**ABSTRACT**

This paper demonstrates the creation, testing and performance evaluation of a game Connect-Four with the help of three state-of-the art AI algorithms namely Minimax algorithm, Expectimax Algorithm with alpha-beta pruning and Depth-Limited Search for improved performance and Monte Carlo Tree Search algorithm. Connect-Four is a familiar and well-known board game whose objective is to get four slots in a row.

**INTRODUCTION**

Besides constructing intelligent systems, Artificial Intelligence is also the endeavour to understand them. It includes an assortment of applications ranging from general purpose areas to particular tasks like proving mathematical theorems and diagnosing diseases. Game playing is one of the oldest fields of ventures in AI. General Game Playing (GGP) is the composition of AI programs’ ability to play games successfully.

Connect-Four a two-player game, where the objective of the game is to win by making four or more discs of your colour in a line horizontally, vertically or diagonally. The board is fashioned with six rows and seven columns where the discs are dropped vertically down occupying the last available slot within that respective column by taking turns. Although it is a solved game, it has 10^13 possible positions on the board making it absurd and illogical to store the moves in memory. As ardent players of board games and an inclination towards working on Artificial Intelligence, in this paper, we aim to depict and illustrate the different ways the AI can be utilized to create gameplay agents for the Connect-Four board game. Agents in AI are entities which act and directs its movements towards accomplishing goals. However, goals alone are insufficient to develop high-quality behaviour. Utility based agents comprise of a utility function that maps a state onto a real number representing the associated degree of happiness (1). For the agent to achieve the goal, it needs to be equipped with the capability of knowing the states, possible actions, transition model, goal test and path cost. The Connect-Four environment (3) comprises of properties as illustrated in figXXX.

|  |  |
| --- | --- |
| Properties of Environments | Connect-Four environment |
| Accessible vs Inaccessible | Accessible – It is completely observable since it consists of the board with constant dimensions and discs belonging to either the player or opponent. |
| Deterministic vs Non-Deterministic | Deterministic – It is considered deterministic since no random elements are involved. |
| Static vs Dynamic | Static – Since the present state cannot be altered be changed until the agent performs a move it is static. |
| Discrete vs Continuous | Discrete – Since the number of states is finite it is discrete. |
| Episodic vs Non-Episodic | **DOUBT** |

There are many approaches that can be employed to solve the Connect-Four game based on the difficulty level set for the AI. As per (2) they are random, defensive and aggressive AI. Random AI as the name suggests randomly performs an action and is easiest to beat, Defensive AI prioritizes to oppose and block the opponents moves from winning and Aggressive AI gives precedence to only winning.

Connect-Four makes a good candidate for implementing and studying search algorithms to find the best possible solution from all the possible solutions. This paper demonstrates the creation, testing and performance evaluation Connect-Four with the help of three state-of-the art AI algorithms namely Minimax algorithm, Expectimax Algorithm with alpha-beta pruning and Depth-Limited Search for improved performance and Monte Carlo Tree Search algorithm. In this paper, we aim to demonstrate a variation to the program with a mixture of ways to select the type of players to play and solve the game. The possible combinations of players are Human versus Human, Human versus Agent and Agent versus Agent. Our experiment involves five types of agent-players namely: Agent1 – Random, Agent2 – Forward checking, Agent3 – Minimax algorithm with alpha-beta pruning and depth-limited search, Agent4 – Expectimax algorithm with alpha-beta pruning and depth-limited search and Agent5 – MCTS.

Search strategies are evaluated by measuring their effectiveness of performance in solving the problem. Ideally, effectiveness of a search is calculated using the search cost associated with the time and memory consumed to obtain the solution. For the Connect-Four game, we chose to record the time and number of moves taken by both the players.

The rest of the paper is organized as follows: Section 2 mentions the Related Work on approaches used to solve the problem, Section 3 comprises of the Problem Definition and Algorithms implemented, Section 4 describes the Experimental Results obtained by the utilised methodology and lastly, Section 5 includes the Conclusions and final discussions of the main results along with an acknowledgement of future work.

In this section you will discuss possible approaches to solve the problem you are addressing, justifying your choice of the 3 you have selected to evaluate. Also, briefly introduce the approaches you are evaluating with a specific emphasis on differences and similarities to the proposed approach(es).

