Three Chess

Analysis of ThreeChess agents

Jun Han Yap (22507198)
Jonathan Neo (21683439)

CITS3001 – Algorithms, Agents and Artificial Intelligence
The University of Western Australia
2020

Contents

1 Introduction

ThreeChess is a variation of chess played on a special board between three players, with the colours Blue, Green and Red. Each player takes turns moving their pieces, where the available moves depend on the type of piece. The goal is to capture one of the opponents' Kings. When a King is captured the game ends; the person who took the King is the winner, the person who lost the King is the loser, and the third player neither wins nor loses.

Our goal of this report is to analyse and determine suitable algorithms for agents in ThreeChess. To achieve that, we review algorithms for similar types of problems, implement algorithms which are most suitable, test and validate implemented algorithms, and analyse the results.

2 Literature review

Selecting an algorithm to act as an agent for a problem depends on the characteristics of the problem. The characteristics below should be taken into consideration when deciding which algorithm would suit the problem:

- Number of players
- Cooperative or non-cooperative

In the context of three player chess, the number of players is three, the game is non-cooperative, and only the first person to capture an opponent’s king is the winner.
Based on that, the methods below would be suitable as an agent for three player chess:

1. Monte-Carlo algorithm
2. Search algorithms
    a. MaxN
    b. Paranoid
    c. Best reply

2.1.1 Monte-Carlo algorithm

The Monte-Carlo Tree Search (MCTS) algorithm is a heuristic driven search decision-making algorithm that utilises probability to choose the optimal move in a given situation. In game playing scenarios, it works by building a tree containing nodes that represents states of the game and explores all possibilities while estimating the rewards of the actions. (Palma and Lanzi, 2014)

There are four main steps in MCTS: Selection, Expansion, Simulation and Backpropagation. In the selection step, the algorithm traverses down the tree from the root and selects optimal child nodes using an evaluation function, Upper Confidence Bound (UCB), which returns the estimated value.

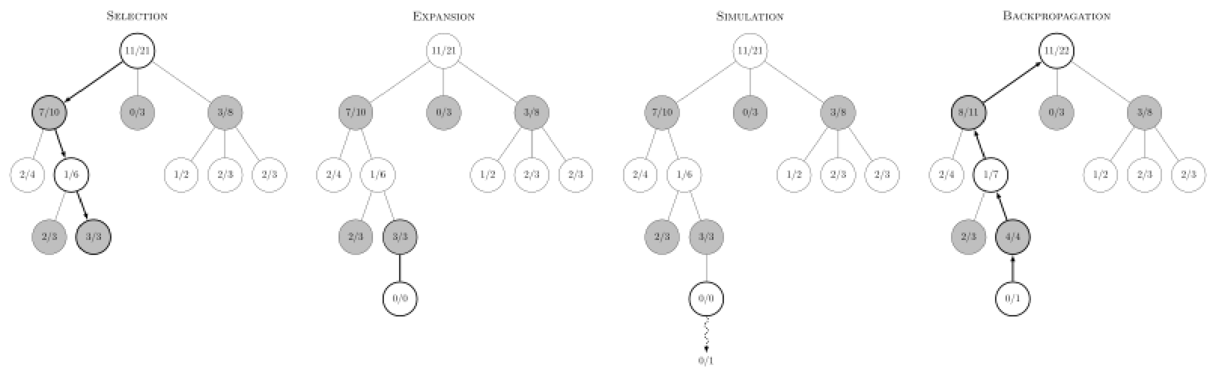
The UCB formula (Kocsis and Szepesva´ri, 2006):

$$\bar{x}_i + c\sqrt{\frac{\ln n}{n_i}}$$

Where $\bar{x}_i$ is the win score, or the ratio of simulations won to the total simulations.
The constant, *c*, controls the exploitation and exploration rate.
n, the number of times the parent node was visited.
n_i, the number of times node *i* was selected.

In the expansion phase, all unvisited child nodes are added to the node that was selected.

In the simulation phase, the game is then played out to the end using random moves.
In the backpropagation phase, the nodes are then visited and its scores added up and updated.



