UNIVERSIDADE
DE ÉVORA

OURI

Relatório

Programação I

Miguel Pombeiro, 57829

Miguel Rocha, 58501

Rafael Prezado, 57355

Docentes: Daniela Schmidt

João Sequeira

Rúben Teimas

2023/2024

Índice

Introdução

Este relatório descreve a criação de uma simulação do jogo "Ouri" ("Oware", em inglês), em ambiente de terminal de computador, com recurso à Linguagem de Programação C.

O Ouri é um jogo para dois jogadores, jogado num tabuleiro com doze (12) casas, dois (2) depósitos e quarenta e oito (48) pedras. Tendo todas as pedras o mesmo valor, o objetivo deste jogo é recolher mais pedras para o seu depósito que o adversário. Assim, será considerado vencedor, o jogador que conseguir acumular vinte e cinco (25) ou mais pedras. O tabuleiro de jogo é constituído por duas filas de seis casas cada, sendo os depósitos localizados nas extremidades. No início do jogo, os jogadores sentam-se frente a frente, sendo que as casas que lhe pertencem são as que estão diretamente à sua frente e o seu depósito será aquele que se encontra à sua direita. Inicialmente todas as casas contêm quatro pedras cada e os depósitos estão vazios.

Para a implementação em linguagem C desta simulação, foram seguidas as regras disponibilizadas pelos docentes da disciplina de Programação I no Enunciado deste Projeto.

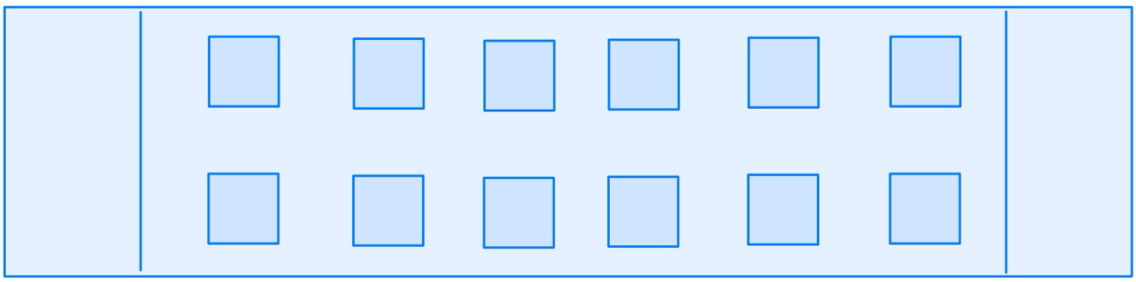

Figura 1 - Desenho de um tabuleiro de Ouri usado no desenvolvimento deste projeto.

O Programa

No início do desenvolvimento do projeto, o grupo decidiu segmentar o programa em várias funções, de forma a facilitar não só o seu desenho bem como a sua implementação e posterior *debug*. Estas funções seriam, posteriormente, chamadas/utilizadas numa outra função *main* onde decorreria todo o procedimento de jogo. Desta forma, foram projetadas quinze (15) funções, cujo código se encontra escrito no ficheiro "ouri_func.c", e ainda uma outra função *main*, escrita no ficheiro "ouri_main.c". A maioria destas funções foi desenhada de forma a poderem ser utilizadas em várias ocasiões, nomeadamente no procedimento normal de jogo (*main*), mas também na função que determina a jogada do computador (*computerMove*).

Um dos problemas iniciais com que o grupo se deparou logo de início e que iria definir a forma como seriam implementadas as restantes funções seria a forma de armazenar o tabuleiro de jogo. Para isso, foram propostas três formas:

- Usando uma matriz de 2x7, em que cada jogador teria o seu lado do tabuleiro armazenado numa das linhas da matriz e o seu depósito na coluna 0.
- Usando dois vetores de tamanho 7, sendo que cada um iria armazenar o lado do tabuleiro de cada um dos jogadores.
- Usando um vetor de tamanho 14, que iria armazenar todo o tabuleiro por ordem crescente do número de casas, correspondendo as casas de um jogador aos índices 1-6 do vetor e as do outro jogador aos índices 7-12 do vetor. Os depósitos de cada um dos jogadores corresponderiam, respetivamente aos índices 0 e 13 do vetor.

