

**KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY**

**Department of Computer Science and Engineering**

CSE 4106: Artificial Intelligence Laboratory

AI Project: **Crown Clash**

**Date of Submission:** 04 September, 2024

|  |  |
| --- | --- |
| **Submitted By** | **Submitted To** |
| Shrabanti Debnath  (1907094)  Rubaiya Raktin Raha  (1907111)  Year: 4th , Semester: 1st  term | Md. Shahidul Salim  Lecturer  Department of Computer Science and Engineering  Most. Kaniz Fatema Isha  Lecturer |

**Contents**

1. Objective
2. Introduction
3. Project Overview
4. Methodology
5. Algorithm Description
6. Programming Environment and Language
7. Game Evaluation
8. Outcomes
9. Limitations
10. Future Scope
11. Discussion
12. Conclusion
13. References

**Objectives**

* To develop "Crown Clash" a Python-based strategy game featuring a grid-based battleground with unique movement and capture rules inspired by classic board games.
* To implement an AI opponent with difficulty levels utilizing minimax algorithm, alpha beta pruning, fuzzy logic and genetic algorithm.
* To design adaptive game mechanics that adjust AI strategy based on the chosen difficulty.
* To optimize AI decision-making to ensure a challenging yet fair experience for players.

**Introduction**

"Crown Clash" is an innovative Python-based strategy game where a player has to outsmart an AI opponent in a grid battleground. Being inspired by the most classic board games (checkers & chess), in "Crown Clash," there exists an interesting modification of the rules where each piece-the soldier, the queen, and the king-moves and captures uniquely. Meanwhile, the strategic depth of this game is further enhanced by an AI, which is adaptable, with a difficulty level ranging from a genetic algorithm opponent to a very strong AI opponent using fuzzy logic in combination with the minimax algorithm. The goal is simple and yet powerful: to outsmart and defeat AI at every level by capturing all of your opponent's pieces for the crown.

