connect 4

Criação e análise de algoritmos para jogos de oponentes

Inteligência Artificial

Introdução

*Adversarial Search* é um tipo de pesquisa utilizado para calcular a melhor jogada em jogos de dois jogadores onde toda a informação é dada. A pesquisa consiste em procurar todas as jogadas possíveis no jogo onde para cada movimento é representado por um valor dependendo da chance de ganhar ou perder.

O importante neste tipo de procura é maximizar a vitória da máquina e minimizar a possível derrota. Para calcular isto, abordaremos três tipos de algoritmos de busca contraditória: *Minimax*, *Alpha-beta Pruning* (Corte Alfa Beta) e *MCTS* (Busca em Árvore Monte-Carlo). e quanto mais rápido e otimizado for esse algoritmo, melhor será para chegar à configuração final do quebra-cabeças.

Minimax Algorithm

O algoritmo *Minimax* consiste em determinar qual a melhor jogada, prevendo todas as possibilidades dentro dos próximos m movimentos, sendo m a profundidade máxima dada pelo utilizador ou o número de jogadas até o tabuleiro estar completo.

Para calcular este movimento, o programa deve **minimizar** a possível derrota e **maximizar** as chances de vitória. Para calcular isto utilizaremos uma estrutura de árvore na qual cada nó é uma jogada possível sobre um nó/tabuleiro pai. A partir deste estado, o algoritmo pode gerar todos os possíveis estados futuros que podem resultar após o adversário jogar.

Quando o programa alcança a profundidade máxima (ou atinge um ponto crítico), este retorna uma pontuação do tabuleiro que alcançou, podendo avaliar assim qual o melhor caminho a seguir.

# Complexidade temporal:

# Complexidade espacial:

**b** = fator de ramificação, **m** = profundidade máxima



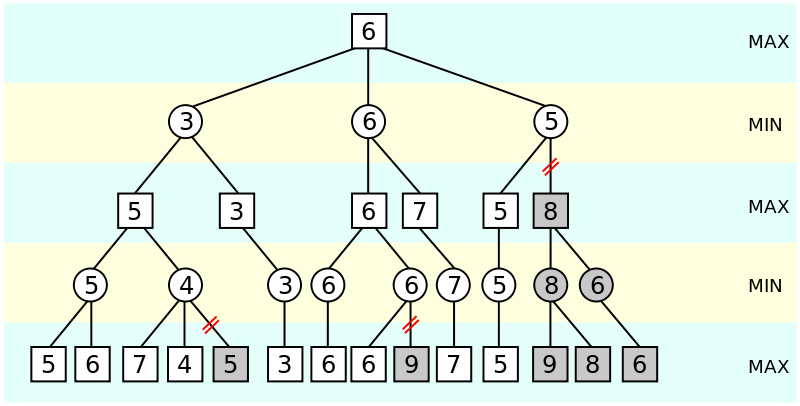
Alpha-beta Pruning

Esta técnica é uma otimização do algoritmo *Minimax* e serve para reduzir significativamente o tempo de processamento. Isto permite ao programa procurar mais rapidamente e até ir mais aprofundadamente na árvore do jogo.

Consiste em cortar caminhos que sejam inúteis, por exemplo, se o programa já sabe que consegue ter uma certa pontuação alta, não terá necessidade de procurar uma pontuação mais pequena.

# Complexidade temporal:

**b** = fator de ramificação, **m** = profundidade máxima



Monte Carlo tree search (MCTS)

*MCTS* é um algoritmo que encontra o melhor movimento a através de uma **seleção**, **expansão**, **simulação** e **atualização** de nós na árvore para encontrar a solução final.

**Seleção**: Primeiramente o programa vai selecionar o nó que tem mais possibilidades de ganhar. Nas imagens seguintes utilizamos nós com as devidas chances de o fazer.

**Expansão**: Para aumentar o número de opções, expande-se o nó selecionado criando-se vários nós filho (Nesta demonstração apenas usaremos um). Estes nós são os futuros movimentos que podem ser jogados no futuro.

