Poda α-β aplicada em *gomoku*

Emmanuel Podestá Jr., Gustavo Zambonin*
Inteligência Artificial (UFSC – INE5430)

- **Funções de heurística e utilidade**

  A função heurística $H(T)$, onde T é um nodo do grafo que caracteriza um tabuleiro do jogo *gomoku*, foi definida como a soma da quantidade de n- uplas, $n \in \{1, 2, 3, 4\}$, afetadas por certos fatores multiplicativos.

  Tais fatores derivam-se do que pode ser identificado como ''boas'' característica da organização de peças de um determinado jogador: alinhamento em algum eixo, falta de obstrução pelo adversário e tamanho da n-upla.

  A obstrução é codificada como um fator que altera gravemente o valor do tabuleiro caso uma n-upla não consiga ser estendida pelo jogador – ou seja, se existirem peças do adversário em cada um dos lados de modo a impossibilitar essas jogadas. Caso apenas um dos lados esteja bloqueado, a punição não é tão severa. Pode ser representada como

$$s(x) = \begin{cases} 1 \text{ if } x = 2 \\ \frac{1}{2} \text{ if } x = 1 \\ \frac{1}{10} \text{ if } x = 0 \end{cases}$$

  onde n é o número de lados disponíveis para jogada.

  n-uplas alinhadas recebem valores de acordo com seu tamanho. Tomando v_n como o valor de uma n-uplas de tamanho n, é possível calcular v_{n+1}, assumindo alguns fatos na construção do raciocínio:

  - $v_{n+1} > \sum v_n$, para que o algoritmo sempre busque construir as maiores n-uplas;
  - um valor inicial não-nulo para v_1, a menor n-upla que a função considera;
  - existem $15 - n + 1$ possíveis arranjos de elementos consecutivos com o mesmo valor em uma palavra binária de tamanho 15;
  - $\lceil \frac{15^2}{2} \rceil = 113$ é o maior número de peças de um jogador num tabuleiro, o que implica em $\frac{113}{15} \approx 7.5$ linhas de peças iguais;
  - existem 3 orientações onde um arranjo pode ser considerado como n-upla.

  Então, $v_{n+1} = \sum v_n \times 3 \times 7.5 \times 113 \times (15 - n + 1)$, e assim tem-se os valores para cada uma das n-uplas. Codificando este cálculo como uma função $v(t) = v_n$, onde t é uma n-upla de tamanho n dentro do tabuleiro T, tem-se então a função heurística final

$$H(T) = \sum_{n=1}^{4} v(t) * s(x), \forall t \in T$$

  onde x é obtido analisando as peças adjacentes à n-upla.

  A função de utilidade foi definida de forma semelhante, levando em conta a dificuldade de representar todos os nodos possíveis do grafo em virtude da quantidade de tabuleiros possíveis. Dessa forma, foi necessário especificar uma profundidade limite para o algoritmo minimax. Este fato dificulta a introdução de um fator heurístico relacionado à profundidade do grafo, pois seria custoso armazenar e processar tal informação ao longo da execução do algoritmo.

  Então, tomando $U(T)$ como a função utilidade, estendeu-se a análise de n-uplas para $n = 5$, aumentando a pontuação final do tabuleiro com um grande valor caso tal n-upla exista. Assim,

$$U(T) = \sum_{n=1}^{5} v(t) * s(x), \forall t \in T$$

  onde o grande valor obtido quando encontra-se uma 5-upla apontará corretamente a importância do nodo, que neste caso é vitorioso.

*`{emmanuel.podesta,gustavo.zambonin}@grad.ufsc.br` — todos os algoritmos utilizados podem ser encontrados também neste repositório.

- **Otimizações e estratégias**

  O algoritmo minimax com poda α-β foi utilizado para buscar melhores jogadas. A estrutura de dados que representa o tabuleiro é uma lista de listas simples, onde cada elemento pode ser um inteiro em $0, 1, -1$: respectivamente espaço vazio e dois jogadores. Dado uma configuração de entrada, o algoritmo verificará possíveis movimentos ao redor das peças já inseridas no tabuleiro, armazenará os possíveis movimentos em uma lista, e vários cenários de jogadas com cada uma dessas coordenadas serão testados, recursivamente. A avaliação de melhor jogada será feita sobre os valores de cada um dos tabuleiros. Após o processo de verificação e poda, a melhor jogada escolhida, juntamente com sua pontuação, é retornada para que o algoritmo possa proceder.

