Relatório de trabalho número 2

Gelson Stalino Varela – número: 202109347

João Vivas – número: 202108177

Index

Introdução

Os jogos com oponentes diferenciam-se dos jogos com um único, pois não podem ser resolvidos com os mesmos métodos de pesquisa que usamos para jogos como o puzzle dos 15. Isto porque estes métodos não assumem a presença de um oponente. Em jogos com oponentes nós temos que lidar com a incerteza das jogadas do oponente, com a memória extra que é necessária devido à essa incerteza e com a performance com que o computador faz as jogadas. Neste relatório vamos só abordar os algoritmos que foram usados no **jogo 4 em linha**, como os algoritmos ***Minimax***, ***Alpha-Beta Pruning*** e ***Monte Carlo tree search (MCTS)***.

Algoritmos

Minimax

O algoritmo *Minimax* é uma regra de decisão onde se tenta minimizar a possível perda para um cenário com o pior caso. Para calcular as jogadas usa-se um algoritmo para calcular a utilidade (pontuação) do estado do jogo onde o estado com maior pontuação indica uma melhor jogada e o estado com menor pontuação significa uma pior jogada (melhor para o adversário) [1]. Como são dois jogadores vamos assumir que um jogador vai ser o MAX e o outro o MIN. O MAX vai tentar encontrar uma sequência que vai gerar uma vitória porém vai ter que ter que considerar as jogadas do MIN. Para o MAX poder jogar de forma ótima tem que assumir que o MIN vai jogar também de forma ótima. Vamos assumir uma árvore de jogo em que o primeiro a jogar é o MAX. Quando for MAX a escolher ele vai procurar o nó que vai lidar para uma maior pontuação e passar a vez para o MIN que vai escolher o nó que menor pontuação. Este processo vai ser repetido até se chegar a um estado final de preferência um que dê a vitória ao MAX. Temos que ter cuidado, pois MAX pode fazer uma jogada que ao longo termo pode ser pior então temos que calcular o jogo até uma certa profundidade e verificar se é necessário sacrificar a melhor jogada se isso nos vai lidar para a solução ótima [2].

Alpha-Beta Pruning

O algoritmo ***Alpha-Beta Prunning*** ou **Corte Alpha-Beta** é uma modificação do algoritmo ***MiniMax*** que vai otimizar a memória utilizada. No algoritmo *MiniMax* vão ser analisados todos os nós da árvore do jogo para tentar encontrar uma solução ótima, porém o número de estados aumenta exponencialmente em função com a profundidade da árvore [2]. O corte *Alpha-Beta* vai cortar alguns nós da árvore que são garantidos que nunca serão escolhidos por nenhum jogador se ambos jogarem de forma ótima. A desvantagem deste algoritmo é sobre como os nós estão ordenados ora se quando for o MAX a selecionar o nó, se o primeiro nó filho que vai selecionar

for o máximo só tem que expandir esse nó e não precisa de se preocupar com os outros nós filhos, porém se os nós não tiverem bem ordenados, no pior caso o corte *Alpha-Beta* pode não ter nenhuma vantagem em relação ao algoritmo *MiniMax*.

Monte Carlo Tree Search (MCTS)

O algoritmo *Monte Carlo Tree Search* é diferente dos outros algoritmos até agora falados, porque não utiliza uma heurística para selecionar que nó vai expandir em vez disso vai calcular a utilidade com base em simulações de jogos completos a partir de um certo estado. A vantagem do *MCTS é* não usar heurísticas falíveis em vez disso usa uma "probabilidade de ganhar" como uma "utilidade média" para um nó [2].

O *Monte Carlo Tree Search* tem 4 etapas:

- **Seleção**: Começamos na raiz da árvore e escolhemos um movimento (com base numa "política de seleção" e isso vai dar a um nó sucessor. Este processo é repetido para podermos descer ao longo da árvore de pesquisa. O movimento que escolhemos tem haver com a probabilidade de um nó tem de guiar para uma vitória por isso escolhemos sempre o que tem maior probabilidade.
- **Expansão**: Nós aumentamos a árvore ao gerar novos sucessores do nó selecionado.
- **Simulação**: Fazemos um jogo a partir do novo nó gerado escolhendo os movimentos dos dois jogadores segundo uma "política de jogo". Estas jogadas não vão ser guardadas na árvore.
- **Retro-Propagação**: Quando a simulação chega a um fim vai ser usado para atualizar a estatística de cada de todos os nós do caminho até à raiz. Cada nó vai guardar o número total de simulações e o número de simulações que lidaram para uma vitória do jogador. Ou seja no jogo **4 em linha** se o computador for as peças vermelhas, cada nó vai guardar o número de simulações total e o número de simulações que deram uma vitória às peças vermelhas. Se uma simulação der vitória ao adversário (peças amarelas) aumentam só o número de simulações.

