Alberto Barbosa

Universidade da Maia *alberto.barbosa@umaia.pt*

Sistemas Inteligentes - Teoria de Jogos

April 30, 2024

# Jogos de Soma Zero

Em Inteligˆencia Artificial, o tipo de jogos mais estudado s˜ao os jogos determin´ısticos de dois jogadores de soma zero (com informac¸˜ao perfeita). Informa¸c˜ao perfeita significa que o estado do jogo ´e completamente observ´avel (n˜ao existe informac¸˜ao ”escondida”).

Soma zero significa que o que ´e bom para um jogador ´e, na mesma medida, mau para o outro jogador. Isto significa que n˜ao existem situa¸c˜oes ”win-win”.

# Jogo: Defini¸c˜ao

Vamos chamar a dois jogadores MIN e MAX. MAX joga primeiro e depois os jogadores jogam `a vez at´e o jogo acabar. No final do jogo, o vencedor recebe uma pontua¸c˜ao e o vencido recebe uma penaliza¸c˜ao. 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*.

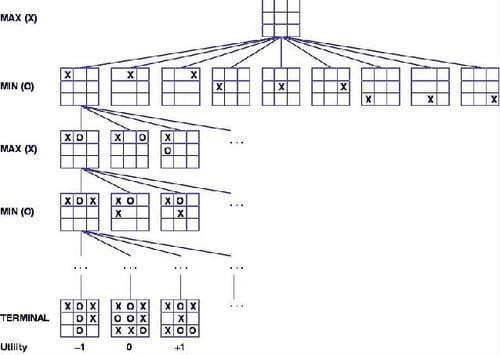
RESULTS(*s*,a): O modelo de transi¸c˜oes que define o estado que resulta de aplicar a ac¸c˜ao *a* no estado *s*.

IS-TERMINAL(*s*): Teste terminal que retorna verdade se o jogo acabou e falso caso n˜ao tenha acabado.

UTILITY(*s*,*p*): Func¸˜ao de utilidade que define um valor num´erico para o jogador *p* quando o jogo termina no estado *s*. No xadrez, por exemplo, o resultado pode ser vit´oria, derrota ou empate com valores 1, 0 ou 1/2 respectivamente. No gam˜ao, por exemplo, a fun¸c˜ao de utilidade pode variar entre 0 e 192.

# Arvore de Jogo´

Tal como vimos anteriormente, o estado inicial, a fun¸c˜ao ACTIONS e a fun¸c˜ao RESULTS definem o grafo do espa¸co de estados e podemos modelar este espac¸o de estados como uma ´arvore de pesquisa. A ´arvore de jogo ´e uma ´arvore de pesquisa que cont´em todas as sequˆencias de jogadas desde o estado inicial at´e um estado terminal.



# MINMAX: Ideia-chave

Vamos assumir que MAX quer encontrar uma sequˆencia de ac¸c˜oes que levam a uma vit´oria.

No entanto, MIN pode influenciar o desfecho desta sequˆencia de ac¸c˜oes com as suas jogadas.

Ent˜ao, MAX tem de elaborar um plano condicional que tenha uma resposta para cada uma das poss´ıveis 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 ´arvore de jogo, a estrat´egia ´optima pode ser descoberta atrav´es da determina¸c˜ao do valor minmax de cada estado da ´arvore. Este valor descreve o valor de utilidade de um determinado estado, assumindo que ambos os jogadores jogam de forma ´optima. O valor minmax de um estado terminal ´e igual ao seu valor de utilidade.

