Sistemas Inteligentes - Teoria de Jogos

Alberto Barbosa

Universidade da Maia

alberto.barbosa@umaia.pt

April 30, 2024

Jogos de Soma Zero

Em Inteligência Artificial, o tipo de jogos mais estudado são os jogos determinísticos de dois jogadores de **soma zero** (com informação perfeita).
Informação perfeita significa que o estado do jogo é **completamente observável** (não existe informação "escondida").
Soma zero significa que o que é bom para um jogador é, na mesma medida, mau para o outro jogador. Isto significa que não existem situações "win-win".

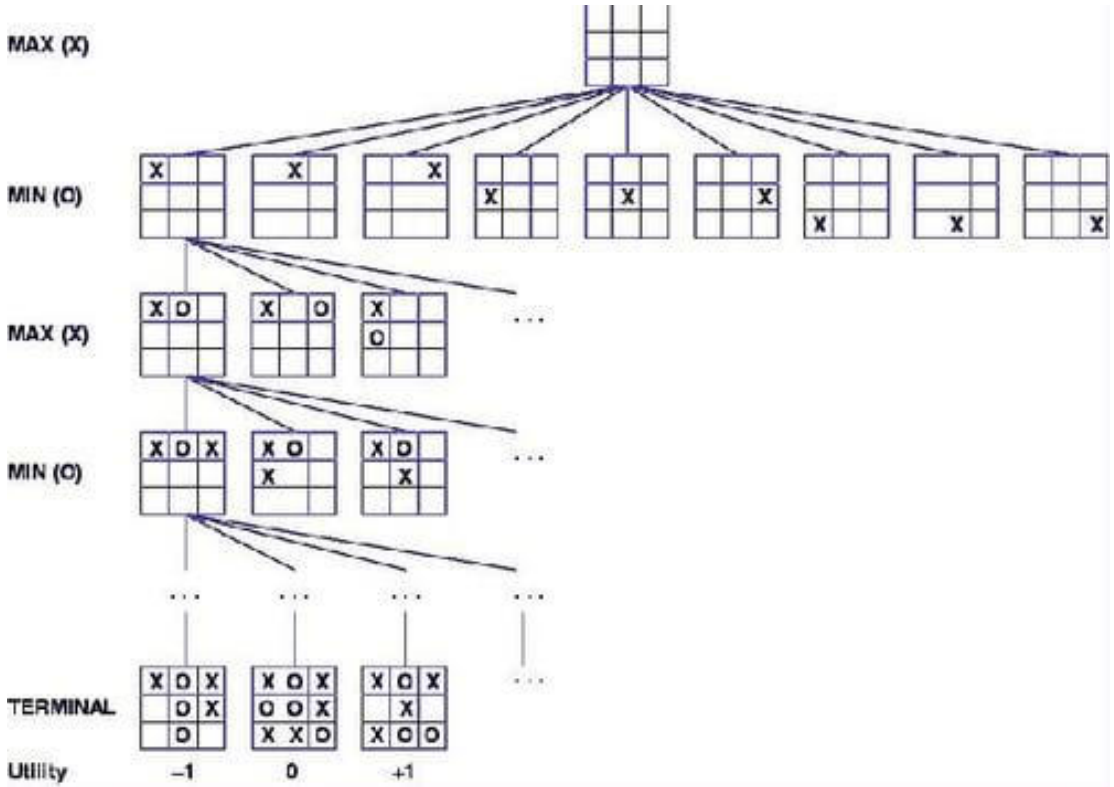
Jogo: Definição

Vamos chamar a dois jogadores MIN e MAX. MAX joga primeiro e depois os jogadores jogam à vez até o jogo acabar. No final do jogo, o vencedor recebe uma pontuação e o vencido recebe uma penalização.
Podemos definir um jogo formalmente:

- S_0: Estado inicial do jogo.
- **TO-MOVE(*s*)**: O jogador que tem a vez de jogar.
- **ACTIONS(*s*)**: Conjunto de jogadas legais no estado *s*.
- **RESULTS(*s*,a)**: O modelo de transições que define o estado que resulta de aplicar a acção *a* no estado *s*.
- **IS-TERMINAL(*s*)**: Teste terminal que retorna verdade se o jogo acabou e falso caso não tenha acabado.
- **UTILITY(*s*,*p*)**: Função de utilidade que define um valor numérico para o jogador *p* quando o jogo termina no estado *s*. No xadrez, por exemplo, o resultado pode ser vitória, derrota ou empate com valores 1, 0 ou 1/2 respectivamente. No gamão, por exemplo, a função de utilidade pode variar entre 0 e 192.

Árvore de Jogo

Tal como vimos anteriormente, o estado inicial, a função ACTIONS e a função RESULTS definem o grafo do espaço de estados e podemos modelar este espaço de estados como uma árvore de pesquisa.
A árvore de jogo é uma árvore de pesquisa que contém todas as sequências de jogadas desde o estado inicial até um estado terminal.