**RELATED WORK**

Game playing is an important area of artificial intelligence since they are interesting means in comparing interaction between user and machine’s behaviour directly. Once the rules, permissible moves and conditions to succeed or lose the game are acknowledged search procedures can be implemented to see good moves to be generated and explored first. Game theory broadens decision theory to circumstances in which multiple agents interact. There has been extensive research performed in understanding different approaches to solving two-player games, strategical and turn-taking games and tactical adversarial games like Backgammon, Connect-Four, Othello and Chess. Typically, these games are modelled as decision trees using different AI algorithms. The general notion of characterizing and assessing the quality of a game is by playing the game with various controllers and algorithms and evaluating the performance through comparisons. As per (6), by investigating the relative performance of different general game-playing algorithms results suggest a significant positive relationship between intelligent agents, game design and success rate of the algorithm. In fact, these parameters differentiate between good and bad game playing algorithms on a well-designed game with the restriction to human-designed games.

In (5), provides literature based on current challenges faced in real-time strategy games. The paper describes the comparative analysis conducted on different AI based machine learning techniques like naïve Bayes classifier and support vector machines to evaluate the performance issues faced trying to achieve optimality. Comparison between the techniques was helpful and utilized in designing the characters of video games.

In (10), paper illustrates the usage of dynamic scripting to provide actions in a turn-based strategic game. The proposed algorithm incorporates rule ordering of dynamic scripting through reinforcement learning combined with minimax algorithm to obtain better performance. The performance is evaluated through a series of matches against a manually designed static AI.

In (8), demonstrates the use and evaluation of Minimax Tree Optimisation (MMTO) algorithms’ performance in shogi to learn a heuristic evaluation function of a practical alpha-beta search program. This study involves a depth search where it repetitively searches and updates using partial derivatives. The paper concludes that MMTO outperformed existing methods and can adapt to forty million parameters. **(can be removed).**

In (9), paper proposes MCTS-minimax hybrids that integrate shallow minimax searches into the MCTS framework. It investigates ways to combine the strategic toughness of MCTS with the tactical strength of minimax to exercise better results from universally useful hybrid search algorithms. The paper explores its effectiveness on games like Connect-Four with the objective to relate their performance to the strategies of the domains.

In (7), presents a study involving the Connect-Four game in a real-time environment with incorporation of time restraints. The paper uses an artificial intelligence based on influence mapping, minimax, minimax with alpha-beta pruning and A\* to evaluate its performance with time constraints.

Inferences from above papers

6 – using 3 different algorithms to analyse the performances and identify the best among them since there is a relationship between the results and agent incorporated with types of algorithms.

10 – The performance was evaluated through series of matches.

8 – uses MMTO with alpha-beta pruning on shogi. We aim to attempt the same on Connect-Four.

**PROBLEM DEFINITION AND ALGORITHM**

Connect-Four being a two-player mathematical 6x7 board game, it can be solved strategically using a mathematical solution. FigXXX depicts the factors that are to be defined to formulate the problem.

|  |
| --- |
| Factors – as a diagram |
| Initial State |
| States |
| Actions |
| Transition Model |
| Goal Test |
| Path Cost |

From a plethora of artificial intelligence algorithms available, a mixture of brute-force techniques and knowledge-based approaches can be utilised to solve the game. There are a few conclusions resolved from solving the game for instance, the first player has a higher probability of winning if the game is started by placing the disc in the middle column. However, this highly depends on the agent’s capacity to look ahead based on the algorithm imposed on it. The layout of the Connect-Four game and design flow of the game is shown in figVVV and figBBB respectively.