Steps of a Monte-Carlo Tree Search
(https://commons.wikimedia.org/wiki/File:MCTS-steps.svg)

Pseudocode for Monte-Carlo Tree Search function (Rahul, 2019)

```
function monte_carlo_tree_search(root):
   while resources_left(time, computational power):
      leaf = traverse(root)
      simulation_result = rollout(leaf)
      backpropagate(leaf, simulation_result)
   return best_child(root)
end function

function traverse(node):
   while fully_expanded(node):
      node = best_uct(node)
   return pick_univisted(node.children) or node
end function

function rollout(node):
   while non_terminal(node):
      node = rollout_policy(node)
   return result(node)
end function

function rollout_policy(node):
   return pick_random(node.children)
end function

function backpropagate(node, result):
   if is_root(node) return
   node.stats = update_stats(node, result)
   backpropagate(node.parent)
end function

function best_child(node):
    pick child with highest number of visits
end function
```

Advantages
Advantages for MCTS include its heuristic nature, meaning that strategic or tactical knowledge of the game is not needed as it is sufficient to only know its legal moves and end conditions, asymmetric, allowing the spending of computational resources on the most promising areas of the trees, and anytime, allowing the algorithm to stop and return the best possible move at any time.

Disadvantages
The MCTS algorithm has a possibility of failing to find more effective moves for games of even medium complexity. This is because of the size of possible moves and because some nodes may not be visited enough to give good estimates. In all the algorithm may choose to ignore deeper tactical moves as it does not have enough resources.

2.1.2 Search algorithms
In search based algorithms, the algorithm requires a search method and an evaluation function. An evaluation function is a function which estimates the score of the game when either the search depth has reached its limit, or the game has ended.

Three search methods that are commonly cited in research are discussed below.

2.1.2.1 MaxN
MaxN assumes that each player seeks to maximise their score in the game (Luckhart and Irani, 1986). The algorithm achieves this by creating a recursive function maxN(node, depth). The pseudocode for MaxN algorithm is shown below.

Pseudocode for MaxN algorithm

```
function maxN(node, depth)
    if (depth == 0 or node is terminal)
        return evaluate(node)
    maxValues = maxN(firstChild, depth-1)
    ForEach (child of node)
        values = maxN(child, depth-1)
        if (values[playerOnTurn] > maxValues[playerOnTurn])
            maxValues = values
    return maxValues
end function
```

Starting from the root player, the tree is traversed downwards where each node represents a player's legal moves, and each level represents a player's turn. The tree traverses down until it reaches a depth of 0, or the node is terminal (a player's king is captured). When the termination occurs, the evaluation function is called which returns the score of each player. At each level of the tree, each player searches for the move that will maximise their score. Each player's decision is passed back up to the previous player's turn until it reaches back to the root player who will then decide what move they will pick to maximise their score.

Advantages
A more optimistic algorithm as compared to paranoid which assumes that all players are forming a coalition against the root player and could lead to conservative moves.

Disadvantages
Only shallow pruning is possible without speculation and thus time complexity increases (Luckhart and Irani, 1986). Due to time constraints, this would likely lead to lesser search depth as compared to the paranoid algorithm.

2.1.2.2 Paranoid
Paranoid search algorithm assumes that all players form a coalition against the root player (Sturtevant and Korf, 2000). The root player aims maximise their score, and the coalition aims to minimise the root player's score. In essence, this assumption converts a multiplayer game into a two-player game and allows the paranoid algorithm to implement the minimax algorithm (von Neumann, 1928). The pseudocode for minimax (paranoid) algorithm is shown below.

Pseudocode for minimax algorithm (paranoid algorithm)

```
function minimax(node, depth, maxPlayer)
        if (depth == 0 or node is terminal)
                return evaluate(node)
        if (maxPlayer)
                maxValue = −INF
                ForEach (child of node)
                        maxValue = max(maxValue, minimax(child, depth-1, false))
                return maxValue
        if (not maxPlayer)
                minValue = +INF
                ForEach (child of node)
                        minValue = min(minValue, minimax(child, depth-1, true))
                return minValue
end function
```

Starting from the maximising player (also the root player), the tree is traversed down in an alternating manner between the maximising player and the minimising player. The maximising player's maxValue is instantiated with the value of negative infinity initially to find moves that will maximise the root player's score. The minimising player's minValue is instantiated with the value of positive infinity initially to find moves that will minimise the root player's score. When the depth reaches 0, or the node is terminal (a player's king is captured), then the evaluation function is called which returns the root player's score.

Each player makes moves that will achieve the best outcome for them, and pass their decision to the previous player until it is the root player's turn who will then pick the move that will maximise their score.

Advantages
Alpha-beta pruning can be applied to the minimax algorithm which significantly improves time complexity (Knuth and Moore, 1975).

Disadvantages
The paranoid algorithm would likely lead to conservative moves as it assumes that both players have formed a coalition against the root player, which is not the case as three player chess is a non-cooperative game.

