Implementação e Avaliação do Algoritmo MCTS-UCT para o jogo Chinese Checkers

Jhonny Manuel Campos Moreira
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2016

Orientador
Inês de Castro Dutra, Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

In this work, we explore the game of Chinese checkers, with the goal of understanding how the algorithms behave in this game, and how we can improve on such algorithms. The main focus went to the Monte Carlo Tree Search version, Upper Confidence Bound for Trees, that presented excellent results in many other games such as Go. In our implementation, we parameterized the number of players, the number of explored nodes, the ramification factor and the node selection. We tested various different game scenarios and concluded that the best value for each individual parameter is dependent on these scenarios. Our results show in which conditions we can maximize the chance of victory for each scenario. More specifically, we maximize the chance of winning if, for the node selection, we use a heuristic-based choice, like a *Greedy* selection. By limiting the ramification factor, we also improve the chances of winning, in particular to the value of 10. And for the number or nodes explored, we conclude that exploring the maximum amount of nodes that we can, within our time limitations, help to improve the chances of winning, for both two and three players.

Resumo

Neste trabalho, exploramos o jogo de *Chinese Checkers* com o objetivo de entender como os algoritmos existentes na literatura se comportam neste jogo e como podem ser adaptados para melhorar os seus resultados. Em particular, estudamos e implementamos uma versão do *Monte Carlo Tree Search*, *Upper Confidence Bound for Trees*, que tem apresentado excelentes resultados em outros jogos, como, por exemplo, o Go. Na nossa implementação parametrizamos o número de jogadores, o número de nós explorados, o fator de ramificação e a seleção de nós. Testamos diferentes cenários de jogos e concluímos que os valores ideais para os diferentes parâmetros de cada jogo está dependente de cada um dos cenários, mostrando como podemos maximizar as chances de vitória para cada um. Mais especificamente, vimos que conseguimos maximizar as chances de vitória se, para a seleção de nós, utilizarmos uma escolha mais "inteligente" como, por exemplo, uma escolha *Greedy*, e se limitarmos o fator de ramificação, em particular com o valor 10, pois apresentou os melhores resultados. Ao explorar o número máximo de nós possível, dentro das limitações de tempo que forem estipuladas, também temos oportunidades de maximizar as vitórias, tanto para dois como para três jogadores.

Conteúdos

Lista de figuras

Lista de tabelas

Lista de algoritmos

Capítulo

1

Introdução

Na área de inteligência artificial (IA) a motivação é sempre conseguir por os computadores a fazer tarefas que só os humanos conseguem realizar ou até mesmo tentar superar os resultados alcançados por eles. Nesta área, os jogos sempre tiveram grande interesse pois, mesmo conseguindo por um computador a jogar, conseguir derrotar um humano experiente sempre foi uma tarefa difícil. A primeira vez que um programa conseguiu derrotar um jogador campeão mundial de xadrez foi em 1997, quando o sistema *Deep Blue* derrotou Garry Kasparov [4]. Este sistema deve a sua vitória à combinação de um processador específico, pesquisa de possíveis jogadas em paralelo, base de dados de jogadas iniciais e finais e funções de avaliação dinâmicas. Os progressos conseguidos com este programa inspiraram o estudo de algoritmos para outros jogos e mesmo a criação de um novo jogo, mais difícil para um computador jogar, o arimaa [1].

No universo dos jogos, os de tabuleiro são os que mais se destacam, pois normalmente conseguimos representar com facilidade o tabuleiro do jogo e os movimentos possíveis são limitados e seguem uma pequena lista de regras bem definidas, em contraste a jogos onde existe um número quase ilimitado de possíveis jogadas como jogos em tempo real. O xadrez e o jogo Go estão particularmente bem estudados, e existem algoritmos desenhados e ajustados especificamente para esses jogos, conseguindo competir com os melhores jogadores do mundo [2]. Estes são jogos de apenas dois jogadores. Em geral, os algoritmos para dois jogadores estão bem estudados, e têm comportamentos bastantes previsíveis. Contudo, no mundo real, os problemas geralmente envolvem mais de dois agentes que podem tanto agir de forma cooperativa quanto competitiva. O comportamento com mais de dois jogadores não é tão previsível, pois com apenas dois jogadores queremos maximizar o nosso ganho e minimizar o do adversário, e o adversário mais forte vai geralmente vencer. Em jogos com mais que dois jogadores, esta situação pode não acontecer. O que levanta questões de como se adaptam os algoritmos de jogos a jogos com dife-

rentes número de jogadores, desta maneira jogos que possam ser jogados por diferentes número de jogadores levantaram a nossa curiosidade. O *Chinese Checkers* (CC) é um jogo de tabuleiro que apresenta a peculiaridade de poder ser jogado por 2, 3, 4 ou 6 jogadores a competir para tentar vencer o jogo. Existem implementações para este jogo, contudo o estudo mais detalhado neste jogo foi realizado num tabuleiro de dimensão reduzida [13], deixando ainda espaço para o estudo de algoritmos para jogos, na versão de tamanho normal do tabuleiro. Normalmente, os algoritmos utilizados para encontrar respostas para este tipo de problema [10] costumam ser uma versão do *Min-Max* ou *Alpha Beta*. Este tipo de algoritmo tenta encontrar uma resposta que maximize a chance de vitória, criando uma árvore de jogo com todas as possíveis jogadas, e a partir daí atribuindo uma utilidade aos nós, escolhendo por fim o nó que tiver maior utilidade. Estes algoritmos são utilizados para jogos de dois jogadores com a característica "soma zero". Os algoritmos mais utilizados [15] para jogos com mais de dois jogadores são o Max^n e o *Paranoid*, o Max^n tem um funcionamento idêntico ao Min-Max mas em vez de retornar um valor de utilidade para um determinado nó, retorna um vetor de tamanho n para n jogadores com o valor de utilidade de cada um dos jogadores. O *Paranoid* tenta reduzir um jogo com mais de dois jogadores num jogo de dois jogadores. O *Monte-Carlo Tree Search* (MCTS) [3] é um algoritmo de pesquisa em árvores, que tem ganho recente popularidade nos últimos anos graças aos resultados que tem conseguido no jogo de Go, culminando com o *AlphaGo* [12], tornando-se no primeiro programa de computador capaz de derrotar um jogador profissional, no tabuleiro de tamanho normal do jogo de Go. Os excelentes resultados que são conseguidos no jogo de Go pelo MCTS criam um interesse para tentar perceber como este algoritmo se vai comportar em diferentes jogos, especialmente em jogos com mais de dois jogadores. O MCTS [3] é um algoritmo que, na sua versão mais genérica, é simples e capaz de retornar respostas satisfatórias. Contudo, assim que é adaptado ao problema, apresenta resultados excelentes. Apesar de já existir uma implementação do MCTS para o CC, existe espaço para continuar a estudar, já que foi numa versão com um tabuleiro de dimensão reduzida e avaliava só o comportamento de 3 jogadores, tornando interessante um estudo no tabuleiro de tamanho normal, tal como uma comparação do funcionamento do algoritmo com dois e três jogadores.

Neste trabalho implementamos o jogo de CC com versão de tamanho normal do tabuleiro. Para jogar o jogo foi implementada uma versão do MCTS, que jogou o CC na versão de dois jogadores assim como a de três jogadores. Neste trabalho foi estudado como alterando diferentes parâmetros do algoritmo, tais como: o número de nós explorados, o fator de ramificação e a escolha dos nós adicionados à árvore afetam a qualidade do algoritmo em diferentes cenários do jogo. Resultando num estudo detalhado do MCTS, que demonstra o comportamento do algoritmo quando alteramos os diferentes parâmetros, servindo como guia para futuras implementações, ajudando a implementar um algoritmo adaptado ao cenário de jogo existente.

Este trabalho encontra-se organizado da seguinte forma:

No Capítulo 2 definimos o conceito de jogo e discutimos sobre suas principais componentes. É explicado com detalhe o jogo de CC, definindo as regras do jogo assim como, como são distribuídos os jogadores pelo tabuleiro. Apresentamos os principais conceitos na área de algoritmos

para jogos e discutimos sobre os algoritmos mais utilizados para a implementação de jogos com dois ou mais jogadores. No Capítulo 3 discutimos alguns dos trabalhos mais relevantes, relacionados com o jogo de CC. No Capítulo 4 apresentamos a nossa implementação do tabuleiro de jogo, e mostramos a nossa implementação do MCTS, discutindo as opções tomadas assim como limitações iniciais da implementação. No Capítulo 5 apresentamos a metodologia experimental, neste capitulo é explicado que testes foram feitos com o MCTS, como foram feitos esses testes assim como o porque de os fazer. No Capítulo 6 apresentamos os resultados dos testes, que estão divididos pelo número de jogadores, e separados pelo parâmetro que está sob estudo. e discutimos os resultados. Finalmente, no Capítulo 7, retiramos as nossas conclusões assim como apresentamos ideias para trabalhos futuros.

Capítulo

2

Conceitos Fundamentais

Neste capítulo vamos rever conceitos fundamentais para compreender o trabalho que foi realizado.

2.1 Chinese checkers

Tal como apresentado na introdução, o CC é um jogo que pode ser jogado com 2 a 6 jogadores, onde cada um tem 10 peças que não são possíveis de remover do tabuleiro de jogo. O objetivo é conseguir chegar com as 10 peças às posições finais antes que o(s) adversário(s). Para conseguir perceber melhor as regras de jogo convém analisar primeiro o tabuleiro.

O tabuleiro do CC é uma estrela hexagonal com 121 espaços jogáveis, onde cada canto do tabuleiro tem 10 espaços que servem como as posições iniciais e finais dos jogadores. A posição final é sempre o canto oposto à posição inicial, ou seja, para a Figura 2.1, a posição final do jogador de peças vermelhas é o canto com as peças verdes.

As peças podem ser movidas para qualquer posição livre adjacente (movimentos simples). Por exemplo, na Figura 2.2 podemos ver que a peça verde em destaque, no topo da figura, apenas pode ser movida para a posição apontada pela seta. As peças também podem saltar por cima de peças adjacentes (movimentos com saltos) desde que caiam em posições livres do tabuleiro. Estes saltos podem ser concatenados se depois de saltarem tiverem outra peça adjacente. Por exemplo, na Figura 2.2 podemos ver que a peça vermelha em destaque, na parte inferior da figura, pode concatenar 3 saltos. Não é necessário concatenar os saltos, contudo é benéfico, pois conseguimos percorrer uma maior distância em uma única jogada e, desta forma, chegar mais rápido à posição final.

Dependendo do número de jogadores, a configuração inicial do tabuleiro vai mudar ligeiramente. Para dois jogadores estes são distribuídos em cantos opostos. Por exemplo, na Figura

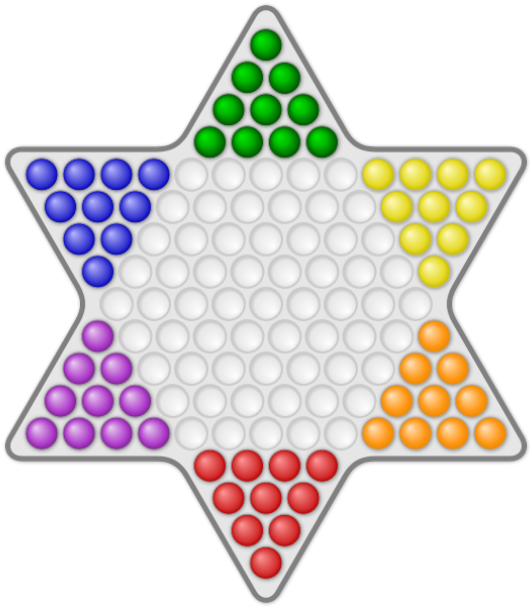


Figura 2.1: Ilustração do Tabuleiro de Jogo do *Chinese Checkers* e as posições iniciais (retirado de: https://commons.wikimedia.org/wiki/File:Chinese_checkers_start_positions.svg)

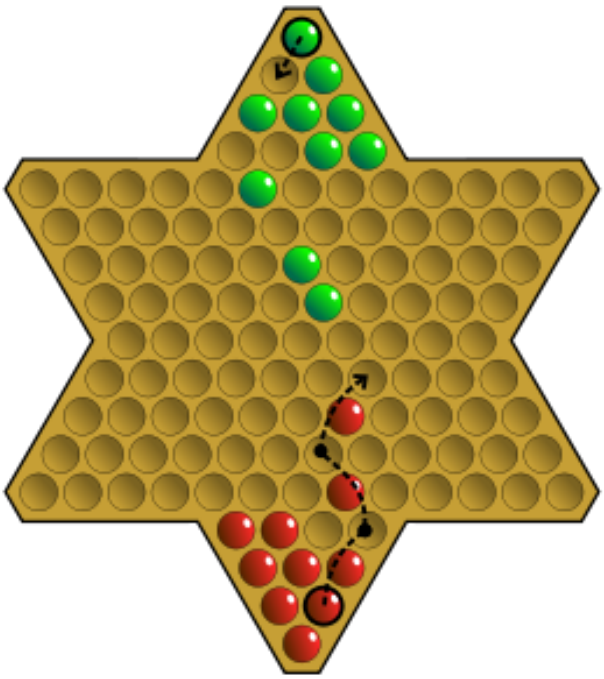


Figura 2.2: Demonstração das possíveis jogadas no jogo do *Chinese Checkers* (retirado de: https://en.wikipedia.org/wiki/Chinese_checkers)

2.1, os dois jogadores poderiam ser os jogadores com peças verdes e vermelhas. Com três jogadores, estes são distribuídos de maneira a que a posição final de cada um seja um canto oposto desocupado. Por exemplo, na Figura 2.1 poderiam ser distribuídos pelos cantos com peças vermelhas, amarelas e azuis. Com quatro, os jogadores são distribuídos de maneira a que o canto oposto esteja ocupado por um adversário e as duas posições vazias sejam opostas. Por fim, com 6 jogadores, estes seriam distribuídos pelos seis cantos do tabuleiro. As posições

podem ser permutadas pelos diferentes cantos desde que respeitem a configuração do tabuleiro estabelecida.

