Manual Técnico

Inteligência Artificial - Projeto 2

Projeto realizado por: Francisco Zacarias nº190221105

Índice

1 Introdução

Este manual, sendo a segunda parte de um projeto cujo jogo a ser implementado é o mesmo do presente, irá conter algumas redundâncias em termos de explicações e apresentação de conceitos. É, no entanto, uma decisão consciente para evitar que seja necessário ler o primeiro manual para compreender este.

Este manual apresenta um desenvolvimento mais técnico sobre o projeto. Vão ser apresentadas as estruturas de dados desenvolvidas para o funcionamento do projeto, uma análise compreensiva ao algoritmo *negamax* com cortes *alpha beta*, à função de utilidade implementada e discussão sobre as limitações técnicas do projeto, comparadas com o enunciado. Este manual assume que o leitor tem conhecimentos básicos do *syntax* da linguagem *Lisp*, assim como conhecimento das regras do jogo do Quatro. O objetivo deste projeto foi desenvolver um programa que capaz de jogar o jogo do quatro através de uma tomada de decisões inteligente. É então possível jogar contra o computador (PC vs Humano) ou fazer dois computadores jogarem um contra o outro (PC1 vs PC2).

1.1 Ficheiros *Lisp*

O projeto está separado em 3 ficheiros .lisp:

- algoritmo.lisp - Inclui a implementação genérica do algoritmo *negamax* com cortes *alpha beta*.
- jogo.lisp - Inclui código relacionado com problema, I.e. funções especificas ao domínio de aplicação, assim como operadores e a função que calcula a utilidade dos nós.
- interact.lisp - Inclui o código relacionado com a interação com o utilizador, assim como toda a escrita e leitura de ficheiros.

1.2 Ficheiro *Dat*

O projeto 1 ficheiro .dat:

- logs.dat - Ficheiro onde serão escritas as estatísticas de cada jogada. Para as jogadas do computador, será escrito o tempo de execução do algoritmo, quantos cortes *alpha beta* foram feitos, nós analisados, utilidade do nó e tabuleiro devolvido. Para as jogadas do utilizador, são escritas as coordenadas da jogada, a peça jogada e o tabuleiro resultante do operador.

2 Implementação do problema

Neste capítulo será feita uma visão global das decisões de implementação tomadas, tanto na forma como as estruturas de dadas foram estruturadas, como nas funções auxiliares mais relevantes para o funcionamento do jogo.

2.1 Estrutura de dados

O jogo do Quatro é constituído por um tabuleiro de dimensões 4 x 4 e pecas que podem estar dentro do mesmo, ou fora (na chamada reserva). Para representar um tabuleiro vazio, onde todas as peças estão na reserva, iriamos utilizar a seguinte estrutura de dados:

```
(
    ; Tabuleiro
    (
        (0 0 0 0)
        (0 0 0 0)
        (0 0 0 0)
        (0 0 0 0)
    )
    ; Reserva
    '(
        (branca quadrada alta cheia)
        (branca quadrada alta oca)
        (branca quadrada baixa oca)
        (branca quadrada baixa cheia)
        (preta quadrada alta cheia)
        (preta quadrada alta oca)
        (preta quadrada baixa oca)
        (preta quadrada baixa cheia)
        (branca redonda alta cheia)
        (branca redonda alta oca)
        (branca redonda baixa cheia)
        (branca redonda baixa oca)
```

```
            (preta redonda alta cheia)
            (preta redonda alta oca)
            (preta redonda baixa cheia)
            (preta redonda baixa oca)
        )
    )
```

Esta lista em *lisp* é então a representação do estado do problema, onde neste caso, representa um tabuleiro vazio.

Aqui identificamos uma lista que contém duas sublistas.

- A primeira sublista representa o tabuleiro do jogo, quatro listas com quatro posições, onde a posição 0 representa uma célula vazia.
- A segunda sublista representa as peças na reserva, i.e. peças que ainda não foram jogadas.

3 Negamax com cortes *alpha beta*

O algoritmo implementado neste projeto foi o *negamax* com cortes *alpha beta*. O *negamax* é essencialmente uma otimização do algoritmo *minimax* ao tirar partido da relação `max(a,b) = -min(-a,-b)`.

3.1 Negamax

O algoritmo *negamax* surge perante uma otimização do algoritmo *minimax*, como já foi falado. Isto significa que essencialmente os algoritmos são bastante parecidos em termos implementação e por essa mesma razão, vou começar por explicar como funciona o algoritmo *minimax*.

3.1.1 Minimax

O *minimax* é um algoritmo de procura que procura a melhor jogada seguinte ao estado que lhe foi dado. Esta escolha é feita através da travessia da árvore de espaço de estados, onde cada nível de profundidade representa a possível jogada de um dos dois jogadores. Cada jogador é representado como o *max* e o *min* no algoritmo, sendo que o nível de profundidade 0 (o estado inicial passado ao algoritmo) é sempre o *max*.