2.1.2.3 Best reply

The best reply algorithm assumes that only the opponent with the strongest move against the root player may move (Schadd and Winands, 2011). The pseudocode for best reply algorithm is shown below.

Pseudocode for best reply algorithm

```
function bestReply(node, depth, maxPlayer)
        if (depth == 0 or node is terminal)
                return evaluate(node)
        if (maxPlayer)
                maxValue = -INF
                ForEach (child of node)
                        maxValue = max(maxValue, bestReply(child, depth-1, false))
                return maxValue
        if (not maxPlayer)
                minValue = +INF
                children = getChildrenOfAllMinPlayers(node)
                ForEach (children)
                        minValue = min(minValue, bestReply(child, depth-1, true))
                return minValue
end function
```

Starting from the maximising player (root player), the tree is traversed down in an alternating manner between the maximising player and the minimising player with the highest minimising score.

The maximising player's score is instantiated to negative infinity initially such that the maximising player searches for values that maximise its score. Likewise, the minimising player's score is instantiated to positive initially such that the minimising player searches for scores that minimises the root player's score. Both opponents' scores are evaluated and the player with the strongest move against the root player may move.

When depth reaches 0 or the node is terminal (a player's king is captured), the evaluation function is called which returns the maximising player's score.

The scores are passed up the tree until it reaches the root player who will pick the move that maximises their score.

Advantages

Alpha-beta pruning can be applied to the best reply algorithm which significantly improves time complexity. More max nodes are visited along the search path, leading to more long term planning (Schadd and Winands, 2011).

Disadvantages

Opponent moves that are weaker are skipped, leading to illegal moves (Schadd and Winands, 2011). The Java board class we are using prevents us from implementing this algorithm as moves cannot be skipped.

3 Selected techniques

For our analysis, we have selected algorithms that we believe would yield the highest performance:

Type	Algorithm
Monte-Carlo	Monte-Carlo Tree Search
Search and evaluate	MaxN
Search and evaluate	Paranoid

3.1 Monte-Carlo tree search

3.1.1 Justification

Despite the large branching factor in threeChess, Monte-Carlo tree search (MCTS) is still suitable as the structure of the algorithm can be implemented into threeChess due to the fact that at any given state, moves can be taken which will create new states.

Within a timelimit of 200 milliseconds, the algorithm performs every step of the MCTS to explore all possible nodes with the given time and return the node with the best move.

A value of 1.0 was chosen and assigned to the *c* constant . Other values did not make much of a difference in winrate and thus we decided to remain with 1.0.

Upon testing, we discovered our algorithm missing obvious winning moves in favor of moves that would prolong or even cause the Agent to lose the game. We then decided to hardcode an algorithm that would check for a possible winning move at every state and take it immediately.

There was also an issue in which the Agent would irregularly attempt to perform an illegal move, this issue seemed to only occur in the first move but we could not figure out why. Instead we added another check, before returning the move to play, to ensure that the move is legal and if not, to play a random move instead.

3.2 MaxN

3.2.1 Justification

Despite not being able to apply alpha beta pruning, MaxN is still a suitable algorithm as the logic closely aligns to how players would make decisions in multi-player chess.
MaxN assumes that players are rational and trying to maximise their own payoffs. This assumption is closely matched with how real players would make decisions in three player chess.
MaxN also assumes that players are not in coalition with each other, which is a fair assumption in three player chess as the game does not dictate that players should form coalitions and there can only be one winner.

3.2.2 Evaluation function

For MaxN to yield good performance, an evaluation function that matches human heuristics (decision making) needs to be used. We use a simple evaluation function that mimics a human behaviour of choosing moves that will yield net benefits to the player as shown below in the pseudocode.

Pseudocode for MaxN evaluation function

```
function evaluate(board)
    score = []
    ForEach(player)
        score[player] = getScore(player)
        return score
end function

function getScore(player)
        score = 0
        ForEach(player.Pieces)
                score += pieceValue

        ForEach(player.CapturedPieces)
                score += pieceValue
        return score
end function
```

Each piece is given a value depending on how valuable each piece is in the game of chess. The piece values are shown in the table below.

Piece	Value
King	40
Queen	9
Rook	5
Bishop	3
Knight	3
Pawn	1

3.3 Paranoid

3.3.1 Justification

Although the paranoid algorithm assumes that all opponents form a coalition against the root player which tends to lead to conservative moves, the paranoid has benefits in that it can perform a larger search depth when using alpha-beta pruning (Sturtevant and Korf, 2003).
Therefore, paranoid may outperform MaxN due to the further lookahead (Sturtevant, 2003) and thus we have chosen to analyse the paranoid algorithm.