Para simplificar a forma de iterar pela variável que iria armazenar o tabuleiro de jogo, o grupo decidiu utilizar o vetor de tamanho 14 uma vez que, apesar de serem mais fáceis de visualizar, as outras duas formas implicariam um aumento de complexidade no código (quer no número de condicionais, quer no número/complexidade dos *loops* que seriam necessários). Por fim, no que diz respeito a esta variável, restava apenas decidir se esta seria declarada de forma global (fora de qualquer função) ou se seria apenas declarada na função *main*. Assim, e de forma a tornar mais fácil de seguir a sequência do código, o grupo optou por declarar a variável apenas na função *main* e utilizar *pointers* de cada vez que uma função necessitasse fazer alterações a esta variável.

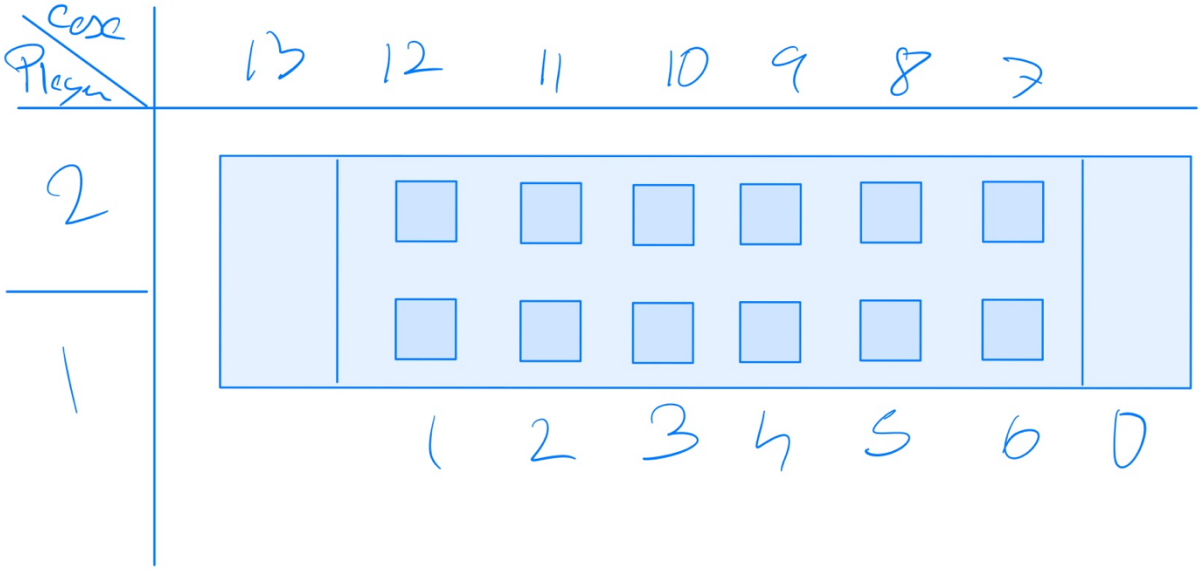

Figura 2 - Esquematização do tabuleiro de jogo com numeração de cada uma das casas e depósitos de forma a corresponderem à numeração dos índices do vetor utilizado.

A fim de otimizar as jogadas do computador, o grupo consultou materiais que tratassem a construção de algoritmos para jogos de tabuleiros[1][2]. Isto posto, destacou-se o algoritmo *minimax* básico, *minimax* com alfa-beta *prunning* e o algoritmo *Greedy*.

O algoritmo *minimax* com alfa-beta *prunning* é eficaz em reduzir "nós" na árvore de decisão. Contudo, a sua implementação era complexa e pequenos erros comprometiam a integridade do computador. Visando simplificar, o grupo optou pelo algoritmo *Greedy*. Este algoritmo prioriza as jogadas com melhor rendimento[3].

Os quatro ficheiros que são necessários para correr o programa são "ouri_func.c", "ouri_func.h", "ouri_main.c" e "makefile" e deverão estar contidos no ficheiro .zip onde se encontra este relatório.

[1] Gabriele Rovaris, «Design of Artificial Intelligence for Mancala Games» (POLITECNICO DI MILANO, 2017), 31–39.
[2] Ifeanyi Ndukwe e Evarista Nwulu, «Computerized Board Game: Dara. International Journal of Advanced Research in Computer Science», *International Journal of Advanced Research* 5 (1 de setembro de 2018): 59–61.
[3] Kumar Ap, «Greedy Algorithm Based Deep Learning Strategy for User Behavior Prediction and Decision-Making Support», *Journal of Computer and Communications* 06 (1 de janeiro de 2018): 47–48, https://doi.org/10.4236/jcc.2018.66004.