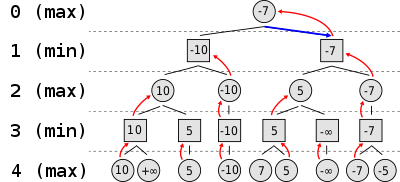
Em cada passo, MAX vai querer fazer a jogada que maximiza este valor de minmax e MIN vai querer fazer a jogada que minimiza este valor. Ent˜ao, temos que podemos definir uma fun¸c˜ao MINMAX(*s*) da seguinte forma:

Se *s* ´e um estado final, retornamos UTILITY(*s*,MAX)



Se ´e a vez de MAX jogar, ent˜ao retornamos *maxa*∈*ACTIONS*(*s*)*MINMAX*(*RESULT*(*s,a*)). Se ´e a vez de MIN jogar, ent˜ao retornamos *mina*∈*ACTIONS*(*s*)*MINMAX*(*RESULT*(*s,a*)).

# MINMAX exemplo



Agora que j´a 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.

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* ←−∞

for all *a* in *game*.ACTIONS(*state*) do

*v*2, *a*2 ← MIN-VALUE(*game*, *game*.RESULT(*state*,*a*))

if *v*2 *> v* then

*v*, *move* ← *v*2, *a*

end if

end for return *v*, *move*

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

*v*2, *a*2 ← MAX-VALUE(*game*, *game*.RESULT(*state*,*a*))

if *v*2 *< v* then

*v*, *move* ← *v*2, *a*

end if

end for return *v*, *move*

# Alpha-Beta

O nu´mero de estados pode crescer exponencialmente com a profundidade da ´arvore.

No entanto, ´e poss´ıvel dar a resposta correcta com o algoritmo MINMAX sem examinar a ´arvore completamente, atrav´es de t´ecnicas de pruning. Pruning nesta abordagem consiste essencialmente consiste em descartar partes da ´arvore que n˜ao influenciem o resultado.

Em concreto, iremos abordar o algoritmo alpha-beta pruning.

# Alpha-Beta

A ideia principal do algoritmo ´e fazer uma pesquisa igual a MINMAX. Por´em, n˜ao ser˜ao analisados n´os que saibamos que garantidamente n˜ao ir˜ao ter influˆencia no resultado, poupando-nos recursos computacionais. O algoritmo pode ser aplicado a ´arvores de qualquer profundidade. Consideremos um n´o *n* na ´arvore, tal que um jogador pode mover-se para

*n*. Se existir uma melhor op¸c˜ao para esse jogador quer no mesmo n´ıvel da ´arvore, quer num n´ıvel superior, ent˜ao o jogador nunca ir´a mover-se para *n*. Ent˜ao, mal saibamos o suficiente sobre *n* para perceber que o mesmo nunca ser´a uma opc¸˜ao vi´avel, podemos podar este n´o.

# Algoritmo Alpha-Beta

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

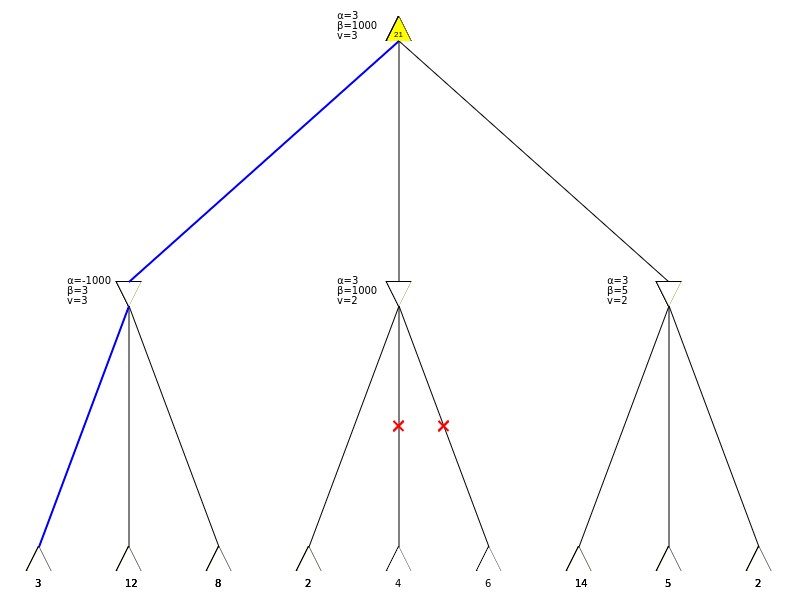
*α*: valor da melhor escolha ao longo do caminho para MAX (podemos ver *α* como ”consigo este valor no m´ınimo”).