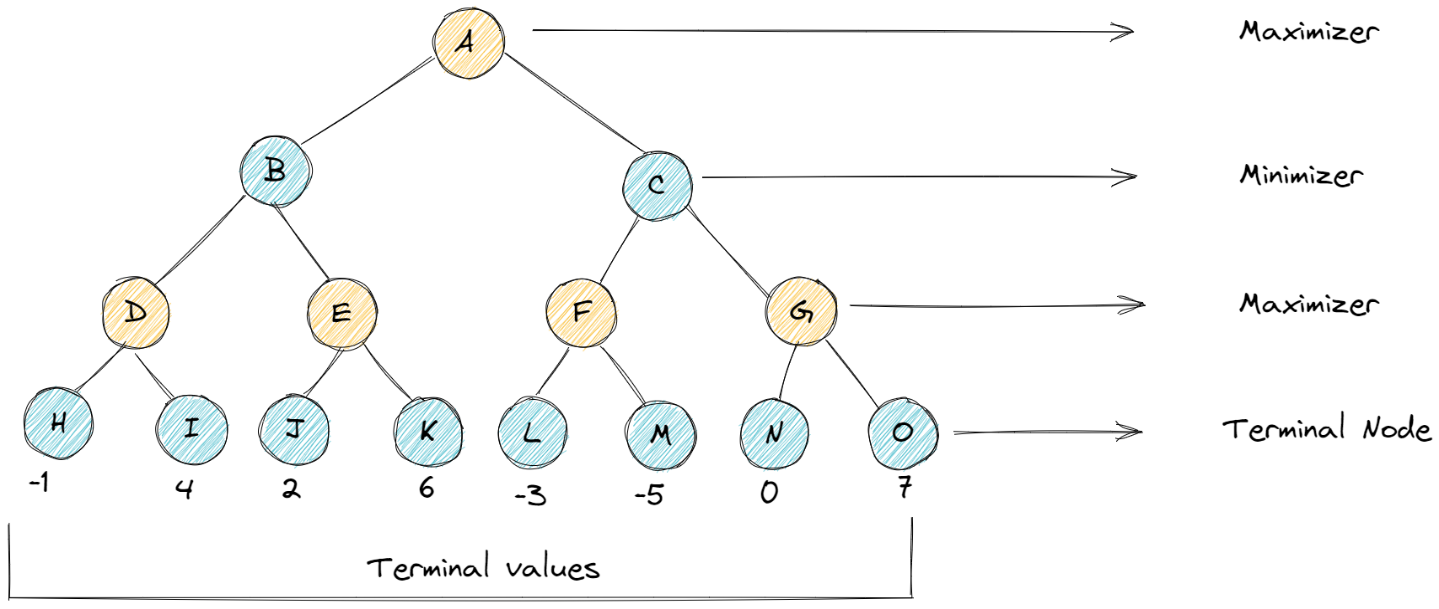
**Project Overview**

* Players compete to complete specific grid patterns on a NXN grid where N=9.
* The game is played between a human player and an AI agent.
* The game has four difficulty levels: Easy, Medium, Hard and Very Hard.
* In the **Easy** level, the AI agent uses genetic algorithm with minimax to select its moves.
* In the **Medium** level, the AI agent uses Minimax algorithm to select its moves.
* In the **Hard** level, the AI agent employs Alpha Beta pruning to optimize its decisions.
* In the **Very Hard** level, the AI agent uses Fuzzy logic to enhance it’s moves.
* Players and the AI take turns, and the first one to capture all the opponent’s pieces wins.
* The AI’s strategy adapts based on the level, providing a progressively challenging experience for the player.

**Theory:**

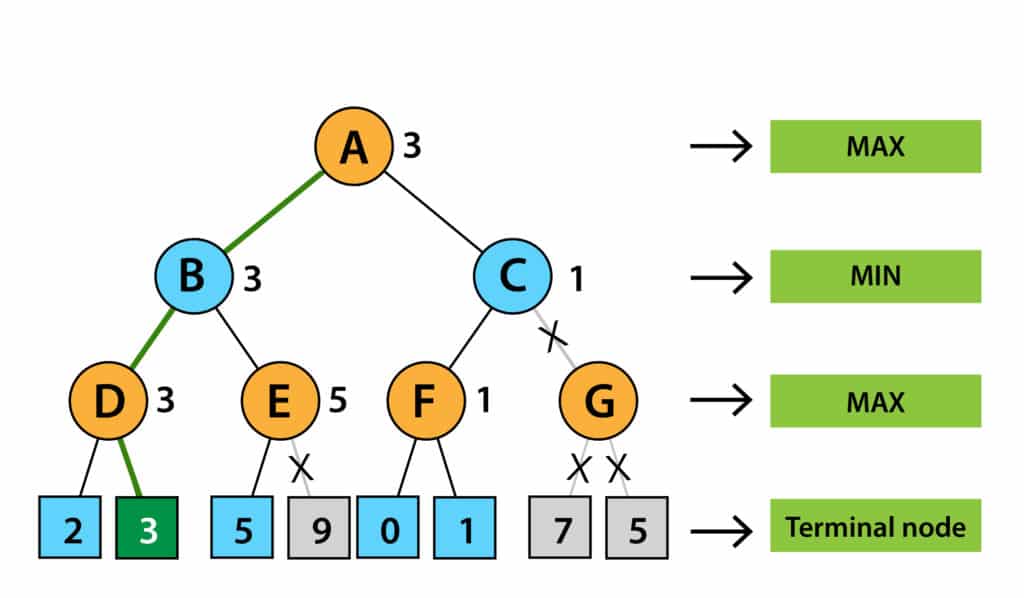
**Minimax Algorithm:**

Minimax: an algorithm in game theory and decision making, mostly within two-player turn-based games-chess or checkers. It simulates all the moves of both players; one player is assumed to maximize, while another player tries to minimize the score of the first player. This algorithm recursively considers a game tree and makes an optimal move on the basis of a worst-case scenario for the performing player who is doing maximization. It generally applies along with alpha-beta pruning that, by eliminating branches not necessary, brings efficiency in it. Minimax basically forms the backbone for making intelligent-playing strategic game AIs.