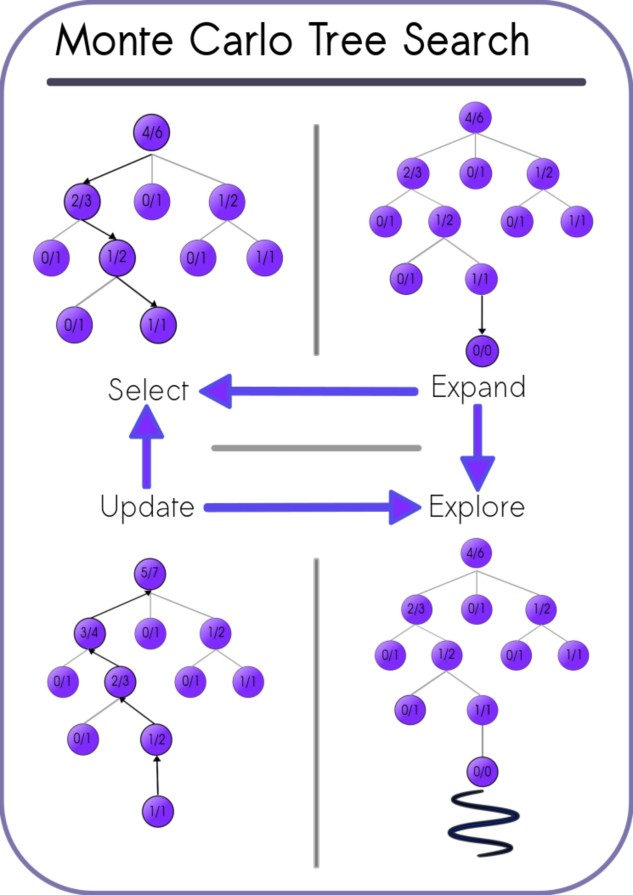
**Simulação**: Simulamos para descobrir qual o melhor nó expandido. Como obtemos isto? Utilizamos **Reinforcement Learning** (goal-oriented algorithms que aprendem a chegar a um objetivo ou a maximizar uma certa dimensão através de movimentos) para fazer decisões aleatórias no jogo a cada nó filho, sendo dado uma *recompensa* a este (ao calcular quão próximo está esse movimento aleatório do output final).

“*When it is not in our power to determine what is true, we ought to act in accordance with what is most probable*”- *Descartes*

*(Neste caso vamos assumir que a simulação devolveu um nó otimista com probabilidade de 1/1)*

**Atualização**: Segundo as pontuações dos novos nós, as pontuações dos seus pais devem ser alteradas subindo a árvore, um a um. Esta atualização muda o estado da árvore e pode mudar o próximo nó selecionado.

Depois de atualizarmos todos os nós, recomeçamos a técnica para encontrar um novo nós que será o resultado vencedor.



Connect 4

**4 em linha** (ou connect 4) é um jogo de tabuleiro entre 2 jogadores sobre um tabuleiro.

Cada jogador pode introduzir apenas uma peça por turno numa das colunas, fazendo a peça cair verticalmente até a primeira casa disponível, de baixo para cima. O primeiro jogador a conseguir colocar as próprias 4 peças seguidas na horizontal, diagonal ou vertical é o vencedor. Se o tabuleiro ficar completo, o jogo termina empatado.

No código usamos um tabuleiro com 7 colunas e 6 linhas e identificadores de peças são uma cruz(x) e um círculo(o).

“Object: Connect four of your checkers in a row while preventing your opponent from doing the same. But, look out -- your opponent can sneak up on you and win the game!”- *Milton Bradley, Connect Four "Pretty Sneaky, Sis" television commercial, 1977*

A interpelação para a vitória da máquina pode ser feita de forma algorítmica, apesar do 4 em linha ser um jogo **resolvido**, ou seja, o primeiro jogador tem sempre maior vantagem e poderá sempre ganhar, se forem feitas as jogadas certas.

