Atividade Prática 2: Busca Competitiva

Inteligência Artificial

Raoni F. S. Teixeira

Introdução

Este documento é parte da segunda avaliação prática da disciplina de inteligência artificial ministrada na UFMT no segundo semestre de 2016. Nesta atividade, você irá implementar um algoritmo de busca competitiva para o jogo Lig 4. Lembre-se de que esta é uma atividade individual e é muito importante que você escreva a sua própria solução e o seu próprio relatório.

Antes de iniciar o exercício, recomendamos fortemente a leitura atenta do Capítulo 5 do livro texto [1] e a consulta ao material sobre Octave/MATLAB disponível na página da disciplina [2].

Arquivos incluídos na atividade

- `game.m` Script Octave/Matlab de teste que orientará você no exercício.
- `tournament.m` Script de teste que orientará você na tarefa opcional.
- `agent.p` Função Octave/MATLAB que implementa o agente inteligente utilizado na tarefa extra (arquivo protegido que nao deve ser alterado).
- `eval_game_a.p` Função de avaliação do agente (arquivo protegido que nao deve ser alterado).
- `do_move.m` Função que implementa o modelo de transição do jogo.

[1] Stuart Russell e Peter Norvig. Inteligência Artificial. Tradução da 3a edição. Editora Campus/Elsevier.

[2] `http://www.students.ic.unicamp.br/~ra089067/ensino/2016_2/ia.html`

- `find_diagonal_streak.m` Função que devolve a quantidade de sequências de peças na diagoboal que existem em um tabuleiro.
- `find_horizontal_streak.m` Função que devolve a quantidade de sequências de peças na horizontal que existem em um tabuleiro.
- `find_vertical_streak.m` Função que devolve a quantidade de sequências de peças na vertical que existem em um tabuleiro.
- `find_streak.m` Função que devolve a quantidade de sequências de peças na vertical, horizontal e vertical que existem em um tabuleiro.
- `is_game_over.m` Função que verifica se o jogo acabou e devolve o *id* do jogador ganhador.
- `is_terminal.m` Função que verifica se um terminado nó de busca é terminal (jogo acabou ou profundidade máxima foi atingida).
- `opposite_player.m` Função que devolve o *id* do jogador adversário.
- `eval_game.m`$^+$ Deve conter o código da função de avaliação criado na atividade.
- `minimax_alpha_beta.m`$^+$ Deve conter a implementação do algoritmo minimax com poda alpha-beta.

Todos os arquivos marcados com $^+$ devem ser implementados (alterados).

---

1 Busca competitiva

Na primeira atividade, consideramos um problema em que ambiente de atuação do agente era muito bem comportado/controlado. Naquele ambiente simplificado, o agente sempre sabia qual estado seria alcançado e por isto podia "fechar os olhos" (i.e., ignorar as informações percebidas pelos sensores) e agir de acordo com uma estratégia previamente planejada.

Agora que já temos um pouco mais de intimidade com o assunto, vamos afrouxar um pouco estas restrições e considerar situações em que o laço entre o agente e o ambiente não pode ser completamente quebrado. Este é o caso,

por exemplo, de ambientes competitivos em que os objetivos de vários agentes estão em conflito. Este tipo de problema é frequentemente chamado de *jogo*.

Em particular, consideramos um tipo de jogo bastante comum que se passa em um ambiente *determístico* completamente *observável* em que dois agentes agem alternadamente até o jogo terminar. Ao fim de cada partida deste jogo, alguns pontos são dados ao ganhador e uma penalidade é aplicada ao perdedor. Em caso de empate, os dois jogadores ganham a mesma quantidade de pontos. Alguns exemplos deste tipo de jogo são xadrez, dama, go e jogo da velha.

Com um pouco de atenção é possível perceber que a estratégia de resolução considerada na atividade anterior pode ser ao menos parcialmente aplicada aqui. Cada estágio da partida, por exemplo, pode ser representado como um estado e a solução do jogo é uma sequência de jogadas (i.e., ações) que leva a um estado em que o agente é vitorioso. O objetivo do agente é obviamente vencer o jogo. Agora, no entanto, não é possível saber de antemão como o outro jogador irá se comportar [3], e precisamos considerar os desdobramentos de cada ação executada por agente e por seu adversário no jogo.