*β*: valor da melhor escolha ao longo do caminho para MIN (podemos ver *β* como ”no m´aximo consigo este valor”).

O algoritmo vai actualizando os valores de *α* e *β* `a medida que percorre a ´arvore e vai podando os restantes ramos da ´arvore assim que o valor de um n´o ´e pior que *α* ou *β* para MAX ou MIN, respectivamente.

# Alpha-Beta exemplo



# Algoritmo Alpha-Beta

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

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

for all *a* in *game*.ACTIONS(*state*) do *v*2, *a*2 ← MIN-VALUE(*game*, *game*.RESULT(*state*,*a*),*α*,*β*)

if *v*2 *> v* then

*v*, *move* ← *v*2, *a α*← MAX(*α*,*v*)

end if

if *v* ≥*β* 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* ← +∞

for all *a* in *game*.ACTIONS(*state*) do *v*2, *a*2 ← MAX-VALUE(*game*, *game*.RESULT(*state*,*a*),*α*,*β*)

if *v*2 *< v* then *v*, *move* ← *v*2, *a β*← MIN(*β*,*v*)

end if

if *v* ≤*α* then

return *v*,*move*

end if

end for return *v*, *move*

Ainda que seja uma melhoria relativamente ao Minmax, o algoritmo Alpha-Beta tem algumas limita¸c˜oes: a principal delas ´e o facto de n˜ao ser poss´ıvel a aplica¸c˜ao do mesmo, caso a ´arvore de jogo seja demasiado grande. Tamb´em podemos ter um problema em arranjar boas fun¸c˜oes de utilidade para alguns jogos.

Assim sendo, surge uma nova estrat´egia chamada Monte Carlo Tree Search.

O valor de utilidade de um n´o em MCTS ´e estimado como a utilidade m´edia relativamente a um nu´mero de simula¸c˜oes de jogos completos a partir do n´o em quest˜ao.

Uma simula¸c˜ao escolhe jogadas para cada um dos jogadores `a vez at´e que chegamos a um estado final. Nesse ponto, as regras do jogo determinam quem ´e o vencedor e qual a pontua¸c˜ao a atribuir.

Como podemos escolher as jogadas a efectuar durante as simulac¸˜oes? Se escolhermos de forma aleat´oria, estaremos a tentar descobrir qual ´e a melhor jogada a fazer se ambos jogarem de forma aleat´oria. Para tal, precisamos de uma playout policy que determina quais as melhores jogadas (pode ser uma heur´ıstica ou um modelo mais complexo). A partir daqui, fazemos *N* simula¸c˜oes a partir do n´o actual e registamos qual das jogadas ter´a melhor percentagem de vit´oria.

