**Project Report: Vikings Chess**

By

**Ankon Choudhury**

Roll: 1907048

&

**Md. Kaisarul Islam**

Roll: 1907054



**Submitted To:**

Md. Shahidul Salim

Lecturer

Signature

Most. Kaniz Fatema Isha

Lecturer

Signature

**Department of Computer Science and Engineering**

**Khulna University of Engineering & Technology**

**Khulna 9203, Bangladesh**

**3rd September 2024**

1. Introduction
   1. Background

The "Vikings Chess" is a strategy board game inspired by old Viking games and chess. In history, Vikings played games where they had to protect a key piece or try to capture it, using careful planning and strategy. This project recreates that kind of game using Python.

In "Vikings Chess," the computer acts as a smart opponent by using special AI techniques. These techniques, like the Minimax algorithm, help the AI look ahead at possible moves and choose the best one. This makes the game challenging and fun, as players must plan their moves carefully to beat the computer.

* 1. Objectives

The project’s major objective is to develop an AI-powered game with implementing advanced algorithms to make the AI capable of strategic gameplay. The primary goal of this project is

* Developing the Vikings Chess game using the Python Pygame library with integration of advanced AI algorithms.
* Implementing the Minimax algorithm for optimal decision-making in the game.
* Enhancing the AI’s efficiency by incorporating Alpha-Beta Pruning to reduce unnecessary computations.
* Using Genetic Algorithms for evolving strategies.
* Using Fuzzy-Logic to determine the intelligence level of AI
* Designing a user-friendly interface to easily interact.
* Providing a competitive AI opponent
* Gain a deep understanding of game theory principles and how they can be applied to AI based game.

1. Project Design

The Project was designed with the fact that it can be scalable and adaptable, making it flexible for any future addition and modification.

* 1. System architecture

The structure of the project is exhibited in a class diagram, illustrating the encapsulation and association of key classes.

* + 1. Class Diagram

The information and functionality of association and encapsulation of each individual class or building block is shown in Figure 3.3 where the blocks resonate the required information.

Game State

+ Board: String<Array>

+ AI Player: String

+ Human Player: String

Update\_board\_status(board)

Find\_valid\_moves(self)

Capture\_check(self)

Escape\_check(self)

evaluate(self,board)

strategy(self,board)

Chromosome

+ Moves: (int,int)<Array>

Fitness()

<Interface>

write\_text(string,pos)

game\_window()

Draw\_board()

Draw\_pieces(row,col)

Update\_piece\_position(row,col)

Fuzzy Intelligence

+ Total no moves: Int

+ AI Count: Int

+ Human Count: Int

- Score: Int

move\_membership(moves)

piece\_membership\_ai(total moves ai)

piece\_membership\_human(total moves human)

evaluate\_ai()

Defuzzification()

Genetic Algorithm

- No of Generation: Int

- Population Size: Int

- Gene Size: Int

- Mutation : Int

- Chromosomes: Chromosome <Array>

find\_fitness (piece pos with score)

calculate\_generation (chromose)

Selection (Chromosome<Array>)

Crossover (chrom1, chrom2)

Mutation (chrom)

Min Max Alpha Beta

- max\_Depth: Int

- Alpha: Int

- Beta: Int

- game\_over: Bool

- score: Int

evaluate(self,board)

applied\_move(board,committed\_move)

**Figure 2.1:** Class diagram of the project.

* 1. Tools used

A wide variety of tools and libraries were used to complete the project. Each tool played a significant role.

* + 1. Libraries

The latest python libraries, including Numpy, Pygame, Math, random, datetime, were masterfully leveraged to create a highly organized and efficient structure for the project.

* + 1. VS Code

Visual Studio Code serves as a versatile code editor. Various extensions can be found to support both python and user interface which increases coding productivity.

1. Project Implementation

This chapter will describe the steps and individual procedures that was taken to complete the AI project. The game can be played only with ai. The interface is user friendly.