Segundo Bell [5], o jogo mais curto possível entre dois jogadores, com ambos a cooperar para terminar o jogo no menor número de jogadas, tem 30 jogadas (15 para cada um). Com apenas um jogador a tentar acabar o mais rapidamente o valor é de 27 jogadas. Isto são valores teóricos, que servem para perceber a complexidade do problema, e não representam os valores esperados num jogo com jogadores competitivos.

2.2 Jogos

Um jogo pode ser formalmente definido [10] como um problema de procura com os seguintes elementos:

- S0 – Estado inicial, que define como começa um jogo
- Jogador(s) – Define que jogador vai jogar no estado s
- Jogadas(s) – lista das possíveis jogadas (também denominadas ações, filhos ou descendentes) no estado s
- Jogar(s,a) – define o estado resultante da jogada a no estado s
- Terminal(s) – verdade se s é terminal (final do jogo) ou falso se não
- Utilidade(s,p) – retorna um valor numérico para o jogador p caso o jogo termine no estado s (por exemplo 1, 0, -1 para vitória, empate e derrota, respetivamente)

O estado inicial, a função Jogadas e a função Jogar vão definir a árvore de jogo, onde os nós vão ser estados e as arestas vão ser as possíveis jogadas. É importante também realçar a ideia dos jogos com soma zero. Um jogo tem a soma zero se quando um jogador aumenta a sua utilidade, a utilidade dos outros jogadores diminui, ou seja a soma das diferenças da utilidade é zero. Para jogarmos um jogo precisamos de uma estratégia, que normalmente requer que o jogador liste todos os possíveis estados do jogo face às suas escolhas. Ainda que seja possível para jogos pequenos, para a maioria dos jogos isto não vai ser possível. Para definir uma estratégia precisamos de uma função de avaliação e um algoritmo para jogos. A função de avaliação vai atribuir um valor ou utilidade aos nós folha da árvore de jogo e o algoritmo decide como são propagados os valores pela árvore até a raiz. Enquanto a função de avaliação é dependente do jogo a qual está aplicada, o algoritmo para jogos é genérico e pode ser aplicado a qualquer domínio.

2.3 Algoritmos para jogos com dois jogadores

Os algoritmos para jogos podem facilmente ser aplicados a qualquer jogo, contudo convém perceber de que maneira cada algoritmo vai afetar as jogadas e se, na prática, é possível aplicar

estes algoritmos ao jogo que queremos. Começaremos por estudar o algoritmo para jogos padrão para dois jogadores, o *Min-Max* (MM).

2.3.1 Min-Max

O MM [10] realiza uma procura em profundidade recursiva na árvore de jogo até encontrar um estado final e retorna o seu valor. O algoritmo vai ter dois jogadores, o jogador Max e o jogador Min. A primeira jogada pertence sempre ao jogador Max, que procura maximizar o valor da sua jogada. A jogada a seguir pertence ao jogador Min. A cada nível, sob o ponto de vista do jogador Max que começa o jogo, o objetivo é maximizar a sua jogada e minimizar a jogada do adversário. O valor de cada estado não final é igual ao do sucessor com maior valor para Max e ao do sucessor com menor valor para Min. Estes valores são calculados apenas após a exploração de todo um ramo da pesquisa através de propagação dos valores terminais aos seus nós pais. O valor dos estados terminais é dado pela função utilidade. O algoritmo MM, para uma profundidade máxima d e para um fator de ramificação b, vai ter uma complexidade temporal $\mathcal{O}(b^d)$ e uma complexidade espacial $\mathcal{O}(b \times d)$.

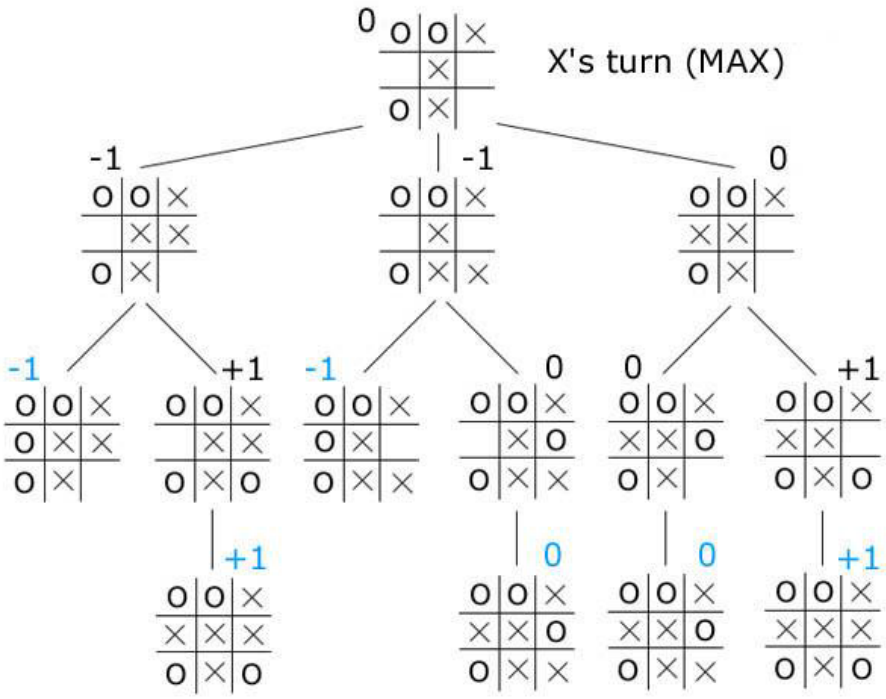


Figura 2.3: Ilustração do algoritmo *Min Max* no jogo do galo (retirado de: https://www.clear.rice.edu/comp405/s15/lectures/gametheory/)

Na Figura 2.3 podemos ver como funciona o algoritmo, que tem que escolher a próxima jogada no jogador 'x' no jogo do galo. Para isto, vai criar a árvore de todas as jogadas possíveis, nos estados finais vai-lhes atribuir a utilidade 1 quando ganha, a utilidade -1 quando perde e a utilidade 0 quando empata. Depois vai recursivamente propagar estes valores pela árvore escolhendo o maior valor quando joga o jogador 'x', tentando maximizar a sua jogada, e o menor valor possível quando joga o jogador 'o'. No final, o nó raiz escolhe uma das jogadas do

nível imediatamente superior da árvore (descendentes diretos da raiz) que apresenta a maior utilidade como a sua próxima jogada, neste caso o nó mais a direita com o valor 0.

A versão original do MM explora a árvore de jogo completa para decidir uma jogada, isto é, analisa todas as jogadas possíveis e para cada uma simula todos os possíveis resultados de jogo. Por razões de tempo e memoria este algoritmo não é mais eficiente para retornar uma resposta, apesar de que irá sempre retornar a melhor resposta se tiver tempo e memória suficiente para explorar a árvore completa. Um algoritmo que tenta minimizar estas desvantagens é o *Alpha Beta* (AB).

2.3.2 Corte Alpha Beta

O algoritmo AB[10] é uma variação do MM que tenta diminuir o número de nós visitados sem alterar o resultado final. Para isso, o algoritmo tem em conta os valores máximo e mínimo encontrados e não vai visitar nós se nas suas sub-árvores sabe que não vai encontrar um valor melhor.

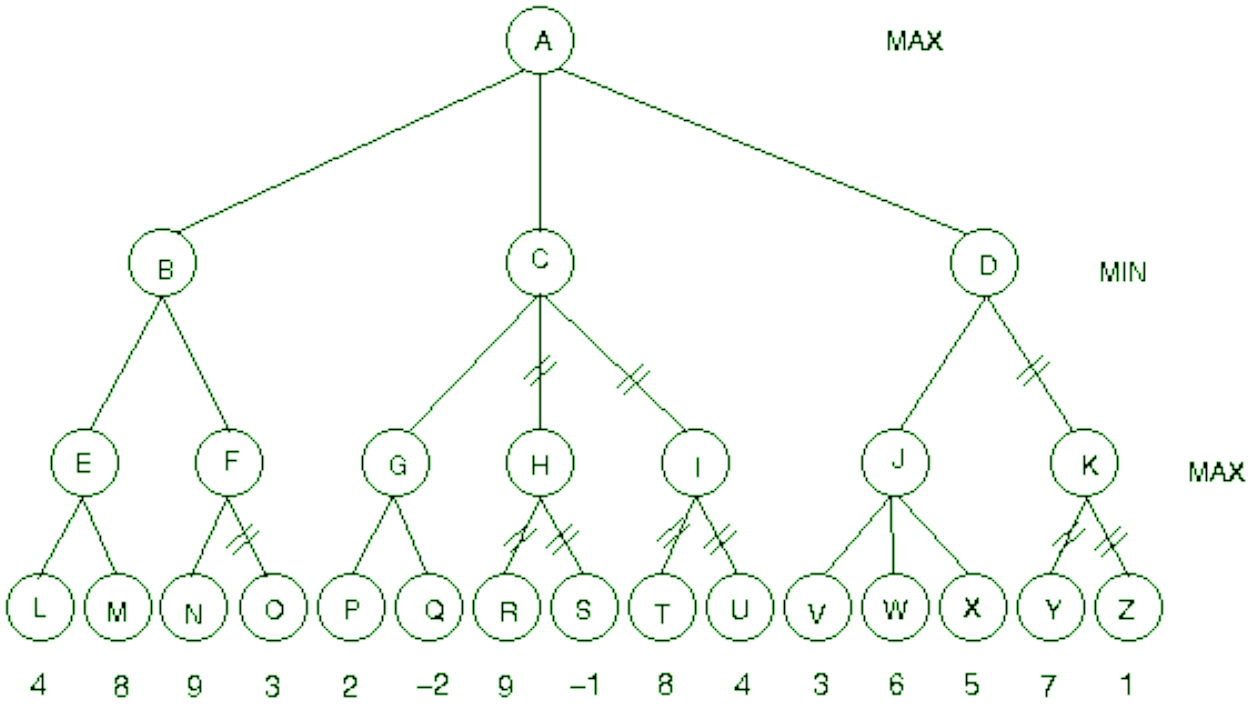


Figura 2.4: Ilustração do algoritmo Corte *Alpha Beta* (retirado de: http://pages.cs.wisc.edu/~dyer/cs540/notes/)

Como podemos ver na Figura 2.4, os caminhos traçados não vão ser explorados pelo AB contudo seriam explorados pelo MM. O algoritmo AB, para uma profundidade máxima d e para um fator de ramificação b, vai ter uma complexidade temporal para o pior caso de $\mathcal{O}(b^d)$ e espacial $\mathcal{O}(b \times d)$. Contudo a eficiência do AB depende da ordenação dos nós na árvore de jogo, ou seja, se o algoritmo explorasse primeiro o melhor caminho, a complexidade temporal seria $\mathcal{O}(b^{\frac{d}{2}})$ para encontrar a solução ótima enquanto o MM tem complexidade $\mathcal{O}(b^d)$, mas como a

ordem dos nós explorados é aleatória a complexidade média é $\mathcal{O}(b^{\frac{3d}{4}})$.

2.4 Algoritmos para múltiplos jogadores

2.4.1 Max^n

Alguns dos conceitos de jogos de dois jogadores podem ser aplicados a jogos com múltiplos jogadores (mais de dois jogadores), e mesmo havendo já alguns estudos sobre pesquisas com múltiplos jogadores, ainda não existem programas capazes de derrotar bons jogadores humanos neste tipo de jogos [15]. Um dos algoritmos mais conhecidos para este tipo de problemas é o Max^n [7]. O Max^n é um algoritmo parecido com o MM que recursivamente vai procurar a melhor jogada. A função utilidade retorna um vetor de tamanho n, com a utilidade para cada jogador, e é escolhida a jogada que tiver o maior valor para o jogador que tem que jogar nessa iteração, ou seja, cada jogador tenta maximizar o seu resultado cada vez que joga.

