Jogos com Oponentes

March 12, 2019

Jogos com Oponentes

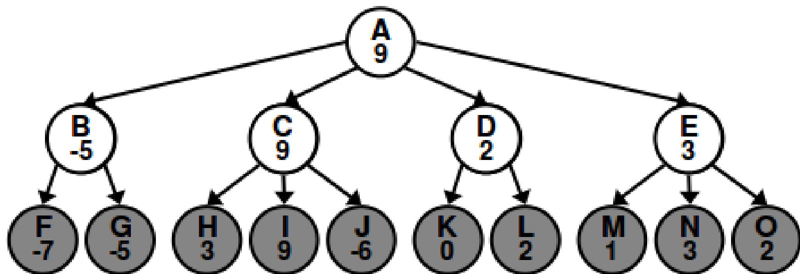
- Problemas de busca: não assumem a presença de um oponente
- Jogos: oponente $\rightarrow$ INCERTEZA!
- Incerteza porque não se conhece as jogadas exatas do oponente e não por causa de falta de informação.
- Jogos são diferentes de busca por duas razões principais:
  - espaço de busca muito grande
  - tempo para cada jogada
- Ex: xadrez, em média fator de ramificação = 35
- em média, 50 jogadas para cada jogador, onde de cada jogada devemos selecionar 1 de 35 $\rightarrow 35^{100}$ nós!!!

Jogos

- Um jogo pode ser definido como um tipo de problema de busca com os seguintes componentes:
  - estado inicial: configuração do tabuleiro e indicação de quem é a vez
  - função para gerar os sucessores: retorna uma lista de sucessores para a jogada corrente
  - teste de terminação
  - Função *utilidade* (utility function) que devolve o valor numérico do jogo. Ex: ganhou (+1, inf), empatou (0), perdeu (-1, $-$inf.
  - dependendo do tamanho do espaço de procura: limite de profundidade
- Jogo com dois jogadores: MIN e MAX
- MAX inicia o jogo

Jogos

- Busca gulosa em Jogos
  - expandir a árvore até os estados terminais
  - avaliar a utilidade de cada estado terminal
  - escolher jogada inicial com valor máximo



(Figura retirada de: http://pages.cs.wisc.edu/~bsettles/cs540/lectures/07_game_playing.pdf)

Jogos

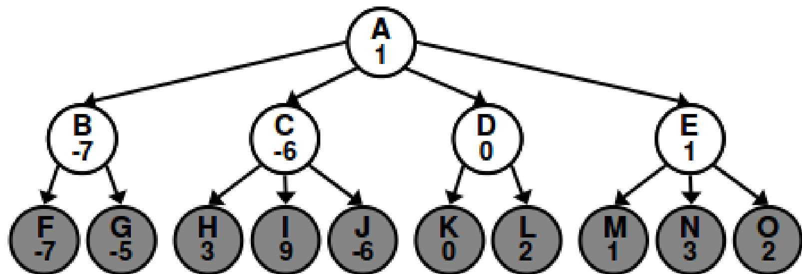
- Este tipo de busca (gulosa) ignora a forma como o oponente está a pensar jogar: no exemplo da árvore anterior, o computador irá escolher a jogada C porque tem a maior (melhor) utilidade. Se o oponente, por sua vez, escolher a jogada J, ganha o jogo (assumindo que valores mínimos de utilidade favorecem o oponente)

Princípio MiniMax

- Max assume o pior cenário possível de jogo (ou seja, assume que o oponente joga de forma ótima):
  - valores mínimos de utilidade favorecem o oponente
  - valores máximos de utilidade favorecem o computador

MiniMax

- O computador assume que depois da sua jogada o oponente vai querer fazer a jogada que minimiza a sua
  - Portanto escolhe a melhor jogada tendo em conta a sua joagada e a do adversário: ou seja, vai escolher a jogada E!



(Figura retirada de: `http://pages.cs.wisc.edu/~bsettles/cs540/lectures/07_game_playing.pdf`)

Corte Alpha-Beta

- Alguns dos ramos do jogo não serão escolhidos se o adversário estiver jogando de forma ótima
- Neste caso, ramos podem ser cortados se o algoritmo guardar:
  - em um nível de máximo (alpha)
    - maior valor encontrado até o momento
    - menor valor de utilidade de um estado
  - em um nível de mínimo (beta)
    - menor valor encontrado até o momento
    - maior valor de utilidade de um estado