**(figures of how the empty connect 4 board game looks)**

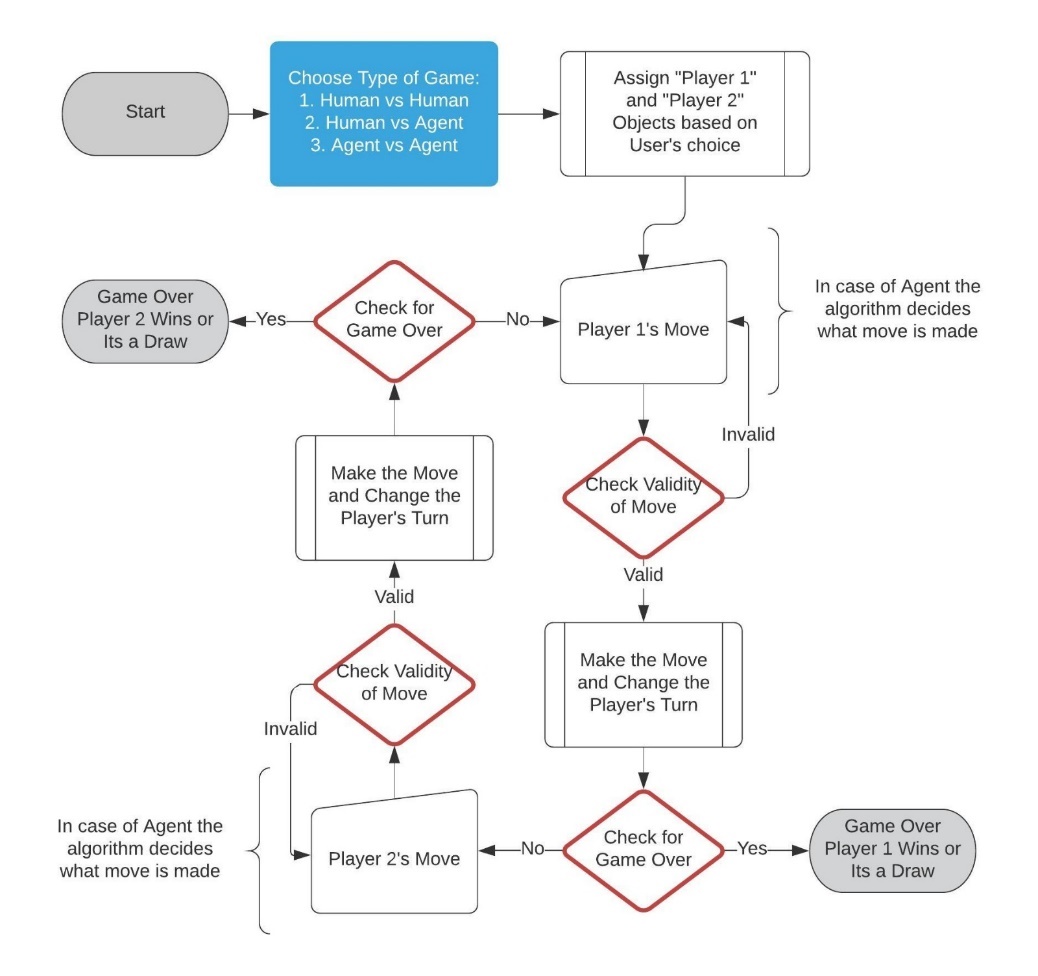


Fig BBB

Apart from a random and forward checking agent for naïve playing, we demonstrate three other agents respectively equipped with three state-of-the-art artificial algorithms Minimax, Expectimax and Monte Carlo Tree Search algorithm by formulating different combinations of games to be played against each other. (10) Their performance is evaluated through series of matches against each other and calculating the time and number of moves made by each of the players.

**3.1 Minimax Algorithm**

Minimax, a decision-making algorithm is typically used in turn-based two-player games aiming to obtain the optimal next move. The general terms of the algorithm are maximiser, generally being the agent and minimizer, the agent’s opponent. The algorithm is based on a zero-sum game theory concept where “the total utility score is divided among the players. An increase in one player’s score results in decrease in another player’s score” (11). This recursive algorithm prepares an optimal move for the agent when always assuming that the opponent plays optimally. To achieve this, the agents looks ahead by using depth-first search for all feasible moves and calculates the resultant score. However, it is not feasible to explore the entire game tree for Connect-Four and hence we opt to backtrack the score using depth-limited search in place of depth-first search which imposes a cut off on the maximum depth of a path. In this paper, we look ahead by 5-6 moves to determine an intermediate score in the “evaluation” function for which the pseudocode is depicted below.

def evaluate\_window(window):

score = 0

if window.count(agents\_coin) == 4:

score += 100

elif window.count(agents\_coin) == 3 and window.count(EMPTY) == 1:

score += 5

elif window.count(agents\_coin) == 2 and window.count(EMPTY) == 2:

score += 2

if window.count(opponent\_coin) == 3 and window.count(EMPTY) == 1:

score -= 4

return score

def evaluate(state)

score = 0

## Score center column

score += state.center\_column.count(agents\_coin) \* 3

## Score Horizontal

for window in state.all\_possible\_horizontal\_windows:

score += evaluate\_window(window)

## Score Vertical

for window in state.all\_possible\_horizontal\_windows:

score += evaluate\_window(window)

## Score positive sloped Diagonal

for window in state.all\_possible\_pos\_sloped\_windows:

score += evaluate\_window(window)

## Score negative sloped Diagonal

for window in state.all\_possible\_neg\_solved\_windows:

score += evaluate\_window(window)

return score

Pruning is a technique implemented in search algorithms to reduce the length of the tree by eliminating the sections that are significantly negligent in terms of accuracy or efficiency. Therefore, we apply a pruning method to limit the number of nodes that the agent needs to examine when looking ahead. Alpha-beta pruning, a technique that can be applied to any depth of a tree is implemented on the standard minimax algorithm to obtain optimal moves. Below code depicts the pseudocode for the minimax algorithm.