3.3.2 Evaluation function

For paranoid to yield good performance, an evaluation function that matches human heuristics (decision making) needs to be used. We use a simple evaluation function that mimics a human behaviour of choosing moves that will yield net benefits to the player as shown below in the pseudocode.

Pseudocode for paranoid evaluation function

```
function evaluate(board, maxPlayer)
        if(board.gameOver and board.winner == maxPlayer)
                score = +INF
        elif(board.gameOver and board.winner != maxPlayer)
                score = -INF
        else
                score = getScore(maxPlayer)
        return score
end function

function getScore(player)
        score = 0
        ForEach(playersPieces)
                score += pieceValue

        ForEach(playerCapturedPieces)
                score += pieceValue

        return score
end function
```

Only scores for the maximising player is returned in the paranoid evaluation function.
Each piece is given a value depending on how valuable each piece is in the game of chess. The piece values are shown in the table below.

Piece	**Value**
King	40
Queen	9
Rook	5
Bishop	3
Knight	3
Pawn	1

4 Validation tests and metrics

In this section, we evaluate the performance of our selected algorithms developed in Java JDK 11. The results of the experiments are presented below in separate sections. All experiments we carried out on a Microsoft Windows 10 Pro Operating System (64-Bit), and 32 GB RAM.

For consistency, each experiment is carried out with 20 games and 300 second time limit.

4.1 Baseline experiments

Before we compare the selected algorithms against each other, we first establish a baseline of how each agent performs against a random agent.
The scores for each agent against a random agent are summarised below.

4.1.1 Paranoid vs Random vs Random

Agent	**Specifications**
Paranoid	• Alpha-beta pruning • Depth of 2
Random	• Random legal move
Random	• Random legal move

Win Rates

Time	**Games**	**ParanoidPruning**	**Random**
300	20	55.0%	45.0%

Lose Rates

Time	**Games**	**ParanoidPruning**	**Random**
300	20	20.0%	80.0%

4.1.2 MaxN vs Random vs Random

Agent	**Specifications**
MaxN	• Depth of 2
Random	• Random legal move
Random	• Random legal move

Win Rates

Time	**Games**	**MaxN**	**Random**
300	20	50.0%	50.0%

Lose Rates

Time	**Games**	**MaxN**	**Random**
300	20	20.0%	80.0%

4.1.3 MCTS vs Random vs Random

Agent	**Specifications**
MCTS	• 200ms timelimit

Random	• Random legal move
Random	• Random legal move

Win Rates

Time	Games	MCTS	Random
300	20	85.0%	15.0%

Lose Rates

Time	Games	MCTS	Random
300	20	10.0%	90.0%

4.2 Two agent comparison experiments

After establishing a baseline, we play two selected agents against a random agent. Summarised below are the results.

4.2.1 MaxN vs Paranoid vs Random

Agent	Specifications
MaxN	• Depth of 2
Paranoid	• Alpha-beta pruning • Depth of 2
Random	• Random legal move

Win Rates

Time	Games	MaxN	ParanoidPruning	Random
300	20	20.0%	30.0%	50.0%

Lose Rates

Time	Games	MaxN	ParanoidPruning	Random
300	20	20.0%	30.0%	50.0%

4.2.2 MCTS vs MaxN vs Random

Agent	Specifications
MCTS	• 200ms timelimit
MaxN	• Depth of 2
Random	• Random legal move

Win Rates

Time	Games	MCTS	MaxN	Random
300	20	70.0%	20.0%	10.0%

Lose Rates

Time	Games	MCTS	MaxN	Random
300	20	0.0%	45.0%	55.0%

4.2.3 MCTS vs Paranoid vs Random

Agent	Specifications

MCTS	• 200ms timelimit
Paranoid	• Alpha-beta pruning • Depth of 2
Random	• Random legal move

Win Rates

Time	Games	MCTS	ParanoidPruning	Random
300	20	80.0%	5.0%	15.0%

Lose Rates

Time	Games	MCTS	ParanoidPruning	Random
300	20	15.0%	45.0%	40.0%

4.3 Three agent comparison experiments

Finally, we compare the three agents we have selected. The results are summarised below.