* 1. Game UI

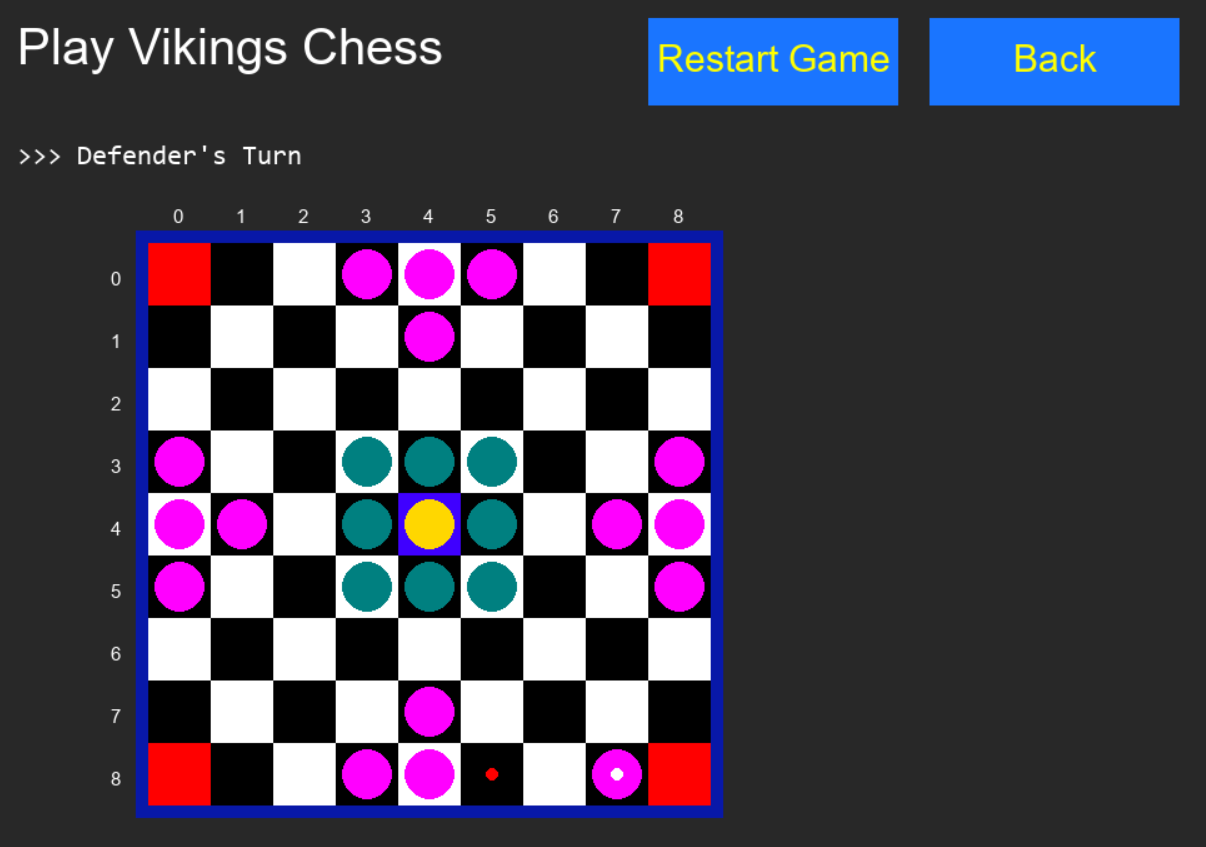
There is only one option for a player to choose at first, he/she can choose to play as player vs AI.

A screenshot of a game

Description automatically generated

**Figure 3.1:** Initial Game Screen.

The game contains a square board of 9\*9 consists of 81 squares of black and white similar to chess board.



**Figure 3.2:** The Board of Vikings Chess Game.

At each corner of the board, there are 4 red squares. There are mainly three types of pieces. The board contains 8 green pieces,16 pink pieces and 1 yellow piece which is surrounded by the green pieces. The first move has always been done by AI.