def minimax(state, depth, alpha, beta, maximizingPlayer):

valid\_moves = state.get\_valid\_moves()

is\_terminal = state.is\_terminal\_node()

if depth == 0 or is\_terminal:

if is\_terminal:

if state.check\_win(agent):

return (None, 100000)

elif state.check\_win(opponent):

return (None, -100000)

else: # Game is over, no more valid moves

return (None, 0)

else: # Depth is zero

return (None, evaluate(state))

if maximizingPlayer:

utility = -math.inf

best\_move = random.choice(valid\_moves)

for move in valid\_moves:

state.agent\_makes\_move(move)

new\_utility = minimax(state, depth-1, alpha, beta, False)[1]

if new\_utility > utility:

utility = new\_utility

best\_move = move

alpha = max(alpha, utility)

if alpha >= beta:

break

return best\_move, utility

else: # Minimizing player

utility = math.inf

best\_move = random.choice(valid\_moves)

for move in valid\_moves:

state.simulate\_opponent\_move(move)

new\_utility = minimax(state, depth-1, alpha, beta, True)[1]

if new\_utility < utility:

utility = new\_utility

best\_move = move

beta = min(beta, utility)

if alpha >= beta:

break

return best\_move, utility

**3.2 Expectimax Algorithm**

There are games that mirror the unpredictable external events by including a random element. Expectimax, a variant of minimax is an algorithm commonly used in zero-sum games where the outcome depends on the probability of the external element along with the player’s skill. It generalises the games where transitions between states are made probabilistic. It uses a predefined opponent strategy to treat opponent decision nodes as chance nodes (1). In other words, the opponent generally being the minimizer in minimax is replaced with the role of a player whose moves depend on the probability element which is played by chance. These are no longer definite with a minimax value and calculated as an average or expected value. Below code depicts the pseudocode for the minimax algorithm.

else:

utility = math.inf

best\_move = random.choice(valid\_moves)

for move in valid\_moves:

state.simulate\_opponent\_move(move)

new\_utility = expectimax(state, depth-1, alpha, beta, True)[1]

if new\_utility <= utility:

utility = new\_utility

best\_move = move

beta = utility/no\_of\_valid\_moves

if alpha >= beta:

break

return best\_move, utility

Apart from the probabilistic element, the algorithm works similar to minimax and hence we apply the same depth-limited search and alpha-beta pruning technique to improve the performance of the algorithm.

**3.3 Monte Carlo Tree Search Algorithm**

Monte Carlo Tree Search (MCTS) is a deeply utilized heuristic search algorithm known for incorporating an element of randomness into the search for variety of games. It was highly recommended as a search and planning structure for identifying optimal results in a given situation. This is demonstrated and confirmed because Monto Carlo rollouts of games gives it a strategic advantage over traditional depth-limited searches. Upper Confidence Bounds for Trees (UCT), a manifestation of MCTS was considered revolutionary for game-playing agents in artificial intelligence (4). The expression for UCT is depicted below from which the highest value calculated is utilised in decision making models.

Where,

* *wi* – number of wins for the node after the ith move
* *ni* – number of simulations for the node after the ith move
* *Ni* – total number of simulations after the ith move run by the parent node
* *c* – the exploration parameter, chosen as for the game.

Pseudocode for UCT:

# returns child with largest UCT value

def selection():

uct\_value = lambda node: node.wins / node.visits + sqrt(2 \* log(parent\_node.visits) / node.visits)

# applies the above uct\_value lamba function to all the child nodes (children)

return sorted(children, key = uct\_value)[-1]

The MCTS planning methodology incrementally constructs an asymmetric search tree guided in most assuring direction which is estimated iteratively and enhances its performance with an increase in the number of iterations ie., it samples moves rather than considering all legal moves from a given state (4). An MCTS iteration usually consists of four consecutive phases: *(1)Selection* where the tree choses the node with largest UCT value, in other words the node which possess the best chance of winning out of all, *(2)Expansion* into new child nodes of the tree after randomly choosing the node, *(3)Rollout* where simulations are conducted on the expanded nodes to review the status of the game - win, loss or tie, *(4)Backpropagation* updates the results obtained in the rollout step along the chosen path like number of wins for the selected node, total number of simulations conducted on the selected node and parent node. Fig XXX depicts a working example of Connect-Four game by following the steps mentioned for the MCTS iteration which is followed by the pseudocode of the above mentioned MCTS steps.