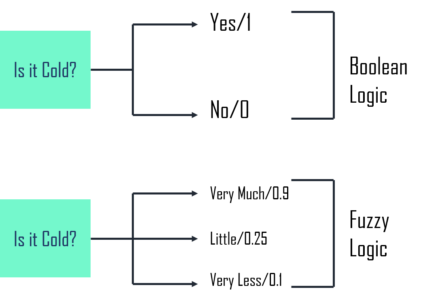
**Alpha-Beta Pruning:**

Alpha-beta pruning reduces the number of nodes to be assessed by the minimax algorithm. Then, the branches which will not affect the final decision are pruned. The alpha-beta technique uses two values, alpha, and beta, to decide at what level to cut branches. Thus, it can go deeper in a search without excessive computation. It keeps the same final decision as before but now is fastened.



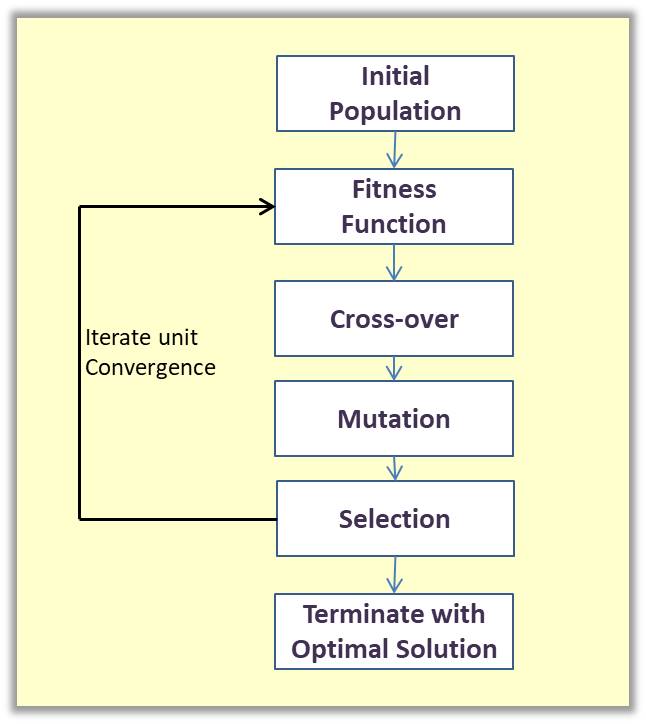
**Fuzzy Logic:**

Fuzzy logic is, therefore, a methodology in reasoning that involves uncertainty or inexact information. It models human-like reasoning in which the truth values are allowed to range in degree between 0 and 1. It has been applied in several control systems and AI for handling situations involving imprecise input data so as to allow smoother and more adaptive operations. It relies on the representation by membership functions for the fuzzy sets and on rules in order to draw conclusions from these fuzzy values. This becomes particularly useful when the system in which one is dealing is very complex and binary logic can be too rigid for it.



**Genetic Algorithm:**

Genetic algorithms represent search algorithms of an optimizing nature, with their original inspiration from natural selection. The method of solving a problem iteratively evolves a population of candidates into better solutions through operations of selection, crossover, and mutation. Solutions are evaluated through the subject of a fitness function, where the best individuals are more likely to pass their genetic material to the next generation. In time, the algorithm converges to an optimum or near-optimum solution. Especially, genetic algorithms are helpful in complex problems for which the more traditional methods cannot find an efficient solution.



**Methodology**

1. **Game Setup:**
2. **How to play:**

As this is a two players game, there are two colour of pieces –

* White (AI)
* Red (Human)

There are three types of pieces in this game:

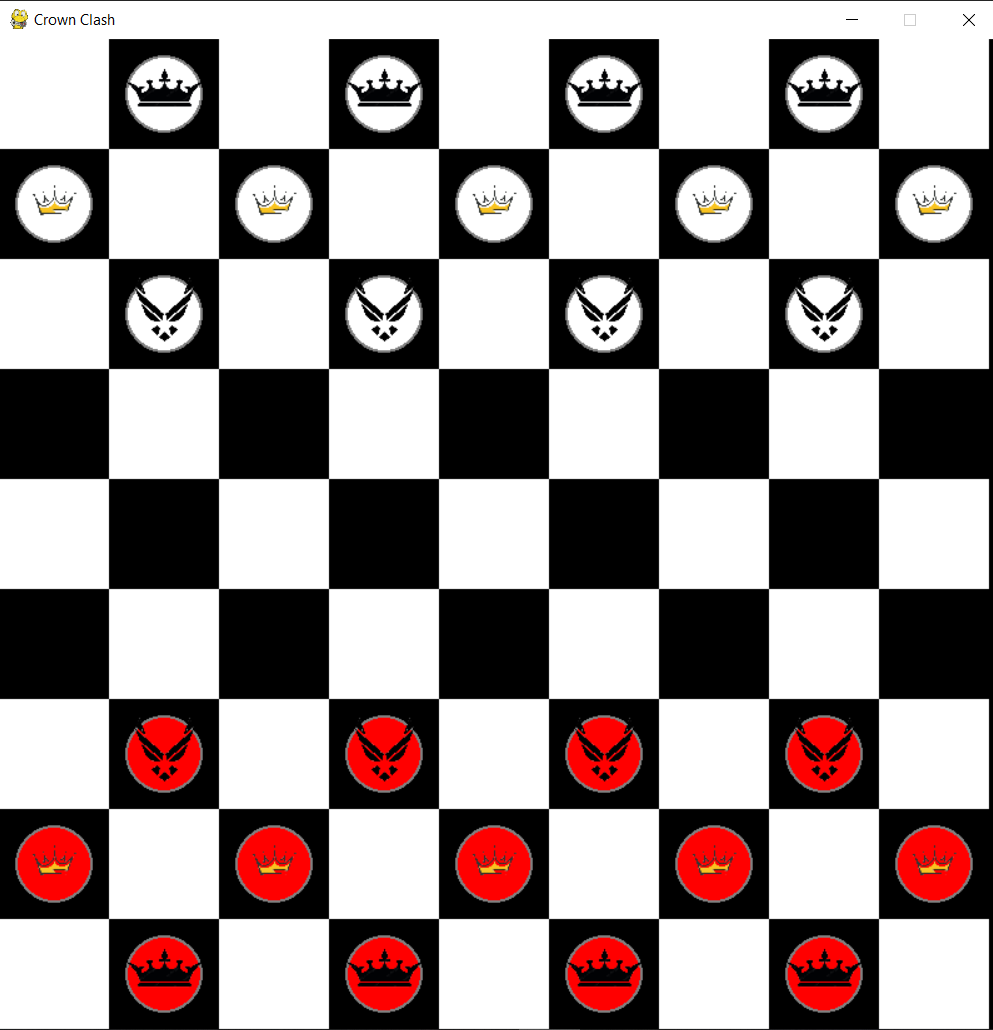
Fig. Queen Pieces Soldier Pieces

King pieces

1. **Board Setup:**

There are 13 pieces available for each player in the 9\*9 board.

****

Game board & piece placements

1. **Game Rules:**

|  |  |
| --- | --- |
| 1. Soldier can move forward 1 cell, and capture opponent pieces in front of it and replaces it in that cell. |  |
| 1. Queen can move up, left, right 1 cell, capture and replace opponent’s piece on that cell. |  |
| 1. King can move up down left right 1 cell,capture and replace opponent’s piece on that cell. |  |
| 1. When a soldier or queen reaches the last cell of opponent’s side, it becomes king. |  |

1. **Algorithm Implementation on Different level:**
   1. **Easy level ( Genetic with MiniMax ):**
   2. **Very Hard level (Fuzzy logic):**

1. Setup of Fuzzy Logic Variables:

* piece\_value: Represents the value of a piece (soldier, queen, king).
* game\_phase: Represents the current phase of the game (early, midgame, late, etc.).
* move\_strength: Represents the strength of a move (how good or strong the move is).

2. Membership Functions:

* Membership functions define how input values (e.g., piece\_value, game\_phase) map to fuzzy sets (e.g., soldier, queen, king for piece\_value).
* They use triangular membership functions (fuzz.trimf) to create overlapping ranges for the variables.