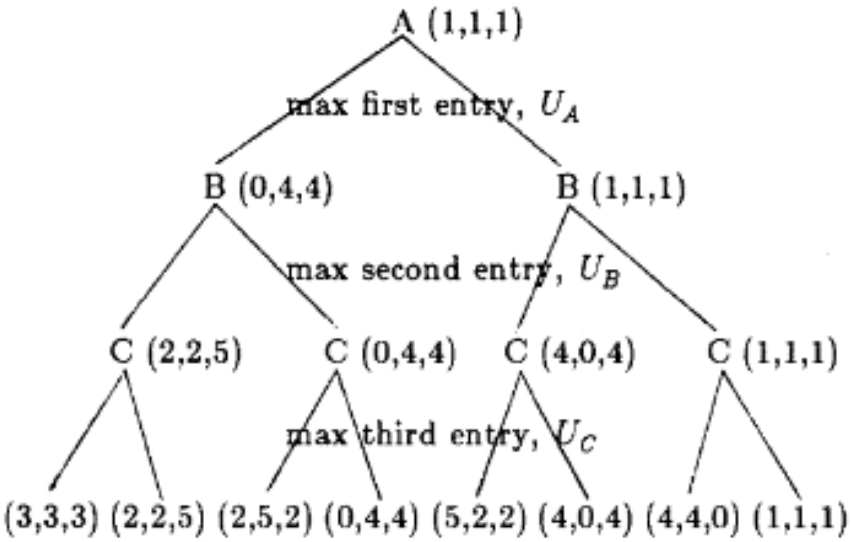


Figura 2.5: Ilustração do algoritmo Max^n [7]

Na Figura 2.5 podemos ver um exemplo do funcionamento do Max^n, nos nós folha a função utilidade retorna o vetor com o valores para os jogadores A, B e C nesta ordem, e vai sempre selecionar o nó que maximiza o valor para o jogador que tem que jogar. Por exemplo, na sub-árvore mais a esquerda, o jogador C tem duas possíveis jogadas com as utilidades (3,3,3) e (2,2,5), como vai selecionar a ação que maximiza a sua utilidade, vai escolher a segunda jogada com utilidade (2,2,5). A versão original do algoritmo não aplica qualquer corte na árvore de jogo, contudo existem variações do algoritmo [15] que tentam podar a árvore para tentar melhorar o seu desempenho.

Duas versões do Max^n que procuram podar a árvore de jogo de maneira a reduzir o número

de nós irrelevantes explorados são o *Speculative* Max^n e o *Approximate deep* Max^n. Estes algoritmos partem do teorema [15] que numa sequência de jogadores únicos, se a soma do mínimo valor garantido de cada jogador é igual ou superior ao valor máximo do jogo, a utilidade dos nós da jogada do último jogador da sequência não vai chegar a raiz. Ou seja, se tivermos três jogadores, se o valor mínimo de utilidade calculada pelo Max^n de cada um corresponder ao valor máximo de utilidade possível no jogo, a utilidade dos nós gerados de uma jogada do terceiro jogador nunca poderá ser a utilidade final encontrada pelo Max^n.

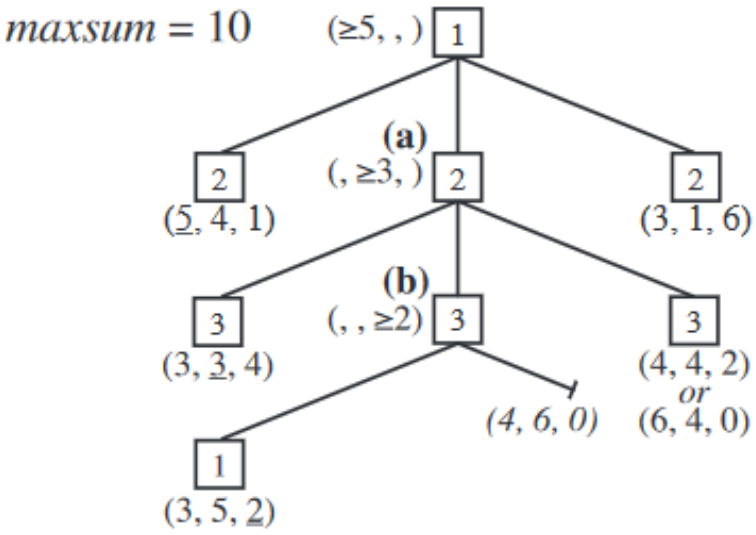


Figura 2.6: Ilustração do algoritmo *Speculative* Max^n [15]

O *Speculative* Max^n quando encontra um caminho da árvore em que a soma dos valores mínimos garantidos a cada jogador é igual ou superior ao valor máximo do jogo, poda a sub-árvore descendente. Porém quando os valores mínimos do jogador atual e do jogador na raiz são alterados, o algoritmo tem que voltar a explorar os caminhos podados. A ordem dos nós influencia quantos caminhos cortados vão voltar a ser explorados, ou seja, com uma ordem ótima nenhum caminho cortado teria que voltar a ser explorado. Na Figura 2.6 podemos ver um exemplo do algoritmo *Speculative* Max^n. Na raiz da árvore, o primeiro jogador tem 5 valores garantidos. No nó (a) o segundo jogador tem 3 valores garantidos e no nó (b) o terceiro jogador tem 2 valores garantidos. Porque a soma dos valores garantidos $5 + 3 + 2 = 10$, que é o valor da soma máxima do jogo, podemos podar os restantes filhos do nó (b). O último nó filho de (a) tem o valor (4,4,2), por isso o primeiro jogador não vai escolher o nó (a), porque tem um melhor valor garantido por outro nó. Mas, se a utilidade do último nó for (6,4,0), a utilidade do nó (a) vai ser (6,4,0), e com esta utilidade, o valor garantido do primeiro jogador e do segundo jogador é alterado, a árvore cortada do nó (b) terá que ser revista, porque pode alterar a utilidade do nó (a) e a utilidade final.

O *Approximate deep* Max^n tal como o *Speculative* Max^n, usando os valores mínimos garantidos de cada jogador, poda a sub-árvore descendente sempre que a soma destes valores for

igual ou maior que o valor máximo do jogo. A diferença entre os algoritmos é que quando no *Approximate deep* Max^n os valores mínimos garantidos são alterados ele não volta a explorar as sub-árvores podadas, tornando as podas permanentes. A vantagem deste algoritmo é que desta maneira, em média, vai conseguir explorar a árvore com mais profundidade se tiver o mesmo limite de exploração que o *Speculative* Max^n. Contudo, como podem ser podadas sub-árvores onde a melhor utilidade podia ser encontrada, a resposta pode não ser a mais precisa.

2.4.2 Paranoid

A falta de cortes na árvore do Max^n torna-se um problema para árvores muito grandes. Mesmo aplicando variações que podam a árvore, quantos mais jogadores existirem menor vai ser a quantidade de nós podados. Para tentar colmatar esta limitação o algoritmo *Paranoid* [15] reduz um jogo de vários jogadores a um jogo de dois jogadores, assumindo que todos os adversários formam uma aliança contra o jogador da vez. Desta maneira, podemos aplicar as regras e os cortes do AB.

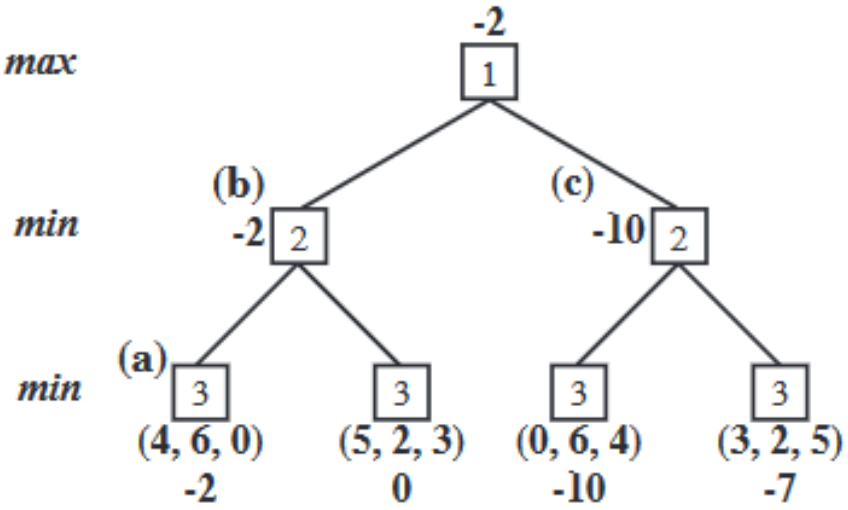


Figura 2.7: Ilustração do algoritmo *Paranoid* [15]

Num jogo com múltiplos jogadores, as diferentes utilidades de cada jogador são representadas num tuplo onde para o jogador i a entrada número i no tuplo corresponde à sua utilidade. Como podemos ver na Figura 2.7, para converter o problema de 3 jogadores para dois jogadores precisamos alterar o conjunto das três utilidades para apenas um valor, que será a diferença entre o valor da utilidade do primeiro jogador e a soma dos valores de utilidade dos restantes jogadores. Por exemplo, no nó (a) o primeiro jogador tem utilidade 4 pontos, o segundo jogador tem utilidade 6 pontos e o terceiro jogador tem utilidade 0 pontos, assim sendo o valor calculado pelo *Paranoid* será $4 - 6 - 0 = -2$. O jogo será jogado como o MM onde no nó (b) o segundo jogador vai escolher o valor mínimo calculado pelo *Paranoid*, neste caso -2, tal como no nó (c) onde vai escolher o valor -10. O primeiro jogador vai tentar maximizar a sua jogada e por

isso escolhe o maior valor que estiver disponível que neste caso é -2. Como é de suspeitar o *Paranoid* poderá tomar más decisões devido a sua natureza paranoica de ver a árvore de jogo.

2.5 Monte Carlo Tree Search

O *Monte Carlo Tree Search* (MCTS) é um algoritmo de pesquisa em árvores que resolve o problema de pesquisa com adversários, e tem ganho popularidade na IA graças aos excelentes resultados que tem demonstrado na área dos jogos [3], em particular com o jogo Go. O MCTS pode ficar a correr indefinidamente até encontrar uma resposta ótima, ou podemos limitar o tempo ou o número de iterações do algoritmo, fazendo com que este retorne sempre uma resposta. Contudo, quanto mais tempo dermos ao algoritmo, maior a probabilidade da resposta ser ótima, ou seja, se o tempo de execução tender para o infinito, o algoritmo irá retornar a melhor solução possível.

O algoritmo é simples, vai construindo a árvore de maneira assimétrica. Cada iteração do algoritmo vai realizar quatro passos:

- Seleção – a partir do estado inicial S0, usando uma política de seleção, o algoritmo vai percorrer a árvore até encontrar um nó expansível. Um nó é expansível se não for terminal e não tiver filhos já visitados.
- Expansão – depois de encontrado um nó expansível, o algoritmo vai gerar um ou mais filhos, e estes vão ser adicionados à árvore de jogo.
- Simulação – utilizando uma política de simulação, vão ser simulados jogos a partir dos nós adicionados durante a expansão, procurando desta maneira encontrar uma utilidade para cada nó.
- Propagação – os resultados das simulações vão ser propagados pela árvore, atualizando todos os nós no caminho até a raiz.

A execução destes passos precisa de duas políticas:

- Política de árvore – determina qual vai ser o nó escolhido e expandido, nos passos de seleção e expansão
- Política de simulação – escolhe um caminho a partir de um estado não-terminal para chegar a um resultado, permitindo fazer a simulação

A política mais simples para encontrar um nó expansível e correr a simulação é escolher aleatoriamente o nó e o caminho. Apesar de o MCTS conseguir alcançar bons resultados desta maneira, para tirar maior proveito do algoritmo é preciso ajustar estas políticas [3] para o problema em questão. Durante a expansão o número de nós adicionados à árvore não está estabelecido, devendo também ser adaptado ao problema.

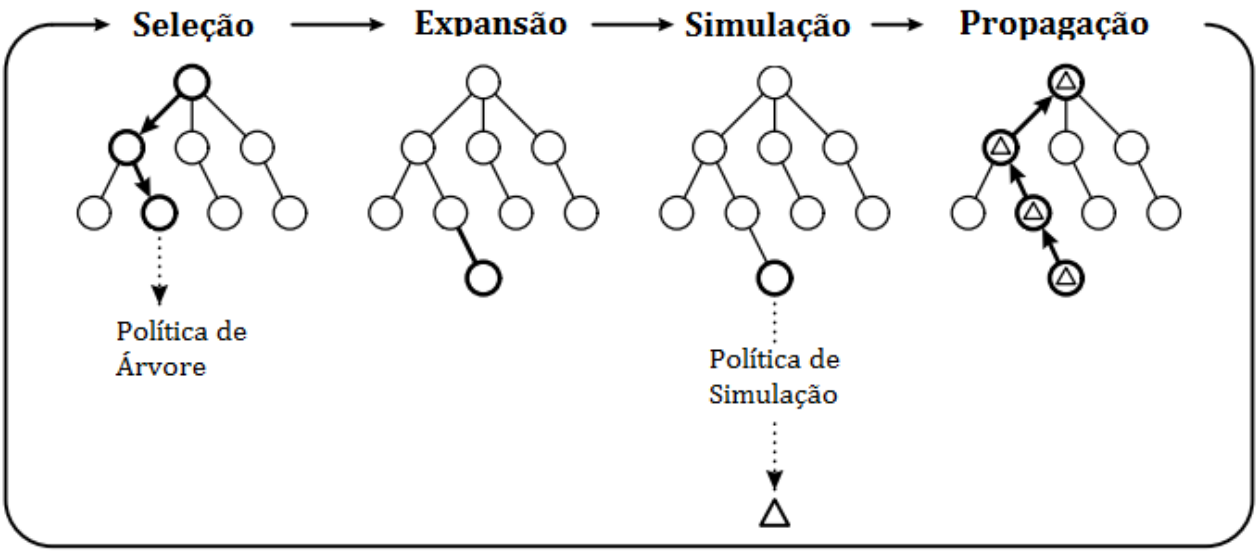


Figura 2.8: Ilustração do algoritmo *Monte Carlo Tree Search*
Ilustração do algoritmo *Monte Carlo Tree Search* (adaptado de: http://stackoverflow.com/questions/23803186/monte-carlo-tree-search-implementation-for-tic-tac-toe)

Na Figura 2.8 podemos ver o funcionamento do algoritmo. No primeiro passo ele vai utilizar a política de seleção para encontrar um nó expansível, em seguida o nó vai ser expandido e vai ser adicionado um filho à árvore. A seguir, o algoritmo vai encontrar uma utilidade para o nó adicionado à árvore usando a política de simulação. Por fim, depois de encontrada a utilidade, esta vai ser propagada pelo caminho até a raiz da árvore.