Principais Variáveis

mode – É uma variável do tipo inteiro, que representa o modo de jogo atual (jogador vs. jogador ou jogador vs. computador).

currentPlayer – É uma variável do tipo inteiro, que representa o jogador atual (jogador 1 ou jogador 2).

board – É um vetor de tamanho 14, neste caso um *array* do tipo inteiro, onde está armazenado o estado atual do tabuleiro de jogo.

win – É uma variável do tipo inteiro, que representa qual o estado atual do jogo, ou seja, se já existe um vencedor ou não. Pode tomar quatro valores diferentes: 0, caso ainda não haja vencedor e o jogo deva continuar; 1, caso o jogador 1 tenha alcançado as condições para ganhar; 2, caso o jogador 2 tenha alcançado as condições para ganhar; -1 caso tenham sido alcançadas as condições de empate.

landed – É uma variável do tipo inteiro, que representa o índice do vetor tabuleiro onde foi colocada a última pedra, após uma jogada.

move – É uma variável do tipo inteiro, que representa o movimento válido que o utilizador escolhe.

Funções

drawBoard

Esta função é do tipo *void* e recebe como argumentos as variáveis **currentPlayer**, **mode** e o *pointer* para **board**. A sua função é imprimir um tabuleiro, de acordo com o jogador atual e o modo atual. No modo de jogador vs. jogador irá rodar o tabuleiro sempre que mude de jogador, para facilitar a jogada de ambos os jogadores diante do mesmo ecrã e de forma que a representação do depósito de cada jogador fique sempre à sua direita. No modo jogador vs. computador, não fará rotação do tabuleiro, uma vez que o "computador" não necessita desta abstração.

playerRotate

Esta função é do tipo inteiro e recebe como argumento a variável **currentPlayer**. Esta função faz a rotação entre o jogador 1 e jogador 2 de forma a definir os turnos do jogo. Assim, em circunstâncias normais, se o último jogador a jogar foi o jogador 1, o próximo jogador será o 2 e vice-versa. A função retorna o número do jogador a jogar no turno seguinte de forma atualizar a variável que especifica o número do jogador.

checkWin

Esta função é do tipo inteiro e recebe como argumento o *pointer* para **board**. Esta função analisa os depósitos de cada jogador. Se algum jogador cumprir os requisitos para vencer (mais de 25 pedras no depósito), retorna o número do jogador que ganhou (1 ou 2); se existiu empate (24 pedras em cada depósito), retorna -1; se ainda não há vencedores, retorna 0.

askNextMove

Esta função é do tipo inteiro e recebe como argumentos o *pointer* para **board**, e as variáveis **currentPlayer**, e **mode**. Esta função, pede ao jogador atual um movimento entre 0 e 6, correspondendo a seleção 0 à situação para guardar o estado atual do jogo e 1-6 à seleção de uma casa do seu lado do tabuleiro de onde quer jogar. Esta função recorre à função *checkIfValid*, para verificar se o movimento introduzido pelo jogador é válido. Enquanto não for selecionado um movimento válido, a função irá pedir um novo movimento. Por fim, caso seja introduzido um movimento válido, a função irá retornar o valor entre 0-12, correspondente

à seleção de gravar o estado do jogo (0) ou à posição no vetor tabuleiro de onde será feito esse movimento (1-12).

checkIfValid

Esta função é do tipo inteiro e recebe como argumentos o *pointer* para **board**, e as variáveis **currentPlayer**, **mode** e **move** que, neste caso, representa a seleção do movimento cuja validade se vai testar – quer seja introduzido pelo jogador, através de *askNextMove*, ou "pensado" pelo computador, através de *computerMove*. A função serve para verificar se o movimento selecionado pelo jogador é válido, de acordo com todas as limitações impostas nas regras do jogo. Para tal, são consideradas as seguintes situações:

- Se o jogador pretende guardar o jogo (seleção 0), o movimento é considerado válido.
- Se o jogador selecionar uma casa que não tenha pedras, o movimento é considerado inválido.
- Se um dos jogadores não tiver pedras no seu lado do tabuleiro, o outro jogador é obrigado a selecionar uma casa que contenha pedras suficientes de modo a colocar pelo menos uma pedra no outro lado do tabuleiro. Caso esta situação se verifique e o jogador selecione uma casa que cumpra este requisito, o movimento é considerado válido; se este requisito não for cumprido, o movimento é considerado inválido.
- Se o jogador selecionar uma casa com apenas uma pedra e tenha, no seu lado do tabuleiro, casas com mais do que uma pedra, o movimento é considerado inválido.
- Se o jogador fizer uma seleção fora do intervalo correto (0-6), o movimento é considerado inválido.

