Inteligência Artificial. Caderno № 4

Manuel Pita, Universidade Lusófona. May 29, 2024

This is version 0.1 of this booklet. if you find any errors, please send and email to
manuel.pita@ulusofona.pt

Busca Competitiva (adversária)

Os algoritmos de busca em IA que temos aprendido até agora consistem num único agente que explora um espaço de busca pré-existente (com ou sem informação) ou tenta otimizar alguma métrica num espaço de busca desconhecido. Os conceitos fundamentais de busca em IA podem ser utilizados também para situações nas quais existe mais do que um agente. Nessas situações, os agentes podem colaborar ou competir. O objetivo desta secção é estudar a implementação básica de um algoritmo de busca em que existem dois agentes que competem um contra o outro, que é a situação clássica, por exemplo, nos videojogos. A abordagem que define a implementação que vamos estudar está enquadrada na teoria dos jogos – ou em inglês, *game theory*, que é uma das sub-disciplinas da inteligência artificial.

Esta abordagem inicial funciona com base nos seguintes pressupostos:

- Os estados são completamente observáveis pelos agentes
- O estado de busca é determinístico
- Os agentes efetuam ações de forma alternada
- O valor de utilidade para ganhar o jogo é igual para os dois agentes

No vocabulário da teoria de jogos isto é: um jogo determinístico de dois jogadores que operam em turnos alternados com informação perfeita e no qual, para que um ganhe, o outro tem de perder, isto é referido em inglês com o termo *zero-sum game*.

A inteligência artificial tem estudado jogos desde o início da disciplina porque os mesmos constituem o próximo passo lógico nos fundamentos de busca: estados e ações são representados facilmente utilizando grafos, os jogadores estão restringidos a um número pequeno de ações, e o objetivo está definido por regras simples com uma utilidade numérica que podemos otimizar (de forma similar ao que é feito em A*). Alguns exemplos dos jogos mais estudados são o Xadrez, Othello, Go e Tic-tac-toe.

Características da busca adversária

Embora o espaço de busca seja determinístico, temos um aspeto imprevisível, nomeadamente as ações do oponente. Isto adiciona uma componente associada a problemas de contingência no qual o raciocínio feito por um agente está determinado pela execução de ações por parte do oponente. Isto não é o tipo de imprevisibilidade clássica em IA porque não tem a ver com aleatoriedade ou falta de informação no espaço de busca, mas sim com as ações de um agente competitivo. Sendo um agente competitivo, assumimos que as ações do oponente sempre vão afastar o nosso agente do seu objetivo o mais possível.

Considerando estas características específicas da busca adversária, não é prático tentar explorar o espaço de busca inteiro. Se tentássemos pré-calcular todas as possíveis ações do nosso agente e as do oponente nos diferentes turnos do jogo, o espaço de busca seria demasiado grande para qualquer jogo não trivial. Portanto, uma abordagem baseada em otimização global (como a que utilizamos no A*) está fora de questão. A próxima pergunta lógica é então, como avançar? Como formalizar o problema de busca adversária?

Para responder a esta pergunta podemos, em primeiro lugar, pensar no jogo específico que estamos a implementar e de acordo com a teoria do jogo, provisionar os agentes com informação que permita discriminar a melhor ação que podem executar em diferentes situações. Por norma, uma abordagem teórica poderá ajudar a reduzir algumas incertezas e dinamizar o jogo, mas a verdadeira inteligência neste contexto está na manipulação combinatória do espaço de busca. Tal manipulação permite que o agente pré-calcule a sua utilidade, assim como a do oponente numa janela de turnos do jogo relativamente pequena, e portanto, computável (não pode ser muitos turnos a futuro porque teríamos rapidamente o problema de explosão combinatória no grafo). Estes cálculos permitem a cada agente determinar a próxima ação que, por um lado, maximiza a sua utilidade, e por outro lado, minimiza as possibilidades de maximizar a utilidade do oponente. Quando um agente faz estes cálculos, é possível reduzir o sub-grafo de busca que está a ser analisado através de ignorar segmentos completos que surgem a partir de ações que não interessam porque, por exemplo, aumentam a utilidade do oponente. O termo utilizado para referirmos ao corte de segmentos inteiros no espaço de busca utilizado por um algoritmo de IA é (em inglês) *pruning*.