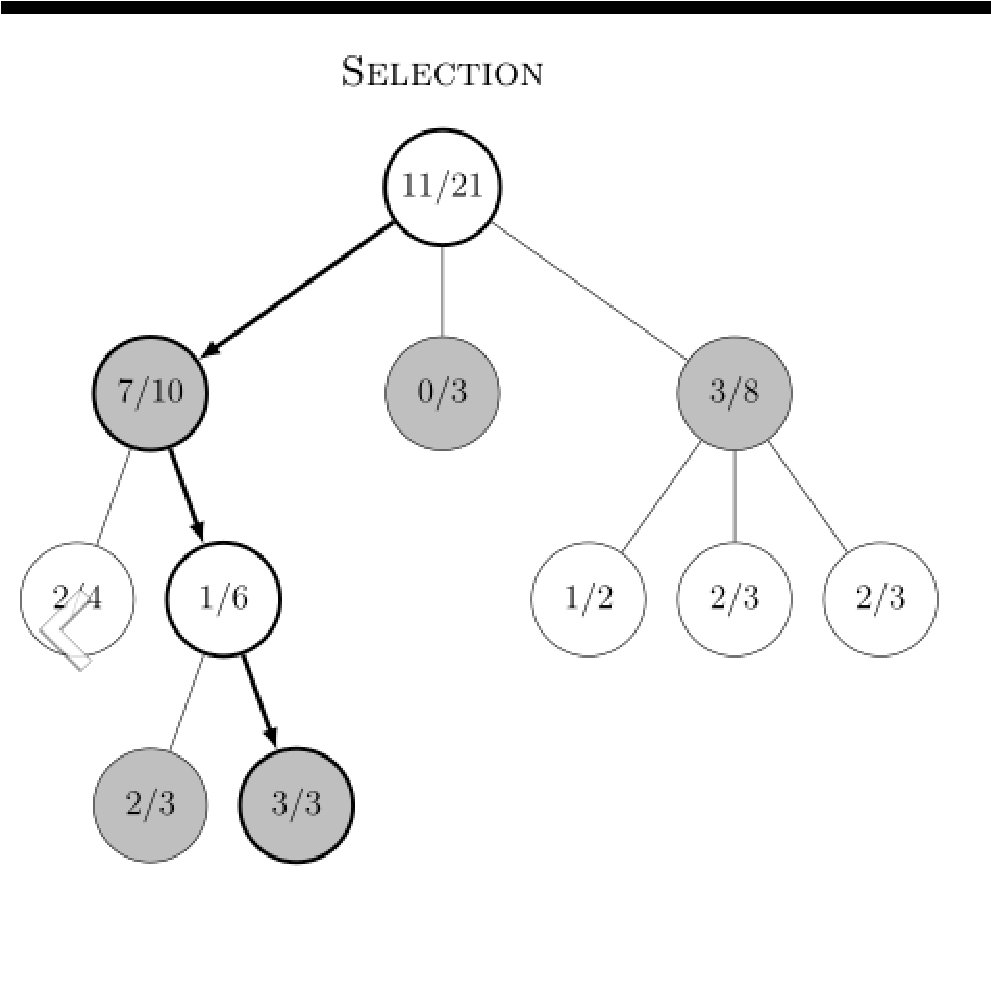
No entanto, com esta estrat´egia, podemos n˜ao convergir para jogadas ´optimas. Ent˜ao precisamos de uma selection policy que ir´a concentrar a maior parte dos recursos computacionais nos n´os mais promissores, equilibrando dois factores: explora¸c˜ao de estados que tiveram poucas simula¸c˜oes e aproveitamento de estados que tiveram bom desempenho em simula¸c˜oes passadas.

A esta gest˜ao chamamos o exploration/exploitation tradeoff.

A forma de o MCTS gerir estas duas componentes ´e atrav´es da manuten¸c˜ao de uma ´arvore de jogo que cresce a cada itera¸c˜ao de acordo com quatro passos:

Selecc¸˜ao: Come¸cando na raiz, escolhemos uma jogada de acordo com a selection policy, gerando um n´o sucessor e repetimos o processo at´e que chegamos a uma folha (n´o terminal).