Corte Alpha-Beta

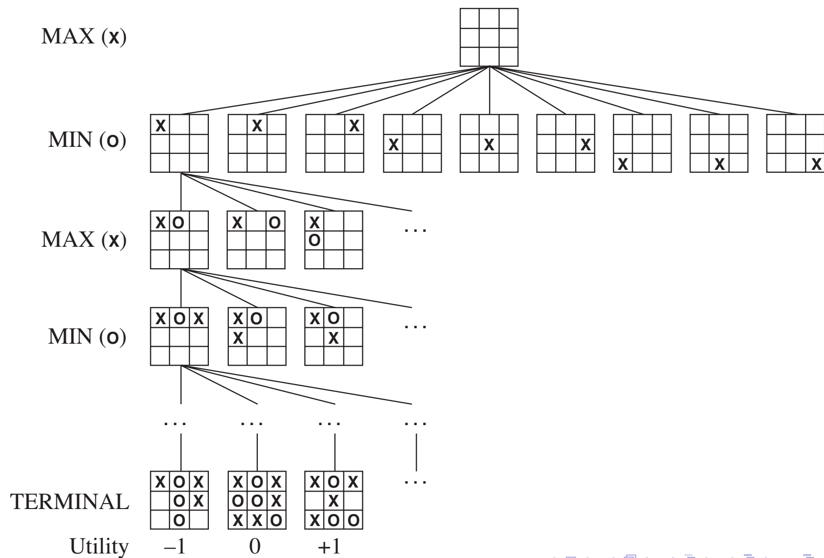
- Quando estiver num nível de **máximo** (vez do computador):
  - se *alpha* $\geq$ que o *beta* do pai, interromper geração de jogadas neste nível
  - num jogo competitivo, o oponente não deixaria o computador fazer aquela jogada
- Quando estiver num nível de **mínimo** (vez do adversário):
  - se *beta* $\leq$ que o *alpha* do pai, interromper geração de jogadas neste nível
  - o computador não deveria fazer estas jogadas

Complexidade

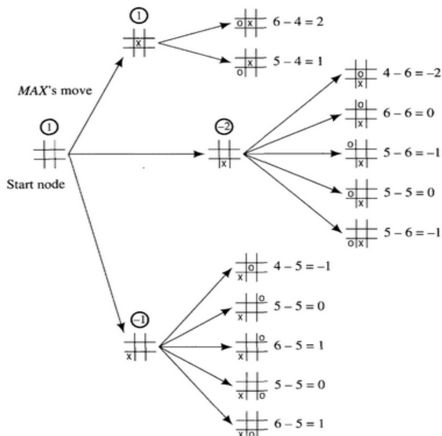
- espacial: O(bd)
- temporal:
  - minimax: $O(b^d)$
  - alpha-beta:
    - na prática, geralmente $O(b^{d/2})$
    - Exemplo: reduz o fator de ramificação do xadrez de ≈ 35 para ≈ 6
    - pior caso ainda é $O(b^d)$

Exemplo: jogo do galo

(velha - BR ou tic-tac-toe - US ou noughts and crosses - UK)



Exemplo: jogo do galo (fonte: Nilsson)



Observações

Observações:

- fator de ramificação dos nós mais próximos da raiz pode ser reduzido removendo estados simétricos
- fator de ramificação dos nós mais profundos da árvore de procura vai automaticamente sendo reduzido (sem necessidade de verificação de simetrias, porque o número de posições vagas no tabuleiro diminui)
- se expandir todos os nós até o último nível da árvore, a função utilidade pode ser: 0 para empate, -1 para perdeu e +1 para ganhou
- se não expandir os nós até o último nível, pode utilizar a função utilidade que devolve o número de filas (linhas, colunas e diagonais) vazias para o MAX menos o número de filas vazias para o MIN.

Jogos: Minimax

MINIMAX-VALUE(n) =

$$\begin{cases} UTILITY(n) & \text{if } n \text{ is a terminal state} \\ max_{s \in \text{successors(n)}} \text{MINIMAX-VALUE(s)} & \text{if } n \text{ is a MAX node} \\ min_{s \in \text{successors(n)}} \text{MINIMAX-VALUE(s)} & \text{if } n \text{ is a MIN node} \end{cases}$$

