Projeto : Fase 2

Inteligência Artificial 22/23

Prof. Joaquim Filipe

Eng. Filipe Mariano

Dots and Boxes

Dots and Boxes é um jogo de lápis e papel para dois jogadores. Este tem como objetivo fechar o maior numero de caixas possíveis. Aplicando apenas arcos ligando os pontos apresentados no tabuleiro.

Manual Técnico

Realizador por:

Dinis Pimpão -> 201901055

Pedro Peralta -> 202002153

Índice

1 - Introdução

O objetivo deste projeto é, a partir de um tabuleiro vazio do jogo **Dots and Boxes**, cada um dos jogadores, ir colocando arcos de forma fechar o maior número de caixas possível, ganhando assim o jogo.

O projeto foi implementado na linguagem de programação funcional Common Lisp.

2 - Arquitetura do Sistema

O projeto é composto por 3 ficheiros de código e 1 auxiliar de "logs" das jogadas:

- **jogo.lisp** -> Carrega os outros ficheiros de código, escreve e lê ficheiros, e trata da interação com o utilizador.
- **puzzle.lisp** -> Código relacionado com o problema.
- **algoritmo.lisp** -> Contem a implementação do algoritmo minimax com cortes alfabeta.
- **log.dat** - ficheiro do historico das execuções do algoritmo alfa-beta cada vez que o computador joga. Guarda as jogadas realizadas, nomeadamente o novo estado, o número de nós analisados, o número de cortes alfa e beta e o tempo de execução.

Para iniciar o programa, deve-se carregar o ficheiro `jogo.lisp` e este irá carregar os outros 2 ficheiros auxiliares.

3 - Entidades e Tipos Abstratos de Dados

Tabuleiro

O tabuleiro é basicamente uma matriz 7x6, representado por uma lista com duas listas:

- **Lista de Linhas Horizontais**: representa os arcos horizontais. Este lista é composta por 6 listas, em que cada uma delas é composta por 6 elementos. O valor de cada elemento é 0 se não existir arco nessa posição, 1 se tiver sido o Jogador 1 a colocar o arco e 2 se tiver sido o Jogador 2 a colocar o arco.
- **Lista de Linhas Verticais**: representa os arcos verticais. Esta lista é composta por 7 listas, em que cada uma delas composta por 5 elementos. O valor de cada elemento é 0 se não existir arco nessa posição, 1 se tiver sido o Jogador 1 a colocar o arco e 2 se tiver sido o Jogador 2 a colocar o arco.

Nó

Um nó é constituido por um estado, que é nada mais que um tabuleiro, acompanhado do número de caixas fechadas por cada jogador. Possui também a sua profundidade e o número de caixas que deu origem àquele nó (explicação a ser vista mais à frente).

Representacao grafica do problema (com alguns arcos colocados e caixas fechadas):

```
+--+--+--+--+  +--+
|  |  |  |  |     |
+--+--+--+--+--+  +
   |  |  |  |  |  |
+  +--+--+--+--+  +
|  |  |  |  |  |  |
+  +--+  +--+--+  +
   |  |  |     |  |
+--+--+  +  +  +  +
   |     |     |
+  +--+--+--+--+--+
```

Operadores

Para esta segunda fase do projeto, os operadores em si mantiveram-se iguais à primeira.

Existem dois tipos de operadores, um operador para aplicar uma linha na horizontal e outra na vertical. É passado por parâmetro a localização da linha a ser aplicada e o estado do tabuleiro.

Exemplo de utilização:

- `(arco-horizontal 0 1 '(((0 0 0) (0 0 1) (0 1 1) (0 0 1)) ((0 0 0) (0 1 0) (0 0 1) (0 1 1)))`
- `(arco-vertical 0 2 '(((0 0 0) (0 0 1) (0 1 1) (0 0 1)) ((0 0 0) (0 1 0) (0 0 1) (0 1 1)))`

Para sabermos onde podemos aplicar os operadores, foi desenvolvida uma função auxiliar para obter os indíces de arcos vazios.

```
(get-arcos-horizontais (tabuleiro-teste)) => ((0 0 0) (0 0 1) (0 1 1) (0 0 1))

(get-indices-arcos-vazios (get-arcos-horizontais (tabuleiro-teste))) => ((1 1) (1
2) (1 3) (2 1) (2 2) (3 1) (4 1) (4 2))
```

No caso do jogador humano realizar a sua jogada e tentar colocar um arco fora dos limites do tabuleiro, ambas as funções de aplicação dos arcos irão retornar o valor nil em vez do novo estado do tabuleiro, podendo assim garantir que o utilizador irá sempre realizar uma jogada lógica e válida.

4 - Algoritmo e Implementações

Como algoritmo utilizado neste projeto havia 2 opções de algoritmos, sendo estes MiniMax ou Negamax, com cortes alfa-beta. Neste caso, foi escolhido implementar o minimax com cortes alfa-beta.