* 1. Game Rules

This is mainly a turn-based board game. The center cell and four corner cells are called restricted cells. The rules are explained below:

1. The yellow piece in the middle of the board is called the king piece. All pieces except the king pieces can move any number of cells horizontally and vertically.
2. King piece (yellow) can move only one cell at a time.
3. The main goal of the game is to move the king piece in any 4 corners (red squares) of the board which is played by the player and AI will try to capture the king so that the king can’t be placed in the red square.
4. If the player is able to move the king to any 4 corners of the board, then the player wins.
5. If the king piece is captured by attackers (pink pieces), then AI wins.
6. Pieces, except king, can be captured by sandwiching them from both sides. (fig 3.3).
7. Restricted cells can be used to sandwich the opponent.

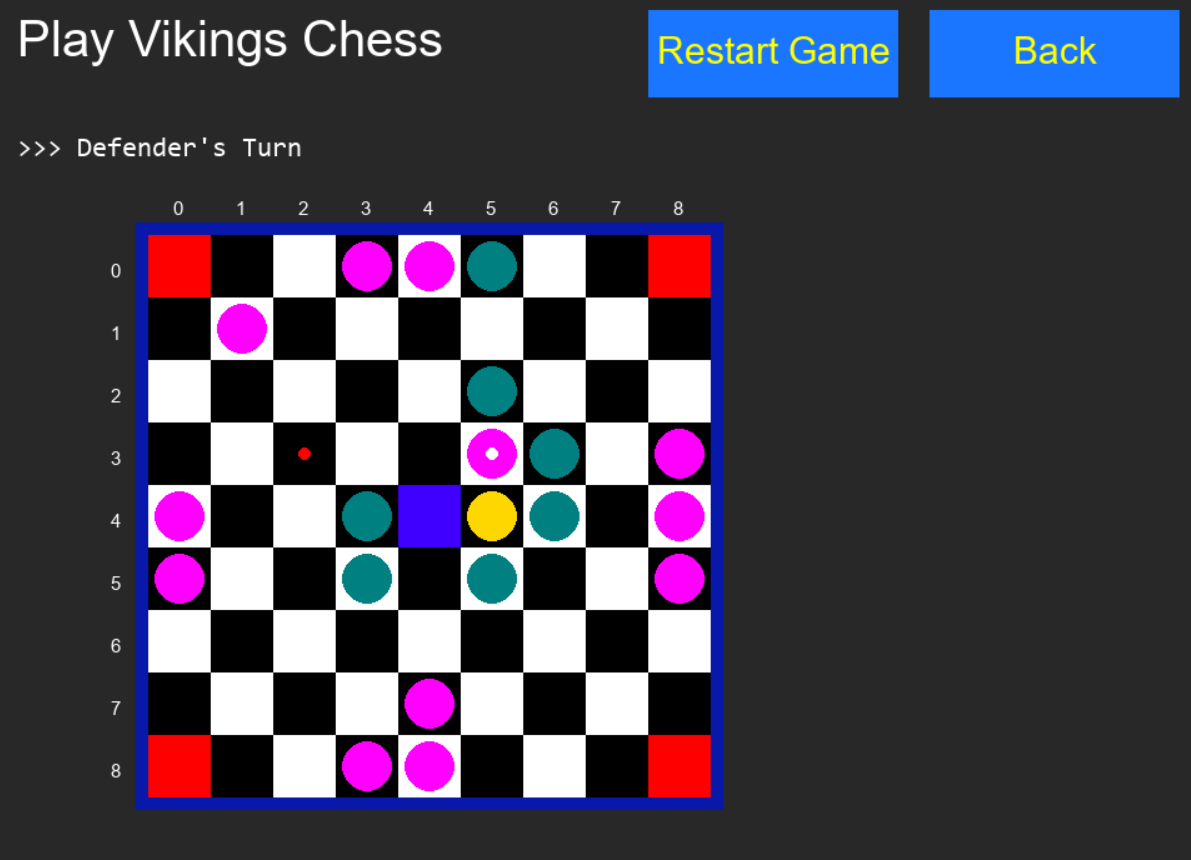


Figure 3.3: The attacker piece (pink piece) is captured by sandwiching with two

Defender pieces (green pieces).