3. Fuzzy Rules:

* Rules combine piece\_value and game\_phase to determine move\_strength.
* The rules prioritize aggressive moves, especially those involving captures (e.g., giving a strong move strength if a soldier is in the early game phase).

4. Fuzzy Control System:

* A fuzzy control system is created (move\_ctrl) based on the defined rules.
* This system is simulated (move\_simulation) to compute the move strength for given inputs.

5. Calculate Fuzzy Move Strength:

calculate\_fuzzy\_move(board, row, col):

* Retrieves the piece at a specific board location.
* Assigns a piece\_value based on the type of the piece (soldier, queen, king).
* Calculates the game\_phase based on the number of pieces left on the board.
* Inputs these values into the fuzzy system to compute a move\_strength.
* It then evaluates all possible moves for that piece and adjusts their strengths, especially boosting moves that involve capturing an opponent's piece.

6. Simulate Move:

Simulate\_move (piece, move, board, skip):

* Simulates a move on a copy of the board, potentially removing skipped (captured) pieces.

7. Determine Best Move:

determine\_best\_fuzzy\_move(board):

* + Iterates over all pieces on the board.
  + For each piece, calculates the possible moves and their strengths using the fuzzy system.
  + Chooses the move with the highest strength and simulates it on a deepcopy of the board.
  + Returns the board state after the best move.

**Algorithm Description**

**1. Minimax Algorithm Implementation**

import deepcopy

import pygame

define colors:

red = (255, 0, 0)

white = (255, 255, 255)

function minimax(position, depth, max\_player, game):

if depth equals 0 or position has a winner:

return position.evaluate(), position

if max\_player:

set maxEval to negative infinity

set best\_move to None

for each move in get\_all\_moves(position, white, game):

evaluation = minimax(move, depth-1, false, game)[0]

set maxEval to the maximum of maxEval and evaluation

if maxEval equals evaluation:

set best\_move to move

return maxEval, best\_move

else:

set minEval to positive infinity

set best\_move to None

for each move in get\_all\_moves(position, red, game):

evaluation = minimax(move, depth-1, true, game)[0]

set minEval to the minimum of minEval and evaluation

if minEval equals evaluation:

set best\_move to move

return minEval, best\_move

function simulate\_move(piece, move, board, game, skip):

if piece exists:

if skip exists:

for each skipped\_piece in skip:

remove skipped\_piece from board

move piece to new position on board

return board

function get\_all\_moves(board, color, game):

set moves to empty list

for each piece in board.get\_all\_pieces(color):

get valid\_moves for piece

for each move and skip in valid\_moves:

create a deepcopy of the board

get temp\_piece from temp\_board at piece's current position

simulate move on temp\_board

add new\_board to moves list

return moves

**2.Alpha beta Implementation**

define colors:

red = (255, 0, 0)

white = (255, 255, 255)

black = (0, 0, 0)

blue = (0, 0, 255)

grey = (128, 128, 128)

function alpha\_beta\_minimax(position, depth, alpha, beta, max\_player, game):

if depth equals 0 or position has a winner:

return position.evaluate(), position

if max\_player:

set max\_eval to negative infinity

set best\_move to None

for each move in get\_all\_moves(position, white, game):

evaluation, \_ = alpha\_beta\_minimax(move, depth-1, alpha, beta, false, game)

set max\_eval to the maximum of max\_eval and evaluation

if max\_eval equals evaluation:

set best\_move to move

update alpha to the maximum of alpha and max\_eval

if beta is less than or equal to alpha:

break loop

return max\_eval, best\_move

else:

set min\_eval to positive infinity

set best\_move to None

for each move in get\_all\_moves(position, red, game):

evaluation, \_ = alpha\_beta\_minimax(move, depth-1, alpha, beta, true, game)

set min\_eval to the minimum of min\_eval and evaluation

if min\_eval equals evaluation:

set best\_move to move

update beta to the minimum of beta and min\_eval

if beta is less than or equal to alpha:

break loop

return min\_eval, best\_move

function simulate\_move(piece, move, board, game, skip):

if piece exists:

if skip exists:

for each skipped\_piece in skip:

remove skipped\_piece from board

move piece to new position on board

return board

function get\_all\_moves(board, color, game):

set moves to empty list

for each piece in board.get\_all\_pieces(color):

get valid\_moves for piece

for each move and skip in valid\_moves:

create a deepcopy of the board

get temp\_piece from temp\_board at piece's current position

simulate move on temp\_board

add new\_board to moves list

return moves

**3. Fuzzy Logic Implementation**

import necessary libraries:

numpy as np

skfuzzy as fuzz

skfuzzy.control as ctrl

board class from checkers.board

alpha\_beta\_minimax function from minimax.algorithm

deepcopy from copy

random

define colors:

red = (255, 0, 0)

white = (255, 255, 255)

black = (0, 0, 0)

blue = (0, 0, 255)

grey = (128, 128, 128)

rows, cols = 9, 9 # board dimensions

define fuzzy variables:

piece\_value = fuzzy antecedent range 0 to 10, step 1

game\_phase = fuzzy antecedent range 0 to 10, step 1

move\_strength = fuzzy consequent range 0 to 100, step 1

define fuzzy membership functions:

piece\_value['soldier'] = trimf [0, 2, 4]

piece\_value['queen'] = trimf [3, 5, 7]

piece\_value['king'] = trimf [6, 8, 10]

game\_phase['very\_early'] = trimf [0, 1, 3]

game\_phase['early'] = trimf [2, 4, 6]

game\_phase['midgame'] = trimf [5, 7, 9]

game\_phase['late'] = trimf [8, 10, 10]

game\_phase['endgame'] = trimf [9, 10, 10]

move\_strength['very\_weak'] = trimf [0, 15, 30]

move\_strength['weak'] = trimf [20, 35, 50]

move\_strength['medium'] = trimf [45, 60, 75]

move\_strength['strong'] = trimf [70, 85, 100]

move\_strength['very\_strong'] = trimf [90, 100, 100]

define fuzzy rules:

for each combination of game\_phase and piece\_value:

create rule setting move\_strength to 'very\_strong'

create rules to prioritize capturing moves

create fuzzy control system:

move\_ctrl = control system using defined rules

move\_simulation = simulation of move\_ctrl

function calculate\_fuzzy\_move(board, row, col):

get piece at (row, col)

if piece is white:

set piece\_value\_value based on piece type

calculate game\_phase\_value based on total\_pieces on board

set move\_simulation inputs for piece\_value and game\_phase

try to compute move\_strength using fuzzy logic

catch ValueError and set move\_strength to 0 if error occurs

get all valid moves for piece

for each move in valid\_moves:

adjust move\_strength based on whether move is a capture

return move\_strengths

return empty

function simulate\_move(piece, move, board, skip):

if piece exists:

if skip exists:

remove each skipped\_piece from board

move piece to move position on board

return board

function determine\_best\_fuzzy\_move(board):

initialize best\_moves as empty list

set best\_strength to -1

for each row in rows:

for each col in cols:

get piece at (row, col)

if piece is white:

calculate move\_strengths using calculate\_fuzzy\_move

for each move and strength in move\_strengths:

update best\_moves if strength is greater than best\_strength

if best\_moves exists:

choose random best\_move from best\_moves

make a deep copy of board and simulate best\_move

return new\_board

return none

if this is the main program:

initialize board with board class

set game to none

determine best\_move using determine\_best\_fuzzy\_move

print best\_move

apply alpha\_beta\_minimax for deeper analysis

print alpha\_beta\_result and best\_move

**1. Genetic Algorithm Implementation**

import deepcopy from copy

define colors:

red = (255, 0, 0)

white = (255, 255, 255)

black = (0, 0, 0)

blue = (0, 0, 255)

grey = (128, 128, 128)

import random

import deepcopy from copy

import red and white from checkers.constants

function GA\_minimax(position, depth, alpha, beta, max\_player, game, evaluation\_function):