Estes passos são repetidos sempre até atingirmos um número de iterações ou ficarmos sem tempo para continuar. Quando esse ciclo acaba é retornado o movimento com o maior número de simulações [2]. Ao selecionarmos temos que seguir uma política de seleção, existe uma eficiente chamada ***"upper confidence bounds applied to trees"*** ou ***UCT*** que avalia cada movimento possível com uma fórmula de confiança chamada ***UCB1***. Para cada nó *n* a fórmula é a seguinte*:*

$$UCB1 = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log(N(Parent(n)))}{N(n)}}$$

Onde *U(n)* é o número de simulações que passaram por *n* que resultaram numa vitória, *N(n)* é o número total de simulações que passaram por *n* e *Parent(n)* é o pai do nó *n* na árvore. Então $\frac{U(n)}{N(n)}$ é a utilidade média de *n*. O termo com a raiz quadrada vai calcular a exploração. Como tem *N(n)* no denominador quer dizer que o termo vai ser mais alto para nós que só foram pouco explorados. No numerador temos o logaritmo do número de vezes que exploramos o pai do nó quer que quanta maior a percentagem de explorarmos um nó *n* o termo de exploração vai diminuir para 0 à medida que aumentamos o número de explorações e eventualmente retornar o nó com maior utilidade média.

MCTS é mais vantajoso em jogos com árvores com grande fator de ramificação muito alto como o *Go* onde o algoritmo do *corte Alpha-Beta* não consegue fazer um corte de forma eficiente, pois precisa de explorar até ao fim da árvore. Outra vantagem é que este algoritmo pode ser usado em jogos, mais recentes onde não há uma heurística tão bem definida, apenas necessita da informação das regras e mais nada. Uma desvantagem que tem vem por causa da sua natureza aleatória o algoritmo pode por vezes falhar uma jogada crucial para vencer o jogo .

Jogo 4 em linha

O jogo abordado neste trabalho é o jogo **4 em linha**. Este jogo consiste em dois jogadores inserirem alternadamente uma peça de uma cor num tabuleiro retangular vertical constituído por seis linhas e sete colunas. O objetivo de cada jogador é tentar colocar 4 peças da sua cor ao lado umas das outras de forma a formar uma linha seja esta vertical, horizontal ou diagonal. Este jogo é um jogo resolvido, ou seja, o primeiro jogador pode sempre vencer se fizer as jogadas corretas[3].

Implementação do jogo e dos algoritmos

Nós decidimos fazer este trabalho em Java, por ser uma linguagem orientada a objetos, que nos permite a nossa estrutura de dados para alguns dos algoritmos como o *Monte Carlo Tree Search*.

O código está dividido em quatro ficheiros: *Game, Board, Strategies* e *Mcts*. O ficheiro *Game* é o ficheiro que vai ser executado e controla a lógica do jogo como o turno dos jogadores e verifica se o jogo já acabou seja por empate ou se um jogador ganhou. Também é onde é escolhido o tipo de estratégia usada pelo computador para jogar o jogo. O ficheiro *Board* controla o que acontece relacionado com o tabuleiro que está implementado numa matriz. Este ficheiro vai ter as funções responsáveis pela inserção de peças no tabuleiro (função *put*), verificar se o tabuleiro está cheio (*isFull*), a utilidade de uma jogada (*evaluate*), se existe um vencedor (*thereIsWinner*), quem é o vencedor (*winner*), bem como imprimir o tabuleiro (*printBoard*). Já o ficheiro

Strategies é onde são calculadas as estratégias *MiniMax* e *AlphaBeta* também tem uma chamada para uma função *mcts* que é executada num ficheiro à parte. O gicheiro *Mcts* é o ficheiro onde está implementado o algoritmo *Mcts*. Nós decidimos colocar esse algoritmo num ficheiro à parte por causa do seu grande volume de implementação.

