U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Professora Inês Dutra

Inteligência Artificial

Segundo Trabalho: Jogos com 2 Jogadores

Abril de 2022

Trabalho realizado por:

João Marrucho - 201804960

Bruno Dias - 201907828

Pedro Leite - 201906697

Índice

1. Introdução

Nos trabalhos anteriores, estudamos estratégias de procura que estão associadas a um único agente que visa encontrar a solução por uma sequência de ações.

Mas existem certas situações onde mais do que um agente está a procurar pela solução no mesmo espaço de procura, e esta situação normalmente ocorre em jogos com adversários. Nesses jogos de adversários, cada jogador precisa considerar as ações do outro jogador e o efeito dessas ações para o seu desempenho. Os jogos são modelados principalmente como um problema de busca e uma função de avaliação de utilidade.

Os algoritmos usados para resolver estes problemas são o Minimax, Alpha-Beta Pruning. Estes serão os algoritmos que iremos usar neste trabalho para avaliar o seu comportamento e eficácia, aplicando-os ao jogo Quatro em linha.

2. Algoritmo Minimax

Minimax é um tipo de algoritmo que visa encontrar o movimento ideal para um jogador, assumindo que o seu adversário também jogue de forma otimizada.

No Minimax, os dois jogadores são chamados de "max" e "min", respetivamente. O max tenta obter a pontuação mais alta possível, enquanto o min tenta fazer o oposto e obter a pontuação mais baixa possível.

Cada estado do jogo tem um valor associado a ele. Num determinado estado, se o max tiver vantagem, a pontuação do tabuleiro tenderá a ser algum valor positivo. Se o min tiver a vantagem nesse estado do jogo, ele tenderá a ser algum valor negativo. Os valores do tabuleiro são calculados por algumas heurísticas únicas para cada tipo de jogo.

3. Algoritmo Alpha-Beta Pruning

Alpha-Beta Pruning é uma versão modificada do algoritmo minimax. É uma técnica de otimização para o algoritmo minimax.

Sem verificar todos os nós da árvore do jogo, podemos calcular a decisão minimax correta, e essa técnica é chamada de Pruning. Isso envolve dois parâmetros de limite Alpha e Beta para expansão futura.

A Alpha-Beta Pruning retorna o mesmo movimento que o algoritmo minimax padrão, mas remove todos os nós que não estão realmente a afetar a decisão final. Portanto, ao remover esses nós, torna o algoritmo mais eficiente.

Os dois parâmetros podem ser definidos como:

Alpha: A melhor escolha (de maior valor) que encontramos até agora em qualquer ponto ao longo do caminho do Maximizer. O valor inicial de alfa é -∞.

Beta: A melhor escolha (de menor valor) que encontramos até agora em qualquer ponto ao longo do caminho do Minimizer. O valor inicial de beta é +∞.

A Alpha-Beta Pruning para um algoritmo minimax padrão retorna o mesmo movimento que o algoritmo padrão, mas remove todos os nós que não estão realmente a afetar a decisão final, mas a tornar o algoritmo lento. Portanto, ao remover esses nós, torna o algoritmo rápido.

4. Algoritmo Monte Carlo Tree Search

O algoritmo básico do MCTS é simples: uma árvore de busca é construída, nó por nó, de acordo com os resultados dos playouts simulados. O processo pode ser dividido nas seguintes etapas:

Seleção: Começando no nó raiz R, seleciona-se recursivamente os nós filhos ótimos (explicados abaixo) até que um nó folha L seja alcançado.

Expansão: Se L não for um nó terminal (ou seja, não encerrar o jogo), cria um ou mais nós filhos e seleciona um C.

Simulação: Executa um playout simulado de C até que um resultado seja alcançado.

Retropropagação: Atualiza a sequência de movimento atual com o resultado da simulação.

Cada nó deve conter duas informações importantes: um valor estimado com base nos resultados da simulação e o número de vezes que foi visitado.

Na sua implementação mais simples e eficiente de memória, o MCTS adicionará um nó filho por iteração. No entanto, pode ser benéfico adicionar mais de um nó filho por iteração, dependendo do aplicativo.

5. Quatro em linha

O Quatro em linha é jogado usando 42 fichas (normalmente 21 fichas vermelhas para um jogador e 21 fichas pretas para o outro jogador), e uma grade vertical com 7 colunas de largura. Cada coluna pode conter no máximo 6 fichas. Os dois jogadores jogam por turnos. Um movimento consiste em um jogador deixar cair uma das suas fichas na coluna de sua escolha. Quando a ficha é colocada numa coluna, ela cai até atingir a parte inferior da coluna ou a ficha superior dessa coluna. Um jogador ganha criando um arranjo no qual pelo menos quatro das suas fichas estão alinhadas em uma linha, coluna ou diagonal.

6. Linguagem, Estrutura de Dados e Funções Auxiliares

A linguagem que decidimos utilizar para este trabalho foi **Python**. Já que trabalhamos com esta linguagem para o projeto anterior, e achamos mais fácil de utilizar do que as suas concorrentes (C++, Java, etc), devido à sua sintaxe simples e intuitiva e vastos recursos disponíveis online.

Para representar o tabuleiro do jogo, decidimos utilizar uma **Matriz** de 6 por 7, já que é a maneira mais fácil de visualizar, modificar e imprimir a configuração do jogo.

Definimos a opção de o jogador, poder escolher jogar contra alguém na função "**twoplayers**", onde cada jogador, consoante a sua vez, escolhe uma coluna, entre 0 e 6, enquanto nenhum chegar à vitória. Após cada jogada, é imprimido o tabuleiro. Também pode escolher jogar contra o computador, na função "**one_player**", onde também poderá escolher o nível de dificuldade que corresponde à profundidade dos algoritmos que vamos abordar. O jogador também tem a opção de escolher quem começa primeiro, e se for o computador, ele escolhe aleatoriamente que coluna escolhe na primeira jogada.