Algoritmo Minimax

No contexto de busca definido até agora, o algoritmo clássico que é utilizado é o *Minimax*. Este algoritmo iterativo pressupõe que existem dois jogadores com turnos alternados, e que na iteração decorrente o jogador que tem o turno vai maximizar a sua

utilidade. O jogo decorre até o seu fim, o qual está definido por duas condições possíveis: (1) Um dos jogadores ganha (e o outro perde) e (2) não existem mais ações possíveis para nenhum jogador.

O estado inicial está definido pela configuração atual do jogo, e qual é o jogador que tem o turno. Em cada iteração, o jogador decide qual das próximas ações possíveis (no âmbito da configuração atual do jogo) vai executar, a qual é determinada através de uma função de utilidade que tem de estar definida matematicamente, e que devemos poder calcular a partir da configuração atual do jogo. O teste para terminar o algoritmo está definido através da identificação dos estados terminais que cumprem com as condições de finalização acima descritas (um agente ganha/o outro perde, ou não há mais ações possíveis).

Se estivéssemos a usar busca tradicional em IA, focaríamos então na maximização da utilidade do nosso agente, ignorando os efeitos possíveis das ações do oponente (Max). Neste algoritmo consideramos tais efeitos (Min), e é por este motivo que o nome do algoritmo é MiniMax. O nosso agente (Max) tem de encontrar uma estratégia para ganhar o jogo, ainda que o nosso oponente (Min) execute ações que coloquem o nosso agente na pior das situações. Dito de outra forma, Max precisa de encontrar a próxima melhor ação a executar, pressupondo que Min vai executar a melhor ação possível para o seu próprio benefício (piorando a situação para Max ao máximo).

Adicionalmente, será necessário considerar que não vamos poder calcular todas as possibilidades para todos os futuros possíveis do jogo. Isto significa que vamos escolher a estratégia para Max utilizando informação reduzida. No caso do MiniMax, o jogador vai analisar N ações no futuro. No caso do xadrez, por exemplo, existem aproximadamente 10^{40} estados, e em cada momento do jogo, existem em média 35 ações disponíveis para o jogador (explosão combinatória).

Na Figura 1 podemos observar uma porção do espaço de busca para o jogo da velha (*tic-tac-toe*) definido a partir do início do jogo. Para o primeiro turno, Max tem disponíveis as nove posições do jogo. A ação que o Max execute, como por exemplo colocar o 'X' na primeira posição no canto superior esquerdo, vai constranger o que Min poderá fazer no seu turno. Neste exemplo a utilidade é definida no final do jogo usando o valor -1 para a situação em que Max perde o jogo, 1 quando Max ganha e 0 quando nenhum ganha.

O algoritmo base para MiniMax é relativamente simples. Em primeiro lugar, devemos gerar o grafo de busca (o qual para busca adversária é uma árvore) até a profundidade d que vamos considerar. A profundidade (d) é um parâmetro definido por nós o qual determina quantos turnos a futuro para Max que vamos considerar na decisão da próxima ação a executar. No exemplo da Figura 2, $d = 2$, portanto geramos o grafo de busca até o segundo turno de Max e adicionamos para cada ação de Max nesse turno a utilidade correspondente a cada ação. Os nós com a informação da utilidade de