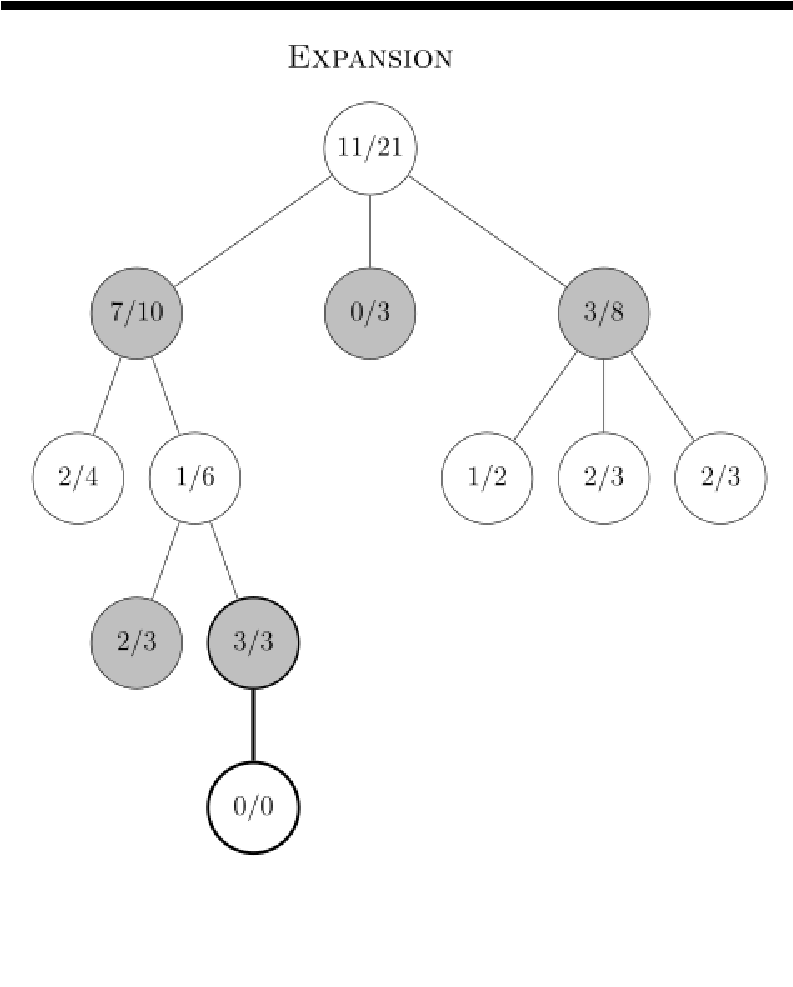




A forma de o MCTS gerir estas duas componentes ´e atrav´es da manuten¸c˜ao de uma ´arvore de jogo que cresce a cada itera¸c˜ao de acordo com quatro passos:

Expans˜ao: Fazemos crescer a ´arvore atrav´es da gerac¸˜ao de um novo filho do n´o selecionado. Algumas vers˜oes geram mais que um filho.

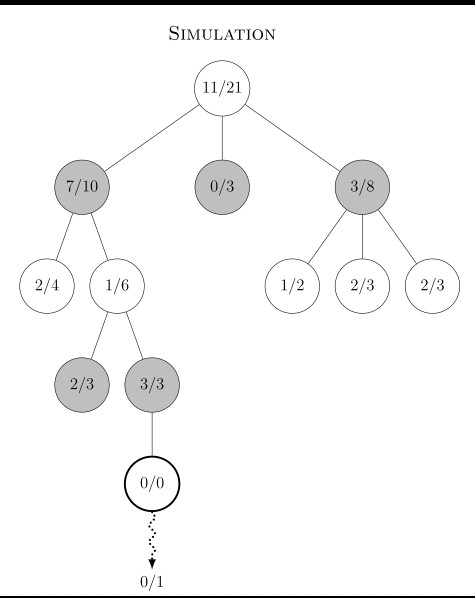




A forma de o MCTS gerir estas duas componentes ´e atrav´es da manuten¸c˜ao de uma ´arvore de jogo que cresce a cada itera¸c˜ao de acordo com quatro passos:

Simulac¸˜ao: Fazemos uma simula¸c˜ao a partir do novo n´o, escolhendo jogadas para ambos os jogadores de acordo com a playout policy. Estas jogadas n˜ao ficam registadas na ´arvore. Registamos apenas o resultado desta simulac¸˜ao.