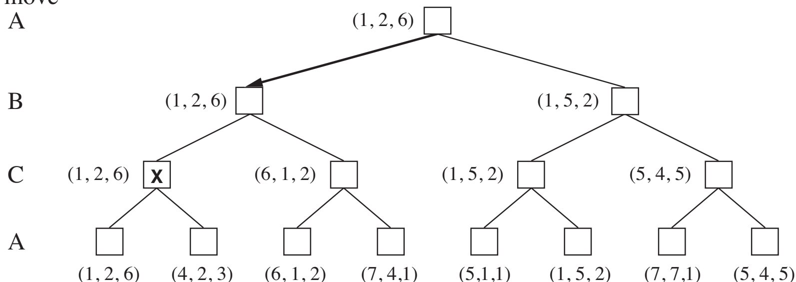
Jogos: Algoritmo Minimax

```
function MINIMAX_DECISION(state): returns an action
    inputs: state (estado corrente no jogo)
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
if TERMINAL_TEST(state) then
   return UTILITY(state)
end if
v ← -infinito
for s in SUCCESSORS(state) do
   v ← MAX(v, MIN-VALUE(s))
end for
return v

function MIN-VALUE(state) returns a utility value
if TERMINAL_TEST(state) then
   return UTILITY(state)
end if
v ← infinito
for s in SUCCESSORS(state) do
   v ← MIN(v, MAX-VALUE(s))
end for
return v
```

Jogos: Algoritmo Alfa-Beta

```
function ALPHA-BETA-SEARCH(state): returns an action
     inputs: state (estado corrente no jogo)
     v ← MAX-VALUE(state,-inf,+inf)
     return the action in SUCCESSORS(state) with value v
function MAX-VALUE(state,alfa,beta) returns a utility value
inputs: state, alfa → melhor alternativa para MAX, beta → melhor alternativa para MIN
if TERMINAL_TEST(state) then
   return UTILITY(state)
end if
v ← -infinito
for s in SUCCESSORS(state) do
   v ← MAX(v, MIN-VALUE(s,alfa,beta))
   if (v >= beta ) then
      return v // momento da poda
   end if
   alfa ← MAX(alfa,v)
end for
return v
function MIN-VALUE(state,alfa,beta) returns a utility value
if TERMINAL_TEST(state) then
   return UTILITY(state)
end if
v ← +infinito
for s in SUCCESSORS(state) do
   v ← MIN(v, MAX-VALUE(s,alfa,beta))
   if (v <= alfa ) then
      return v // momento da poda
   end if
   beta ← MIN(beta,v)
end for
return v
```

Exemplo: jogos com múltiplos jogadores (fonte: Russell)



Exemplo: jogos com múltiplos jogadores

Observações:

- representação de valores para cada jogador em forma de vetor (tuplas), onde cada elemento indica a função utilidade para um jogador, na ordem de jogada
- algoritmo aplicado similar ao minimax
- complicações:
  - mudanças dinâmicas de estratégia durante o jogo quando acontecem alianças entre jogadores para derrubar o jogador mais forte e mais tarde estas alianças desaparecem (jogo flutua entre cooperação e competição)
  - quando no jogo é introduzido algum fator aleatório (jogos com utilização de dados, por exemplo)

Efeito horizonte (Horizon effect)

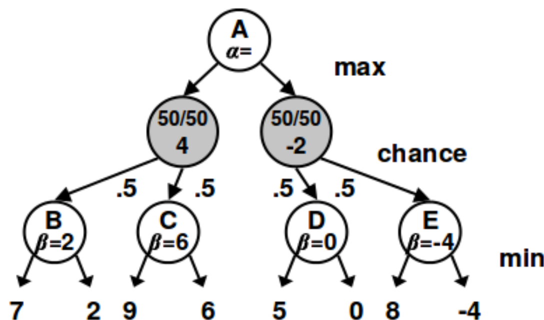
- Quando utilizamos um limite de profundidade para a árvore de jogo, pode acontecer do minimax comportar-se mal (o adversário poderia ganhar na próxima jogada – "short-sightedness")
- Alternativas
  - "quiescence search"
  - "secondary search"

Efeito horizonte (Horizon effect)

- Quiescence search
  - continua gerando a árvore de jogo além do limite de profundidade procurando um estado "estável"
- Secondary search
  1. Encontra a melhor jogada até a profundidade d
  2. Procura k passos além da profundidade d e verifica se aquela jogada continua boa
  3. Se não continuar, repete o passo (2) para a próxima melhor jogada

Jogos com elemento aleatório

- árvore de jogo precisa representar a aleatoriedade
- atribuição de pesos às utilidades de acordo com as probabilidades das jogadas
- valor esperado das jogadas: soma
- escolher jogada com maior valor



Outras estratégias

- limitar o tempo de busca (iterativa em profundidade)
- utilizar base de dados de jogadas mais comuns
- Monte Carlo Tree Search (MCTS)
- etc...

