IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

TIAGO ANDRÉ MENDES DE ALMEIDA RIBEIRO

TIAGO ALEXANDRE QUARESMA ALVES

Adji–boto - Manual Técnico

Inteliêngia Artificial, Engenharia de Software

Ano letivo 2018 / 2019

Docentes

Professor, Joaquim Filipe

Professor, Hugo Silva

Engenheiro, Filipe Mariano

Índice

1. Introdução

Adji-boto é uma aplicação desenvolvida baseada num jogo em Common Lisp para a unidade curricular de Inteligência Artificial. Esta fase do projeto consiste em adapter o que foi feito anteriormente de modo a adicionar uma componente "multiplayer", onde existem dois modos: Máquina contra Máquina e Humano contra Máquina. Para este efeito, foi utilizado o algoritmo Minimax com cortes alfabeta, também conhecido por alfabeta.

O modo de jogo é escolhido pelo utilizador, bem como o tempo máximo de execução do algortimo. Caso o utilizador escolhesse o modo Humano contra Máquina, também tinha que indicar quem era o primeiro jogador. Ao longo do jogo são mostrados "logs" no ecrã (e gravados em ficheiro) sobre o estado atual do jogo.

Em termos de jogo, neste caso é uma variante do Adji–boto e Oware, especialmente concebida para este projeto, e tem como objetivo permitir ao computador vencer o jogador humano ou um outro computado.

2. Estrutura do Sistema

Esta aplicação está dividade em três ficheiros *.lisp*, de modo a facilitar a manutenção e compreensão humana do programa: puzzle, procura e projeto. São ainda utilizados dois ficheiros *.dat* de *input* e *output* : problemas, resultados.

Jogo

Este ficheiro é o antigo "puzzle", um ficheiro que contém toda a informação relativa ao domínio de aplicação. Praticamente nada foi alterado, apenas acrescentadas mais uma ou duas funções que foram necessárias.

Algoritmo

O antigo "procura", neste ficheiro encontram-se todas as funções, quer principais como de auxílio, associadas à execução do algoritmo alfabeta

Interact

Por fim, o ficheiro interact que tem tudo o resto, nomeadamente funções I/O (compilar e carregar os outros ficheiros, gravar output) e trata de toda a interação com o utilizador. Também funções de testes para facilitar o teste e desenvolvimento do algoritmo.

Log

Ficheiro produzido à medida que o programa é executado onde são registadas as jogadas feitas, tanto pela máquina como pelo humano, ao longo do jogo. Este ficheiro nunca é substituído em jogos subsequentes, sendo que há uma seperação com recurso ao caracter "-" entre os vários jogos. Também para ajudar quem estiver a ler o ficheiro a perceber os resultados, no inicio de cada jogo é escrito no ficheiro o tabuleiro inicial que será utilizado no jogo.

3. Implementação técnica

Tirando o facto de ter sido utilizado um paradigma funcional devido ao LISP ser uma linguagem funcional, foram-nos impostas algumas limitações ténicas extra. A implementação deste projeto foi toda feita com foco na recursividade, não havendo nenhum ciclo existente no programa, tirando alguns casos exceptions onde optamos por não utilizar na mesma. Foi-nos também impedido o uso de sequênciação e funções com efeitos secundários, à exceção de aquando de interação com o utilizador, sendo que nestes casos era necessário sempre imprimir mensagens na consola ou ficheiro (com uso de format que é uma função com efeitos secundários) e executar um outro comando.

4. Alfabeta

Na implementação deste algoritmo foram necessárias algumas funções auxiliares de modo a conseguirmos atingir sucesso. Não há necessidade de falar de todas ao pormenor, mas faz sentido mencionar uma ou outra antes de passar às análise dos resultados.

Também é de notar que nesta parte, tanto por questões de simplicitade como performance foram utilizadas variáveis globais. Apenas duas com de facto impacto na aplicação, sendo as outras três meramente para efeitos estatísticos de cada jogada calculada pelo algoritmo.