MINMAX: Ideia-chave

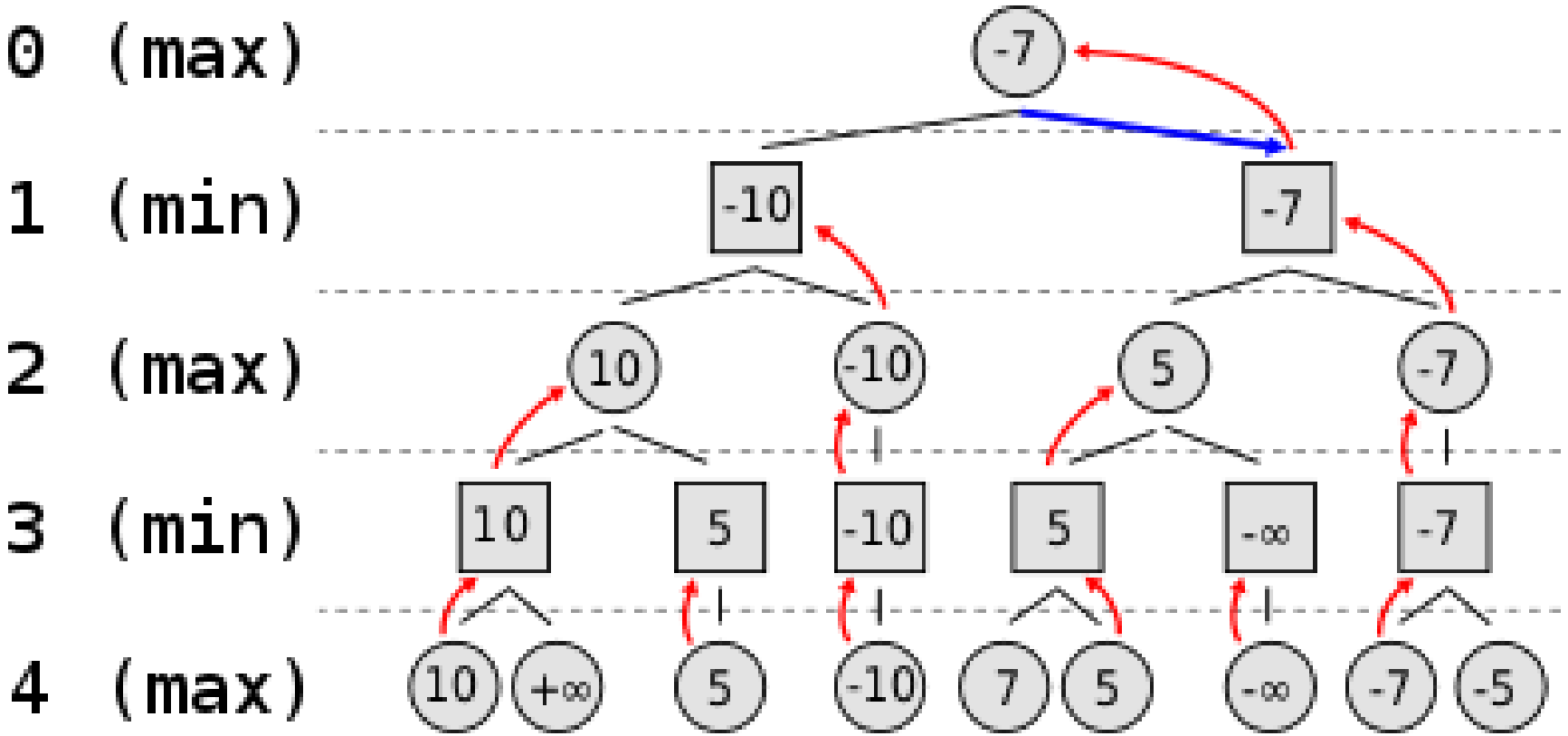
Vamos assumir que MAX quer encontrar uma sequência de acções que levam a uma vitória.
No entanto, MIN pode influenciar o desfecho desta sequência de acções com as suas jogadas.
Então, MAX tem de elaborar um plano condicional que tenha uma resposta para cada uma das possíveis jogadas de MIN.
Para determinar quais os melhores passos a dar por parte do MAX, iremos usar um algoritmo chamado **MINMAX**.

Valores minmax

Dada uma árvore de jogo, a estratégia óptima pode ser descoberta através da determinação do valor **minmax** de cada estado da árvore.
Este valor descreve o valor de utilidade de um determinado estado, assumindo que ambos os jogadores jogam de forma óptima. O valor minmax de um estado terminal é igual ao seu valor de utilidade.
Em cada passo, MAX vai querer fazer a jogada que maximiza este valor de minmax e MIN vai querer fazer a jogada que minimiza este valor. Então, temos que podemos definir uma função MINMAX(s) da seguinte forma:

- Se s é um estado final, retornamos UTILITY(s,MAX)
- Se é a vez de MAX jogar, então retornamos $max_{a \in ACTIONS(s)} MINMAX(RESULT(s, a))$.
- Se é a vez de MIN jogar, então retornamos $min_{a \in ACTIONS(s)} MINMAX(RESULT(s, a))$.