A função retorna 1 se o movimento for válido, e, caso não seja, retorna 0.

theMove

Esta função é do tipo inteiro e recebe como argumentos o *pointer* para **board** e a variável **move**. Quando chamada, irá realizar o movimento válido selecionado pelo jogador, fazendo a atualização do vetor que armazena o estado atual do tabuleiro de jogo. A função irá retornar o índice do vetor onde a última pedra foi depositada.

capture

Esta função é do tipo *void* e recebe como argumentos o *pointer* para **board**, e as variáveis **currentPlayer** e **landed**. Esta função verifica se é possível realizar a captura de algumas pedras após a finalização do movimento, a partir da posição onde a última pedra foi depositada e de acordo com as regras do jogo. Caso a captura seja possível, a função vai realizar essa captura e atualizar o vetor onde está armazenado o estado atual do tabuleiro de jogo.

checkNRocks

Esta função é do tipo inteiro e recebe como argumentos o *pointer* para **board**. A função vai verificar se algum jogador não tem pedras no seu lado do tabuleiro. Se isso se verificar irá retornar o número do jogador (1 ou 2) que não tem pedras, caso ambos os jogadores tenham pedras nos respetivos lados do tabuleiro, irá retornar 0.

whenNoRocks

Esta função é do tipo inteiro e recebe como argumentos o *pointer* para **board**. Esta função chama a função *checkNRocks* para verificar o número do jogador que não tem peças no seu lado do tabuleiro. De seguida, irá verificar se existem movimentos para o jogador contrário, que terá pedras, que permitam colocar pedras no lado do tabuleiro que não as tem. Retorna 1 se existir algum movimento disponível e 0 se não existir nenhum movimento disponível.

allRocksToDeposit

É uma função do tipo *void* e recebe como argumentos o *pointer* para **board**. Esta função apenas será chamada em condições de fim de jogo muito específicas: quando não existem movimentos possíveis para colocar pedras do outro lado do tabuleiro; quando o jogo entra num ciclo infinito de movimentos em que não será possível fazer mais capturas. A função vai atualizar o vetor que armazena o estado atual do tabuleiro de jogo, recolhendo todas as pedras em jogo, em cada um dos lados do tabuleiro para os respetivos depósitos. Assim, todas as pedras do lado do jogador 1 irão para o depósito do jogador 1 e, todas as pedras do lado do jogador 2 irão para o depósito do jogador 2.

checkForEnd

É uma função do tipo inteiro e recebe como argumentos o *pointer* para **board** e o *pointer* para **currentPlayer**. Esta função permite verificar se o jogo terá alcançado condições para o fim, verificando as seguintes situações:

- Algum dos jogadores tenha mais de vinte e cinco (25) pedras no seu depósito ou ambos os jogadores tenham vinte e quatro (24) pedras nos respetivos depósitos. Para isso, é chamada a função *checkWin*.
- Algum dos jogadores não tem pedras no seu lado do tabuleiro e o adversário não tem nenhum movimento possível, no turno seguinte, que coloque lá pedras. Para fazer esta verificação são chamadas as funções *checkNRocks* e a função *whenNoRocks*. Caso não seja possível realizar mais movimentos é chamada a função *allRocksToDeposit*.
- O jogo tenha entrado numa condição de ciclo infinito, em que não é possível fazer mais capturas. A única situação considerada para se verificar este estado foi a que está descrita no enunciado do projeto, em que existem apenas 2 pedras no tabuleiro e estas estão localizadas em posições completamente opostas no tabuleiro. Se esta situação se verificar, é chamada a função *allRocksToDeposit*.