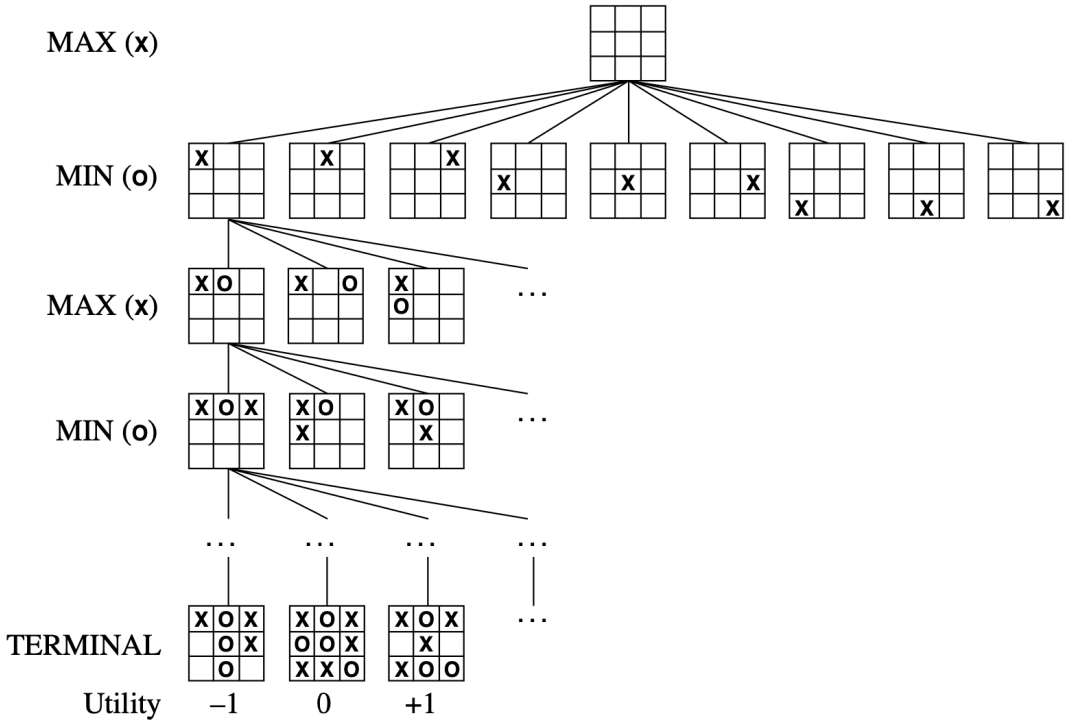

Figure 1: Exemplo da definição do espaço de busca para a execução de MiniMax no jogo *Tic-Tac-Toe*

cada ação são terminais. A seguir, na fase iterativa do algoritmo, começamos a iterar no nível mais profundo dos estados possíveis para Max (os estados ligados aos nós terminais). Para cada estado s nesse nível de profundidade, definimos uma variável $u(s)$. Se estivermos num nível onde os estados correspondem ao jogador Max, $u(s)$ terá o valor maior dos nós sucessores (filhos), caso contrário, se estivermos a processar um estado correspondente ao jogador Min, então o valor $u(s)$ será o menor. Os valores $u(s)$ são, portanto, instanciados desde o fundo da árvore, iterativamente, até chegarmos ao nó que corresponde ao estado atual do jogador Max. Com estes valores o jogador Max informa a sua decisão relativamente a qual ação executar, maximizando a sua utilidade e pressupondo que Min fará a sua melhor jogada sempre. Ver exemplo concreto na Figura 2 e a sua legenda. É importante destacar que a fase iterativa deste algoritmo base funciona de forma similar ao algoritmo *Breadth First Search* (no sentido em que as iterações operam por camadas).

