PMR3201 - Computação para Mecatrônica

Prof. Thiago de Castro Martins
Prof. Newton Maruyama
Prof. Marcos de S.G. Tsuzuki
Monitor: Pietro Teruya Domingues

Exercício Programa 2 - Versão 2017

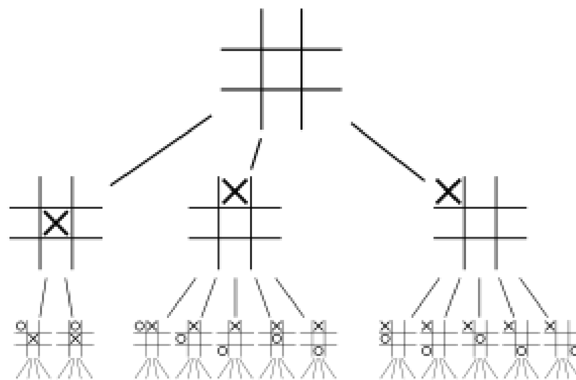
Resolvendo o Jogo da Velha (Tic-Tac-Toe)

- **DATA FINAL DE ENTREGA: Terça-Feira 06/06/2017**
- **O exercício deve ser feito individualmente.**
- **Submeta através do sistema MOODLE:**
  - **Relatório de no máximo 2 páginas A4 explicando o projeto do seu programa (formato PDF).**
  - **Código fonte (arquivo *.py).**
  - **Compactar os dois ítens descritos acima num arquivo .zip.**

1 Introdução

Neste exercício programa propõe a implementação de um algoritmo que pode decidir sobre qual a melhor jogada que pode ser realizada para se tentar vencer o Jogo da Velha.

O algoritmo é baseado na construção de uma estrutura denominada Árvore de Jogo que pode ser interpretada como uma simulação de todas os possíveis sequencias de jogadas que podem ser realizadas por jogadores oponentes.



2 Game tree

Na teoria dos jogos a Árvore de Jogo é uma estrutura do tipo grafo direcionado que representa o jogo. Os nós (*nodes*) representam estados do jogo enquanto que os arcos (*arcs*) representam ações. Uma árvore

de jogo completa contêm todos os possíveis 'jogos' (todos os possíveis caminhos ou sequencias de ações e estados) que podem ocorrer. Os nós-terminais (*leaf-nodes*) representam um estado correspondente ao final de um jogo, ou seja, vitória de um dos jogadores ou empate.

Uma árvore de jogo pode ser interpretada como uma simulação que gera todos os possíveis jogos.

Partindo-se de um determinado estado, a utilização de uma Árvore de Jogo permite estabelecer uma ação (jogada) ótima para um determinado jogador analisando-se idealmente todos os possíveis caminhos a serem percorridos.

Devido à complexidade exponencial, na prática, somente é possível analisar a simulação de algumas jogadas a frente. Por exemplo, para um jogo de xadrez seria possível analisar todos os possíveis jogos ? Existem estimativas do número de jogos possíveis para um jogo de xadrez da ordem de 1.0×10^{37} o que torna bastante difícil o cálculo da melhor jogada em tempo real.

Em 1996 o supercomputador da IBM Deep Blue com 256 processadores projetado especialmente para jogar xadrez e capaz de analisar 200 milhões de posições por segundo realizou um confronto com Gary Kasparov. Placar: Kasparov 3 vitórias, 2 empates e 1 derrota. Um novo confronto em 1997 resultou em: Deep Blue 2 vitórias, 3 empates e 1 derrota.

Para um jogo de GO é estimado um número de jogos em torno de 1.74×10^{172}.

O Jogo da Velha clássico com um tabuleiro de 3×3 é (outras versões existem com tabuleiros 2D de maior dimensão ou tabuleiros 3D) obviamente um jogo de baixa complexidade.

3 Complexidade do jogo

Uma abordagem ingênua sobre o jogo seria considerar que existem 9 posções no tabuleiro que podem ser preenchidas pelos símbolos 'X', 'O' e 'b' (blank) o que resultaria em 19.683 ($= 3^9$) estados do tabuleiro e 362.880 (= 9!) possíveis jogos diferentes.

Entretanto uma análise mais cuidadosa deve considerar os seguintes aspectos:

- O jogo termina assim que um jogador consegue estabelece uma sequencia de 3 símbolos consecutivos,
- Se o jogador 'X' começa o jogo o número de símbolos 'X' é sempre maior ou igual o número de símbolos 'O'.

Se forem considerados ainda simetrias (rotações e reflexões) somente existem 138 estados terminais do jogo. Assumindo que o jogador 'X' sempre inicia o jogo, obtem-se:

- 91 posições distintas com vitória do jogador 'X'
- 44 posições distintas com vitória do jogador 'O'
- 3 posições distintas com empate.

4 Algoritmo Minimax

Na Teoria dos Jogos, o algoritmo minimax calcula ações que possam minimizar a perda máxima de um jogador. O cálculo da melhor jogada requer a construção de uma árvore de jogo.

Por exemplo, no Jogo da Velha existem dois jogadores que alternam jogadas. O jogador 1 será identificado aqui pelo símbolo 'X' que utiliza no tabuleiro e o jogador 2 será identificado pelo símbolo 'O'.

Por hipótese vamos considerar que o jogador 'X' sempre inicia o jogo. O algoritmo é executado dentro da perspectiva de que 'X' deve minimizar a perda máxima.