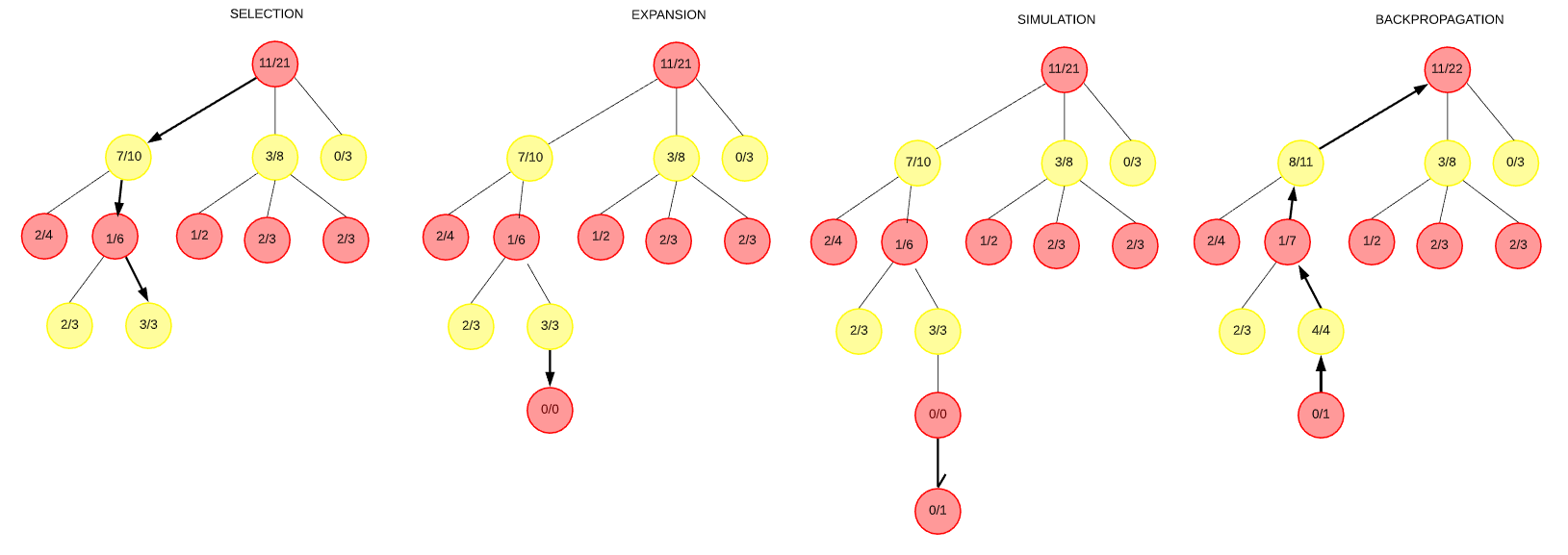


Fig XXX

Pseudocode of MCTS:

def montecarlo\_tree\_search(currentNode, state):

rootnode = Node(state, PREV\_PLAYER)

if currentNode is not None:

rootnode = currentNode

for i in range(max\_iterations):

node = rootnode

# selection

while node.available\_moves == [] and node.children != []:

node = node.selection()

player\_makes\_move(state, CURR\_PLAYER, node.move)

# expand

if node.available\_moves != []:

move = random.choice(node.available\_moves)

player\_makes\_move(state, CURR\_PLAYER, move)

node = node.expand(move)

# rollout

while get\_available\_moves():

move = random.choice(get\_available\_moves())

player\_makes\_move(state, CURR\_PLAYER, move)

if check\_player\_win(PREV\_PLAYER):

break

# backpropagate

while node is not None:

node = node.parent

if timeout:

break

win\_ratio = lambda node: node.wins/node.visits

sorted\_children = sorted(rootnode.children, key = win\_ratio)[::-1]

return rootnode, sorted\_children[0].move

**REFERENCES**

1 - Artificial Intelligence A Modern Approach

2 - Real-Time Connect 4 Game Using Artificial Intelligence

3 - <https://0xadada.pub/2003/12/15/connect-four-playing-ai-agent/>

4 – On Monte Carlo Tree Search and Reinforcement Learning

5 – A comprehensive review on artificial intelligence based machine learning techniques for designing interactive characters.

6 – General video game evaluation using relative algorithm performance profiles.

7 – Real-Time Connect 4 game using Artificial Intelligence\*

8 – Large-scale optimization for evaluation functions with minimax search

9 – MCTS-Minimax Hybrids

10 – Minimax Guided reinforcement learning for turn-based strategy games

11 - <https://en.wikipedia.org/wiki/Zero-sum_game>