Para facilitar a implementação, o algoritmo foi dividido em 3 funções principais:

- **alfabeta** -> Função principal que verifica se o nó é terminal (tabuleiro preenchido), se chegou à profundidade máxima ou se esgotou o tempo máximo de pesquisa, avaliando o nó e retornando o resultado. Caso não seja um nó terminal, conforme a sua profundidade, irá chamar uma das funções auxiliares (max ou min), sendo que caso a profundidade seja par, irá chamar max, caso contrário, min.
- **alfabeta-max** -> Função auxiliar à função principal alfabeta, que inicialmente avalia se há cortes beta `(alfa >= beta)`, caso haja retorna o valor beta. Caso contrário, chama a função alfabeta com o 1º elemento dos sucessores para o resultado deste ser avaliado e colocado numa função `max` com o valor atual do alfa. Por fim, chama a própria função para avaliar os restantes elementos na lista de sucessores.
- **alfabeta-min** -> Função semelhante à `alfabeta-max` mas que avalia se há cortes alfa `(beta <= alfa)` e caso haja, retorna o valor de alfa. Chama a função alfabeta tal como anteriormente referido, mas desta vez, avaliando através de uma função `min` o beta atual e o valor retornado da chamada da função alfabeta.

Um exemplo das funções `alfabeta-max` e `alfabeta-min` pode ser visualizado abaixo:

```
(alfabeta-max (cdr sucessores) (max a (alfabeta (car sucessores) max-depth a b))
b)
(alfabeta-min (cdr sucessores) a (min (alfabeta (car sucessores) max-depth a b)))
```

Para o algoritmo funcionar corretamente, foi também necessário recorrrer a funções auxiliares, dependentes do dominio do problema, nomeadamente para avaliar o valor de um nó e para ordenar os nós sucessores gerados, de forma a obter um maior número de cortes alfa-beta durante a execução do algoritmo, aumentando assim a sua eficiência.

Função utilizada para avaliar um nó, confome quem é o jogador inicial (Max)

```
(defun avaliar (no)
  (cond ((= *initial-player* *jogador1*) (- (caixas-fechadas-jogador *jogador1*
no) (caixas-fechadas-jogador *jogador2* no)))
        (t (- (caixas-fechadas-jogador *jogador2* no) (caixas-fechadas-jogador
*jogador1* no)))
  )
)
```

Ordena os nós através da função avaliação anterior

```
(sort nos-sucessores #'< :key #'avaliar)
```

Um extra, que poderá ajudar a ganhar o campeonato que existe após a entrega deste projeto, foi o desenvolvimento de uma 2ª versão da função de avalição, tem em conta mais variáveis, sem ser apenas o número de caixas dos jogadores. Neste caso, será beneficiado com 10 pontos por cada caixa que feche (favorecer um estado que fecha mais que 1 caixa, caso possível) e "prejudicado" por 5 pontos por cada "caixa" que deixe a 1 arco de fechar (caixas com 3 lados completos, na próxima jogada, o jogador oposto fecha e ganha o ponto).

```
(defun avaliar-v2 (no)
  (cond ((= *initial-player* *jogador1*) (- (- (* (caixas-fechadas-jogador
*jogador1* no) 10) (* (3-side-boxes (no-tabuleiro no)) 5)) (* (caixas-fechadas-
jogador *jogador2* no) 10)))
        (t (- (- (* (caixas-fechadas-jogador *jogador2* no) 10) (* (3-side-boxes
(no-tabuleiro no)) 5)) (* (caixas-fechadas-jogador *jogador1* no) 10)))
  )
)
```

5 - Resultados Finais

Fazendo uma breve análise das estatisticas geradas no ficheiro de log, com um tempo máximo de 5000ms por jogada do computador, podemos observar os seguintes valores:

Tempo	Nós Analisados	Cortes Alfa	Cortes Beta
328ms	7037	68	1225
313ms	6632	66	1158
288ms	6363	64	1088
344ms	6256	62	1053
677ms	6114	60	1014
1916ms	5340	58	967

Tempo	Nós Analisados	Cortes Alfa	Cortes Beta
4085ms	5459	56	994
4526ms	4933	54	718
4522ms	4530	52	741
3561ms	4012	50	677
4231ms	3882	48	643
4471ms	3264	46	608
4844ms	2824	44	341
4592ms	2970	42	400
4513ms	2247	40	366
3354ms	2200	38	375
3412ms	2082	36	346
4545ms	441	0	46
5546ms	547	0	40
4730ms	962	62	264
3ms	21	0	10

Como se pode observar na tabela acima, no inicio do jogo, quando o tabuleiro ainda se encontra práticamente vazio, são analisados muitos mais nós, em muito menos tempo, devido ao elevado número de cortes alfa-beta feitos. Conforme o jogo progride, vai havendo menos possibilidade de cortes, o que faz com que jogada demore mais tempo a analisar todos os estados sucessores possiveis, e por vezes, chegando ao tempo limite de execução, forçando assim a devolver a melhor solução encontrada nesse tempo.

6 - Limitações Técnicas

O nosso projeto contém algumas limitações, nomeadamente:

1. Poderia ter sido implementada uma técnica de memoização através do uso de uma hash-table acelerar o processo de geração/avaliação de sucessores, uma vez que não seria necessário fazer a geração e respetiva avalição de nós "repetidos", uma vez que esta ficaria em "cache" nesta hash-table.
2. Utilização de uma depth máxima maior, uma vez que daria ao algoritmo uma maior "visão" das jogadas possíveis, podendo assim avaliar e escolher uma melhor jogada. O problema de uma maior depth, é que demora mais tempo a avaliar todos os nós e sendo que também existe uma limitação no tempo máximo de uma jogada do computador, poderia não compensar.