Redução $\alpha - \beta$ (*Pruning*)

O algoritmo base de busca adversária tem um problema importante, nomeadamente a necessidade de pré-calcular e explorar a árvore inteira até a profundidade (d), o que dependendo da complexidade do jogo poderá resultar numa complexidade com-

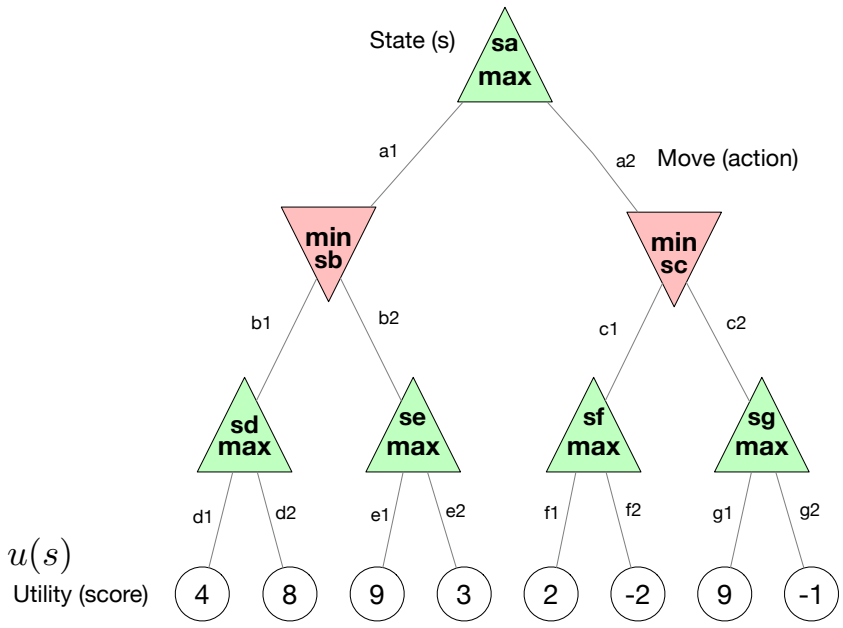

Figure 2: Exemplo da execução do algoritmo MiniMax. O algoritmo começa a sua fase iterativa no estado sd (Max) instanciando $u(sd) = 8$ correspondente á maior utilidade associada aos nós filhos. Da mesma forma $u(se) = 9; u(sf) = 2$ e $u(sg) = 9$. No nível acima (correspondente a Min) $u(sb) = 8$, que corresponde ao menor valor dos nós filhos, nomeadamente sd e se e da mesma forma $u(sc) = 2$. Finalmente $u(sa) = 8$, e Max opta então por executar a ação $a1$.

putacional inaceitável. Portanto, seria muito útil termos alguma ferramenta heurística que nos permita cortar secções inteiras do grafo de busca cuja exploração seja irrelevante para determinar a melhor estratégia que o jogador Max deve seguir. Um dos fundadores da inteligência artificial, John McCarthy, definiu tal heurística em 1956, a qual é conhecida como *alpha-beta pruning*.

Sendo uma heurística, a mesma atua como um *add-on* no algoritmo original, e neste caso, adicionalmente altera também o tipo de percurso iterativo do algoritmo MiniMax. No MiniMax original fazemos um percurso por camadas, desde a base do grafo (árvore) de busca até o nó inicial (análogo a um BFS), mas quando incluímos *alpha-beta pruning*, tal percurso é análogo a um Depth First Search (também feito desde os nós terminais, até o nó origem). Outra diferença é que, em vez de instanciar uma variável $u(s)$ para cada estado, instanciamos duas: $\alpha(s)$ e $\beta(s)$. A primeira (α) guarda valores máximos e é inicializada com o valor $-\infty$, e a segunda (β) valores mínimos, inicializada com o valor ∞.

Durante a fase iterativa, começamos na camada mais profunda, e fazemos um percurso da esquerda para a direita. Se o estado que entra na iteração corresponde ao jogador Min, então atualizamos $\beta(s)$ com o valor menor considerando o seu valor atual, e todos os valores da utilidade presentes nos filhos do nó s. Caso contrário, atualizamos $\alpha(s)$ com o valor máximo que considera o seu valor atual, e os valores

dos filhos de s. Os valores atualizados para s são propagados ao seu nó pai, o qual corresponde ao outro jogador, e portanto irá atualizar a outra variável, α ou β, diferente da atualizada no nó filho que está a propagar a sua informação. Uma vez atualizado o nó pai, avançamos para o próximo nó filho à direita (caso o mesmo exista). Nesse momento perguntamos se $\alpha \geq \beta$. Caso esta condição seja verificada, toda a sub-árvore que tem início com a ligação do nó pai ao próximo nó filho pode ser ignorada nos cálculos. Caso contrário, os valores do nó pai são propagados ao próximo nó filho e o processo é repetido.