MINMAX exemplo



Algoritmo MINMAX

Agora que já sabemos calcular o valor de minmax, podemos descrever um algoritmo de pesquisa que se baseie nestes valores e descubra qual a melhor jogada a fazer por parte de MAX.

Algoritmo MINMAX

Algorithm 1 MINMAX-SEARCH(game,state)

player ← *game*.TO-MOVE(*state*)
value, *move* ← MAX-VALUE(*game*,*state*)
return *move*

Algorithm 2 MAX-VALUE(game,state)

if *game*.IS-TERMINAL(*state*) **then**
    return *game*.UTILITY(*state*,*player*),*null*
end if
v, *move* ← $-\infty$
for all *a* **in** *game*.ACTIONS(*state*) **do**
    $v2$, $a2$ ← MIN-VALUE(*game*, *game*.RESULT(*state*,*a*))
    if $v2 > v$ **then**
        v, *move* ← $v2$, a
    end if
end for
return v, *move*

Algoritmo MINMAX

Algorithm 3 MIN-VALUE(game,state)

```
if game.IS-TERMINAL(state) then
    return game.UTILITY(state,player),null
end if
v, move ← +∞
for all a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state,a))
    if v2 < v then
        v, move ← v2, a
    end if
end for
return v, move
```

Alpha-Beta

O número de estados pode crescer exponencialmente com a profundidade da árvore.
No entanto, é possível dar a resposta correcta com o algoritmo MINMAX sem examinar a árvore completamente, através de técnicas de **pruning**.
Pruning nesta abordagem consiste essencialmente consiste em descartar partes da árvore que não influenciem o resultado.
Em concreto, iremos abordar o algoritmo **alpha-beta pruning**.

Alpha-Beta

A ideia principal do algoritmo é fazer uma pesquisa igual a MINMAX. Porém, não serão analisados nós que saibamos que **garantidamente** não irão ter influência no resultado, poupando-nos **recursos computacionais**. O algoritmo pode ser aplicado a árvores de **qualquer profundidade**. Consideremos um nó n na árvore, tal que um jogador pode mover-se para n. Se existir uma melhor opção para esse jogador quer no mesmo nível da árvore, quer num nível superior, então o jogador nunca irá mover-se para n. Então, mal saibamos o suficiente sobre n para perceber que o mesmo nunca será uma opção viável, podemos **podar** este nó.