Através da função de utilidade definida (específica ao domínio da aplicação), o algoritmo calcula, para cada nó, o valor da sua utilidade. Isto é fundamental para identificar a melhor jogada, sendo que se estivermos a calcular a jogada *max*, queremos escolher o nó sucessor (a jogada seguinte) com a maior utilidade, e para calcular a jogada do *min* queremos escolher a jogada seguinte com a utilidade menor, a fim de reduzir as chances do adversário e aumentar as nossas.

Isto significa que o algoritmo minimax tem que identificar, para cada nó, se deve calcular o min ou o max através de uma condição. É esta ineficiência que o *negamax* vai colmatar.

3.1.2 Negamax - Otimização do *minimax*

O algoritmo *negamax* segue a mesma filosofia do *minimax*, onde a sua diferença está na identificação do *max* e do *min*, 1 e -1 respetivamente. Assim, a cada iteração, não é necessário fazer a condição que verifica o *min e max* a cada nó, basta multiplicar o valor do jogador respetivo com a utilidade do nó e aplicar a propriedade respetiva `max(a,b) = -min(-a,-b)`.

3.1.2 Negamax - Implementação

Aqui vou apresentar a minha implementação do *negamax*, em lisp

Esta implementação do algoritmo foi retirada diretamente do projeto, pelo que contém algumas chamadas a funções de *closures* a fim de guardar dados estatísticos da sua execução. Esta implementação não ordena os nós dos sucessores. A razão será aprofundada no capítulo 4. Limitações técnicas.

O algoritmo foi implementado em 3 funções diferentes:

```
(defun negamax (estado profundidade-maxima avaliacao tempo-limite)
"Executa o algoritmo negamax"
    (funcall 'restaura-valores)
    (let*(
            (tempo-inicial (get-internal-run-time))
            (resultado (funcall 'negamax-algorithm estado profundidade-maxima
avaliacao 'sucessores 0))
            (tempo-total (- (get-internal-run-time) tempo-inicial))
        )

        ; Se excedeu o limite de tempo, não retorna nada
        (if (> tempo-total tempo-limite)
            (append (list -1 -1 -1 -1 -1 estado))
            (append
                (list tempo-total)
                (list resultado)
                (funcall 'algoritmo-resultado)
            )
        )
    )
)
```

A função *negamax* é a função que deve ser chamada para executar o algoritmo. Serve essencialmente para abstrair algumas operações que têm que ser realizadas a cada chamada, tal como calcular o tempo de execução e criar a lista de retorno das estatísticas.

```
(defun negamax-algorithm (estado profundidade-maxima avaliacao gera-sucessores
profundidade-atual &optional (alpha *minus-infinity*) (beta *plus-infinity*)
(maximizing-player 1))
    ; Incrementa um no nos analisados
    (funcall 'adicionar-analisado)

    (if
        ; Condições de paragem
        (or
            (equal profundidade-maxima 0)
            (tabuleiro-solucao (tabuleiro estado))
        )
        (* (funcall avaliacao estado) maximizing-player)
        (let* (
```

```
                (sucessores (funcall gera-sucessores estado))
                ;(sucessores (funcall 'ordena-nos sucessores avaliacao maximizing-
player))
            )

            (cond
                ; Se não houver sucessores retorna a utilidade deste nó
                ((null sucessores) (* maximizing-player (funcall avaliacao
estado)))
                ; Senão devolve a utilidade dos nós sucessores
                (t (funcall 'negamax-sucessores sucessores profundidade-maxima
avaliacao gera-sucessores alpha beta maximizing-player (1+ profundidade-atual)))
            )
        )
    )
)
```