Algoritmo MiniMax

Nós implementamos este algoritmo em três funções *minimax*, *maxValue*, *minValue*. Primeiro é desabilitado a função de corte *Alpha-Beta* definida pela variável global *ALPHA_BETA* e a profundidade que o algoritmo vai pesquisar, neste caso procura em 5 níveis de profundidade a contar com o nível de *root*. Como a função vai ser chamada quando for o computador a jogar então vai primeiro procurar o sucessor com maior utilidade, depois de descer um nó vai procurar o nó com menor utilidade e descer o nível e assim por adiante. A função que procura o sucessor com maior utilidade é *maxValue* e que procura o que tem menor utilidade é *minValue*. Quando estas funções são executadas, os sucessores do nó selecionado são guardados numa em memória e para cada sucessor é executado a função seguinte e calculado qual a jogada com maior utilidade e qual a coluna que vai ser selecionada pelo computador. As funções estão implementadas em código desta forma:

```
private static int bestColumn, rootDepth;
private static boolean ALPHA_BETA;
static int minimax(int depth, Board board) {
		rootDepth = depth-1;
		ALPHA_BETA = false;
		maxValue(rootDepth, board, Integer.MIN_VALUE, Integer.MAX_VALUE);
		return bestColumn;
}
static int maxValue(int depth, Board board, int alpha, int beta) {
		if (depth == 0) return board.evaluate();
		int value = Integer.MIN_VALUE; // constante da classe Integer
		Map<Board, Integer> successors = board.successors(Board.YELLOW);
		for (Board successor : successors.keySet()) {
				int value2 = minValue(depth-1, successor, alpha, beta);
				if (value2 > value) {
						value = value2;
						alpha = Math.max(alpha, value);

						if (depth == rootDepth) bestColumn = successors.get(successor).intValue();
				}
				if(ALPHA_BETA && value >= beta) return value;
		}
		return value;
}
static int minValue(int depth, Board board, int alpha, int beta) {
		if (depth == 0) return board.evaluate();
		int value = Integer.MAX_VALUE; // constante da classe Integer
		Map<Board, Integer> successors = board.successors(Board.RED);
		for (Board successor : successors.keySet()) {
				int value2 = maxValue(depth-1, successor, alpha, beta);
				if (value2 < value) {
						value = value2;
						beta = Math.min(beta, value);
				}
				if(ALPHA_BETA && value <= alpha) return value;
		}
		return value;
}
```

Algoritmo Alpha-Beta

Este algoritmo é basicamente igual ao *MiniMax*, a maior diferença é que *ALPHA_BETA* vai ter um valor de *true*, ou seja, não vai executar as funções *maxValue* e *minValue* para os sucessores que garantidamente que o programa não vai escolher. A única função nova é a função *alpha-beta* que basicamente é a função que atribui o valor de *true* a *ALPHA_BETA*. O algoritmo está implementado assim:

```
static int maxValue(int depth, Board board, int alpha, int beta) {
	if (depth == 0) return board.evaluate();
	int value = Integer.MIN_VALUE; // constante da classe Integer
	Map<Board, Integer> successors = board.successors(Board.YELLOW);
	for (Board successor : successors.keySet()) {
		int value2 = minValue(depth-1, successor, alpha, beta);
		if (value2 > value) {
			value = value2;
			alpha = Math.max(alpha, value);
			if (depth == rootDepth) bestColumn = successors.get(successor).intValue();
		}
		if(ALPHA_BETA && value >= beta) return value;
	}
	return value;
}

static int minValue(int depth, Board board, int alpha, int beta) {
	if (depth == 0) return board.evaluate();
	int value = Integer.MAX_VALUE; // constante da classe Integer
	Map<Board, Integer> successors = board.successors(Board.RED);
	for (Board successor : successors.keySet()) {
		int value2 = maxValue(depth-1, successor, alpha, beta);
		if (value2 < value) {
			value = value2;
			beta = Math.min(beta, value);
		}
		if(ALPHA_BETA && value <= alpha) return value;
	}
	return value;
}

static int alpha_beta(int depth, Board board) {
	rootDepth = depth-1;
	ALPHA_BETA = true;
	maxValue(rootDepth, board, Integer.MIN_VALUE, Integer.MAX_VALUE);
	return bestColumn;
}
```

Algoritmo MCTS

O algoritmo MCTS foi feito num ficheiro diferente ***Mcts.java***. Neste ficheiro nós criámos uma classe *Node* para podermos fazer as várias etapas do algoritmo Monte-Carlo. Esta classe *Node* vai ter atributos *board*, *currentPlayer*, *parent*, *childrenList*, *wins* e *numberOfVisits*. O atributo *board* descreve o estado atual do tabuleiro, *currentPlayer* diz se é o utilizador ou o computador a jogar, *childrenList* é uma lista dos nós descendentes do nó atual, *numberOfVisits* é o número de simulações que já passaram pelo nó e *wins* é o número dessas simulações que resultaram numa vitória.