Se alguma das condições de fim de jogo descritas acima se verificarem, a função irá retornar 1, caso contrário irá retornar 0.

saveFile

Esta função é do tipo *void* que recebe como argumentos o *pointer* para **board**. A função permite guardar o estado atual do vetor que armazena o estado do tabuleiro de jogo. Caso chamada, irá pedir o nome desejado para o ficheiro onde irá guardar o progresso atual do jogo, sem a extensão do ficheiro. A função irá criar esse mesmo ficheiro de texto (.txt), no diretório onde estão os ficheiros do programa e com o nome escolhido pelo utilizador.

computerMove

É uma função do tipo inteiro e recebe como argumentos o *pointer* para o **board**, e as variáveis **currentPlayer** e **mode**. Esta função irá atuar como um "computador" no modo jogador vs. computador, retornando um movimento válido, como se fosse o jogador 2. O objetivo desta função é determinar a jogada que tem mais retorno para o depósito do computador. A função

faz uso de um *loop* para iterar por todas as casas do seu lado do tabuleiro, verificando se cada uma das jogadas será ou não válida, recorrendo, para isso, à função *checkIfValid*. Caso a jogada seja válida, a função vai copiar o vetor que armazena o estado atual do tabuleiro de jogo, para um vetor interno e temporário. Neste vetor temporário, será realizada a jogada (função *theMove*) e a captura (função *capture*). De seguida, é feita uma análise da diferença dos valores do depósito temporário e do depósito do tabuleiro de jogo. Se houver uma diferença nos valores dos depósitos, significa que esse movimento deu origem a capturas de pedras. Assim, e se o valor de capturas for maior do que o valor de capturas de outros movimentos já testados, o valor deste movimento será guardado.

Caso a função percorra todo o lado do tabuleiro do computador e não consiga fazer nenhuma captura, o computador irá realizar um movimento aleatório. Este movimento é selecionado com recurso à função *rand ()*, contida na *standard library* de C, e que retorna um valor pseudoaleatório. É, ainda, feita uma verificação sobre a validade deste movimento aleatório e, caso não seja válido, é feita uma nova tentativa a um movimento aleatório, até que um movimento válido seja encontrado. A função retorna um valor entre 7 e 12, correspondente ao movimento determinado.

loadGame

Esta função é do tipo *void* e recebe como argumentos o *pointer* para **board**, e as variáveis **argc**, do tipo inteiro, que indica o número de argumentos chamados ao correr o programa, e o vetor **argv** do tipo vetor de caracteres que guarda o conteúdo de cada argumento chamado para correr o programa. A função vai ler um ficheiro de estado de um jogo anterior, guardado pelo utilizador, e "carregar" esse mesmo estado de jogo para o vetor que armazena o estado atual do tabuleiro de jogo. Para isso, deverá ser inserido apenas um argumento extra, que corresponderá ao nome de um ficheiro .txt, guardado no mesmo diretório e que contenha os dados guardados no formato correto (descrito no enunciado do projeto).

printWinner

Esta função é do tipo *void* e recebe como argumentos o *pointer* para o **board**, e as variáveis **mode** e **win**. A função serve para imprimir o tabuleiro após uma das condições de fim de jogo tenha sido alcançada, bem como o jogador (ou computador) que venceu, de acordo com o modo

de jogo escolhido. A função imprime, ainda, um resumo do final do jogo, com o total de peças capturadas por cada jogador.

main

Esta função está no ficheiro "ouri_main.c" e é onde está codificada toda a dinâmica e procedimento da simulação do jogo. É aqui que são chamadas as outras funções já descritas e necessárias para executar o jogo. A função começa por declarar e inicializar as funções já descritas na secção "Principais Variáveis" deste relatório. A função é depois dividida em três (3) ciclos diferentes: o primeiro, onde é pedido ao jogador que escolha o modo de jogo (jogador vs. jogador ou jogador vs. computador); o segundo, onde está codificado o procedimento para o modo jogador vs. jogador; o terceiro, onde está codificado o procedimento para o modo de jogador vs. computador. Por fim, quando é encontrado um vencedor, é chamada a função *printWinner*.

Bibliografia

Ap, Kumar. «Greedy Algorithm Based Deep Learning Strategy for User Behavior Prediction and Decision-Making Support». *Journal of Computer and Communications* 06 (1 de janeiro de 2018): 45–53. https://doi.org/10.4236/jcc.2018.66004.

Ndukwe, Ifeanyi, e Evarista Nwulu. «Computerized Board Game: Dara. International Journal of Advanced Research in Computer Science». *International Journal of Advanced Research* 5 (1 de setembro de 2018).

Rovaris, Gabriele. «Design of Artificial Intelligence for Mancala Games». POLITECNICO DI MILANO, 2017.

The Unicode Standard, Version 15.1