Funções de avaliação

- função heurística de características do tabuleiro: function(f_1,f_2,f_3,...,f_n)
- características são numéricas. Por exemplo, no xadrez:
  - f_1 = # de peças brancas
  - f_2 = # de peças pretas
  - $f_3 = f_1/f_2$
  - f_4 = estimativa de ameaça ao rei

Funções de avaliação

- função pode ser linear:

$$(w_1 \times f_1) + (w_2 \times f_2) + (w_3 \times f_3) + \ldots + (w_n \times f_n)$$

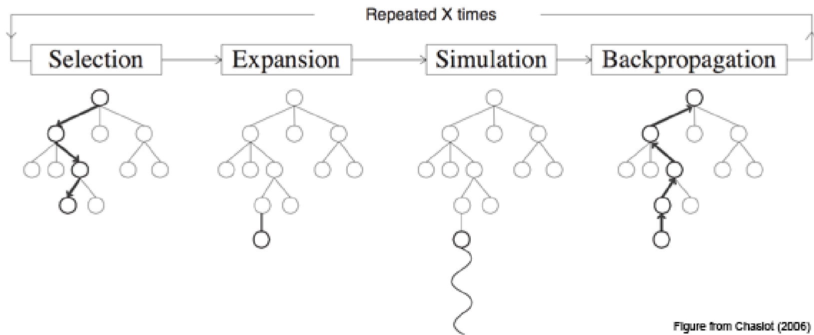
- Pesos w_i podem ser aprendidos

Monte Carlo Tree Search (MCTS)

- Minimax e alpha-beta podem não conseguir resolver alguns jogos cujo espaço de estados seja muito grande. Por exemplo:
  - Battleship Poker com informação imperfeita e jogos não determinísticos tais como Backgammon e Monopoly
  - Go: fator de ramificação: 300!

Monte Carlo Tree Search ajuda a resolver estes tipos de jogos.

Monte Carlo Tree Search (MCTS)

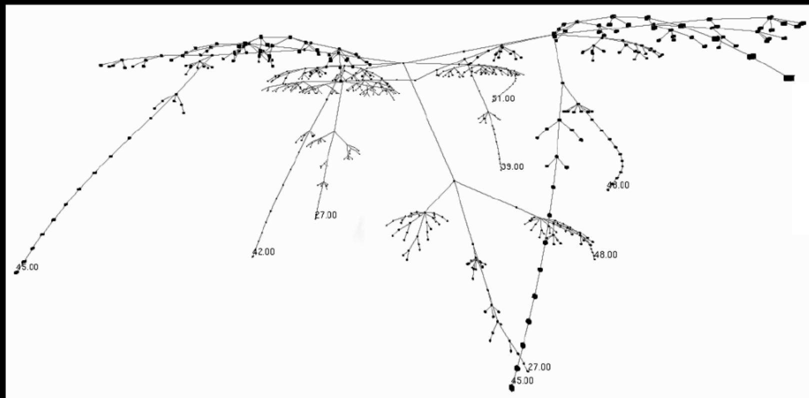


Monte Carlo Tree Search (MCTS)

- Usado no Go
- Go utiliza MCTS + Convolutional Neural Networks (CNN) + Reinforcement Learning

Mais info sobre AlphaGo e MCTS:
`https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-al`

MCTS tree example



Algoritmo MCTS

MCTS Algorithm for Action Selection

```
repeat N times {  // N might be between 100 and 1,000,000
    // set up data structure to record line of play
    visited = new List<Node>()
    // select node to expand
    node = root
    visited.add(node)
    while (node is not a leaf) {
      node = select(node, node.children)  // e.g. UCT selection
      visited.add(node)
    }
    // add a new child to the tree
    newChild = expand(node)
    visited.add(newChild)
    value = rollOut(newChild)
    for (node : visited)
        // update the statistics of tree nodes traversed
        node.updateStats(value);
    }
}
return action that leads from root node to most valued child
```

Algoritmo MCTS: como selecionar o melhor nó

Upper Confidence Bounds for Trees

$$UCB1 = \bar{X}_j + C\sqrt{\frac{2\ln n}{n_j}}$$

Algoritmo MCTS: como selecionar o melhor nó

- X_j recompensa estimada da escolha j
- n número de vezes em que o pai foi visitado
- n_j número de vezes em que a escolha j foi feita
- *Exploitation*: 1ª parcela da soma (esquerda):
  - enfatiza a recompensa
  - torna a busca mais guiada
- *Exploration*: 2ª parcela da soma (direita):
  - reforça a exploração de nós menos frequentemente visitados
  - reduz o efeito de "rollouts" com pouca sorte
  - Constante C equilibra *Exploitation* e *Exploration*