Algoritmo 1: Algoritmo genérico MCTS

```
function MCTS(S0)
    Cria no-raiz V0 a partir do estado S0
    while Dentro dos limites computacionais do
        V1 ⇐ Politica_de_Arvore(V0)
        Δ ⇐ Politica_de_Simulação(S(V1))
        Propagar(Δ, V1)
    return Melhor_Filho(V0)
end function
```

No Algoritmo 1 podemos ver como podemos construir a implementação do MCTS. O algoritmo inicializa o nó raiz $V0$ a partir do estado $S0$, o estado $S0$ é o estado atual para o qual pretendemos decidir uma jogada. Enquanto o algoritmo estiver dentro do limite computacional estabelecido, vai utilizar a política de árvore que, a partir do nó raiz, vai encontrar um nó expansível $V1$ e vai expandi-lo $S(V1)$. De seguida vai utilizar a política de simulação para encontrar uma utilidade Δ para os nós resultantes da expansão e depois vai propagar a utilidade encontrada pela árvore até chegar à raiz. Quando chega ao limite computacional, o algoritmo retorna o filho que apresenta a melhor utilidade.

O objetivo do MCTS é de tentar aproximar o valor das possíveis ações de um estado ao seu valor real. Isto vai sendo conseguido construindo uma árvore parcial (Figura 2.8). A árvore

construída depende muito de como os nós são selecionados. A versão que deu popularidade ao MCTS foi *Upper Confidence Bounds for Trees* (UCT). O sucesso desta versão do algoritmo deve-se principalmente à política de árvore UCB1 proposta por Kocsis e Szepesvári [6], que trata a escolha de um nó como um problema *Multi-armed Bandit*. Os problemas *Bandit* são problemas em que é preciso fazer uma escolha entre várias ações possíveis com o objetivo de maximizar a recompensa acumulada através de escolhas consecutivas da ação ótima. Com o UCB1, o valor de um nó corresponde a um valor aproximado da recompensa, calculado a partir de simulações pelo MCTS. Estas recompensas são variáveis aleatórias com distribuições desconhecidas. Assim sendo, a política de árvore reduz-se a um problema que vai selecionar um nó j que maximize a função:

$$UCT = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}} \tag{2.1}$$

Onde n é o número de vezes que o nó atual foi visitado, n_j é o número de vezes que o nó filho j foi visitado e $C_p > 0$ é uma constante. Se houver mais que um nó filho com o valor máximo o desempate resume-se à escolha aleatória de um deles. O valor $\overline{X}_j$ é a recompensa média do nó filho j e vai estar compreendido entre $[0, 1]$. O princípio está em que se $n_j = 0$ o valor de UCT vai tender para ∞, logo os nós que não tiverem sido visitados vão ter os maiores valores possíveis, assegurando que todos os nós são considerados pelo menos uma vez antes de expandir filhos com mais profundidade na árvore. Esta política resulta numa espécie de *iterated local search*. Cada vez que um nó é explorado o seu valor vai diminuir. Por outro lado, cada vez que um nó não é escolhido a probabilidade de ser escolhido aumenta. É essencial não ter nós com uma probabilidade zero de ser escolhidos, devido a aleatoriedade das simulações. A constante C_p pode ser ajustada para aumentar ou diminuir a probabilidade de escolher nós menos visitados. O valor $C_p = \frac{1}{\sqrt{2}}$ foi provado por Kocsis e Szepesvári de satisfazer a desigualdade de Hoeffding para valores entre $[0, 1]$.

As contribuições de Kocsis e Szepesvári provaram que a probabilidade de escolher uma ação sub-ótima converge para zero numa relação polinomial quando o número nós explorados tende para o infinito. Esta prova implica que, dado tempo suficiente, o UCT permite que o MCTS convirja para a árvore do MM e, sendo assim, retorne uma resposta ótima.

Capítulo

3

Trabalhos Relacionados

O jogo de CC não tem tido o mesmo foco do xadrez ou do jogo de Go no que toca a IA, contudo o trabalho de Sturtevant destaca-se por ter estudado como diferentes algoritmos se comportam numa versão de 3 jogadores do CC. Os trabalhos relacionados nesta secção foram selecionados com base nos resultados dos motores de pesquisa do Google Scholar, usando as palavras chaves: "multiplayer games", "multiplayer algorithms", "chinese checkers". Desses resultados os trabalhos que mais se relacionaram com o nosso trabalho são: "Multi-Player Games: Algorithms and Approaches" [15] e "An Analysis of MCTS-UCT in Multi-Player Games" [13].

3.1 Algoritmos e Abordagens para jogos com múltiplos jogadores

Neste trabalho [15], os autores executaram os testes com o jogo de CC em duas versões do jogo, inicialmente uma versão menor do tabuleiro, onde cada jogador teria 6 peças em vez de 10. Depois foram testados no tabuleiro normal de CC. Na versão de 6 peças cada jogador tem em média 25 ações possíveis, nos piores casos pode chegar a ter mais de 50 ações possíveis. O jogo normal pode chegar a ter mais de 100 possíveis ações por jogada. Os algoritmos estudados neste trabalho foram o Max^n e o *Paranoid*. No caso do Max^n foram implementadas varias versões, incluindo a versão padrão do algoritmo e diferentes versões com métodos de corte, *Speculative Pruning* e *Approximate Deep* , sendo que as versões com corte foram testadas apenas na versão normal do tabuleiro.

3.1.1 Tabuleiro Menor

Nos testes foram jogados jogos entre o Max^n e o *Paranoid*, para evitar os jogadores repetirem sempre as mesmas jogadas. Os empates de ações eram resolvidos escolhendo aleatoriamente uma das ações.

		Paranoid	Max^n
3-Jogadores 250k nós	Vitorias	60,6%	39,4%
	Jogadas	3,52	4,92
	Profundidade	4,9	3,1
4-Jogadores 250k nós	Vitorias	59,3%	40,7%
	Jogadas	4,23	4,73
	Profundidade	4,0	3,2
6- Jogadores 250k nós	Vitorias	58,2%	41,8%
	Jogadas	4,93	5,49
	Profundidade	4,6	3,85

Tabela 3.1: Resultados do *Paranoid* e Max^n para 3, 4 e 6 jogadores

Os resultados dos testes encontram-se na tabela 3.1. Para três jogadores existem 6 configurações diferentes e foram jogados 100 jogos em cada configuração num total de 600 jogos. O algoritmo *Paranoid* ganhou $60,6\%$ dos jogos e o Max^n ganhou $39,4\%$. A linha das jogadas corresponde à média das jogadas que faltavam para o algoritmo que perdeu terminar o jogo. O jogador *Paranoid* estava em média 1,4 jogadas à frente do jogador Max^n. A linha da profundidade corresponde à média da profundidade que o algoritmo conseguiu chegar expandindo os 250k nós, o *Paranoid* conseguia ver jogadas com cerca de 50% mais profundidade que o Max^n.

Para quatro jogadores existem 14 configurações diferentes. Foram jogados 50 jogos em cada configuração num total de 700 jogos. O algoritmo *Paranoid* ganhou $59,3\%$ dos jogos enquanto o Max^n ganhou $40,7\%$, tendo resultados bastante próximos a versão de 3 jogadores. O *Paranoid* quando perdeu ficou em média a 4,23 jogadas de acabar e o Max^n ficou a 4,73 de terminar, não havendo uma vantagem muito significativa para o *Paranoid*. Finalmente o *Paranoid* conseguia, em média, alcançar uma profundidade equivalente a 4 jogadas, enquanto o Max^n ficava-se por 3,2 jogadas, fazendo com que o *Paranoid* conseguisse ver cerca de $33,3\%$ mais fundo que o Max^n.

Para seis jogadores existem 64 configurações diferentes. Foram jogados 20 jogos para cada um, num total de 1280 jogos. O jogador *Paranoid* ganhou $58,2\%$ dos jogos enquanto o Max^n ganhou $41,8\%$. Em média, o *Paranoid*, quando perdia, ficava a 4,93 jogadas de terminar, enquanto o Max^n, quando perdia, ficava a 5,49 jogadas de terminar. O algoritmo *Paranoid* conseguia ver jogadas com cerca de 20% mais profundidade que o Max^n que alcançou em média 3,85 jogadas de profundidade enquanto o *Paranoid* chegou em média a 4,6 jogadas de profundidade.

Foram feitos outros testes a estes algoritmos, mas, desta vez, para apenas 3 jogadores. Jogando 100 jogos em cada possível configuração num total de 600 jogos, limitando o fator de ramificação de cada algoritmo de maneira a que apenas as 6 melhores ações eram consideradas

		Paranoid	Max^n
250k nós, fator de ramificação limitado	Vitorias	71,4%	28,6%
	Jogadas	2,47	4,4
	Profundidade	8,2	5,8
profundidade limitada	Vitorias	56,5%	43,5%
	Jogadas	3,81	4,24

Tabela 3.2: Resultados do *Paranoid* e Max^n com fator de ramificação ou profundidade limitada

em cada ramo. Os resultados destas experiências podem ser vistas na tabela 3.2. Com essas condições o *Paranoid* superou os resultados anteriores vencendo $71,4\%$ dos jogos, conseguindo explorar em média 8,2 jogadas de profundidade em comparação com 5,8 jogadas de profundidade do Max^n e ficando a 2,47 jogadas de terminar quando perdia em contraste com as 4,4 jogadas para o Max^n. Podemos também ver os resultados dos algoritmos jogando, mas com a profundidade limitada a 4 jogadas, independente do número de nós expandidos. Nestas condições, o *Paranoid* voltou a superar o Max^n, mas agora por uma margem menor, ganhando $56,5\%$ dos jogos e ficando a 4,24 jogadas de terminar quando perdia contra 3,81 jogadas para o Max^n. Com estes resultados, consegue-se concluir que o *Paranoid* tem melhores resultados no jogo de CC conseguindo procurar em mais profundidade e, em geral, retornar melhores jogadas para a mesma profundidade.

3.1.2 Tabuleiro Normal

No tabuleiro normal de CC foram feitos teste similares, mas como o fator de ramificação é bastante alto, em cada ramo são consideradas apenas as melhores 10 ações. Nestes testes jogaram duas versões do Max^n que tentam podar a árvore, *Speculative* Max^n e *Approximate deep* Max^n contra o *Paranoid*. Primeiro jogaram o *Speculative* Max^n contra o *Paranoid*, com o limite de 500k nós expandidos, seguido de *Approximate deep* Max^n contra *Paranoid*, também com o limite de 500k nós expandidos. Os resultados podem ser vistos na Tabela 3.3. No primeiro caso, o *Paranoid* vence 65% dos jogos contra o *Speculative* Max^n, mas apenas 52% contra o *Approximate deep* Max^n. Em média, o *Paranoid* explorou 8 jogadas de profundidade, na árvore de jogo, enquanto o *Speculative* Max^n explorou, em média, 6,24 jogadas de profundidade e o *Approximate deep* Max^n 7,56. Em geral, o *Approximate deep* Max^n teve resultados melhores em comparação com o *Speculative* Max^n contra o *Paranoid*. Isto deve-se a duas razões. Primeiro, como são escolhidas as 10 melhores ações, a probabilidade de cortar ramos bons é reduzida. Segundo, como o *Approximate deep* Max^n consegue explorar em média 7,56 jogadas de profundidade, enquanto *Speculative* jogadas Max^n explora apenas 6,24 jogadas, ao chegar a profundidade 7, na árvore de jogo, o algoritmo consegue ver o resultado de 3 jogadas próprias, sendo particularmente importante no inicio e no fim onde as peças dos jogadores não se cruzam, permitindo selecionar combinações de jogadas mais complexas que não seriam possíveis de ver explorando menos que 7 jogadas de profundidade.

	Vitorias	Jogadas	Profundidade
Speculative Max^n	35%	6,66	6,24
Paranoid	65%	3,54	8,05
Approx. Deep Max^n	48%	5,53	7,56
Paranoid	52%	4,75	8,04

Tabela 3.3: Variações do Max^n contra o *Paranoid*

3.2 MCTS-UCT em jogos com múltiplos jogadores

Desta vez os autores [13] estudaram a performance do MCTS-UCT em comparação com o Max^n e o *Paranoid*. Foram estudados vários jogos, entre eles o CC. Na avaliação feita ao jogo de CC, Sturtevant utilizou uma versão menor do tabuleiro (6 peças) para três jogadores. Para fazer a comparação direta entre cada algoritmo foram jogados 50 jogos em cada uma das 6 diferentes combinações de jogo que os algoritmos podiam assumir, num total de 300 jogos. Em cada jogo os algoritmos podiam explorar 250k nós. Foram usadas duas funções de avaliação diferentes para o Max^n e para o *Paranoid*, (dist) que tenta minimizar a distância até terminar o jogo e (diff) que tenta maximizar a diferença entre a distância do jogador até terminar o jogo com a distância dos adversários. Paras a simulações do MCTS-UCT foi utilizada uma política *Epsilon-Greedy farthest-first*, que dá uma probabilidade de 95% para jogar a peça que se encontra mais atrás no tabuleiro e 5% de hipótese de fazer uma jogada aleatória.

