Relatório de I.A.: Gomoku, Parte 2

Cauê Baasch de Souza
João Paulo Taylor Ienczak Zanette

26 de Setembro de 2018

1 Decisões de Projeto

1.1 Geral

O programa foi escrito na linguagem Rust, separado em 8 arquivos:

- `ai.rs`: Contém a implementação do **"Smart Bot"**, que representa a I.A. implementada com o algoritmo do *minimax* com podas *alpha-beta*.
- `axes.rs`: Contém a implementação de um iterador que itera pelos eixos do tabuleiro.
- `board.rs`: Contém a implementação do tabuleiro em si.
- `coordinates.rs`: Contém funções de parsing de coordenadas (para entrada do usuário).
- `game.rs`: Contém a implementação do jogo em si, com funcionalidades para interagir em uma partida (avançar turnos, simular o jogo até o fim…).
- `main.rs`: Contém a execução principal do programa, com uma função de heurística para teste e a configuração de um jogo de exemplo.
- `players.rs`: Contém a implementação de alguns jogadores, como o "Human", que joga conforme a entrada do usuário, e o "Random Bot", que apenas seleciona uma célula aleatória livre.
- `tests.rs`: Contém os testes unitários para validar algumas implementações feitas, como: validar os algoritmos de detecção de vitória (se não consideram uma sequência de 5 elementos, porém em linhas diferentes, como vitória, por exemplo).

OBS: Em Rust, quando não há ";" no último comando de um escopo, então aquele comando é o valor retornado pelo escopo.

1.1.1 O jogo

O jogo foi estruturado considerando que fosse possível vincular duas implementações quaisquer de jogadores, desde que implementem uma função `decide(self, board, last_move)`, em que `self` é o próprio jogador, `board` é o tabuleiro no momento em que o jogador irá decidir sua jogada, e `last_move` é a última jogada feita (ou nenhuma, caso seja a primeira jogada). Isso é feito utilizando uma *trait* (semelhante a interface) chamada "Player":

```
pub trait Player {
    fn decide(
        &mut self,
        board: &Board,
        last_move: Option<(usize, usize)>,
    ) -> (usize, usize);
}
```

A partir disso, o jogo pode ser simulado de três formas: apenas um turno, "N" turnos ou até o fim (empate ou vitória). A criação de um jogo é dada definindo qual implementação será utilizada para os jogadores 1 e 2, como no exemplo abaixo:

```
let mut game = Game::new(player1, player2);

match game.play_to_end() {
```

1.1.2 "Smart Bot" e o *minimax*

Em `ai.rs` se encontra a implementação da `struct SmartBot`, que implementa a `trait Player`, fazendo seu `decide(...)` retornar a função "minimax":

```
let (alpha, beta) = (isize::min_value(), isize::max_value());

self.minimax_aux(board, depth, alpha, beta, true)
```

A função "minimax" é subdividida em uma outra função, "minimax_aux", Mas essa parte o Cauê sabe melhor AVANÇA NESSA CAVALA IMUNDA.

1.2 Limitações

AVANÇA NESSA CAVALA IMUNDA.

2 Principais Métodos

Provavelmente vai ficar embutido nos outros tópicos. AVANÇA NESSA CAVALA IMUNDA.

2.1 Utilidade e Heurística

Uma mesma função foi utilizada para Utilidade e Heurística. O resultado dessa função é apenas a diferença da pontuação do jogador atual com a do oponente para o tabuleiro dado:

```
fn heuristic(board: &Board, player: PlayerIndicator) -> isize {

    ...

    let player_score = score_for_player(&board, player);
    player_score
}
```

O cálculo da pontuação para o tabuleiro, dado um jogador, está definido em uma função interna à de heurística, em que, para cada caminho possível nos eixos (todas as horizontais, verticais e diagonais principais e secundárias), conta-se um *streak* (sequência de casas que não estejam sendo ocupadas pelo oponente, limitando a até 5 elementos). Esse *streak* então é elevado a uma constante arbitrária (`SCORE_PER_MY_SPACE`). Sendo assim, sequências muito boas de *streaks* geram pontuações muito altas. Quando o tamanho do *streak* é 5, porém, a função imediatamente para retornando o maior número inteiro com sinal possível).

```
fn score_for_player(board: &Board, player: PlayerIndicator) -> isize {
    let max = Axes::new(&board)
        .map(|axis| {
            axis.streaks_with_room(player)
                .iter()
                .map(|x| x.len())
                .max()
                .unwrap_or(0) as isize
        }).max()
        .unwrap_or(0);

    if max == 5 {
        isize::max_value()
    } else {
        max
    }
}
```

Tiz, descreve aqui bonitinho as heurísticas. AVANÇA NESSA CAVALA IMUNDA.

2.2 Outros métodos