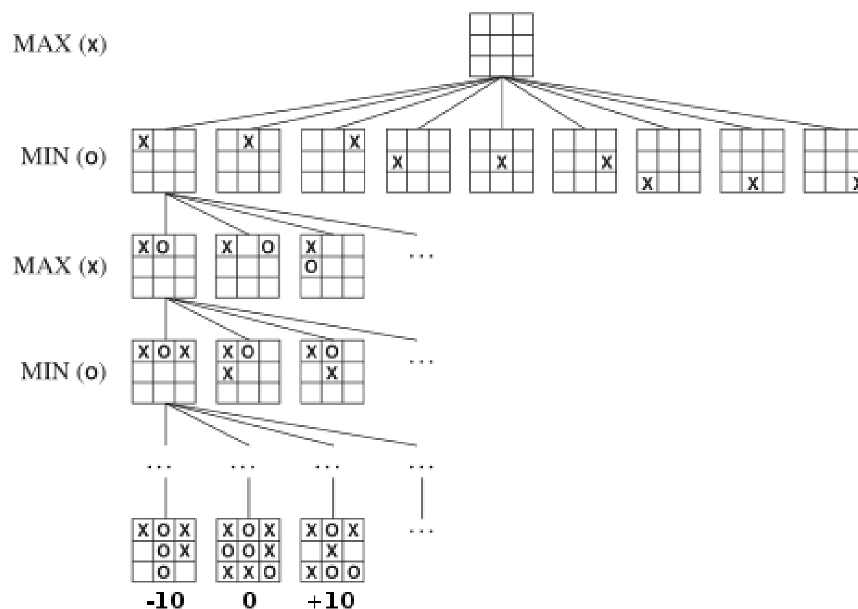
Obviamente a perspectiva do jogador 'O' é de estabelecer jogadas para maximizar a perda máxima do outro jogador.

Se o problema tem baixa complexidade é possível executar o programa para todas as jogadas. Obviamente os cálculos são sempre os mesmos. Uma estrutura de dados do tipo árvoe poderia ser mantida para armazenar os cálculos para cada estado do jogo.

O algoritmo proposto não utiliza uma árvore explicitamente mas sim uma função recursiva. cada instancia da função `minimax()` representa um possível estado do jogo.

O algoritmo deve atribuir uma pontuação positiva, por exemplo $+10$, para todo o jogo em que 'X' é vencedor, e atribuir a mesma pontuação mas negativa, -10, para todo o jogo que 'O' for vencedor. Obviamente, todos os jogos que resultam em empate recebem a pontuação nula 0.

A figura abaixo ilustra um trecho de uma árvore do Jogo da Velha.



Um possível algoritmo minimax é apresentado a seguir:

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state :
        return value of the board

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            create newboard
            value = minimax(newboard, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal
    else :
        bestVal = +INFINITY
        for each move in board :
            create newboard
            value = minimax(newboard, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Uma pontuação fixa positiva ou negativa não leva a uma situação de decisão ótima pois todos os caminhos que levam à vitória do jogador 'X' recebem a mesma pontuação. Mas certamente existem jogadas melhores que outras. As melhores jogadas são as que permitem o jogador 'X' vencer mais rápido, ou seja, com um número de jogadas menor.

Dessa forma, podemos uma pontuação básica para a vitória e descontar o valor da profundidade da árvore em que a vitória é obtida.

Um algoritmo para a pontuação considerando a profundidade da árvore é apresentado a seguir:

```
if maximizer has won:
    return WIN_SCORE - depth
else if minimizer has won:
    return LOOSE_SCORE + depth
```

`WIN_SCORE=10`, `LOOSE_SCORE=-10` e `depth` é a profundidade da chamada recursiva (ou equivalentemente à profundidade da árvore)

O jogador 'X' pode durante um jogo analisar qual a melhor jogada através da função `findBestMove()` ilustrada a seguir:

```
function findBestMove(board):
    bestMove = NULL
    for each move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

A avaliação da melhor jogada obviamente deve ser feita utilizando a função `minimax()`

5 Codificando um estado

Um estado do jogo será codificado de duas maneiras diferentes na linguagem Python:

1. lista de 3 listas de 3 elementos onde:
   - 'b': indica posição vazia;
   - 'X': indica posição ocupada pelo jogador X;
   - 'O': indica posição ocupada pelo jogador O.

   Por exemplo:

   Um tabuleiro totalmente vazio

   seria representado como:

   ```
   [['b','b','b'],['b','b','b'],['b','b','b']]
   ```

X		
	O	
		X

   seria representado como:

   ```
   [['X','b','b'],['b','O','b'],['b','b','X']]
   ```

6 Como um jogo termina ?

Obviamente o jogo termina em 3 circunstâncias:

1. Vitória do 'X': o jogador 'X' vence se conseguir inserir um arranjo de três símbolos consegutivos (horizontal, vertical ou diagonal).
2. Vitória do jogador 'O': o jogador 'O' vence nas mesmas condições.
3. Empate: nenhum jogador consegue a vitória após todas as posições do tabuleiro estarem prenchidas.

7 Para você fazer

7.1 Cálculo experimental da complexidade

- Implemente uma função que calcule a complexidade do Jogo da Velha.

- Considerando que a primeira jogada é sempre realizada pelo jogador 'X', a função deve calcular o número possível de vitórias do jogador 'X', número de derrotas e número de empates.
- Considerar todas as posições como distintas. i.e., não considerar simetrias.

7.2 Implementar findBestMove

- Implementar a função `minimax()`.
- Implementar a função `findBestMove()` utilizando `minimax()`.

7.3 Programa teste

- Implementar um programa teste que demonstre o funcionamento correto do dos algorritmos.

8 Referências

1. Algorithms, Robert Sedgewick and Kevin Wayne, Addison-Wesley Professional, 4th Edition, 2011.