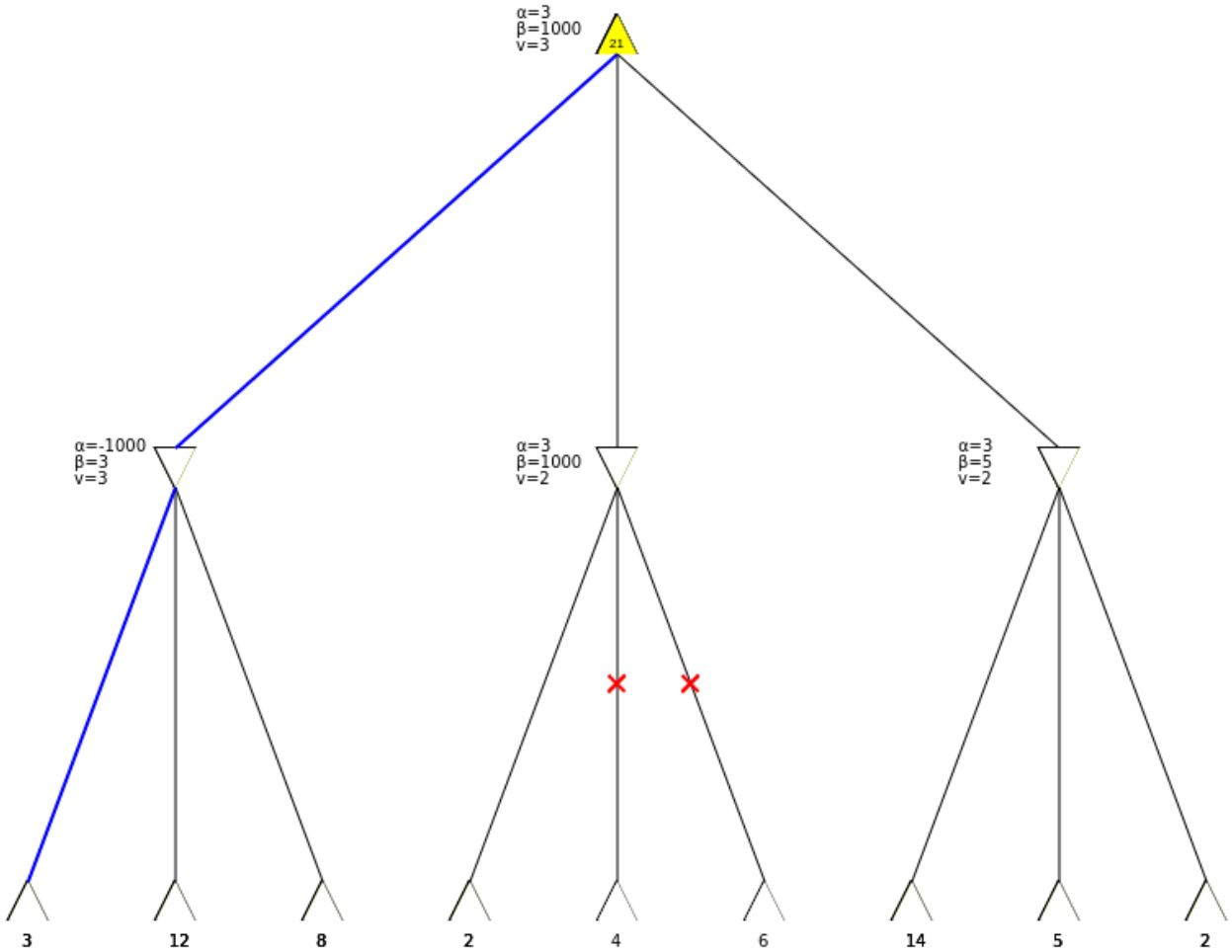
Algoritmo Alpha-Beta

O algoritmo alpha-beta tem este nome porque introduz dois parâmetros novos:

- α: valor da melhor escolha ao longo do caminho para MAX (podemos ver α como "consigo este valor no mínimo").
- β: valor da melhor escolha ao longo do caminho para MIN (podemos ver β como "no máximo consigo este valor").

O algoritmo vai actualizando os valores de α e β à medida que percorre a árvore e vai podando os restantes ramos da árvore assim que o valor de um nó é pior que α ou β para MAX ou MIN, respectivamente.

Alpha-Beta exemplo



Algoritmo Alpha-Beta

Algorithm 4 ALPHA-BETA-SEARCH(game,state)

player $\leftarrow$ *game*.TO-MOVE(*state*)
value, *move* $\leftarrow$ MAX-VALUE(*game*,*state*,$-\infty$,$+\infty$)
return *move*

Algorithm 5 MAX-VALUE(game,state,α,β)

if *game*.IS-TERMINAL(*state*) **then**
    return *game*.UTILITY(*state*,*player*),*null*
end if
v, *move* $\leftarrow -\infty$
for all *a* **in** *game*.ACTIONS(*state*) **do**
    *v*2, *a*2 $\leftarrow$ MIN-VALUE(*game*, *game*.RESULT(*state*,*a*),α,β)
    if *v*2 $>$ *v* **then**
        v, *move* $\leftarrow$ *v*2, *a*
        $\alpha \leftarrow$ MAX(α,*v*)
    end if
    if *v* $\geq \beta$ **then**
        return *v*, *move*
    end if
end for
return *v*, *move*

Algoritmo Alpha-Beta

Algorithm 6 MIN-VALUE(game,state,α,β)

if *game*.IS-TERMINAL(*state*) **then**
  return *game*.UTILITY(*state*,*player*),*null*
end if
$v \leftarrow +\infty$
for all *a* **in** *game*.ACTIONS(*state*) **do**
  $v2$, $a2 \leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*,*a*),α,β)
  if $v2 < v$ **then**
    v, *move* $\leftarrow$ $v2$, *a*
    $\beta \leftarrow$ MIN(β,v)
  end if
  if $v \leq \alpha$ **then**
    return v,*move*
  end if
end for
return v, *move*

Monte Carlo Tree Search

Ainda que seja uma melhoria relativamente ao Minmax, o algoritmo Alpha-Beta tem algumas limitações: a principal delas é o facto de não ser possível a aplicação do mesmo, caso a árvore de jogo seja demasiado grande. Também podemos ter um problema em arranjar boas funções de utilidade para alguns jogos.
Assim sendo, surge uma nova estratégia chamada Monte Carlo Tree Search.

Monte Carlo Tree Search

O valor de utilidade de um nó em MCTS é estimado como a utilidade média relativamente a um número de simulações de jogos completos a partir do nó em questão.
Uma simulação escolhe jogadas para cada um dos jogadores à vez até que chegamos a um estado final. Nesse ponto, as regras do jogo determinam quem é o vencedor e qual a pontuação a atribuir.