A Figura 3 junto com o texto que segue nesta secção correspondem ao exemplo explicado durante as aulas.

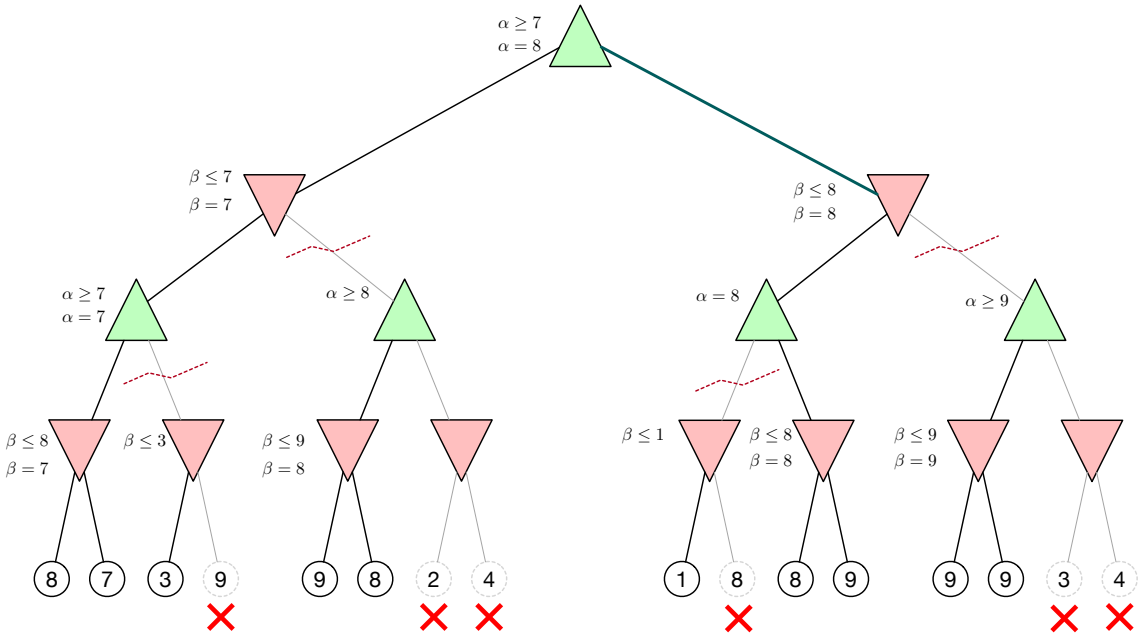

Figure 3: Exemplo da execução do algoritmo MiniMax. O algoritmo começa a sua fase iterativa no estado sd (Max) instanciando $u(sd) = 8$ correspondente á maior utilidade associada aos nós filhos. Da mesma forma $u(se) = 9; u(sf) = 2$ e $u(sg) = 9$. No nível acima (correspondente a Min) $u(sb) = 8$, que corresponde ao menor valor dos nós filhos, nomeadamente sd e se e da mesma forma $u(sc) = 2$. Finalmente $u(sa) = 8$, e Max opta então por executar a ação $a1$.

1. Iniciamos a primeira iteração no nó terminal com utilidade u(s) = 8 que é filho do jogador Min. Neste momento, sem calcular mais nada sabemos que o seu β será, no máximo, 8.

2. Na próxima iteração visitamos o nó irmão, com $u(s) = 7$, e como não há mais nós irmãos para o jogador Min, sabemos que o seu β será exatamente igual a 7.