	MCTS-UCT	$Paranoid(dist)$	$Paranoid(diff)$	$Max^n(dist)$	$Max^n(diff)$
MCTS-UCT	-	8%	4%	3,7%	6%
$Paranoid(dist)$	92%	-	46,3%	25,0%	36,7%
$Paranoid(diff)$	96%	53,7%	-	46,3%	68,7%
$Max^n(dist)$	96,3%	75%	53,7%	-	56,3%
$Max^n(diff)$	94%	63,3%	31,3%	43,7%	-

Tabela 3.4: Resultados do MCTS-UCT, *Paranoid* e Max^n

Os resultados podem ser vistos na tabela 3.4, o MCTS-UCT ganha mais de 90% dos jogos contra os outros algoritmos. Conseguimos também ver que o Max^n tem melhores resultados usando a função (dist) e o *Paranoid* ganha mais jogos usando a função (diff).

Na tabela 3.5 é estudada a qualidade das jogadas do MCTS-UCT quando sucessivamente duplicam-se o número de nós que ele pode expandir. O MCTS-UCT precisa de um mínimo de 25k nós expandidos para conseguir ter resultados bons. Com menos expansões não consegue adquirir informação necessária para selecionar as melhores ações. Sendo que o MCTS-UCT só consegue vencer 16,7% dos jogos contra o *Paranoid* se só forem permitidas 1600 expansões.

Expansões	50k vs 25k	100k vs 50k	200k vs 100k	400k vs 200k	800k vs 400k
Vitorias	60,7%	54,3%	55,2%	60,1%	51,8%

Tabela 3.5: Qualidade das jogadas quando duplicam as expansões permitidas

Com os resultados destes trabalhos, podemos concluir que o algoritmo que teve melhores

resultados para o jogo de CC foi o MCTS-UCT. Contudo é um algoritmo que revela ainda ter muito por estudar e apresenta detalhes que podem ser afinados para melhorar ainda mais o seu desempenho.

3.3 Melhoramentos ao MCTS-UCT

Além dos trabalhos descritos nas secções anteriores, existem outros trabalhos que procuram aplicar novas heurísticas para o MCTS-UCT para tentar melhorar a qualidade de jogo do algoritmo. Roschke [9] utilizou uma base de dados que, dadas as posições das peças de um jogador, retorna o menor número de jogadas necessárias para terminar o jogo. Esta base de dados é usada como heurística nas simulações. Foram estudados dois métodos diferentes para utilizar as bases de dados. O primeiro, durante as simulações, escolhe nós com uma heurística *Epsilon-Greedy*, como a vista na secção anterior, durante um número fixo de jogadas, depois utiliza a base de dados para saber o número de jogadas necessárias para cada jogador terminar o jogo, o que terminar em menos jogadas vence a simulação. O outro método, durante as simulações usa uma heurística *Epsilon-Greedy* para jogar até as peças de cada jogador estarem separadas das peças dos adversários (final de jogo), e depois verifica qual é o jogador que consegue terminar o jogo em menos jogadas para determinar o vencedor da simulação. Ambos os métodos superam os resultados do MCTS-UCT que usa a política de simulação com apenas a heurística *Epsilon-Greedy*. A versão que utiliza a política de simulação que simula um número fixo de jogadas supera os resultados da que apenas utiliza a base de dados no final de jogo. Demonstrando que as bases de dados de jogo podem ser usadas efetivamente como heurísticas para o jogo de CC.

Capítulo

4

Implementação do MCTS-UCT para o Chinese Checkers

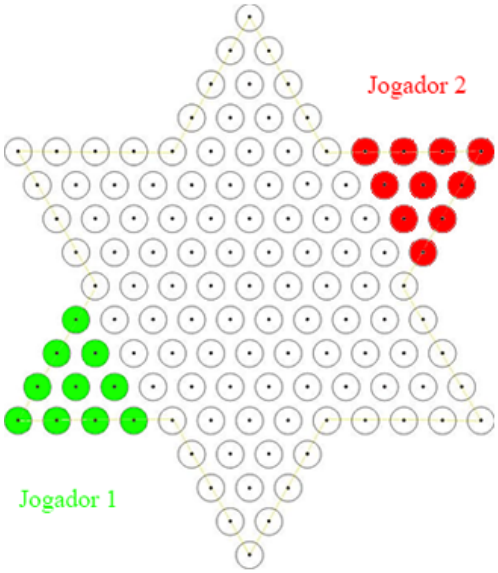
No capítulo anterior vimos que nos trabalhos relacionados tanto o *Paranoid* como o Max^n foram estudados tanto na versão reduzida como na versão de tamanho normal do tabuleiro. O MCTS-UCT, apesar de só ter sido estudado na versão reduzida do tabuleiro, é o que apresenta melhores resultados e parece ter mais potencial para melhorias. Neste trabalho focamo-nos na implementação e estudo do MCTS-UCT. Neste capítulo apresentamos a nossa implementação do algoritmo MCTS-UCT para o jogo de CC. Descrevemos também todos os detalhes necessários para implementação, como o estado de jogo, o tabuleiro, os jogadores, as possíveis jogadas e o algoritmo implementado. A implementação e os testes foram feitos para o tabuleiro de tamanho normal de CC, com 10 peças para cada jogador.

4.1 Estado de Jogo

Um estado é definido por um tabuleiro, a lista dos jogadores e uma referência para o jogador a quem pertence a próxima jogada. Cada estado também guarda uma referência para o estado anterior, para evitar que haja ciclos na árvore de jogo. Cada estado de jogo é capaz de gerar todos as possíveis ações que podem ocorrer nesse estado, de acordo com a referência para o jogador da vez. Para cada peça do jogador verifica-se os possíveis movimentos simples vistos no capítulo 2. Os movimentos que são legais são adicionados à lista de ações possíveis. Para as mesmas peças é verificado se podem realizar movimentos com saltos. Para isso é verificado se pode realizar um movimento com um único salto. Se puder, esta ação é adicionada à lista de jogadas possíveis. Com a ação resultante é verificado se a peça pode voltar a fazer outro salto. Se puder, uma nova ação é adicionada e este processo é repetido até que não haja mais

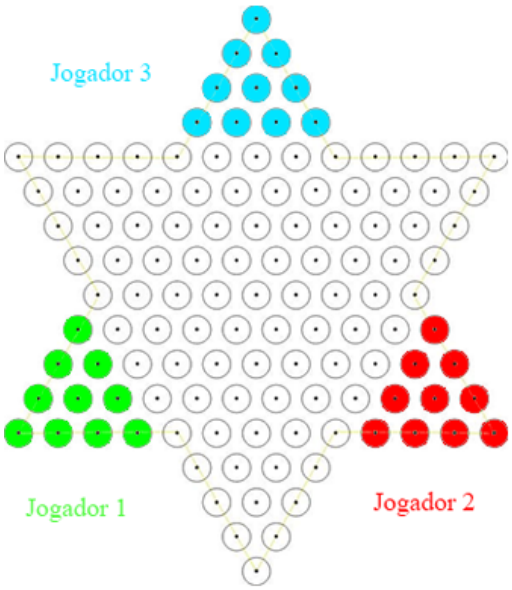
jogadas possíveis. Os movimentos são testados para todas as direções. Os movimentos em que as peças se movem para trás ou para um canto que não seja o inicial ou final do jogador são considerados ilegais nesta implementação e não são adicionados à lista das ações.

Quando um jogo é inicializado gera-se um estado inicial, onde cada jogador tem as peças nas posições iniciais. Para gerar um jogo novo é apenas necessário dizer o número de jogadores que o jogo vai ter. As posições iniciais que foram estabelecidas para esta implementação dependem apenas do número de jogadores. Para 2 jogadores o tabuleiro tem a disposição apresentada na Figura 4.1. Para 3 jogadores a disposição das peças é mostrada na Figura 4.2.




```
0
00
000
0000
0000000002222
000000000222
00000000022
0000000002
000000000
1000000000
11000000000
111000000000
1111000000000
0000
000
00
0
```

Figura 4.1: Ilustração do Tabuleiro com 2 jogadores e a representação no programa




```
3
33
333
3333
0000000000000
000000000000
00000000000
0000000000
000000000
1000000002
11000000022
111000000222
1111000002222
0000
000
00
0
```

Figura 4.2: Ilustração do Tabuleiro com 3 jogadores e a representação no programa

4.1.1 Tabuleiro

O tabuleiro de jogo é representado numa matriz 17×17. Como o tabuleiro do CC tem 121 espaços jogáveis, a matriz necessita de um conjunto de regras que define quais são os espaços que fazem parte do tabuleiro. Na Figura 4.3 podemos ver a representação do tabuleiro numa matriz 17×17. As posições com o cinzento mais escuro correspondem às posições iniciais/finais do tabuleiro. As regras para cada linha da matriz dizem em que posição começa o tabuleiro e qual é o número de casas que o tabuleiro tem nesta linha. Por exemplo, na Figura 4.3, na 4ª linha da matriz, o tabuleiro começa na 5ª coluna e tem tamanho 4.

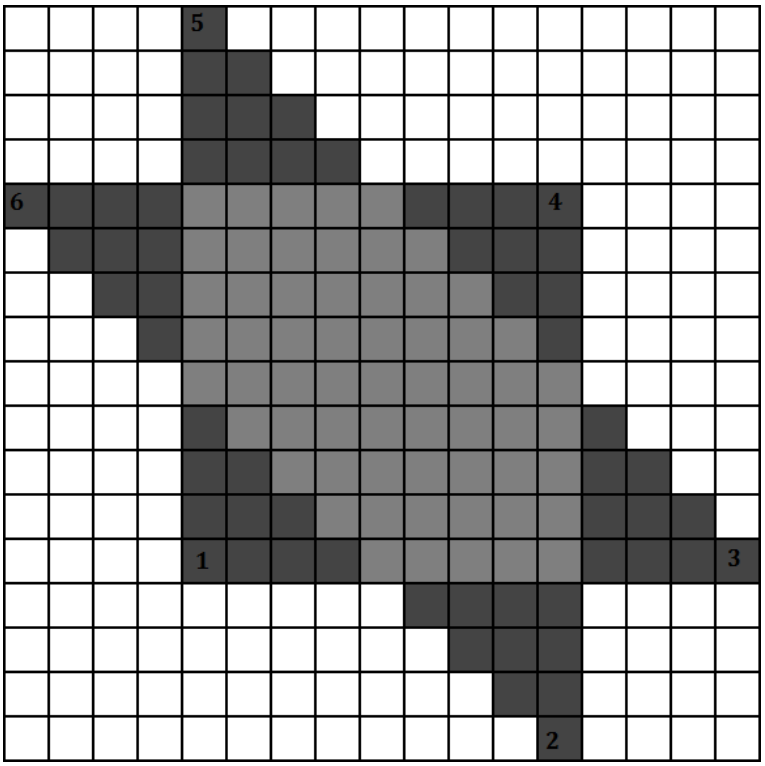


Figura 4.3: Representação do tabuleiro numa matriz 17x17

Quando um jogo é inicializado, na matriz, os espaços em branco são casas não jogáveis, os espaços com o número 0 são casas vazias e os números 1-6 representam uma peça do jogador do seu respetivo número como pode ser visto no lado direito das Figuras 4.1 e 4.2.

4.1.2 Jogadores

Os jogadores são representados pelo seu número, a posição de tabuleiro onde vão começar e as coordenadas das suas respetivas peças no tabuleiro. As posições representam os cantos que têm as 10 casas que marcam o início ou fim de um jogo. Na Figura 4.3 são as posições em cinzento escuro. Nesta implementação o canto inferior esquerdo do tabuleiro foi escolhido como a primeira posição e vai incrementando sempre em 1 por cada canto no sentido anti-horário, como podemos ver na Figura 4.3. Para dois jogadores, o jogador número 1 vai jogar a partir da primeira posição e vai tentar chegar à quarta posição enquanto o jogador número 2 começa

na quarta posição e tenta chegar à primeira. Para três jogadores, o jogador número 1 volta a começar na primeira posição e joga para chegar à quarta posição, o jogador número 2 começa na terceira posição e tenta chegar à sexta e o jogador número 3 começa na quinta posição e tenta chegar à segunda.

4.2 MCTS-UCT

Em primeiro lugar foi implementada uma versão genérica do MCTS-UCT a partir do Algoritmo 2, para o limite computacional escolhido que foi o número de nós explorados. Um nó é considerado explorado se é adicionado à árvore criada pelo MCTS-UCT. A função $(S(V1))$ corresponde à expansão de um nó, ou seja, as ações geradas a partir desse nó. Durante as expansões todas as possíveis ações são adicionadas à árvore.

Algoritmo 2: MCTS-UCT

```
function MCTS-UCT(S0)
    Cria nó-raiz V0 a partir do estado S0
    while Dentro dos limites computacionais do
        V1 ⇐ Política_de_Árvore(V0)
        Δ ⇐ Política_de_Simulação(S(V1))
        Propagar(Δ, V1)
    return Melhor_Filho(V0)
end function
```

Para a política de árvore usamos a que foi proposta por Kocsis e Szepesvári [6] que vai maximizar:

$$UCB1 = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}} \tag{4.1}$$

com $C_p = \frac{1}{\sqrt{2}}$, que como já vimos é uma boa constante para valores entre $[0, 1]$, que vai ser o caso deste jogo porque o valor de cada nó vai ser a sua taxa de vitória. Podemos ver no Algoritmo 3 como foi implementada a política de árvore. Até encontrar um nó não expandido, o algoritmo vai escolher o nó que maximiza a política UCB1, retornando no fim um nó não expandido que maximiza UCB1.

