da lista são iniciados com o valor 4, com exceção das casas que representam os depósitos dos jogadores, sendo estas as casas 6 e 13.

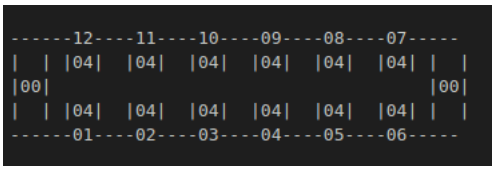


Fig. 3. Estado Inicial

### D. Teste Terminal

O estado terminal é alcançado quando um dos jogadores fica sem peças para poder continuar a jogar.

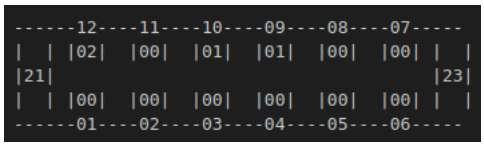


Fig. 4. Estado Final

### E. Função Utilidade

O jogador ganha caso tenha mais sementes no seu depósito do que o seu adversário. A função utilidade simplesmente verifica quem ganhou a partida e retorna um valor positivo em caso de vitória, um valor negativo em caso de derrota e zero em caso de empate.

```python
#Utility function
def utility_function(board, player):
    if(player == 0):
        if(mancala.player_win(board) == 1):
            return (None, 100000000000000)
        elif(mancala.player_win(board) == 2):
            return (None, -100000000000000)
        else:
            return (None, 0)
    if(player == 1):
        if(mancala.player_win(board) == 2):
            return (None, 100000000000000)
        elif(mancala.player_win(board) == 1):
            return (None, -100000000000000)
        else:
            return (None, 0)
```

Fig. 5. Imagem da função utilidade

## IV. Trabalho Relacionado

Na nossa pesquisa encontramos já algum trabalho feito no que toca à implementação deste jogo assim como na implementação de inteligência artificial [2] [3] [9]. Também nas aulas da disciplina foram implementados algoritmos idênticos ao deste projeto, pelo que certamente serve de fonte de auxílio. Apesar de já existirem implementações do jogo mancala com o algoritmo minimax, o projeto do jogo quatro em linha referenciado consegue ser o mais completo, pois a forma como está estruturado separa muito bem a parte referente ao jogo da parte que diz respeito à IA. Com esta estrutura conseguimos ter um excelente ponto de partida para o nosso projeto e ideia bem definida daquilo que tínhamos que fazer.

## V. Implementação do Jogo

Como já foi referido o tabuleiro de jogo é representado põe uma lista de 14 elementos, sendo que cada um deste elementos representa uma casa e o valor guardado refere-se ao número de sementes nessa mesma casa. Os elementos 6 e 13 ficam reservados para o depósito de cada jogador. No que toca ao movimento das peças, criou-se uma função move_piece, que recebe a posição desejada, o jogador que está a jogar e o tabuleiro de jogo. é nesta função que acontece a verificação para saber quando é que se pode comer as sementes do adversário e quando se tem direito a mais uma jogada.

```python
# Handles the logic for moving pieces in the board
def move_piece(pos, player, board):
    i = 0
    if(player == 0):
        hand = board[pos]
        board[pos] = 0
        i = pos+1
        while(hand > 0):
            if(i == PL_2_STORE):
                i = 0
            if(hand == 1 and board[i] == 0 and (i >= 0 and i < PL_1_STORE)):
                board[i] += 1
                eat_seeds(i, player, board)
                hand -= 1
            elif(hand == 1 and i == PL_1_STORE):
                board[i] += 1
                return player
            else:
                board[i] += 1
                hand -= 1
                i += 1

    if(player == 1):
        hand = board[pos]
        board[pos] = 0
        i = pos+1
        while(hand > 0):
            if(i == PL_1_STORE):
                i += 1
            if(i > PL_2_STORE):
                i = 0
            if(hand == 1 and board[i] == 0 and (i > PL_1_STORE and i < PL_2_STORE)):
                board[i] += 1
                eat_seeds(i, player, board)
                hand -= 1
            elif(hand == 1 and i == PL_2_STORE):
                board[i] += 1
                return player
            else:
                board[i] += 1
                hand -= 1
                i += 1

    player += 1
    player = player % 2
    return player
```

Fig. 6. Função move_piece

A função eat_seeds está encarregue de tratar da lógica para efetuar o movimento de captura das sementes. Foi criada uma lista, chamada opposite, que auxilia este processo, onde guarda o lado oposto de todas as casas, sendo assim mais fácil processar esta operação.

Para assegurar que a posição escolhida pelo utilizador é uma posição válida, existe a função verify_move, que verifica se a posição escolhida é um número e que está dentro da parte jogável do tabuleiro referente à vez do utilizador. As funções game_over e player_win, são duas funções que dizem respeito

```python
# Handles the logic for the case of a player eats the seeds of the opponent
def eat_seeds(pos, player, board):
    global opposite

    if(player == 0):
        if(board[opposite[pos]] != 0):
            board[PL_1_STORE] += (board[pos]+board[opposite[pos]])
            board[pos] = 0
            board[opposite[pos]] = 0

    if(player == 1):
        if(board[opposite[pos]] != 0):
            board[PL_2_STORE] += (board[pos]+board[opposite[pos]])
            board[pos] = 0
            board[opposite[pos]] = 0
```