if depth equals 0 or position has a winner:

return evaluation\_function(position) plus small random value, position

if max\_player:

set max\_eval to negative infinity

set best\_moves to empty list

for each move in get\_all\_moves(position, white, game):

evaluation, \_ = GA\_minimax(move, depth-1, alpha, beta, false, game, evaluation\_function)

if evaluation is greater than max\_eval:

update max\_eval to evaluation

set best\_moves to a list containing this move

elif evaluation equals max\_eval:

add move to best\_moves

update alpha to the maximum of alpha and max\_eval

if beta is less than or equal to alpha:

break loop

return max\_eval, choose random move from best\_moves

else:

set min\_eval to positive infinity

set best\_moves to empty list

for each move in get\_all\_moves(position, red, game):

evaluation, \_ = GA\_minimax(move, depth-1, alpha, beta, true, game, evaluation\_function)

if evaluation is less than min\_eval:

update min\_eval to evaluation

set best\_moves to a list containing this move

elif evaluation equals min\_eval:

add move to best\_moves

update beta to the minimum of beta and min\_eval

if beta is less than or equal to alpha:

break loop

return min\_eval, choose random move from best\_moves

function simulate\_move(piece, move, board, game, skip):

if piece exists:

if skip exists:

for each skipped\_piece in skip:

remove skipped\_piece from board

move piece to new position on board

return board

function get\_all\_moves(board, color, game):

set moves to empty list

for each piece in board.get\_all\_pieces(color):

get valid\_moves for piece

for each move and skip in valid\_moves:

create a deepcopy of the board

get temp\_piece from temp\_board at piece's current position

simulate move on temp\_board

add new\_board to moves list

shuffle moves list randomly

return moves

result,discussion,performancecomparison,limitation,future scope,conclusion,references

**Limitations**

.1. The game features only three types of pieces, which might restrict strategic diversity.

2. Over time, players may notice patterns in the AI's decision-making, reducing the challenge.

3. The game does not support multiplayer options, limiting player interaction to AI opponents only.

4.The fixed grid size may limit the complexity of gameplay, especially for advanced players.

**Future Scope**

1.Introducing an online or local multiplayer mode for player-versus-player matches.

2. Incorporating machine learning to enhance AI adaptability and complexity.

3. Adding new pieces with unique movements and abilities to increase strategic depth.

4. Allowing players to modify grid size, piece rules, and difficulty levels for a more personalized experience.

**Discussion**

Minimax provides the best move by thoroughly evaluating all possible scenarios and assuming perfect play from both the AI and its opponent. It is particularly effective in games with deterministic outcomes and a clear win or loss structure. Fuzzy Logic excels in handling uncertainty and situations where decisions need to be made with incomplete information. It offers good moves based on heuristic evaluation, making it effective in scenarios where the environment is complex and not fully deterministic. Genetic Algorithms are strong in exploring large, complex search spaces and evolving strategies over time. However, they might not always yield the single best move in every specific instance, especially in games requiring precise, immediate decisions rather than long-term strategy evolution. Hence, minimax approach is likely to provide the most consistently optimal moves.

**Conclusion**

The grid-based strategic game "Crown Clash" successfully integrates multiple AI decision-making strategies for an engaging experience. By employing genetic algorithms, fuzzy logic, and minimax approaches, the game offers varied difficulty levels to entertain the players of different skill levels. The unique piece movements enhance competitiveness. Overall, "Crown Clash" demonstrates the effective implementation of strategic algorithms, presenting a balanced and interactive platform for both players and the AI opponent.

**References**

[1] “PyGame Tutorial,” GeeksforGeeks, Oct. 19, 2021. <https://www.geeksforgeeks.org/pygame-tutorial/>

[2] Pirovano, Michele. "The use of fuzzy logic for artificial intelligence in games." *University of Milano, Milano* (2012).

[3] Nielsen, Jon Lau, et al. "Ai for general strategy game playing." *Handbook of Digital Games* (2014): 274-304.

[4] Mathew, Tom V. "Genetic algorithm." *Report submitted at IIT Bombay* 53 (2012).