1. Only one piece can be captured at a time.
2. To capture the king, attackers need to surround him on all four cardinal points.
3. AI try to make the best move based on some algorithms (minmax algorithm, genetic algorithm) to capture the king.
4. If the king is captured, attackers win (AI wins).

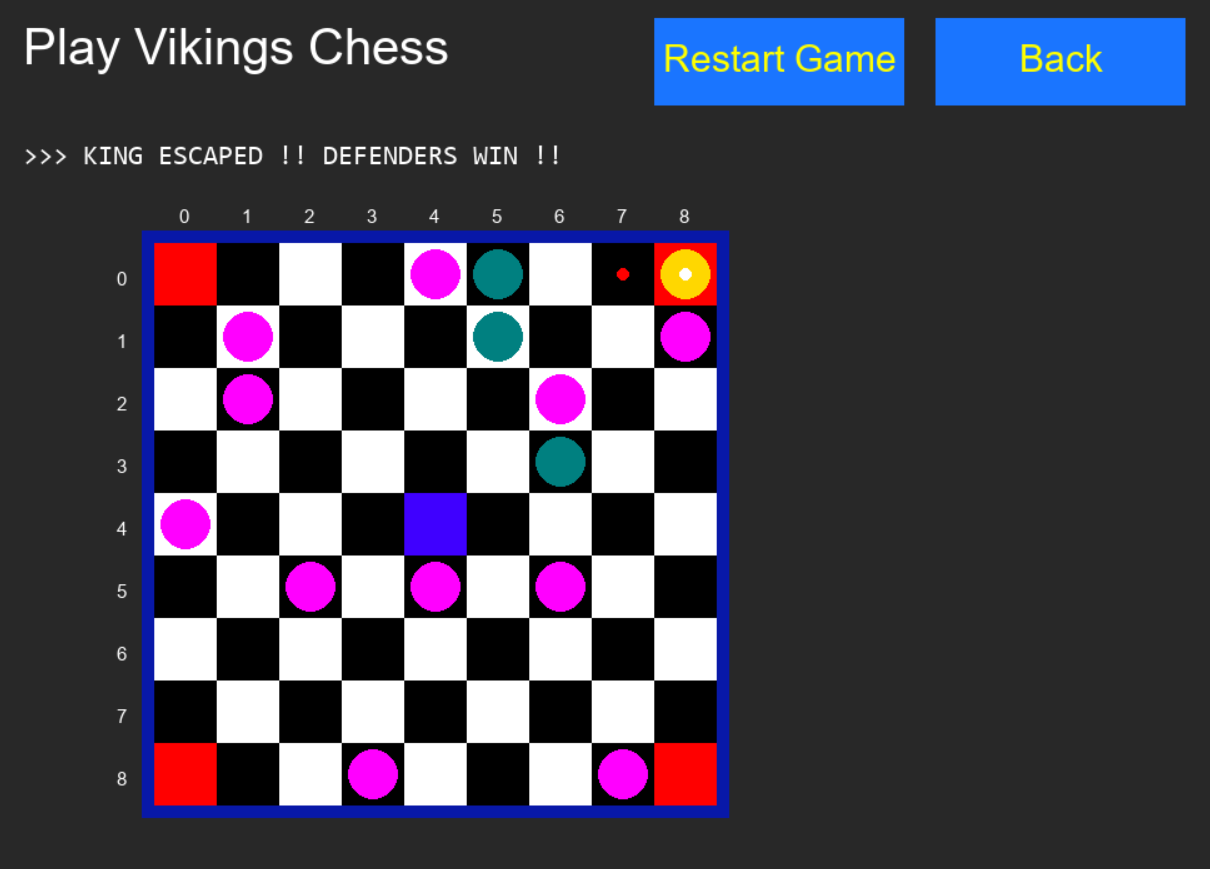


Figure 3.4: The king escaped and defender wins.