Monte Carlo Tree Search

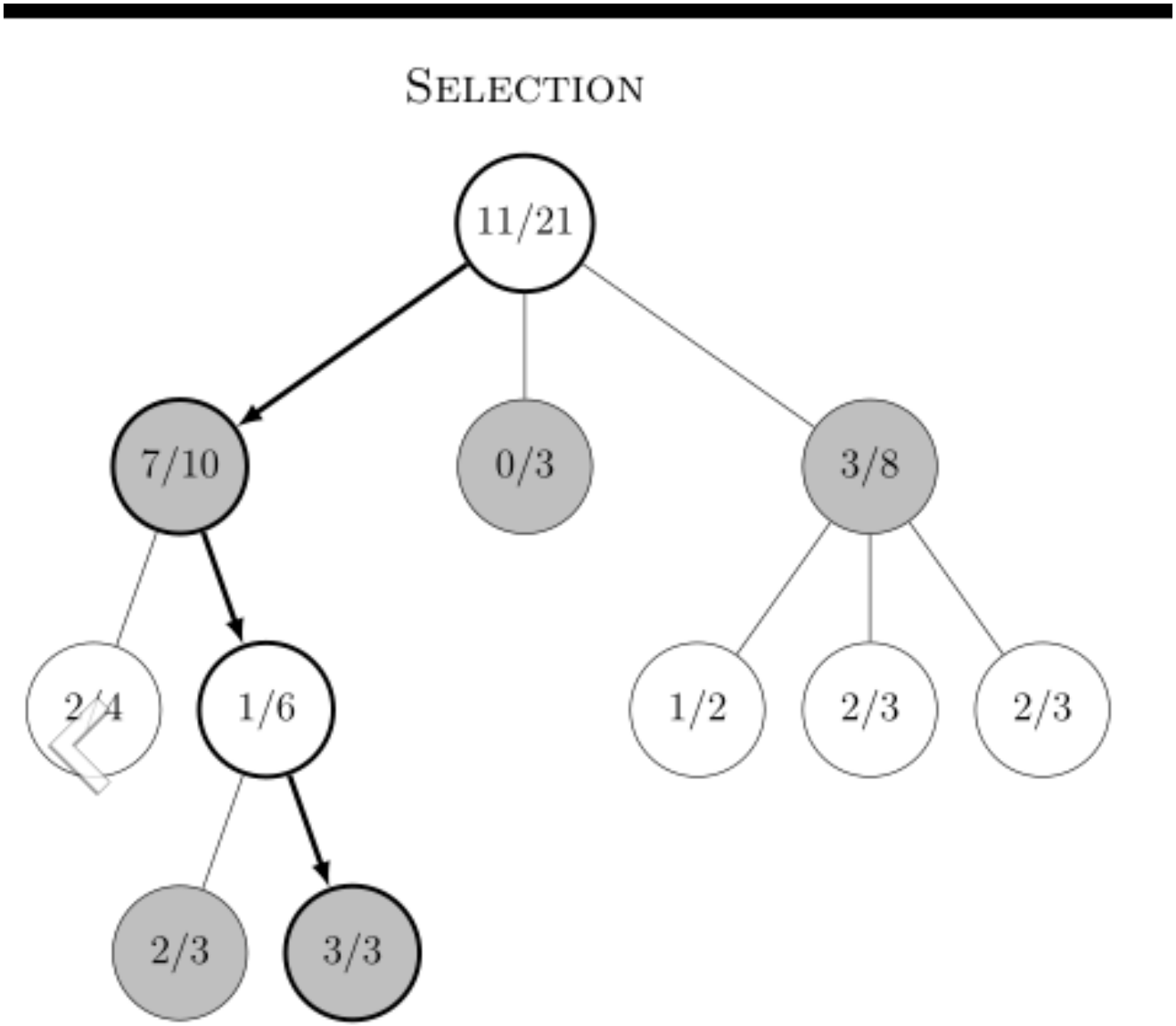
Como podemos escolher as jogadas a efectuar durante as simulações? Se escolhermos de forma aleatória, estaremos a tentar descobrir qual é a melhor jogada a fazer se ambos jogarem de forma aleatória.
Para tal, precisamos de uma playout policy que determina quais as melhores jogadas (pode ser uma heurística ou um modelo mais complexo).
A partir daqui, fazemos N simulações a partir do nó actual e registamos qual das jogadas terá melhor percentagem de vitória.
No entanto, com esta estratégia, podemos não convergir para jogadas óptimas. Então precisamos de uma selection policy que irá concentrar a maior parte dos recursos computacionais nos nós mais promissores, equilibrando dois factores: exploração de estados que tiveram poucas simulações e aproveitamento de estados que tiveram bom desempenho em simulações passadas.
A esta gestão chamamos o **exploration/exploitation tradeoff**.

Monte Carlo Tree Search

A forma de o MCTS gerir estas duas componentes é através da manutenção de uma árvore de jogo que cresce a cada iteração de acordo com quatro passos:

- **Selecção:** Começando na raiz, escolhemos uma jogada de acordo com a selection policy, gerando um nó sucessor e repetimos o processo até que chegamos a uma folha (nó terminal).

