Algoritmo MiniMax com podas α-β utilizado no Gomoku

Fabio Moreira Marcello Klingelfus Junior

22 de Agosto de 2017

- **Heurística e utilidade**

  A função heurística $H(T)$ deste trabalho pode ser definida como o somatório de todos os encadeamentos ao quadrado que pertecem ao computador, multiplicado por um fator de bloqueio de jogada (detalhado adiante), subtraindo o somatório de todos os encadeamentos ao quadrado do adversário. A Equação 1 define matematicamente essa função.

  $$H(T) = \left(\sum_{i=1}^{N} (EC)^2 \right) * TEB - \sum_{j=1}^{N} (EA)^2 \tag{1}$$

  Onde:

  - T: É uma determinada configuração do tabuleiro de *gomoku*
  - EC: Encadeamentos do computador
  - EA: Encadeamentos do adversário
  - TEB: Total de encadeamentos bloqueados, definido como a soma das peças bloqueadas do adversário ao adicionar uma peça pertencente ao computador.
  - N: Total de encadeamentos do computador/adversário

  Em uma partida normal, o adversário e o computador adicionam peças ao tabuleiro até que ou alguém faça uma sequência de cinco peças ou o tabuleiro seja totalmente preenchido, resultando em um empate. Para que se consiga atingir o objetivo do jogo, é necessário que as peças sejam agrupadas até que formem uma sequência de cinco peças. Para isso, é essencial que o computador entenda que peças agrupadas valem mais que peças "soltas" pelo tabuleiro. O objetivo de elevar os encadeamentos ao quadrado é justamente dizer ao computador que quanto maior o encadeamento, maiores serão as chances de atingir o objetivo buscado. O computador não joga sozinho, assim, também é fundamental que ele compreenda que quanto mais encadeamentos grandes o adversário possuir, menores são as chances do computador de atingir seu objetivo, esse é o propósito da segunda parte da equação. O problema com essa heurística é que o adversário pode enganar o computador. Basta adicionar dois encadeamentos na mesma direção deixando um "buraco" entre eles. Com isso, o computador não verá essa jogada como uma ameaça e dará mais valor para outras jogadas, o fator TEB corrige essa deficiência.

A função de utilidade $U(T)$ pode ser definida matematicamente através da Equação 2.

$$H(T) = \left(\sum_{i=1}^{N} (EC)^2 \right) * TEB - \left(\sum_{j=1}^{N} (EA)^2 \right) * FJ \qquad (2)$$

O que difere as equações é o fator FJ (Fim de Jogo). Ao detectar o fim de jogo, seu valor é alterado de 1 para 100.

- **Otimizações e estratégias**

  O tabuleiro foi projetado como uma matriz de tamanho 15×15. As peças são objetos que possuem informações sobre: proprietário (computador, adversário ou vazio), uma variável indicando se ele foi visitado ou não (sua utilidade será detalhada depois) e sua posição no tabuleiro.

  Há também uma lista que armazena o estado atual do jogo em relação aos encadeamentos existentes. A cada jogada, avalia-se o efeito da adição de uma peça (do computador ou do adversário) no tabuleiro. Com isso, evita-se analisar as cadeias toda vez que uma peça é adicionada ao jogo.

  Para o algoritmo MiniMax, a árvore de estados será construída recursivamente ao longo da execução desse algoritmo. O tamanho do tabuleiro pode comprometer o desempenho do algoritmo. Assim, para a construção dos possíveis estados, será considerado um pedaço desse tabuleiro. Se o adversário adicionar uma peça fora desse espaço, o algoritmo passará a considerar um espaço que englobe essa peça.

- **Detecção de fim de jogo e sequência de quatro peças**

  A detecção não é realizada por um algoritmo específico, mas é extraído do resultado do método 'hu' (atualiza o encadeamento) que por sua vez chama o método 'Busca Pontual' (procura alterações). Esse método mantém a lista de encadeamentos do jogo sempre atualizada. Ao adicionar uma peça, o método verifica quantos encadeamentos há ao redor daquela peça, por exemplo, se há um encadeamento de tamanho dois na vertical à esquerda e de tamanho um à direita, o método decrementa dois encadeamentos de tamanho dois da lista e incrementa um encadeamento de tamanho quatro. Quando o método finaliza a execução, basta verificar se há um encadeamento de tamanho quatro ou cinco. Ao contrário do método imaginado inicialmente e detalhado no trabalho parcial, há um grande ganho de desempenho ao analisar somente os encadeamentos modificados ao adicionar uma nova peça ao jogo.

- **Decisões de projeto**

  O trabalho foi dividido em três classes:

  Controlador.py - Inicia e controla o fluxo do jogo, chama os métodos das classes IA e Tabuleiro quando necessário.

  IA.py - Possui o algoritmos: MiniMax, atualização da lista de encadeamentos, heurística e demais ferramentas que auxiliam na implementação destes. Pelo fato de usar intensamente a lista de encadeamentos do tabuleiro, decidiu-se armazená-la nesta classe ao invés da classe Tabuleiro.py, caso contrário acarretaria uma queda no desempenho ao ter que chamar uma função, colocar na pilha, retirar, etc. inúmeras vezes.

  Tabuleiro.py - Cria o tabuleiro, faz todas as inicializações e validações. Responsável por todas as alterações envolvendo o tabuleiro (adição/remoção de peças). Controla o "subtabuleiro" que será utilizado para o algoritmo MiniMax.

- **Limitações**

  - Para tamanhos de tabuleiro grandes, maiores que 11×11, a jogada do computador pode afetar o tempo de processamento da escolha da melhor jogada. Por exemplo, ao adicionar uma peça no início do tabuleiro, o valor de alpha será alto ao adicionar peças ao lado dessa e baixo ao adicionar peças longe dela. Dessa forma, o algoritmo miniMax irá realizar a poda mais cedo, acelerando o algoritmo. Ao contrário, se adicionar uma peça no fim, a poda ocorrerá mais tarde.

- **Principais métodos**

  IA.minimax() - Implementa o algoritmo minimax com poda alpha e beta.

  IA.BuscaPontual() - Busca os vizinhos do último ponto inserido e os tamanhos dos seus encadeamentos e faz a atualização da lista de encadeamentos.

  IA.heuristica() - Retorna a pontuação heurística para um certo nó folha.