3. Como a melhor jogada do Min terá utilidade (7) nesta rama da árvore, o jogador Max sabe que seguindo a estratégia definida nesta rama da árvore vai obter uma

utilidade pelo menos igual a sete.

4. Continuamos o percurso com o nó terminal com valor (3) e determinamos que o β para o Min será, no máximo (3). Neste momento o algoritmo determina que pela rama esquerda o Max poderá sempre obter pelo menos (7) portanto nunca irá optar pela rama que vai dar uma utilidade máxima de (3). Por isso não temos que calcular mais nada para o Min e cortamos essa rama da árvore. Não é necessário processar o nó terminal com utilidade (9) nesta iteração.

5. Como o nó pai (Max) não tem mais filhos, sabemos então que o seu α não é maior ou igual a (7) mas sim, exatamente (7). Isto significa que o β do Min (pai) será menor ou igual a (7).

6. Seguimos para a próxima iteração com o nó terminal na outra rama da árvore com utilidade (9). O β será menor ou igual a 9.

7. Avançamos para o nó irmão, e como não há mais irmãos o valor de $\beta = 8$, o jogador Max sabe que por esta rama vai obter pelo menos uma utilidade (8). Como o Min pode ir pela outra rama e obter uma utilidade (7) nunca vai optar por esta rama onde o Max pode obter (8). Esta é outra situação na qual a relação entre α do Max e o β do Min é tal que $\alpha \geq \beta$ e portanto cortamos essa rama da árvore. Neste momento, como não há mais nós filhos a serem examinados nesta rama, o β do Min será (7), e portanto sabemos que o Max irá obter, pelo menos, uma utilidade (7).

8. Na próxima iteração avaliamos o nó terminal com utilidade (1) da outra rama da árvore, e sabemos que o β para o Min será no máximo (1). Neste momento, como sabemos que o α do Max na outra rama é pelo menos (7), determinamos que o Max nunca irá apostar numa estratégia que o possa levar a obter utilidade (1) ($\alpha \geq \beta$) portanto cortamos essa rama da árvore. Note que neste caso o α e o β não são adjacentes na árvore, mas sim distantes. Esta situação é conhecida como *deep cut-off*.

9. Avançamos para a próxima iteração na qual avaliamos o nó terminal com utilidade (8) e sabemos que o β do Min será pelo menos 8.

10. Seguimos para o próximo nó terminal com utilidade (9) e confirmamos que o β do Min ficará com valor (8). Como esta rama só tem uma alternativa (a outra foi cortada) o α do Max fica com utilidade (8), e portanto o β do Min acima será menor ou igual a (8).

11. Continuamos para o próximo nó com utilidade terminal (9) e sendo que ambos os nós têm a mesma utilidade o β fica com valor nove que é propagado de forma a

que o Max sabe que vai obter utilidade 9 ou mais através desta estratégia. Aqui temos mais uma situação em que o α é maior que o β, e portanto temos um corte e não precisamos de calcular mais nada nesta rama.

12. O β do Min fica com valor (8) e portanto o Max altera o seu $\alpha = 8$ e assim concluímos que o Max vai executar a ação à direita da árvore com utilidade esperada (8).

Resumo

Nesta secção, estudamos a busca adversária em IA, onde dois agentes competem entre si, um conceito comum em jogos. Abordamos o algoritmo Minimax, que ajuda a determinar a melhor ação para um agente maximizador, considerando que o oponente minimizador fará a melhor jogada para reduzir a utilidade do maximizador. Analisamos as características de jogos de soma zero, onde a vitória de um implica a derrota do outro. Exploramos também a técnica de *alpha-beta pruning*, que otimiza o Minimax ao eliminar ramos do grafo de busca que não influenciam a decisão final, reduzindo assim a complexidade computacional. O uso destas técnicas é fundamental para a criação de agentes inteligentes em jogos, permitindo uma análise eficiente das possíveis estratégias e ações.