Para a política de simulação, utilizamos uma política *Epsilon-Greedy* que escolhe jogar a peça mais atrás (*farthest-first*) com uma probabilidade de 95% e uma peça aleatória com probabilidade de 5%. Esta heurística foi utilizada para as simulações porque reduz a profundidade média de um nó terminal de 306 para 120, para dois jogadores, e de 398 para 145, para três jogadores, em comparação com uma política de escolha aleatória. A escolha desta heurística permite explorar os nós em menos tempo, conseguindo diminuir o tempo de execução total do algoritmo, porque estamos a usar o número de nós explorados como limite computacional. Contudo queremos sempre manter alguma aleatoriedade para evitar construir sempre a mesma

Algoritmo 3: Política de Árvore

```
function Política_de_Árvore(V)
    while V tem descendentes do
        V ⇐ UCB1(V)
    return V
end function
function UCB1(V)
    return arg_max_{V'∈FilhosdeV} Q(V')/N(V') + (1/√2)·√(2lnN(V)/N(V'))
end function
```

árvore de jogo. Podemos ver a implementação da política de simulação no Algoritmo 4.

Algoritmo 4: Política de Simulação

```
function Política_de_Simulação(V)
    while V não for terminal do
        V ⇐ arg_min_{V'∈S(V)} Farthest_First(V')
    return V
end function
```

Por fim, os resultados das simulações são propagados iterativamente pela árvore até a raiz. Este processo é fácil porque cada nó guarda uma referência para o estado anterior. Os resultados em cada nó correspondem ao resultado da simulação feita a partir desse nó mais os resultados dos seus nós filhos.

Em qualquer decisão do algoritmo, se existir empates, a escolha é feita de forma aleatória.

O maior problema que o algoritmo apresenta é o elevado fator de ramificação na árvore de jogo. Mesmo não considerando movimentos para trás ou para os cantos, que não sejam os iniciais ou finais, o fator de ramificação médio do jogo é de 26, chegando aos 50 nos piores casos para dois jogadores. Para três jogadores, o fator de ramificação médio é de 30 podendo chegar aos 100 nos piores casos. Durante testes iniciais ao algoritmo, este não conseguia retornar respostas satisfatórias, podendo mesmo não ser capaz de terminar um jogo. Isto deve-se ao facto de que com muitos nós adicionados durante a expansão, o algoritmo não conseguia distinguir as possíveis jogadas para escolher uma melhor. Além disso, quanto maior é o fator de ramificação, menor será a profundidade da árvore de jogo possível de ser explorada por causa das limitações de tempo. A solução que usamos para este problema foi limitar o número de nós que são adicionados à árvore durante a expansão. Desta maneira, diminuímos o crescimento da árvore de jogo. Ao escolher um limite para o número máximo de nós filhos, utilizamos como padrão o valor sugerido por Sturtevant[13], que é uma ação por peça, ou seja, a cada expansão são adicionadas 10 jogadas à árvore de jogo. Desta maneira, quando expandimos o nó selecionado, no máximo 10 das suas possíveis jogadas são adicionadas à árvore de jogo.

Com o algoritmo implementado vimos vários parâmetros que podem ser alterados, estes parâmetros são: o número de nós explorados, o número de ações adicionadas à árvore durante

a expansão e a escolha das ações adicionadas à árvore. Para poder maximizar a qualidade da resposta dada pelo algoritmo temos que perceber como estes parâmetros afetam a resposta.

Capítulo

5

Metodologia Experimental

Para avaliar como os diferentes parâmetros do algoritmo afetam a qualidade das respostas realizamos vários testes com diferentes versões do MCTS-UCT. Os testes foram realizados para dois e para três jogadores, usando diferentes números de nós explorados. Desta maneira conseguimos perceber como cada parâmetro altera o algoritmo conseguindo ver as diferenças ou semelhanças entre dois e três jogadores, assim como em situações com diferentes números de nós explorados. Na realização dos testes, dois algoritmos diferentes jogaram entre si 30 jogos, em configurações diferentes, conforme o número de jogadores. Convencionamos cada versão do MCTS-UCT como um algoritmo diferente, apesar de serem apenas diferentes versões do mesmo algoritmo. Para dois jogadores quando testamos dois algoritmos, temos duas configurações diferentes como podemos ver na Tabela 5.1, sendo então jogados 15 jogos em cada configuração. Para três jogadores quando testamos dois algoritmos, temos seis configurações diferentes como podemos ver na Tabela 5.2 e foram jogados 5 jogos em cada configuração. Os resultados de cada algoritmo correspondem à percentagem de jogos ganhos durante cada teste. Também foi testado o impacto de ser o primeiro jogador. Para este teste não é preciso utilizar diferentes configurações, pois o objetivo é testar para determinado algoritmo como é afetado se for o primeiro a jogar. Para estes testes foram realizados 15 jogos para cada versão do algoritmo, e os resultados correspondem à percentagem de jogos ganhos pelo primeiro jogador.

Jogador 1	Jogador 2
Algoritmo 1	Algoritmo 2
Algoritmo 2	Algoritmo 1

Tabela 5.1: Configurações para 2 jogadores

Os parâmetros que pretendemos testar são o limite de nós explorados, o limite do fator de ramificação, e diferentes políticas de seleção dos nós. Os testes foram divididos em 4 categorias

Jogador 1	Jogador 2	Jogador 3
Algoritmo 1	Algoritmo 2	Algoritmo 1
Algoritmo 1	Algoritmo 1	Algoritmo 2
Algoritmo 1	Algoritmo 2	Algoritmo 2
Algoritmo 2	Algoritmo 1	Algoritmo 2
Algoritmo 2	Algoritmo 2	Algoritmo 1
Algoritmo 2	Algoritmo 1	Algoritmo 1

Tabela 5.2: Configurações para 3 jogadores

descritas nas subsecções seguintes.

5.1 Políticas de Seleção dos nós

Como pretendemos limitar o número de nós adicionados durante cada expansão do algoritmo, vamos ter que de alguma maneira fazer uma seleção de que nós vamos adicionar à árvore de jogo. Para tal foram testadas duas diferentes políticas, uma *Greedy* e uma *Epsilon-Greedy*.

O algoritmo com a política *Greedy*, adiciona à árvore de jogo, a cada expansão, as 10 melhores jogadas, a não ser que existam menos que 10 possíveis jogadas. Sendo este o caso, adiciona-as todas.

O algoritmo com a política *Epsilon-Greedy*, a cada expansão, adiciona 10 jogadas, ou se existir menos que 10 adiciona todas as jogadas existentes, onde 50% das jogadas são selecionadas usando a política *Greedy* e as outras 50% são selecionadas aleatoriamente. Esta proporção de jogadas foi escolhida, porque adicionava tanto jogadas escolhidas com a heurística como jogadas aleatórias. A heurística garante que as peças estão todas a avançar no tabuleiro, enquanto a escolha aleatória de nós pode deixar peças nas casas iniciais, não se conseguindo terminar o jogo. Mas a escolha aleatória serve também para perceber se estamos e enviesar muito a escolha dos nós feita pelo algoritmo. Se existir empates na seleção de nós, o desempate é feito aleatoriamente.

Para este teste usamos uma heurística que avalia os nós conforme o número de movimentos simples para terminar o jogo e a distância das peças mais distantes das posições finais. Desta maneira, uma jogada é considerada melhor se consegue terminar em menos movimentos simples e não deixa peças para trás.

Apenas diferenciando a política usada na seleção dos nós adicionados à árvore, os algoritmos utilizam os mesmos parâmetros. Foram realizados três testes diferentes aos dois algoritmos, diferenciando os testes alterando os limites para o número de nós explorados: 1000, 2000 e 4000.

5.2 Fator de Ramificação

Como já vimos, a árvore de jogo do CC tem um fator de ramificação muito alto, e para combater este problema utilizamos sempre um fator de ramificação limitado, ou seja, limitamos o número de nós que pode ser adicionado à árvore de jogo. O número máximo de nós adicionados, durante a fase de expansão, utilizado nos testes é 10. Este número tem por base os estudos feitos por Sturtevant no jogo de CC, que vimos no Capítulo 3. Contudo queremos perceber o impacto da alteração deste valor no comportamento do algoritmo.

Para este conjuntos de testes foram utilizados três algoritmos, que apenas tinham diferentes limites para o fator de ramificação. Os algoritmos escolhem sempre os melhores nós conforme o limite de nós que podem adicionar à árvore de jogo. O primeiro algoritmo pode adicionar no máximo 10 nós à árvore, o segundo pode adicionar no máximo 5 nós à árvore e o terceiro algoritmo pode adicionar no máximo 20 nós à árvore. Nos testes realizados o primeiro algoritmo jogou contra os outros dois algoritmos e foram realizados testes alterando os limites para o número de nós explorados para 1000, 2000 e 4000.

5.3 Nós Explorados

Como vimos no Capítulo 2, se o número de nós explorados tender para infinito, a resposta vai tender para a solução ótima. Contudo, se pretendermos que o algoritmo retorne uma resposta vamos ter que limitar o número de nós que pode explorar. Com este teste pretende-se perceber o ganho que podemos ter ao aumentar o número de nós que podemos explorar, e tentar encontrar uma relação entre a qualidade das respostas e o número de nós expandidos, sabendo que o tempo de execução do algoritmo aumenta linearmente com o número de nós expandidos. Para estes testes foram usados três algoritmos com os mesmos parâmetros, diferenciando apenas o número permitido de nós explorados. Os algoritmos utilizam uma política de seleção de nós *Greedy*, igual à vista na sub-secção anterior, e podem adicionar no máximo as 10 melhores jogadas durante cada expansão. Onde cada um dos algoritmos se diferencia é no número de nós explorados. O primeiro deles tem um limite de 1000 nós, para o segundo o limite é 2000 nós e para o terceiro algoritmo o limite é 4000 nós.

Nos testes realizados, cada algoritmo jogou contra os outros dois algoritmos.

5.4 Importância da 1ª jogada

Nos jogos por turnos, a primeira jogada traz sempre alguma vantagem ou desvantagem, apesar desta característica não estar diretamente relacionada com o algoritmo, mas sim com o jogo em si. Achamos importante perceber se os algoritmos conseguiam tirar partido desta vantagem. Para tal utilizamos os resultados de todos os testes realizados, fazendo uma análise para determinar se o primeiro jogador tinha alguma vantagem. Para isto usamos os resultados

de cada conjunto de teste, e vemos a percentagem de vitórias que o primeiro jogador tem em cada teste. Se a primeira jogada não oferece nenhuma vantagem ao jogador, será de esperar que este ganhe 50% dos jogos para dois jogadores, e 33,3% dos jogos para três jogadores.

Capítulo

6

Resultados

Neste capítulo apresentamos e discutimos os resultados obtidos nos testes. Em primeiro lugar, são apresentados os resultados para dois jogadores, depois para três jogadores e, por fim, são discutidas as conclusões.

6.1 Dois Jogadores

Em primeiro lugar, vamos apresentar os resultados dos algoritmos, jogando com apenas dois jogadores.

6.1.1 Política de Seleção dos nós

No algoritmo implementado limitamos o número de nós adicionados à árvore de jogo durante cada expansão. Isto significa que vamos ter que fazer uma seleção dentre todas as possíveis ações. A maneira mais simples de fazer uma seleção de descendentes é usar uma heurística para avaliar os possíveis nós e selecionar a partir dessa heurística, mas desta maneira podemos enviesar muito o algoritmo. Com este teste pretendemos então perceber se temos melhores resultados ao adotarmos uma seleção mais *Greedy*, ou uma solução mais híbrida com *Epsilon-Greedy*.

O que sabemos é que a seleção de nós é importante porque, se existir uma jogada ótima e esta não for adicionada à árvore, não interessa o número de nós que o algoritmo explore, nunca retornará a melhor resposta. A política *Greedy*, no total, ganhou 73,3% dos jogos (ver Tabela 6.1), o que significa que consistentemente seleciona sempre melhores nós para adicionar à árvore do que a política *Epsilon-Greedy*. Com diferentes números de nós explorados observamos a percentagem de vitórias alterando. Com 1000 nós a política *Greedy* ganhou 76,7% dos jogos.

Tabela 6.1: 2 jogadores: Política de Seleção *Greedy* vs Política de Seleção *Epsilon-Greedy*

		Greedy	*Epsilon-Greedy*
1000 Nós Explorados	Vitórias	76,7%	23,3%
	Jogadas	65,9	53,5
2000 Nós Explorados	Vitórias	60%	40%
	Jogadas	59	51,6
4000 Nós Explorados	Vitórias	83,3%	16,7%
	Jogadas	56,4	50
Total	Vitórias	73,3%	26,7%
	Jogadas	60,4	51,8

Com 2000 nós explorados, a percentagem diminui para os 60%, mas com 4000 nós explorados a política *Greedy* venceu 83,3% dos jogos. Podemos também ver a média de jogadas que o algoritmo fez nos jogos que venceu. Vemos que o número médio de jogadas vai diminuindo a medida que os algoritmos exploram mais nós. Isto deve-se ao facto de que quando exploram mais nós, os algoritmos retornam melhores respostas. Podemos ver que o algoritmo *Greedy* ganha em média em 60,4 jogadas e que o algoritmo *Epsilon-Greedy* ganha em 51,8 jogadas. Este valores não refletem a qualidade das respostas dos algoritmos, isto porque o *Epsilon-Greedy*, no total, ganhou 26,7% dos jogos retornando melhores respostas, mas os resultados demonstram que não conseguia retornar melhores respostas que o algoritmo *Greedy* com muita frequência. Desta maneira podemos concluir que utilizando uma política *Greedy* para a seleção de nós, o algoritmo vai jogar consistentemente melhor.