- **Detecção de fim de jogo e sequências**

  A detecção de fim de jogo é feita a partir de um método `victory`, que percorre toda a representação matricial do tabuleiro procurando por quíntuplas da mesma cor. Após cada jogada, essa verificação é aplicada e $U(T)$ é chamada caso o resultado seja verdadeiro. De modo similar, sequências são verificadas com uma variante do método acima que observa apenas n-uplas de um jogador e tamanho, fornecidos como argumentos. Note que este método não necessariamente busca por n-uplas com peças consecutivas, e pode identificar sequências com um espaço vazio como boas jogadas. O funcionamento específico tem base em `itertools.groupby` da linguagem Python, que agrupa símbolos semelhantes e sua quantidade em um objeto iterável, como uma lista.

- **Decisões de projeto**

  A implementação foi dividida em três classes, cada qual com sua função imutável:

  - `game.py` é responsável pela parte de acesso aos modos de jogo e validade das entradas, bem como todo o fluxo de controle, comunicando-se com o tabuleiro e chamando métodos necessários para que jogadores humanos e artificial possam progredir no jogo.
  - `gomoku_board.py` é uma implementação simples de um tabuleiro para um jogo de *gomoku*, com métodos variados para suportar a integração com um algoritmo de busca de decisão, utilizando vários recursos da linguagem escolhida, tais como *slices*[1], *unpacking*[2] e *list comprehensions*[3], bem como várias funções nativas (em especial, a biblioteca `itertools`[4]).
  - `minimax.py` é responsável por construir árvores de possibilidades dinamicamente, dado o tabuleiro atual, com uma profundidade escolhida previamente, e decidir qual será a melhor jogada possível respeitando as funções heurística e utilidade. A análise de vitória é feita em todo nodo e não apenas nos que são folhas, pois tal situação pode acontecer em nodos que são pais; apenas as posições vazias e vizinhas às peças de ambos os jogadores são avaliadas, pois deseja-se bloquear o jogador adversário e, ao mesmo tempo, as peças do jogador atual geralmente estão adjacentes.

- **Limitações**

  - As cores representando os jogadores foram fixadas em virtude da falta de abstração na identificação de quem fará a maximização e minimização; assim, a peça de cor branca sempre iniciará, e esta representa o jogador humano.
  - O desempenho é um problema real, em virtude de todas as possibilidades calculadas no algoritmo minimax; seu gargalo apresenta-se na função `neighbor_board`, pois é necessário verificar quais são os espaços válidos ao redor de uma dada coordenada a partir do raio desejado e filtrá-los de modo que tais espaços possam formar n-uplas úteis para o jogador.
  - A profundidade escolhida para o algoritmo minimax também implica diretamente em quão rapidamente o jogador artificial fará sua jogada, e deve-se escolher um valor ótimo para que o algoritmo não demore demais, mas as jogadas ainda sejam plausíveis.
  - A heurística poderia ser refinada para levar em conta a profundidade da árvore, porém isso implicaria em uma grande mudança na implementação do algoritmo minimax, e portanto a sugestão foi descartada.

- **Principais métodos**

  - `minimax.py:ab_pruning()` implementa o algoritmo minimax com poda.
  - `gomoku_board.py:neighbor_board()` verifica vizinhos de uma dada coordenada utilizando `itertools.starmap` para gerar todas as coordenadas válidas no formato de "estrela", e filtra posições válidas através de uma função anônima.
  - `gomoku_board.py:row_values()` calcula o valor (heurístico) de uma lista de listas de coordenadas; pode ser reutilizado para todos os eixos, dado que a construção de tais listas sejam corretas.

[1] https://docs.python.org/3/glossary.html#term-slice
[2] https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists
[3] https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
[4] https://docs.python.org/3/library/itertools.html