1. If the king escapes to any of the four corners, defenders win.
2. If all attackers are captured, the defender wins.
   1. Optimizing AI Strategies

This section describes how AI decides his best move and evaluate his performance using different heuristic algorithms, implementation details along with their corresponding pseudo code.

* + 1. The Role of Alpha-Beta Pruning with Minimax

During the first phase, AI optimizes its decision making process by recursively simulates all possible move till depth 5, evaluating each move and selecting the best one based on a heuristic evaluation function. Alpha-Beta pruning enhances the efficiency of this algorithm by pruning branches that cannot possibly influence the final decision.

Pseudo Code:

The algorithm can be described by the following functions.

Function Minimax(board, alpha, beta, depth, turn):

Initialize bestvalue

If turn is Maximizer (True):

bestvalue = -∞

Else (Minimizer, False):

bestvalue = ∞

moves = find\_all\_possible\_valid\_moves(board, turn)

If depth = 0 or game\_over(board):

Return evaluate(board)

Create current\_board as a copy of board

For each move in moves:

tmp\_fake\_board = make\_move(current\_board, move)

If turn is Maximizer:

value = Minimax(tmp\_fake\_board, alpha, beta, depth-1, False)

bestvalue = max(bestvalue, value)

alpha = max(alpha, bestvalue)

If beta <= alpha:

Break (Beta cutoff)

Else (Minimizer):

value = Minimax(tmp\_fake\_board, alpha, beta, depth-1, True)

bestvalue = min(bestvalue, value)

beta = min(beta, bestvalue)

If beta <= alpha:

Break (Alpha cutoff)

Return bestvalue

* + 1. The Role of Genetic Algorithm

In the game, the AI optimizes its decision-making process using a Genetic Algorithm (GA). The GA generates a population of possible moves, which are then evaluated based on a heuristic function. Through processes like selection, crossover, and mutation, the GA iteratively refines these moves, seeking the most effective strategy. This approach allows the AI to explore a wide range of potential moves and evolve better strategies over generations. By focusing on the most promising options, the GA efficiently guides the AI towards the best possible decisions in the game’s complex mid-phase.

Pseudo Code:

The algorithm can be described by the following functions.

Function GeneticAlgorithm(population, generations, crossover\_rate, mutation\_rate):

# Assume 'population' is a list of chromosomes initialized outside this function

For generation in range(generations):

# Evaluate fitness for each chromosome

fitness\_scores = []

For each chromosome in population:

fitness = find\_fitness(chromosome, dicto) # 'dicto' is the dictionary used for fitness calculation

fitness\_scores.append(fitness)

# Selection: Choose parents based on fitness

selected\_parents = selection(population, fitness\_scores)

# Crossover: Create next generation through crossover

new\_population = []

For i in range(0, len(selected\_parents), 2):

parent1 = selected\_parents[i]

parent2 = selected\_parents[i+1]

If random < crossover\_rate:

crossover\_point = random\_int(1, len(parent1) - 1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

Else:

child1 = parent1

child2 = parent2

new\_population.append(child1)

new\_population.append(child2)

# Mutation: Apply mutation to introduce variability

For each chromosome in new\_population:

If random < mutation\_rate:

mutate(chromosome) # Calls a mutation function to alter genes

# Replace old population with the new generation

population = new\_population

# Return the best chromosome from the final generation

best\_chromosome = select\_best(population, fitness\_scores)

Return best\_chromosome

Function find\_fitness(chromosome, dicto):

summ = 0

For gene in chromosome:

summ += dicto[gene]

Return summ

Function selection(population, fitness\_scores):

# Implement selection logic (e.g., roulette wheel, tournament)

selected\_parents = []

# Select parents based on fitness

Return selected\_parents

Function mutate(chromosome):

# Implement mutation logic (e.g., bit flip, swap)

Return mutated\_chromosome

* + 1. The Role of Fuzzy Logic

During the game, the AI assesses the player's performance using a fuzzy logic-based system. This system assigns membership values to these factors, and based on the fuzzy logic rules, it calculates a performance score for the player, determining whether the player's actions were excellent, good, moderate, bad, or worst.