6.1.2 Fator de Ramificação

Nos Capítulos 4 e 5 foi mencionado que iríamos limitar o número de filhos adicionados à árvore durante a fase de expansão (limitar o fator de ramificação), e que o valor padrão para esse limite iria ser 10. Contudo queríamos perceber como o aumento ou a diminuição desse valor iria alterar o comportamento do algoritmo. Neste teste foram usados valores relativamente extremos, duplicando ou reduzindo para a metade o valor padrão. Com este teste, podemos tentar inferir o comportamento do algoritmo com valores intermédios

Tabela 6.2: 2 jogadores: Fator de Ramificação limitado 5 Nós vs 10 Nós

		5 Nós Adicionados	10 Nós Adicionados
1000 Nós Explorados	Vitórias	56,7%	43,3%
	Jogadas	50,1	51,3
2000 Nós Explorados	Vitórias	33,3%	66,7%
	Jogadas	51,5	50,2
4000 Nós Explorados	Vitórias	43,3%	56.7%
	Jogadas	52,1	46,4
Total	Vitórias	44,4%	55,6%
	Jogadas	51,3	49,1

No primeiro conjunto de testes (Tabela 6.2) procuramos comparar o valor padrão de 10 com um valor menor 5. O algoritmo que adiciona 10 nós, no total, ganhou 55,6% dos testes, mostrando ser genericamente o melhor valor a adotar. Contudo é necessário realçar que para o menor número de nós explorados, 1000, o algoritmo que adiciona 5 nós ganhou 56,7% dos jogos. Isto deve-se ao facto de que ao limitar mais o número de nós adicionados por expansão, o algoritmo vai conseguir alcançar maiores profundidades na árvore de jogo do MCTS e vai ter menos nós como possível resposta, o que significa que cada uma das possíveis respostas terá sido explorada mais vezes, melhorando a qualidade do seu valor estimado (UCT). Ao aumentarmos o número de iterações, esta vantagem diminui pois o número adicional de nós explorados permite ao algoritmo que adiciona 10 nós alcançar maior profundidade e melhorar a qualidade dos valores estimados pelo algoritmo, mas o algoritmo que adiciona 5 filhos irá continuar com um número mais limitado de jogadas possíveis. Tal como vimos no teste anterior o algoritmo vai estar sempre dependente das jogadas que adiciona à árvore, e se não adicionar a melhor jogada à árvore, nunca irá retornar a melhor solução. Como foi já mencionado, o valor 5 é um valor extremo do valor padrão, mas ainda assim conseguiu manter-se competitivo. Podemos também ver na Tabela 6.2, a média de jogadas de cada algoritmo nos jogos que venceu. Os valores desta vez estão próximos para ambos os algoritmos, sendo que o algoritmo que vence mais jogos apresenta sempre um valor mais baixo de jogadas. Vemos também um decréscimo do número médio de jogadas para o algoritmo que adiciona 10 nós consoante o aumento do número de nós explorados, e um acréscimo para o algoritmo que adiciona 5 nós. Neste momento, é importante notar que o número médio de jogadas que um algoritmo faz quando vence, em conjunto com a percentagem de jogos ganhos refletem a consistência do algoritmo. Isto é, um algoritmo com um número médio baixo de jogadas e uma percentagem baixa de vitórias, tem pouca consistência nas respostas dadas e não consegue terminar muitos jogos nesse número de jogadas. Enquanto um algoritmo com uma percentagem maior de vitórias, demonstra mais consistência, a terminar os jogos num número mais constante de jogadas.

Tabela 6.3: 2 jogadores: Fator de Ramificação limitado 10 Nós vs 20 Nós

		10 Nós Adicionados	20 Nós Adicionados
1000 Nós Explorados	Vitórias	83,3%	16,7%
	Jogadas	64,4	56,2
2000 Nós Explorados	Vitórias	86,7%	13,3%
	Jogadas	53,1	54,7
4000 Nós Explorados	Vitórias	70%	30%
	Jogadas	55,6	50,4
Total	Vitórias	80%	20%
	Jogadas	57,7	52,9

No segundo conjunto de testes (Tabela 6.3) comparamos o comportamento do algoritmo que adiciona 10 nós por expansão, valor padrão, contra um algoritmo que adiciona o dobro de nós por expansão, 20 nós. O algoritmo que adiciona 10 nós ganhou, no total, 80% dos jogos.

Da mesma forma que no conjuntos de testes anterior, vemos que aumentando o número de nós explorados o algoritmo que adiciona mais nós vai melhorando os seus resultados. Contudo, o valor 20 demonstra ser um valor muito extremo, tendo um resultado muito negativo na percentagem de vitórias (20%), não conseguindo manter-se competitivo contra o valor padrão. Desta vez, vemos o algoritmo que adiciona 10 nós jogar em média 57,7 jogadas até vencer, e o algoritmo que adiciona 20 nós a fazer 52,9 jogadas quando vence, ou seja, o algoritmo que adiciona 20 nós, é inferior e não consegue consistentemente penalizar o adversário quando faz más jogadas. Consideramos que o fator de ramificação 20 é um valor muito alto e diminui a eficácia do algoritmo.

Como vimos no teste anterior, ao limitarmos mais o número máximo de nós adicionados por expansão, permitimos ao algoritmo melhorar o valor calculado pelo MCTS-UCT, assim como permitimos ao algoritmo explorar a árvore em mais profundidade. A desvantagem de limitar o número de nós adicionados é, que ao fazermos uma escolha de apenas um subconjunto das possíveis jogadas, podemos deixar a melhor jogada de fora desde o início. Tanto a heurística da escolha dos nós como o número de nós escolhidos, vão ajudar a diminuir a probabilidade de isto acontecer. Contudo como temos um número limitado de nós que vamos explorar, quanto menos nós adicionarmos por expansão, melhor o algoritmo vai conseguir distinguir a melhor jogada de jogadas menos boas fazendo com que seja preciso ajustar os valores usados. Com os resultados obtidos conseguimos concluir que o valor 10 para o fator de ramificação é o melhor, para maximizar a chance de vitória. Contudo, nos testes que fizemos não tentamos encontrar os melhores valores para cada cenário de jogo, levando-nos a assumir que este valor pode ser ajustado melhor à quantidade de nós explorados. Para um número inferior de nós explorados, diminuir o fator de ramificação aumenta a chance de vitória, e para um número maior de nós explorados, aumentar o fator de ramificação irá aumentar a chance de vitória. Contudo o valor padrão utilizado apresenta genericamente bons resultados para qualquer dos cenários do jogo.

6.1.3 Nós Explorados

Como foi estudado na literatura, ao aumentar o número de nós expandidos, a resposta retornada pelo algoritmo vai começar a tender para a resposta ótima (resposta do *Min-Max* ou Max^n). Contudo queremos comparar como o algoritmo se vai comportar com diferentes números de nós explorados, e como melhora a chance de vitória com o aumento do número de nós explorados.

Nos testes realizados (Tabela 6.4) vemos que a chance de vitória aumenta com o aumento do número de nós explorados. O algoritmo que explora mais nós (4000 nós) venceu mais jogos (70%) e o algoritmo que explora menos nós (1000 nós) venceu o menor número de jogos (26,7%). O número médio de jogadas que cada algoritmo precisou para vencer está relativamente próximo. Isto é porque os algoritmos estão a usar as mesmas políticas de seleção de nós e a adicionar o mesmo número de nós por expansão e, portanto, os algoritmos têm o mesmo potencial de escolher a melhor jogada. Contudo, como já foi mencionado, a diferença está

Tabela 6.4: 2 jogadores: Nós Explorados: 1000 nós vs 2000 nós vs 4000 nós

Nós Explorados	1000 Nós		2000 Nós		4000 Nós	
	Vitórias	Jogadas	Vitórias	Jogadas	Vitórias	Jogadas
1000 Nós	-		73,3%	51	73,3%	50,7
2000 Nós	26,7%	51,3	-		66,7%	51
4000 Nós	26,7%	51,6	33,3%	49,3	-	
Total	26,7%	51,4	53,3%	50,4	70%	50,8

na consistência em terminar os jogos em poucas jogadas, e desta maneira, podemos ver que o algoritmo que adiciona 4000 nós é o mais consistente e por isso ganhou mais jogos. Com estes resultados concluímos que o ideal será usar o máximo tempo disponível para a execução do algoritmo, pois como vimos, quanto mais nós forem explorados, maior vai ser a chance de vitória.

6.1.4 Primeira Jogada

Como nos jogos por turnos a primeira jogada traz consigo algum tipo de vantagem, decidimos analisar os resultados dos testes feitos, para tentar perceber se ter a primeira jogada aumenta a chance de vitória. A Tabela 6.5 está organizada da seguinte maneira: a primeira coluna corresponde ao número de nós explorados (1000, 2000 ou 4000) durante os testes da Seleção de nós e o Fator de ramificação, ou ao teste que corresponde dos testes com diferentes nós explorados (1000 vs 2000, 1000 vs 4000 ou 2000 vs 4000). Na segunda coluna são mostrados os resultados das vitórias do primeiro jogador no teste da seleção de nós. Na terceira e na quarta colunas temos os resultados dos testes do fator de ramificação para 5 nós vs 10 nós adicionados e 10 nós vs 20 nós adicionados, respetivamente. Na última coluna, temos os testes com diferentes número de nós explorados. A última linha corresponde ao total de vitórias do primeiro jogador no total dos jogos realizados.

Nos testes realizados (Tabela 6.5), vemos que o jogador que tem a primeira jogada ganhou no total 55,2% dos jogos, o que demonstra que existe uma vantagem em ter a primeira jogada. Se não existisse o primeiro jogador devia ter ganho apenas 50% dos jogos. Os jogos onde esta vantagem é mais evidente são aqueles em que os algoritmos podem explorar mais nós (4000 nós). O que nos leva a concluir que existe uma vantagem em ter a primeira jogada, pelo menos para dois jogadores, e que os algoritmos que exploram mais nós tiram maior proveito desta vantagem.

6.2 Três Jogadores

Um dos objetivos deste trabalho é estudar como se comporta o algoritmo para diferentes número de jogadores. Após os testes com dois jogadores foram realizados os testes para três jogadores. Nesta secção vamos analisar os resultados dos testes de três jogadores e fazer algumas

Tabela 6.5: 2 jogadores: Vantagem da Primeira Jogada

	Seleção de Nós	Fator de Ramificação (5 nós vs 10 nós)	Fator de Ramificação (10 nós vs 20 nós)	Nós Explorados
1000/ 1000 vs 2000 Nós Explorados	53,3%	46,7%	53,3%	50%
2000/ 1000 vs 4000 Nós Explorados	56,7%	50%	50%	63,3%
4000/ 2000 vs 4000 Nós Explorados	53,3%	60%	66,7%	60%
Total	54,4%	52,2%	56,7%	57,8%
	55,2%			

comparações com os testes anteriores.

6.2.1 Políticas de Seleção dos nós

Na secção anterior vimos que o algoritmo com a política *Greedy* ganhou mais jogos, e podemos ver um comportamento semelhante com três jogadores (Tabela 6.6). O algoritmo com a política *Greedy* ganhou, no total, 61,1% dos jogos. Para três jogadores vemos a política *Greedy* a ganhar menos jogos do que para dois jogadores (Tabela 6.1), exceto no teste com 2000 nós explorados.

Tabela 6.6: 3 jogadores: Política de Seleção *Greedy* vs Política de Seleção *Epsilon-Greedy*

		Greedy	*Epsilon-Greedy*
1000 Nós Explorados	Vitórias	63.3%	36.7%
	Jogadas	49,4	46,4
2000 Nós Explorados	Vitórias	66,7%	33,3%
	Jogadas	46,2	44,3
4000 Nós Explorados	Vitórias	53,3%	46,7%
	Jogadas	42,3	43,2
Total	Vitórias	61,1%	38,9%
	Jogadas	46,1	44,5

Estes resultados podem significar que a heurística que utilizamos para selecionar os nós pode não ser igualmente boa para dois e três jogadores. Em comparação com os testes para dois jogadores, ambos os algoritmos terminam o jogo em menos jogadas, 46,1 jogadas para a versão *Greedy* e 44,5 para a versão *Epsilon-Greedy*, enquanto estas demoram 60,4 e 51,8 respetivamente para dois jogadores.

6.2.2 Fator de Ramificação

Nos testes a dois jogadores vimos que o algoritmo que adiciona 10 nós por expansão tinha a maior chance de vitória. Contudo, o algoritmo que adiciona 5 nós tinha maior chance de vitória, quando podia explorar apenas 1000 nós. Vimos também um acréscimo das chances de vitória, quando o número máximo de nós aumentava, para o algoritmo que adicionava mais nós por expansão, mas que o valor 20, para o fator de ramificação, era muito elevado e trazia os piores resultados.