Definimos uma função **"can_play"**, que retorna uma string com as diferentes colunas em que é possível chegar.

Para verificar se algum dos jogadores ganhou, temos a função "**solvable**", que verifica se há 4 símbolos iguais seguidos, ao longo da matriz toda. A função faz esta verificação para cima, na função "**cima**", para a direita, na função "**direita**", na diagonal para baixo, na função "**diabaixo**" e na diagonal para

cima, na função "**diacima**". Se houver uma solução a função retorna "True", senão retorna "False", e continua a verificação recursivamente.

Definimos também uma função "**draw**", no caso de empate.

Para modificar a matriz do tabuleiro após a jogada de um determinado jogador, temos a função "**sucessores**". Que na coluna escolhido pelo jogador, procura de baixo para cima pelo símbolo "-", e assim que o encontra troca-o pelo símbolo do respetivo jogador.

Para a medição da utilidade temos a função "**utility**". A utilidade é medida em blocos de 4 ao longo da matriz, na horizontal, na função "**dir_utility**", na vertical, na função "**up_utility**", na diagonal para cima, na função "**upright_utility**" e na diagonal para baixo, na função "**downright_utility**". Se nesses blocos encontrarmos 4 símbolos iguais somamos/subtraímos 512 pontos, se encontrarmos 3 símbolos iguais somamos/subtraímos 50, se encontrarmos 2 símbolos iguais somamos/subtraímos 10 e se encontrarmos 1 símbolo somamos/subtraímos 1, tendo em conta se esse símbolo é o nosso ou do nosso adversário e se o resto do bloco não contem símbolos do jogador adversário.

7. Implementação do Algoritmo Minimax

Para a implementação do algoritmo Minimax, definimos a função "**max_player**". Nesta função começamos por definir uma string com as colunas onde se pode jogar, através da função "**can_play**", esta variável vai ser utilizada para sabermos que colunas são viáveis de calcular os sucessores, quando a ativamos escolhemos aleatoriamente uma das opções possíveis, para que futuramente se tivermos duas opções com a mesma utilidade, escolhemos aleatoriamente a peça que jogamos e não a primeira coluna que aparece. Definimos também uma variável booleana, que é defina como "true" se o jogo acabou (se um dos jogadores ganhou, através da função "**solvable**" ou se houve um empate, através da função "**draw**"), esta função vai ser útil para saber quando terminar o algoritmo.

Para o algoritmo escolher sempre a melhor jogada possível, cada vez que é a sua vez de jogar, vai definir um árvore com uma certa profundidade, quão maior for profundidade mais tempo vai demorar a escolher que peça utilizar, mas a probabilidade de ser uma melhor jogada também aumenta, onde o nó inicial, é

o tabuleiro atual, e os nós filho, correspondem aos respetivos sucessores (jogadas possíveis), e assim em diante até à profundidade definida ou até chegar a uma solução (utliza pesquisa em profundidade). Cada nível de profundidade vai ser respetivamente maximizante, minimizaste, maximizante, minimizante… O nível maximizante é inicialmente definido com um valor muito grande negativo e o nível minimizante com um valor muito grande positivo. Nas profundidades maximizantes (vez do algoritmo de jogar) vão ser definidas utilidades positivas, consoante o quão promissora a jogada é para o algoritmo, e nas profundidades minimizante (vez do oponente do algoritmo jogar) vão ser definidas utilidades negativas, consoante o quão promissora a jogada é para o seu adversário. Assim o algoritmo escolhe sempre um sucessor, tendo em conta o quão boa essa jogada vai ser para si (maximizante), e quão menor as consequências dessa jogada vão ser (minimizante).

Quando a árvore chega ao seu limite, o algoritmo acaba por escolher o sucessor, que tem o caminho com maior utilidade.

8. Implementação do Algoritmo Alpha-Beta Pruning

Alpha-Beta Pruning é uma versão modificada do algoritmo minimax, logo a sua implementação, definida pela função "**max_alpha**", foi baseada na implementação do algoritmo Minimax.

No maximizante, definimos um alpha que vai guardando os maiores valores da utilidade até ao momento. E no minimizante, definimos um beta que vai guardando os menores valores da utilidade até ao momento. No maximizante quando o alpha for maior ou igual ao beta do pai e no minimizante quando o alpha do pai for maior ou igual ao beta, já que vai calcular ramos dispensáveis ele interrompe a recursão, economizando assim tempo e memória.

9. Resultados dos Algoritmos Implementados

Profundidade	Minimax	Alpha-Beta Pruning
3	0.604s	0.457s
5	11s	1s
7	6.3m	3.4s

Profundidade	Minimax	Alpha-Beta Pruning
8	*	10.7s
9	*	35s

* O tempo cresce exponencialmente, por isso é muito difícil de calcular o tempo do Minimax para estas profundidades.

10. Conclusão

Como expectável, o algoritmo Alpha-Beta Pruning demonstrou ser mais eficiente do que o algoritmo Minimax, principalmente para profundidades mais elevadas. Já que o Alpha-Beta Pruning é uma implementação do Minimax, mas aperfeiçoada, cortando todos os nós que não demonstram chegar a uma solução pretendida. Sendo que o Minimax demonstra uma complexidade temporal O(b^m) e o Alpha-Beta Pruning O(b^m/2).

11. Referências Bibliográficas

S. Russell, P. Norvig; Artificial Intelligence: A Modern Approach, 3rd ed, Prentice Hall, 2009

Slides da unidade curricular