Função avaliação

Uma das fundamentais é a denominada funcao-avaliacao. Esta função, embora simples é de extrema importância. O objetivo dela é avaliar o custo do nó aquando de chegar a um nó folha (quer seja por ter chegado ao fim do ramo, limite forçado ou acabado o tempo de execução). Depois de algum debate sobre esta função o grupo decidiu criar uma que prioritizava jogadas que levassem a peças serem comidas. Para descobrir isto, simplesmente subtraímos o número total de peças do tabuleiro do antecessor direto ao tabuleiro atual.

```
(defun funcao-avaliacao (no)
  "Valida quantas pecas foram \"comidas\" entre o tabuleiro atual e o
  (let ((valorA (board-value (get-node-state no)))
        (valorB (board-value (get-node-state (get-node-parent no)))))

    (- valorB valorA))
)
```

Figura 1 - Função avaliação

Estrutura de dados

Embora similar à parte 1, para esta fase foi necessário alterar a estrutura de dados usada. Já não sendo necessários alguns dos atributos, em troca de outros.

```
(defun create-node (board &optional (depth 0) (position -1) (parent nil))
  "Construtor do no das arvores para os algoritmos"
  (list board depth position parent)
)
```

Figura 2 - Estrutura de nó

O custo do nó e qual a heurística a ser usada para o cálculo do nó já não são precisas. No entanto é acrescentado outro campo, apenas para fins estatísticos e de informação ao utilizador que é o "position". Este campo diz-nos que posição foi escolhida para gerar este nó. Isto é útil para depois da jogada ser calculada pelo algoritmo podermos dizer ao utilizador algo do gênero "A máquina jogou na casa x".

Análise

Em termos de algoritmo conseguímos que ele tenta de facto procurar uma jogada ótima, sendo que em jogos sucessivos nas mesmas condições o caminho encontrado é sempre o mesmo. E no caso de Máquina contra Máquina, o resultado final é sempre igual.

Por questões de viabilidade, e utilização de recursos, a geração da árvore está limitada a 5 níveis, sendo que mesmo um nível tão baixo já gera várias sucessores.

Através da estatísticas de cada jogada, podemos observar que as jogadas não demoram muito tempo a serem calculadas, ficando a cerca de 0.1 milisegundos por jogada. Isto deve-se ao facto de apenas ser permitida uma profundidade máxima de 5.

Também conseguímos observar o quanto duas jogadas praticamente idênticas podem ser bastante diferentes em termos de cálculos. Num jogo onde o tabuleiro inicial era composto por apenas casas com 8 peças, podemos ver através das seguintes figuras que a segunda jogada, embora exatamente idêntica à primeira mas na casa oposta demorou quase o quadruplo do tempo a calcular a jogada.

Conseguímos também ver que, devido às nossas máquinas pessoais serem "decentes", e o limite de profundidade, o algoritmo nunca se aproxima do tempo limite de, para este teste, 1 segundo.

No entanto não é tudo pontos positivos. Devido a esta limitação na árvore de procura, sofremos de um problema que é o facto de não conseguírmos chegar ao nó folha "oficial". Isto implica que nunca chegamos a calcular a jogada verdadeiramente ótima, que resulta em jogadas extra. Ou jogadas que no nível 5 parecem que estão a caminho da jogada ótima mas que depois nos próximos 5 níveis, se estes fossem gerados, via-mos que de facto esta jogada leva à vitória do adversário.

```
> Maquina 0 jogou
  - na casa 1
  - cortou 4 no(s) alfa
  - cortou 4 no(s) beta
  - avaliou 18 no(s)
  - demorou 0.016 milisegundos(s)

(0 8 8 8 9 9)
(9 9 9 9 9 9)
```

Figura 3 - Jogada #1

```
> Maquina 1 jogou
  - na casa 6
  - cortou 16 no(s) alfa
  - cortou 16 no(s) beta
  - avaliou 90 no(s)
  - demorou 0.06 milisegundos(s)

(1 9 9 9 10 10)
(10 10 10 9 9 0)
```

Figura 4 - Jogada #2

5. Limitações técnicas

Uma das limitações mais evidentes foi de facto o IDE necessário à criação da aplicação. O LispWorks é um IDE, que embora tenha algumas ferramentas boas como o tracer, debugger, etc, carece várias funcionalidades e "quality of life improvements" que muitos outros IDEs já implementaram, tornando a nossa experiência enquanto programadores horrenda. Não falando no facto de só termos acesso à versão gratuída lmita memória *heap*, forçando-nos a limitar a geração da árvore de procura muito cedo e o número de vezes que o IDE fecha aleatóriamente numa tentativa de nos convencer a comprar uma versão acima.

A falta de tempo por parte do grupo foi ainda mais sentida nesta fase por vários motivos, que nesta parte fez com o código criado não fosse o melhor que o grupo conseguiria.