Quando é o computador a jogar é executada a função *bestMove* do ficheiro *Mcts.java.*

```
static int mcts(Board board) {
	return new Mcts().bestMove(board);
}
```

A função *bestMove* vai criar um nó raiz que vai receber a jogada que o jogador fez e depois entra num ciclo que é executado o número de vezes definido em ***TOTAL_ITERACTIONS***. Em cada iteração é selecionada a melhor folha guardada em memória (função *transverseTheTree*) usando a fórmula de seleção, e verifica-se se é estado final. Se não for, é gerado os filhos dessa folha caso seja possível. Depois é escolhido um filho aleatoriamente e faz-se uma simulação a partir desse filho. Quando a simulação chega a um resultado é incrementado o *numberOfVisits* e o *wins* é incrementado com 1 se for um nó do computador cuja simulação deu uma vitória ao computador, ou caso seja um nó do utilizador cuja simulação deu uma vitória para o jogador.

```
int bestMove(Board board) {
	root = new Node(board, Board.RED, null);
	int iteration = 0;
	while (iteration < TOTAL_ITERATIONS) {
		Node leaf = transverseTheTree(root);
		if (!gameIsOver(leaf.getBoard())) {
			generateChildren(leaf);
		}
		Node node = leaf;
		if (leaf.hasChild()) {
			node = leaf.getAnyChild();
		}
		int simulationResult = simulate(node);
		backPropagate(simulationResult, node);
		iteration++;
	}
	return bestChildColumnNumber();
}
```

Nós fizemos este algoritmo com base nas seguintes fontes [4][5][6][7].

Análise

Nós verificamos a performance dos algoritmos a nível de tempo. As seguintes tabelas mostram os dados em 5 jogos diferentes sendo que na primeira coluna tem o número de jogadas que o computador fez nesse jogo e na segunda coluna o tempo médio para cada jogada.

MiniMax

Nº de Jogadas	Média de cada jogada (em ms)
4	41.75
7	34.96
8	22.12
12	17.58
14	21.51

Alpha-Beta

Nº de jogadas	Média de cada jogada (em ms)
4	8.58
6	12.81
8	12.70
12	11.88
16	9.85

MCTS

Nº de jogadas	Média de cada jogada (em ms)
4	70.41
6	66.08
9	62.00
12	55.43
18	47.77

Em conclusão a nível de rapidez *Alpha-Beta* é melhor, pois não tem tantos nós para processar, como *MiniMax*. Já o *Monte Carlo Tree Search* é o pior dependendo do número de iterações que são feitas. Nestes casos TOTAL_ITERACTION tem valor de 1000, logo vai ter que processar mais nós do que os restantes algoritmos.

Nós verificamos quantos nós ficam guardados na memória em cada algoritmo. Tanto o *MiniMax* como o *Alpha-Beta*, no pior caso, guardam ($\sum_{i=0}^{d} b^i$) nós em cada jogada em que *d* diz quantos níveis de profundidade queremos que faça a pesquisa e *b* o fator de ramificação. No nosso programa definimos que *d=5* e o *b =7*, no máximo. Logo tanto *MiniMax* como *Alpha-Beta* vão guardar em cada jogada, no máximo, 2801 nós na memória. Já *MCTS* vai guardar $b \times k + 1$ sendo *b* o fator de ramificação e *k* o número de iterações e o 1 é o nó raíz. No programa é definido $k = 1000$, logo guarda 7001 nós por jogada, no máximo. O algoritmo poderia guardar menos que *MiniMax* e *Alpha-Beta* se definíssemos *k* para um número menor de iterações.

Notas

Este trabalho encontra-se na sua totalidade no GitHub.

Bibliografia

1: Minimax, 2023, https://en.wikipedia.org/wiki/Minimax
2: Stuart Russel, Peter Norvig, Artificial Intelligence A Modern Approach, 2022,
3: Connect Fout, 2023, https://en.wikipedia.org/wiki/Connect_Four
4: int8, Monte Carlo Tree Search – beginners guide, 2018, https://int8.io/monte-carlo-tree-search-beginners-guide/
5: Ankit Choudhary, Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo, 2019, https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/
6: Winning Strategies for Connect 4 or Four in a Line Games, https://www.qnaguides.com/Winning-Strategies-for-Connect-4-or-Four-in-a-Line-Games.html
7: Monte Carlo Tree Search, https://mcts.netlify.app/mcts/