Esta função está encarregada de pegar num nó (um estado) e a gerar os sucessores do mesmo. Caso não tenha sucessores, vai retornar a utilidade deste nó. Caso tenha sucessores, vai chamar outra função `negamax-sucessores` que vai calcular a utilidade dos sucessores.

```
(defun negamax-sucessores (sucessores profundidade-maxima avaliacao gera-
sucessores alpha beta maximizing-player profundidade-atual &optional (utilidade
*minus-infinity*))
    (if
        ; Quando todos os nós forem avaliados, retorna a utilidade
        (null sucessores)
        utilidade
        (let* (
                (sucessor (car sucessores))
                (utilidade (max utilidade (- (funcall 'negamax-algorithm sucessor
(1- profundidade-maxima) avaliacao gera-sucessores profundidade-atual (- 0 beta)
(- 0 alpha) (- 0 maximizing-player)))))
                (alpha (max utilidade alpha))
                (corte (if (>= alpha beta) t nil))
            )

            ; Verifica novas soluções
            (cond
                ;Se não existe solução e for profundidade 1, adiciona o atual
                ((and (equal profundidade-atual 1) (equal (funcall 'no-solucao)
nil))
                        (funcall 'adiciona-solucao sucessor)
                )
                ; Se a profundidade for 1 e a utilidade for maior que a do atual,
altera a solucao
                ((and (equal profundidade-atual 1) (> (funcall avaliacao sucessor)
(funcall avaliacao (funcall 'no-solucao))))
                        (funcall 'adiciona-solucao sucessor)
                )
            )
```

```
            ; Adiciona estatisticas do corte
            (if (not (null corte))
                (funcall 'adiciona-corte maximizing-player)
            )

            ; Se houve corte, retorna utilidade do sucessor, se não segue para os
restantes sucessores
            (if (not (null corte))
                utilidade
                (negamax-sucessores (cdr sucessores) profundidade-maxima avaliacao
gera-sucessores alpha beta maximizing-player profundidade-atual utilidade)
            )
        )
    )
)
```

Por fim temos a função negamax-sucessores. Esta função e a negamax-algorithm são mutuamente recursivas. O que esta função pretende fazer é chegar ao nó com a maior profundidade definida (a profundidade máxima foi definida durante a chamada à função *negamax*) e devolver a sua utilidade. Assim que chega ao último nó, começa a subir a árvore atribuindo os valores de *alpha* e *beta* respetivos e aplicar os cortes necessários.

3.2 Estatísticas

Desta vez, as estatísticas do algoritmo são guardadas numa *closure*. Uma *closure* é um ambiente léxico onde são guardados valores durante toda a execução do programa, sem que existam repercussões destrutivas para fora desse ambiente.

É portanto implícito que neste ambiente são usadas funções destrutivas, tais como *setf* e *incf* para manipular os valores das variáveis que representam as estatísticas do algoritmo. Estas modificações são feitas através de funções definidas dentro da closure que podem ser chamadas fora da mesma.

3.3 Função de utilidade

A função de utilidade usada para calcular a utilidade de cada nó é bastante simples. É uma abordagem que se baseia apenas no máximo numero de peças alinhadas, numa posição válida para vencer, existem. Portanto a utilidade é calculada da seguinte forma:

1 Peça	10 pontos
2 Peças	100 pontos
3 Peças	1000 pontos
4 Peças	10000 pontos

Isto é uma abordagem que ignora as características das peças, relembrando que para vencer é necessário que 4 peças com pelo menos uma característica em comum estejam alinhadas. Isto significa que é uma função de

utilidade naturalmente inclinada para jogadas arriscadas, na medida em que é mais agressiva a introduzir peças alinhadas para chegar ao fim do jogo.

4. Limitações técnicas

Este projeto sofre de várias limitações técnicas que serão explicadas.

4.1 Ordenação dos nós

O algoritmo não faz a ordenação dos nós sucessores, quando gerados para cada estado. Isto foi uma decisão consciente porque afeta muito a performance do algoritmo.

A ordenação que eu estava a fazer era, com base na utilidade, de forma crescente. Vou apresentar as estatísticas que me levaram a decidir não permitir a ordenação de nós:

Ordenação Nós	**Algoritmo**	**Estado**	**Profundidade**	**Tempo-Execução**	**Nós analisados**	**Cortes Max**	**Cortes Min**
Não	Negamax	(teste2)	3	187ms	18196	216	143
Sim, crescente	Negamax	(teste2)	3	9344ms	20651	120	143
Sim, decrescente	Negamax	(teste2)	3	21547	34962	3117	143

O estado (teste2) está implementado no ficheiro jogo.lisp.

Ao analisar estes resultados, a fim de fazer o jogo mais rápido e dentro do intervalo de tempo pedido [1000ms - 5000ms], decidi não fazer a ordenação de nós.

O meu objetivo com este capítulo sobre a ordenação dos nós é reconhecer que existe esta baixa performance com a ordenação que possivelmente é consequente da minha implementação do *negamax*. Portanto a função da ordenação está implementada, mas o algoritmo não a está a realizar, pelas razões anteriormente apresentadas. Com isto, apenas pretendo justificar a minha escolha, mas obviamente reconheço e assumo que algo não está a funcionar como esperado, e isso é efetivamente um erro da minha implementação.

4.2 Memoização e Procura quiescente

Estas duas funcionalidades não foram implementadas no projeto.

4.3 Verificação de inputs

Este projeto não faz qualquer tipo de verificações de input do utilizador. Quando o jogador inicia o jogo, quando é pedido a profundidade máxima ou tempo máximo por exemplo, o programa não faz qualquer tipo de filtro ao *input*, pelo que qualquer introdução inválida levará ao mau funcionamento do projeto.