Tabela 6.7: 3 jogadores: Fator de Ramificação limitado 5 Nós vs 10 Nós

		5 Nós Adicionados	10 Nós Adicionados
1000 Nós Explorados	Vitórias	46,7%	53,3%
	Jogadas	44,9	45,9
2000 Nós Explorados	Vitórias	26,7%	73,3%
	Jogadas	44,8	44,2
4000 Nós Explorados	Vitórias	40%	60%
	Jogadas	44,6	43,4
Total	Vitórias	37,8%	62,2%
	Jogadas	44,7	44,4

Para três jogadores (Tabela 6.7), temos o algoritmo que adiciona 10 nós a ganhar, no total, 62,2% dos jogos. Nos testes para dois jogadores vimos ainda o algoritmo que adiciona menos nós a ter mais chances de vitória quando se exploravam apenas 1000 nós, mas agora vemos o algoritmo que adiciona 10 nós, apesar que por pouco, vencer mais jogos (53,3%) com este número de explorações, e quando o número de nós explorados aumenta, a superioridade deste algoritmo torna-se mais evidente, vencendo 73,3% dos jogos quando se explora 2000 nós e vencendo 60% dos jogos quando se explora 4000 nós.

Tabela 6.8: 3 jogadores: Fator de Ramificação limitado 10 Nós vs 20 Nós

		10 Nós Adicionados	20 Nós Adicionados
1000 Nós Explorados	Vitórias	76,7%	23,3%
	Jogadas	48,5	48,7
2000 Nós Explorados	Vitórias	66,7%	33,3%
	Jogadas	47,4	45,5
4000 Nós Explorados	Vitórias	93,3%	6,7%
	Jogadas	45,1	45
Total	Vitórias	78,9%	21,1%
	Jogadas	46,9	46,2

Nos testes seguintes (Tabela 6.8), o algoritmo que adiciona 10 nós por expansão jogou contra o que adiciona 20 nós. Mais uma vez vemos o algoritmo que adiciona 10 nós a vencer mais jogos (78,9%). Em média os algoritmos demoraram, aproximadamente, mais duas jogadas do que no teste anterior, o algoritmo que adiciona 10 nós terminou os jogos em 44,7 jogadas no

primeiro conjuntos de testes (Tabela 6.7) e neste conjunto de testes precisou de 46,9 jogadas. Isto deve-se ao facto do algoritmo que adiciona 20 nós apresentar os piores resultados e quando um algoritmo não consegue penalizar o outro quando faz piores jogadas, a média de jogadas sobe. Os algoritmos, no geral, tiveram um comportamento igual para dois e para três jogadores. Podendo-se concluir que o valor 10 para o fator de ramificação foi o melhor dos utilizados.

6.2.3 Nós Explorados

Nos testes com dois jogadores vimos que a chance de vitória aumentava quando o número de nós explorados aumentava, e que o algoritmo que explorava mais nós ganhou o maior número de vezes. Com três jogadores as conclusões retiradas mantiveram-se (Tabela 6.9).

Tabela 6.9: 3 jogadores: Nós Explorados: 1000 nós vs 2000 nós vs 4000 nós

Nós Explorados	1000 Nós		2000 Nós		4000 Nós	
	Vitórias	Jogadas	Vitórias	Jogadas	Vitórias	Jogadas
1000 Nós	-		76,7%	46,2	80%	44,9
2000 Nós	23,3%	45,8	-		66,7%	44.3
4000 Nós	20%	45,3	33,3%	44,5	-	
Total	21,6%	45,5	55%	45,6	73,3%	44,6

O algoritmo que explora mais nós, 4000, ganhou 73,3% dos jogos, conseguindo ganhar 80% dos jogos ao algoritmo que explora 1000 nós, e ganhou 66,7% dos jogos ao algoritmo que explora 2000 nós. Apesar do número médio de jogadas em que cada algoritmo vence os jogos estar próximo para os três algoritmos, o algoritmo que explora 4000 nós termina os jogos em menos jogadas e ganha 73,3% dos jogos, demonstrando que é consistente com as suas jogadas, conseguindo terminar o jogo em menos jogadas. Os resultados obtidos para dois jogadores coincidem com os resultados obtidos para três jogadores. Vemos que os algoritmos que exploram mais nós ganharam mais jogos, para os algoritmos que usamos, o que explora 4000 nós foi o melhor.

6.2.4 Primeira Jogada

Na Secção anterior vimos que o primeiro jogador tinha uma chance de vencer de 55,2%, o que significava que o primeiro jogador tinha uma ligeira vantagem por ter a primeira jogada, e vimos que essa vantagem ficava mais aparente quantos mais nós os algoritmos pudessem explorar.

Para três jogadores (Tabela 6.10) vemos que a primeira jogada tem uma vantagem pequena 35,3%, que seria 33,3% num jogo sem que a primeira jogada tivesse uma vantagem, em relação aos outros jogadores. Tal como para dois jogadores, a vantagem torna-se mais evidente quando os algoritmos podem explorar mais nós. O teste onde a primeira jogada teve mais impacto, foi onde testamos algoritmos a explorar números diferentes de nós. Isto deve-se ao facto dos algoritmos destes testes utilizarem a mesma política de seleção de nós e adicionarem o mesmo

Tabela 6.10: 3 jogadores: Vantagem da Primeira Jogada

	Seleção de Nós	Fator de Ramificação (5 nós vs 10 nós)	Fator de Ramificação (10 nós vs 20 nós)	Nós Explorados
1000/ 1000 vs 2000 Nós Explorados	20%	20%	40%	53,3%
2000/ 1000 vs 4000 Nós Explorados	40%	16,7%	30%	46,7%
4000/ 2000 vs 4000 Nós Explorados	36,7%	36,7%	43,3%	46,7%
Total	32,2%	24,5%	37,7%	46,9%
	35,3%			

número de nós, por isso uma das vantagens que os algoritmos tinham em relação aos outros, alem do número de nós explorados, era a vantagem de ter a primeira jogada.

Capítulo

7

Conclusão

Neste trabalho implementamos o jogo de CC para o tabuleiro de tamanho normal. Para jogar o jogo foi implementado o MCTS-UCT. Para estudar a importância dos parâmetros do algoritmo foram feitos vários testes para dois e três jogadores. Os parâmetros que testamos foram a política de escolha de nós, o número de nós adicionados por expansão e o número de nós explorados. Também analisamos a importância da primeira jogada.

Para a política de seleção de nós, a política *Greedy* ganhou mais jogos, mostrando que a escolha de nós impacta o resultado final, e que deve ser uma escolha bem consciente por parte do algoritmo não devendo deixar ao acaso. Para o fator de ramificação concluímos que o valor 10 maximiza as chances de vitória para os valores testados. Mas vimos que outros possíveis valores podem maximizar as chances de vitórias para diferentes números de nós expandidos. Por exemplo, o valor 5 venceu mais jogos para dois jogadores explorando apenas 1000 nós. Isto leva-nos a crer que um ajuste do valor utilizado conforme o número de nós explorados irá maximizar as chances de vitória, ou seja, diminuir o número de nós adicionados quando exploramos menos nós, e aumentar o número de nós adicionados quando exploramos mais nós. O número de nós explorados é o parâmetro mais limitante do algoritmo pois, se aumentarmos o seu valor, o tempo de execução do algoritmo vai aumentar. Com os resultados obtidos, vimos que ao explorar o maior número de nós estamos a aumentar as chances de vitória.

Este trabalho apresentou alguma limitações, nomeadamente o tempo de execução dos algoritmos. Em média, para cada jogador, o algoritmo demora 35 minutos por cada 1000 nós explorados durante um jogo inteiro, o que resulta em que um jogo de dois jogadores, em que cada algoritmo possa explorar 1000 nós, demore cerca de 70 minutos e um jogo de três jogadores, em que cada algoritmo explore 4000 nós, demore cerca de 7 horas. Por causa disto, não foi possível testar algoritmos que explorassem mais nós, apesar de na literatura [13] estar explícito que o número de nós que usamos é baixo. Isto também fez com que não pudéssemos

realizar mais testes, em particular, para três jogadores, onde foram feitos apenas 5 jogos por cada configuração, tornando mais difícil fazer uma análise individual dos resultados de cada teste.

7.1 Trabalhos Futuros

Na secção da política de seleção de nós, foi mencionado que foi utilizada apenas uma heurística, para a seleção dos nós. Esta mesma heurística foi utilizada na fase de simulação do MCTS-UCT. Sabemos que a seleção dos nós impacta a resposta final do MCTS-UCT, por isso encontrar a melhor heurística irá melhorar a qualidade de jogo do algoritmo. Sturtevant et al. [14] fez um estudo preliminar, do uso de bases de dados de finais de jogo para tentar melhorar as heurísticas para o jogo de CC com 10 peças, baseados nos estudos de Roschke et al. [9], que utilizavam este método para o jogo de CC com 6 peças. E apresentavam uma melhoria nos resultados do MCTS-UCT. Desta mesma maneira, propõe-se uma melhora na heurística usada nas simulações, por exemplo, Nijssen [8] propõe o uso de Playout Search, que em vez de fazer uma escolha *Epsilon-Greedy* na escolha dos nós, realiza uma pesquisa para encontrar o melhor nó para selecionar, assim melhorando a qualidade das simulações.

Nos testes sobre o fator de ramificação vimos que o valor 10 apresentou os melhores resultados, contudo existe espaço para melhorar a qualidade do MCTS-UCT alterando o fator de ramificação, aumentando ou diminuindo conforme o número de nós que o algoritmo pode explorar, ou até mesmo utilizando um valor dinâmico que altere conforme a profundidade da árvore, ou com o jogador da vez. Por exemplo Schadd et al. [11] propuseram um algoritmo que apenas adiciona à árvore de jogo a melhor jogada do adversário mais forte, permitindo ao jogador da vez poder analisar mais jogadas próprias. Este algoritmo superou o Max^n e o *Paranoid* a jogar o jogo de CC. Este método poderia ser adaptado à árvore de MCTS-UCT.

Como já comentamos, uma da limitações deste trabalho foi a falta de testes, especialmente para 3 jogadores, não conseguindo analisar detalhadamente os resultados de cada teste individual. Dando espaço para um estudo mais detalhado do comportamento dos algoritmos com diferentes números de nós explorados, especialmente com mais nós explorados.

Por falta de tempo não foi conseguido criar uma base de dados das jogadas iniciais, contudo o uso de uma base de dados com as jogadas iniciais pode reduzir significativamente o tempo de execução do algoritmo, assim como melhorar a qualidade das jogadas iniciais. Uma das propriedades que descobrimos do jogo de CC é que existe um número limitado de jogadas iniciais ótimas, e esta jogadas não são afetadas pelas jogadas dos adversários, pois as peças entre jogadores inicialmente estão distanciadas. E as simulações iniciais são as que demoram mais tempo, pois está o jogo quase todo ainda por ser simulado.

7.2 Comentários finais

A área dos jogos não para de encontrar avanços no que toca a melhorar a qualidade que um computador consegue jogar e será uma questão de tempo até que os computadores sejam os melhores jogadores. Em particular, o jogo de CC está a começar a chamar o interesse de vários cientistas, pois oferece um desafio diferente: jogar com múltiplos jogadores. O MCTS está a destacar-se como o melhor algoritmo para jogar este jogo, contudo ainda existem melhorias a fazer até conseguir tornar este algoritmo no melhor jogador do jogo. Com este trabalho conseguimos estudar alguns parâmetros do MCTS-UCT para este jogo e identificar como podemos alterá-los para maximizar as chances de vitória. Contudo ainda há muitas oportunidades para melhoramentos.

Bibliografia

[1] Deep blue. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/impacts/. [acedido em 14-01-2016]. Cited in page 1.

[2] Bruno Bouzy and Tristan Cazenave. Computer go: An {AI} oriented survey. *Artificial Intelligence*, 132(1):39 – 103, 2001. Cited in page 1.

[3] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, March 2012. Cited in pages 2 and 12.

[4] Murray Campbell, A.Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57 – 83, 2002. Cited in page 1.

[5] Bell George I. The shortest game of chinese checkers and related problems. *Integers*, 9:17–39, 2009. Cited in page 6.

[6] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006. Cited in pages 14 and 23.

[7] Carol Luckhart and Keki B. Irani. An algorithmic solution of n-person games. In Tom Kehler, editor, *AAAI*, pages 158–162. Morgan Kaufmann, 1986. Cited in page 9.

[8] J. (Pim) A. M. Nijssen and Mark H. M. Winands. *Playout Search for Monte-Carlo Tree Search in Multi-player Games*, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Cited in page 40.

[9] Max Roschke and Nathan R. Sturtevant. *UCT Enhancements in Chinese Checkers Using an Endgame Database*, pages 57–70. Springer International Publishing, Cham, 2014. Cited in pages 19 and 40.

[10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. Cited in pages 2, 6, 7, and 8.

[11] M. P. D. Schadd and M. H. M. Winands. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):57–66, March 2011. Cited in page 40.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. Cited in page 2.

[13] Nathan R. Sturtevant. *An Analysis of UCT in Multi-player Games*, pages 37–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Cited in pages 2, 15, 18, 24, and 39.

[14] Nathan R. Sturtevant. *Challenges and Progress on Using Large Lossy Endgame Databases in Chinese Checkers*, pages 3–15. Springer International Publishing, Cham, 2016. Cited in page 40.

[15] Nathan Reed Sturtevant. *Multiplayer Games: Algorithms and Approaches*. PhD thesis, 2003. AAI3089030. Cited in pages 2, 9, 10, 11, and 15.