Neste sentido, um jogo é uma espécie de problema busca com 6 (seis) componentes importantes. O primeiro deles é o estado inicial $s_i \in \mathcal{S}$ que especifica como o jogo começa. Outro componente é um função J, $J(s) \mapsto [\text{jogador}_1, \text{jogador}_2]$, que especifica para cada estado s qual jogador deve jogar. Esta função controla a alternância entre os jogadores. Novamente, a formulação é baseada em uma descrição das ações válidas (função A) em um estado e de um modelo de transição T que especifica o estado que será alcançado (s') após uma jogada ser executada por um jogador p em um estado s qualquer. O quinto componente é uma função F, $F(s) \mapsto [\text{Verdadeiro}, \text{Falso}]$, que verifica se o jogo terminou ou não. Os estados em que o jogo acaba são chamados de terminais. A função devolve o valor Verdadeiro apenas se o estado s é terminal. O sexto e último componente é uma função U, $U(s, p) \mapsto [-\infty, \infty]$, que avalia um estado terminal s considerando um jogador p. Esta última função atribui um valor númerico para o estado s. Os valores atribuídos aos jogadores em geral são iguais ou opostos. No jogo da velha, por exemplo, a função pode devolver 1 para indicar que o primeiro jogador venceu e -1 para indicar que o segundo jogador foi vencedor. O valor

[3] Conta-se que na Copa de 58, o técnico Feola bolou um esquema infalível contra a seleção soviética: Nilton Santos lançaria a bola pela esquerda para Garrincha, que driblaria três russos e cruzaria para Mazzola marcar de cabeça. Garrincha ouviu o professor atentamente: "Tá legal, seu Feola, mas o senhor combinou com os russos?".

0 (zero) pode ser utilizado para indicar o empate.

Tais componentes podem ser utilizados, por exemplo, para enumenar todas as partidas possíveis. Podemos também utilizar a função U para selecionar as partidas em que nosso agente saíra vitorioso. A enumeração de partidas pode ser encarada como uma árvore cujos nós representam estados obtidos após o movimento de um jogador e as interligações entre os nós indicam as jogadas realizadas. O estado inicial s_i é a raiz da árvore e os estados terminais são obviamente folhas. Cada caminho interligando a raiz à uma destas folhas representa uma partida específica do jogo. Para selecionar uma partida, podemos então considerar o valor que a função U atribuiu para o nó terminal. Quanto melhor o valor, melhor será a partida para o agente.

A seguir, mostramos como este processo pode ser extendido para o projetar agentes que escolhem jogadas.

1.1 Estratégia minimax

A primeira coisa que precisamos fazer para escolher jogadas é modelar o comportamento do jogador adversário. O que sabemos sobre ele? Como ele vai escolher as jogadas? Bem... é difícil responder com precisão, mas para simplificar podemos assumir que o adversário também está jogando para ganhar. Com isto podemos enxergar cada partida do jogo como um problema de busca em que o agente e o adversário tentam alcançar estados objetivos que, respectivamente, maximizam e minimizam o valor devolvido por U.

Embora tudo isto possa parecer complicado, é possível expressar este raciocínio em uma única equação:

$$\mathrm{MM}(s) = \begin{cases} U(s), & \text{se } F(s) \text{ é Verdadeiro.} \\ \mathrm{Max}_{a \in A(s)} \mathrm{MM}(T(s,a,p)) & \text{se } J(s) \text{ é o agente.} \\ \mathrm{Min}_{a \in A(s)} \mathrm{MM}(T(s,a,p)) & \text{se } J(s) \text{ é o adversário.} \end{cases} \quad (1)$$

Obviamente, neste caso, o agente racional deve escolher uma sequência de ações que leva ao estado objetivo com o maior valor calculado por minimax (MM). Uma implementação em Octave/MATLAB é apresentada a seguir.

```
function [ best_value, best_move ] = minimax(State, player)
  best_value = nan;
  best_move = 0;
```

```
  if is_terminal(State),
    best_value =  U(State, player);
  else
    moves = legal_moves(State, player);
    for i=1:size(moves, 1),
      newState = do_move( State, moves(i), player);
      value = -1*minimax(newState, opposite_player(player));
      if isnan(best_value) || value > best_value,
        best_value = value;
        best_move = moves(i);
      end
    end
  end
end
```

2 Implementação principal

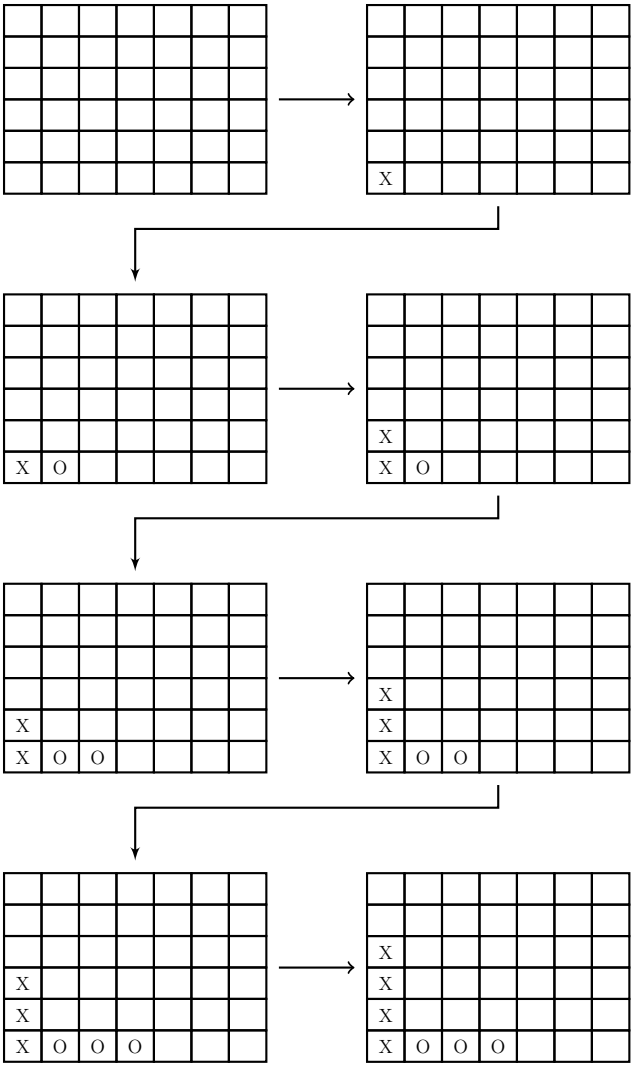
Nesta atividade, você irá escrever um agente inteligente baseado em busca competitiva para o jogo Lig4 [4]. Esta seção apresenta vários detalhes desta implementação.