Monte Carlo exemplo

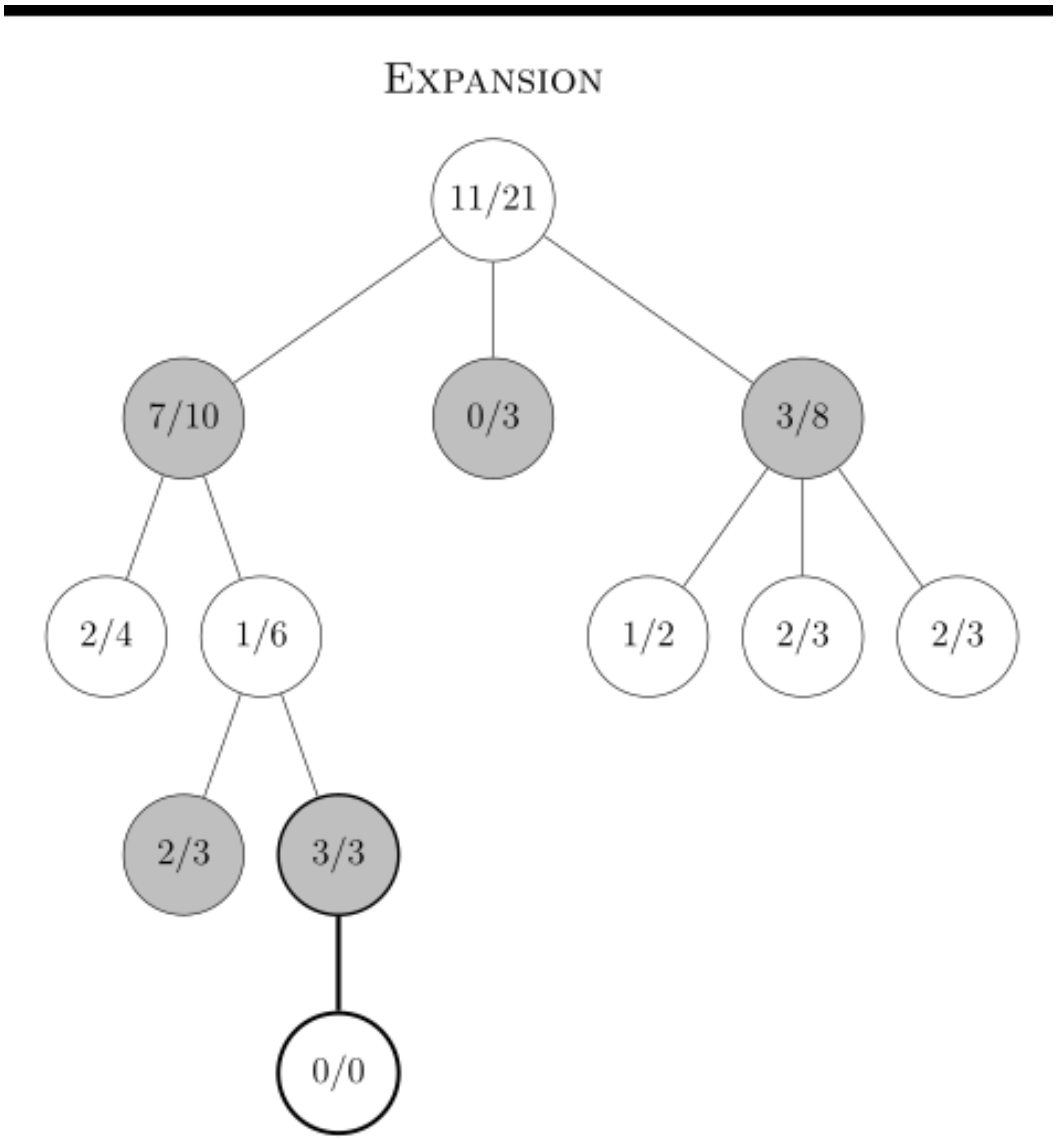


Monte Carlo Tree Search

A forma de o MCTS gerir estas duas componentes é através da manutenção de uma árvore de jogo que cresce a cada iteração de acordo com quatro passos:

- **Expansão:** Fazemos crescer a árvore através da geração de um novo filho do nó selecionado. Algumas versões geram mais que um filho.