Conclusão

O algoritmo de pesquisa em árvore Monte Carlo explica que em vez de usarmos força bruta e vermos as milhões de formas possíveis do caminho certo, podemos usar Reinforcement Learning calcular mais eficazmente e mais rapidamente

Nos tipos de busca não-informados, o BFS é o mais rápido, mas não fica muito longe do IDFS. Isto acontece porque a forma de busca por ambos é igualmente por camada, mas o IDFS acaba por gastar mais memória porque volta a ler os nós que visitou anteriormente. O DFS é de longe o mais pesado destas pesquisas e o que visita mais nós, devido à sua profundidade.

Nos tipos de busca informados a procura pelo resultado é incrivelmente mais linear e mais rápida. Isto acontece porque estes tipos utilizam heurísticas, que servem para atribuir pesos nas operações feitas pelo programa, ou seja, o programa não complicará o jogo pois não lhe interessa ficar mais longe do resultado final, a menos que perceba que não há alternativa.

Quanto às heurísticas usadas nestes últimos tipos, têm algum impacto dependendo se formos a utilizar Greedy ou A-star. Em quase todos os casos que utilizamos, a heurística de somatório na busca Greedy teve uma maior profundidade da solução do que a de Manhattan, mas gastou menos memória, ou seja, com Manhattan temos quase sempre o resultado com menos jogadas, mas sempre mais rapidame

nte do que com o somatório. Na busca A-star costuma demorar menos tempo, usar menos memória mas encontra a mesma profundidade e visita mais nós.

Concluindo, as buscas informadas são mais complexas mas muito mais funcionais que as não-informadas. Neste jogo, se utilizarmos a busca Greedy, devemos optar pela heurística se Manhattan pois é mais precisa, mas se utilizarmos A-star, devemos de usar a heurística de somatório.

Referências

Game theory — Minimax, *by NerdzLab*

https://towardsdatascience.com/game-theory-minimax-f84ee6e4ae6e

Minimax Algorithm in Game Theory, *by  Akshay L Aradhya*

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Game-tree Search and Pruning Algorithms, *by Mohammad T. Hajiaghayi, University of Maryland*

http://www.cs.umd.edu/~hajiagha/474GT15/Lecture12122013.pdf

Monte Carlo Tree Seacrh, *by Sagar Sharma*

https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa

Reinforcement learning, *by Prateek Bajaj*

https://www.geeksforgeeks.org/what-is-reinforcement-learning/

Reinforcement learning, *by Prateek Bajaj*

http://www.cs.umd.edu/~hajiagha/474GT15/Lecture12122013.pdf

Breadth First Search (BFS)

O BFS, também denominada de busca em largura, é um tipo de busca não informada em que ao contrário da DFS, faz a sua pesquisa a partir de níveis, isto é, começando pela raiz, vai procurando em cada nível de profundidade todos os nós lá pertencentes , caso não encontre nenhum , passa para o nível seguinte, e continua este processo até encontrar o nó almejado. Através deste algoritmo podemos garantir que qualquer nó será visitado, pudendo encontrar assim múltiplas soluções , dando prioridade sempre ao nó que se encontra na menor profundidade.

O BFS terá uma complexidade espacial superior à do DFS caso haja um elevado grau de ramificação de nós num determinado nível profundidade, levando ao gasto acentuado da memória.

Complexidade do Algoritmo:

# Complexidade temporal:

# Complexidade espacial:

**b** = fator de ramificação

|  |  |  |  |
| --- | --- | --- | --- |
| **Final** | | | |
| **9** | **5** | **12** | **7** |
| **14** | **13** | **0** | **8** |
| **1** | **3** | **2** | **4** |
| **6** | **10** | **5** | **11** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Estratégia | | Tempo (ms) | Espaço | Profundidade da solução | Nº Nós visitados |
| BFS | | 158 | 9Mb | 13 | 13790 |
| DFS (Depth 80) | | 408 | 113MB | 79 | 2260561 |
| IDFS | | 139 | 14Mb | 13 | 8718 |
| GREEDY | Somatório | 51 | 295Kb | 13 | 16 |
| Manhattan | 33 | 2Mb | 19 | 47 |
| A\* | Somatório | 51 | 296Kb | 13 | 45 |
| Manhattan | 52 | 295Kb | 13 | 16 |

(DFS retirado no Teste 2 no Gráfico 1 por ser um valor demasiado alto, afetando assim a visualização das restantes informações)