2.1 Detalhes da problema

O Lig4 é um jogo em que dois jogadores acrescentam alternadamente uma peça em um tabuleiro. O tabuleiro é uma matriz 6×7 e está inicialmente vazio. O primeiro jogador a construir uma seqüência consecutiva de quatro peças na horizontal, vertical ou diagonal ganha o jogo.

Uma sequência de jogadas válidas é apresentada a seguir. As peças do primeiro e do segundo jogador são marcadas com X e O, respectivamente.

[4] `https://en.wikipedia.org/wiki/Connect_Four`



Como podemos ver neste caso, cada jogador tem no máximo sete opções de movimentos (uma para cada coluna). As peças são *empilhadas* nas colunas e o jogador não pode acrescentar uma peça em uma coluna *cheia*. O jogo acabou quando o primeiro jogador conseguiu construir uma sequência na primeira coluna do tabuleiro.

Para mais detalhes, consulte o modelo de transição implementado no aquivo `do_move.m`. Você também pode jogar contra o computador e ter uma idéia melhor de como o jogo funciona. Basta executar o script `demo.m` no pacote distribuído com a atividade. Have fun!

2.2 Detalhes da implementação

A solução computacional clássica para este problema é o algoritmo minimax (veja Secção 1.1). O principal problema do minimax, no entanto, é que a quantidade de estados do jogo que precisam ser examinados é exponencial. É impossível resolver este problema completamente, mas, em muitas situações, é possível reduzir o número de nós avaliados e garantir um resultado razoável. Duas estratégias são utilizadas na prática e devem ser implementadas por você nesta tarefa. A primeira delas é conhecida como *poda alfa-beta* (ou poda α-β) e define uma variação do algoritmo minimax que interrompe a avaliação quando (comprovadamente) os movimentos posteriores não afetam o valor calculado pelo algoritmo. A sua implementação da poda deve ser feita no arquivo `minimax_alpha_beta.m`. Consulte o livro texto para mais detalhes.

Você também terá que implementar um controle da *profundidade* da busca. Neste caso, como a busca pode ser interrompida em um nó não terminal, precisamos de uma função de avaliação (*heurística*) que estima a utilidade esperada para qualquer estado do jogo. Um exemplo desta dinâmica é apresentado no arquivo `minimax.m` distribuído no pacote da atividade.

Obviamente, o sucesso do seu agente depende da função de avaliação utilizada. A avaliação deve ser implementada no arquivo `eval_game.m`. Tal função deve receber uma instância do jogo (tabuleiro, profundidade e jogador) e devolver um número real que indica o quão próximo o jogador está de ganhar o jogo. Lembre-se de que em se tratando da função de avaliação *simples é melhor que complicado*. É muito mais útil ter tempo para procurar mais nós na árvore de busca do que expressar perfeitamente o valor de uma posição.

2.3 Exercício opcional (extra)

Você pode optar por resolver uma tarefa extra que consiste em vencer o agente inteligente disponibilizado no pacote da atividade. Para testar sua implementação, execute o arquivo `tournament.m`.

3 Relatório, entrega e notas

Depois de terminar a implementação, você deve escrever um pequeno relatório contendo obrigatoriamente:

- uma seção "Resumo" que deve claramente contextualizar e apresentar os principais resultados do trabalho e
- uma seção "Resultados e discussões" em que os resultados (i.e., tabelas, gráficos) devem ser apresentados e interpretados.

A seção "Resultados e discussões" deve apresentar uma comparação entre o desempenho das funções heurísticas utilizadas.

O relatório e os códigos devem ser entregues até o dia **20 de fevereiro de 2017**. A pontuação de cada tarefa deste exercício é apresentada a seguir.

Tarefa	**Arquivo**	**Pontuação (nota)**
Função de avaliação heurística	`eval_game.m`	2.0
Poda *alpha-beta*	`minimax_alpha_beta.m`	5.0
Relatório	`ap2.pdf`	3.0
Vencer o jogador distribuído	`tournament.m`	3.0 (extra)