Monte Carlo exemplo

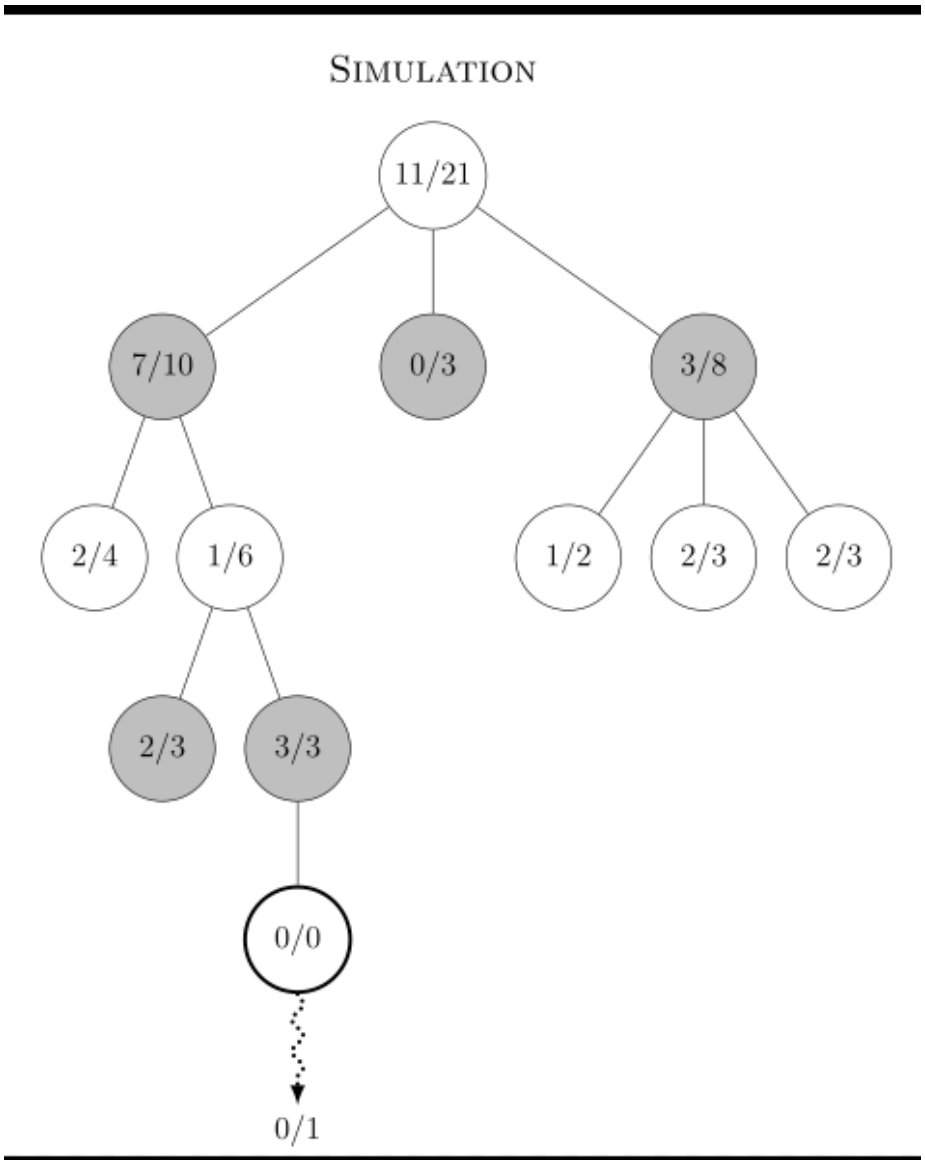


Monte Carlo Tree Search

A forma de o MCTS gerir estas duas componentes é através da manutenção de uma árvore de jogo que cresce a cada iteração de acordo com quatro passos:

- **Simulação:** Fazemos uma simulação a partir do novo nó, escolhendo jogadas para ambos os jogadores de acordo com a playout policy. Estas jogadas não ficam registadas na árvore. Registamos apenas o resultado desta simulação.