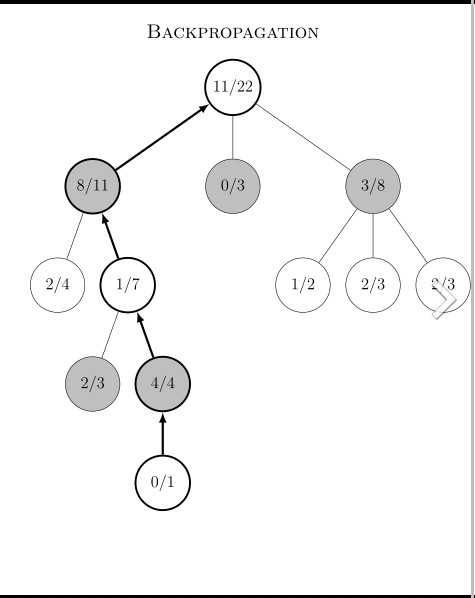


A forma de o MCTS gerir estas duas componentes ´e atrav´es da manuten¸c˜ao de uma ´arvore de jogo que cresce a cada itera¸c˜ao de acordo com quatro passos:

Backpropagation: Usamos a simulac¸˜ao para actualizar os n´os da



´arvore at´e `a ra´ız. Uma vez que o branco perdeu, iremos aumentar apenas o nu´mero de simula¸c˜oes nos n´os brancos e aumentaremos o nu´mero de simulac¸˜oes e de vit´orias nos cinzentos.

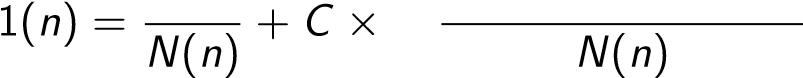


Estes passos s˜ao repetidos durante um determinado nu´mero de iterac¸˜oes ou at´e o tempo alocado `a tarefa ter terminado e retornamos a jogada com maior nu´mero de simula¸c˜oes.

Uma selection policy bastante utilizada ´e a Upper Confidence Bounds Applied to Trees, onde usamos uma f´ormula para calcular um limite superior para confianc¸a, chamada UCB1. Para cada n´o, a f´ormula ´e:

s

*U*(*n*) log*N*(PARENT(n))

*UCB* (1)

Onde *U*(*n*) ´e a utilidade de todas as simula¸c˜oes que passaram por *n*. *N*(*n*) ´e o nu´mero de simula¸c˜oes que passaram por *n*. PARENT(*n*) ´e o n´o pai de *n* na ´arvore.

Ent˜ao, *NU*((*nn*)) ´e o termo de aproveitamento (exploitation): a utilidade m´edia de *n*.

qlog*N*(PARENT(n))

Por outro lado, *N*(*n*) ´e o termo de explora¸c˜ao (exploration).

Tem o nu´mero de simula¸c˜oes no denominador *N*(*n*) o que significa que tomar´a valores maiores para n´os que foram explorados poucas vezes. No numerador tem o logaritmo do nu´mero de vezes que exploramos o n´o pai de *n*.

Isto significa que a forma de escolher *n* ter´a um impacto cada vez menor do termo de explora¸c˜ao `a medida que *N* cresce e portanto eventualmente, ser´a dado maior nu´mero de simulac¸˜oes a n´os com maior utilidade m´edia. *C* ´e uma constante para equilibrar explora¸c˜ao e aproveitamento.√

Normalmente ´e determinada empiricamente (sendo 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.

MCTS ´e uma boa alternativa ao Alpha-Beta quando o branching factor da

´arvore de jogo ´e muito grande e/ou ´e dif´ıcil definir uma boa fun¸c˜ao de avaliac¸˜ao.

Uma vez que o alpha-beta utiliza a fun¸c˜ao de avaliac¸˜ao para decidir qual o n´o com maior qualidade para ser escolhido. Ora, se a fun¸c˜ao de avaliac¸˜ao n˜ao est´a bem definida, estes resultados ter˜ao pouca qualidade. O MCTS, por sua vez, utiliza o agregado de todas as simulac¸˜oes, pelo que se torna menos vulner´avel a estes erros.