Fig. 7. Função eat_seeds

ao final do jogo, sendo que a primeira verifica se a partida acabou e a segunda retorna o jogador que ganhou a partida.

```python
# Verifies if the board is in a final state, wich means the game is over
def game_over(board):
    count1 = 0
    count2 = 0
    for i in range(len(board)):
        if(i >= 0 and i < PL_1_STORE):
            if(board[i] == 0):
                count1 += 1
        if(i > PL_1_STORE and i < PL_2_STORE):
            if(board[i] == 0):
                count2 += 1

    if(count1 == 6):
        for i in range(7,13):
            board[PL_2_STORE] += board[i]
        return True
    elif(count2 == 6):
        for i in range(0,PL_1_STORE):
            board[PL_1_STORE] += board[i]
        return True
    else:
        return False
```

Fig. 8. Função game_over

## VI. Algoritmos Implementados

Para este projeto o algoritmo pedido foi o minimax. Usamos o minimax e o minimax com cortes alpha-beta [5] [6].

Minimax é um algoritmo é uma regra de decisão para minimizar a perda no pior cenário possível, neste caso queremos minimizar os ganhos do adversário. O algoritmo é auxiliado por funções que ajudam a avaliar o quão boa é a solução. São usadas duas funções, utility_function, que avalia a solução caso se trate de uma situação terminal, isto é, se é uma vitória, derrota ou empate e a heuristic, que avalia o tabuleiro no turno do jogador.

```python
# evaluates how good is the move based in the number of seeds in the player deposit
def heuristic(board, player):
    if(player == 0):
        return (board[mancala.PL_1_STORE] - board[mancala.PL_2_STORE])
    if(player == 1):
        return (board[mancala.PL_2_STORE] - board[mancala.PL_1_STORE])
```

Fig. 9. Função heuristic

O minimax avalia todas as jogados possíveis a uma determinada profundidade. Caso existam muitas opções, o algoritmo rapidamente esgota os recursos da máquina. Para torna este algoritmo mais eficiente, existe uma variação, minimax alpha-beta. Isto permite corta ramos de exploração descessários sem alterar a solução final, aumentando assim a eficiência do algoritmo.

## VII. Experiências e Resultados

Para testar as duas versões dos algoritmos, fizemos a média do tempo que o algoritmo demora a tomar uma decisão a diferentes níveis de profundidade. A forma como o cálculo foi efetuado, foi através da soma de todos os tempos ao longo de vinte partidas e dividindo o resultado pelo número total de tempos registados. Os resultados são os que se seguem:

| Minimax | |
|---|---|
| Depth | Time (Average) |
| 1 | 6.355434782608699e-05 |
| 2 | 0.00038020624999999993 |
| 3 | 0.0015099647058823508 |
| 4 | 0.007639854166666668 |
| 5 | 0.03351009166666666 |
| 6 | 0.1736062576923077 |
| 7 | 0.5134596305555558 |
| 8 | 2.5876256178571433 |
| 9 | 9.929812355000013 |
| 10 | — |
| **Minimax alpha-beta** | |
| Depth | Time (Average) |
| 1 | 7.478478260869559e-05 |
| 2 | 0.00029995625000000034 |
| 3 | 0.0006773529411764701 |
| 4 | 0.003206154166666668 |
| 5 | 0.00663799166666667 |
| 6 | 0.026838650000000016 |
| 7 | 0.059131891666666735 |
| 8 | 0.16887341071428577 |
| 9 | 0.34320350250000004 |
| 10 | 1.1970522824999992 |

Para testar a qualidade das soluções simulamos dez jogos consecutivos de um agente com o algoritmo minimax alpha-beta contra um com escolhas aleatórias. O agente aleatório é para verificar o quão consistente é o algoritmo a fazer as melhores escolhas. Os resultados são os que se seguem:

| Depth | Victories | |
|---|---|---|
| | Jogador 1 | Jogador 2 |
| 1 | 5/10 | 5/10 |
| 2 | 8/10 | 8/10 |
| 3 | 7/10 | 7/10 |
| 4 | 8/10 | 9/10 |
| 5 | 7/10 | 9/10 |
| 6 | 9/10 | 9/10 |
| 7 | 9/10 | 9/10 |
| 8 | 9/10 | 10/10 |
| 9 | 10/10 | 9/10 |
| 10 | 8/10 | 10/10 |

## VIII. Conclusão

No que diz respeito à implementação do jogo e dos algoritmos, o trabalho está estável apresentando os resultados esperados. Existiu alguma dificuldade a encontrar a heurística mais correta para o problema, de forma a obter os melhores resultados. De forma a melhorar o trabalho no futuro, seria interessante experimentar um maior número de heurísticas e fazer a transição para uma interface gráfica.

## References


[1] Erik Arneson, https://www.thesprucecrafts.com/how-to-play-mancala-409424, 3/05/2019
[2] https://github.com/mhchong/Mancala-Game
[3] https://github.com/cypreess/py-mancala
[4] https://www.johnpratt.com/items/mancala/index.html
[5] https://en.wikipedia.org/wiki/Minimax#Pseudocode
[6] https://en.wikipedia.org/wiki/Alphabeta_pruning#Pseudocode
[7] https://www.ultraboardgames.com/mancala/strategies.php
[8] https://www.ultraboardgames.com/mancala/best-opening-move.php
[9] https://github.com/KeithGalli/Connect4-Python