Monte Carlo exemplo

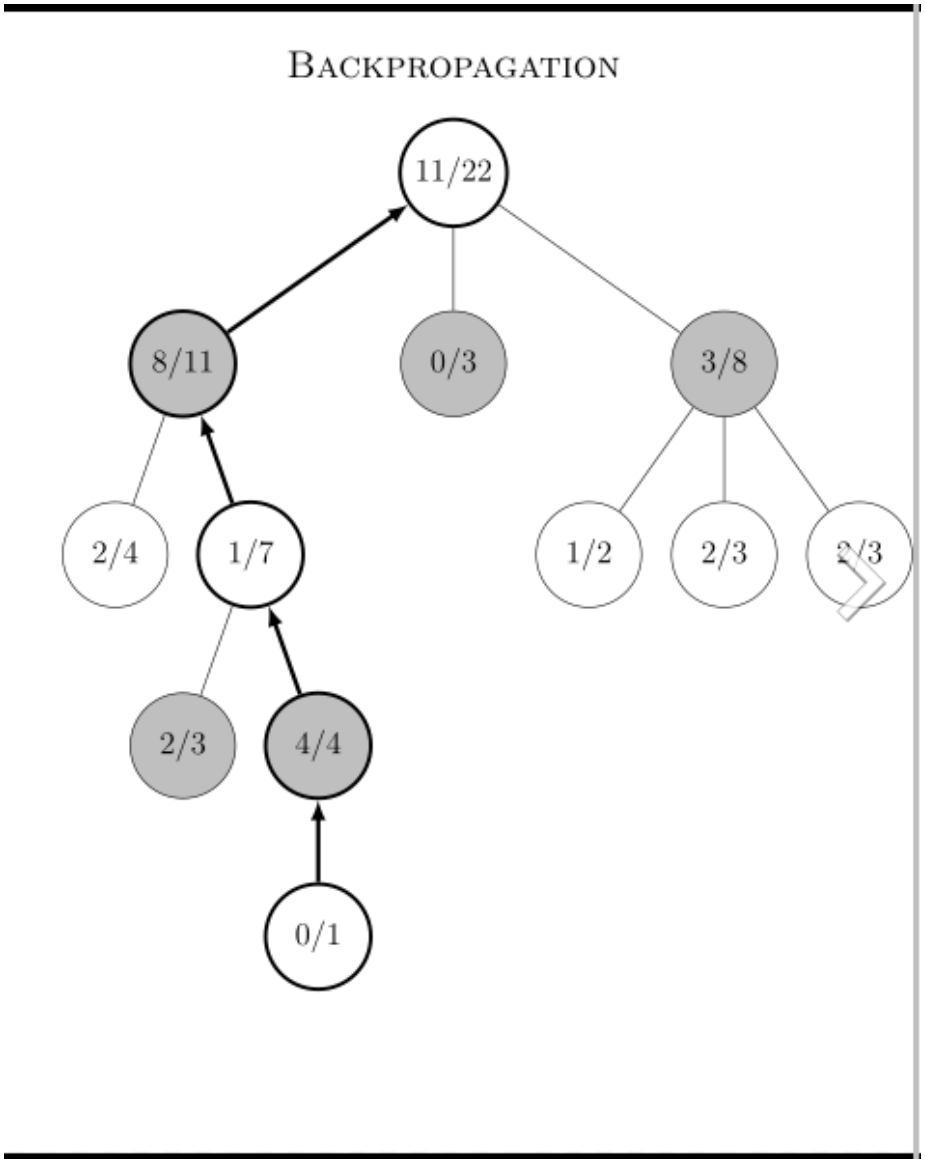


Monte Carlo Tree Search

A forma de o MCTS gerir estas duas componentes é através da manutenção de uma árvore de jogo que cresce a cada iteração de acordo com quatro passos:

- **Backpropagation:** Usamos a simulação para actualizar os nós da árvore até à raíz. Uma vez que o branco perdeu, iremos aumentar apenas o número de simulações nos nós brancos e aumentaremos o número de simulações e de vitórias nos cinzentos.

Monte Carlo exemplo



Monte Carlo Tree Search

Estes passos são repetidos durante um determinado número de iterações ou até o tempo alocado à tarefa ter terminado e retornamos a jogada com maior número de simulações.

Monte Carlo Tree Search

Uma selection policy bastante utilizada é a Upper Confidence Bounds Applied to Trees, onde usamos uma fórmula para calcular um limite superior para confiança, chamada **UCB1**. Para cada nó, a fórmula é:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT(n)})}{N(n)}} \quad (1)$$

Onde $U(n)$ é a utilidade de todas as simulações que passaram por n. $N(n)$ é o número de simulações que passaram por n. PARENT(n) é o nó pai de n na árvore.

Monte Carlo Tree Search

Então, $\frac{U(n)}{N(n)}$ é o termo de aproveitamento (exploitation): a utilidade média de n.
Por outro lado, $\sqrt{\frac{\log N(\text{PARENT(n)})}{N(n)}}$ é o termo de exploração (exploration).
Tem o número de simulações no denominador $N(n)$ o que significa que tomará valores maiores para nós que foram explorados poucas vezes. No numerador tem o logaritmo do número de vezes que exploramos o nó pai de n.
Isto significa que a forma de escolher n terá um impacto cada vez menor do termo de exploração à medida que N cresce e portanto eventualmente, será dado maior número de simulações a nós com maior utilidade média.
C é uma constante para equilibrar exploração e aproveitamento.
Normalmente é determinada empiricamente (sendo $\sqrt{2}$ um bom valor inicial).

Algoritmo Monte Carlo Tree Search

Algorithm 7 MCTS(*state*)

tree ← NODE(*state*)
while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*,*child*)
end while
return move in ACTIONS(*state*) with the highest number of playouts.

Monte Carlo Tree Search

MCTS é uma boa alternativa ao Alpha-Beta quando o branching factor da árvore de jogo é muito grande e/ou é difícil definir uma boa função de avaliação.
Uma vez que o alpha-beta utiliza a função de avaliação para decidir qual o nó com maior qualidade para ser escolhido. Ora, se a função de avaliação não está bem definida, estes resultados terão pouca qualidade. O MCTS, por sua vez, utiliza o agregado de todas as simulações, pelo que se torna menos vulnerável a estes erros.