4.3.1 MCTS vs Paranoid vs MaxN

Agent	Specifications
MCTS	• 200ms timelimit
Paranoid	• Alpha-beta pruning • Depth of 2
MaxN	• Depth of 2

Win Rates

Time	Games	MCTS	ParanoidPruning	MaxN
300	20	65.0%	20.0%	15.0%

Lose Rates

Time	Games	MCTS	ParanoidPruning	MaxN
300	20	30.0%	35.0%	35.0%

5 Analysis of agent performance

Playing against the random agent, Monte-Carlo Tree Search (MCTS) performs the best out of all agents we have implemented with a 85% win rate. Paranoid follows with a 55% win rate and then MaxN with a 50% win rate. When playing against each other, the MCTS surpasses the other 2 with a 65% win rate with Paranoid and MaxN having a 20% and 15% win rate respectively.

In theory, Paranoid is able to look further compared to MaxN due to pruning but we set a depth limit of 2 due to tournament time constraints. MaxN works by assuming that the opponents it plays against will be choosing moves that maximises their benefits. Therefore, its suboptimal win rate can be explained by agents, such as the random agent, not choosing moves that would result in an optimal position.

Paranoid and MaxN can be characterised as more conservative or 'safe' algorithms as they assume that their opponents will be choosing the best available moves to play against them. Hence, even though the win rate for the aforementioned algorithms are low, their lose rate is also relatively low.

For example, both Paranoid and MaxN have a 55% and 50% win rate respectively against the Random Agents, but both have a 20% lose rate.

The Monte-Carlo Tree Search algorithm seemingly performs the best out of all algorithms with a 65% and above win rate while also have an optimal lose rate of 30% and below throughout all tests. The MCTS also has no evaluation function which assumes the motives of its opponents, rather it randomly simulates all moves and chooses moves with a high visit and win rate. The algorithm's high win can be accredited to its aim to perform moves that will result in or have a high chance of it winning. Its low lose rate can be attributed to the fact that the MCTS algorithm will usually perform moves that have a high win rate and therefore will seek to perform moves that will result in wins, an aggressive technique and therefore it seeks to win the game before it gives its opponent's the opportunity to win. Therefore, the algorithm's win and lose rate can be explained by its lack of care regarding the game's mechanics, strategies or the opponents' methods.

6 Conclusion

In conclusion, the Monte-Carlo Tree Search is the most efficient algorithm out of the 3 we implemented. However, improvements could definitely be made to all algorithms as workarounds were implemented, such as the 'Instant Win' algorithm, to ensure that they perform more effectively.

7 References

Kocsis, L., & Szepesva´ri, C. (2006). Bandit based Monte-Carlo Planning. Bandit Based Monte-Carlo Planning, 3–5. https://doi.org/10.1007/11871842_29

Luckhardt, C. A., & Irani, K. B. (1986). An algorithmic solution of N-person games. AAAI'86: Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, 158–162. https://dl.acm.org/doi/10.5555/2887770.2887795#sec-ref

Palma, S. D., & Lanzi, P. L. (2014). Monte Carlo Tree Search algorithms applied to the card game Scopone. Monte Carlo Tree Search Algorithms Applied to the Card Game Scopone, 11–18. https://teaching.csse.uwa.edu.au/units/CITS3001/project/2017/paper1.pdf

Von Neumann, J. (1928). "Zur Theorie der Gesellschaftsspiele". Math. Ann. 100: 295–320. doi:10.1007/BF01448847

Rahul, R. (2019, January 14). ML | Monte Carlo Tree Search (MCTS). GeeksforGeeks. https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Simons S. (1995) Minimax Theorems and Their Proofs. In: Du DZ., Pardalos P.M. (eds) Minimax and Applications. Nonconvex Optimization and Its Applications, vol 4. Springer, Boston, MA. https://doi.org/10.1007/978-1-4613-3557-3_1

Sturtevant N. (2003) A Comparison of Algorithms for Multi-player Games. In: Schaeffer J., Müller M., Björnsson Y. (eds) Computers and Games. CG 2002. Lecture Notes in Computer Science, vol 2883. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-40031-8_8

Sturtevant, N. R., & Korf, R. E. (2000). On Pruning Techniques for Multi-Player Games. Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, 201–207. https://dl.acm.org/doi/10.5555/647288.721604

Schadd, M. P. D., & Winands, M. H. M. (2011). Best Reply Search for Multiplayer Games. IEEE Transactions on Computational Intelligence and AI in Games, 3(1), 57–66. https://doi.org/10.1109/tciaig.2011.2107323

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326. https://doi.org/10.1016/0004-3702(75)90019-3

Sturtevant, N.R., & Korf, R. (2003). Multiplayer games: algorithms and approaches.