Pseudo Code:

The algorithm can be described by the following functions.

Function total\_number\_move\_membership(n):

Initialize degree as a dictionary with 'low', 'avg', 'huge' all set to 0

If 0 <= n < 7:

degree['low'] = 1

Else If 7 <= n < 10:

degree['low'] = (10 - n) / (10 - 7)

degree['avg'] = (n - 7) / (10 - 7)

Else If 10 <= n < 11:

degree['avg'] = (12 - n) / (12 - 10)

Else If 11 <= n < 12:

degree['avg'] = (12 - n) / (12 - 10)

degree['huge'] = (n - 11) / (13 - 11)

Else If 12 <= n < 13:

degree['huge'] = (n - 11) / (13 - 11)

Else If n >= 13:

degree['huge'] = 1

Return degree

Function guti\_membership\_AI(n):

Initialize degree as a dictionary with 'poor', 'good', 'best' all set to 0

If 0 <= n < 8:

degree['poor'] = 1

Else If 8 <= n < 10:

degree['poor'] = (10 - n) / (10 - 8)

degree['good'] = (n - 8) / (10 - 8)

Else If 10 <= n < 12:

degree['good'] = (12 - n) / (12 - 10)

degree['best'] = (n - 10) / (12 - 10)

Else If n >= 12:

degree['best'] = 1

Return degree

Function guti\_membership\_HUMAN(n):

Initialize degree as a dictionary with 'poor', 'good', 'best' all set to 0

If 0 <= n < 2:

degree['poor'] = 1

Else If 2 <= n < 3:

degree['poor'] = (3 - n) / (3 - 2)

degree['good'] = (n - 2) / (3 - 2)

Else If 3 <= n < 5:

degree['good'] = (5 - n) / (5 - 3)

degree['best'] = (n - 3) / (5 - 3)

Else If n >= 5:

degree['best'] = 1

Return degree

Function evaluate\_black\_player\_rating(M\_degree, W\_degree, B\_degree):

Initialize rating as a dictionary with 'worst', 'bad', 'moderate', 'good', 'excellent' all set to 0

rating['excellent'] = max(

min(

min(max(B\_degree['good'], B\_degree['best']), max(W\_degree['good'], W\_degree['poor'])),

max(M\_degree['avg'], M\_degree['low'])

),

min(

min(B\_degree['best'], max(W\_degree['good'], W\_degree['poor'])),

max(M\_degree['avg'], M\_degree['low'])

)

)

rating['good'] = max(

min(

min(max(B\_degree['good'], B\_degree['best']), max(W\_degree['good'], W\_degree['poor'])),

max(M\_degree['huge'], M\_degree['avg'])

),

min(

min(max(B\_degree['good'], B\_degree['best']), max(W\_degree['good'], W\_degree['poor'])),

max(M\_degree['avg'], M\_degree['low'])

)

)

rating['moderate'] = max(

min(

min(max(B\_degree['poor'], B\_degree['good']), max(W\_degree['best'], W\_degree['good'])),

M\_degree['huge']

),

min(

min(max(B\_degree['poor'], B\_degree['good']), max(W\_degree['best'], W\_degree['good'])),

M\_degree['avg']

)

)

rating['bad'] = max(

max(

min(max(B\_degree['poor'], max(W\_degree['best'], W\_degree['good'])), M\_degree['huge']),

min(max(B\_degree['poor'], max(W\_degree['best'], W\_degree['good'])), M\_degree['huge'])

),

max(

min(max(B\_degree['poor'], max(W\_degree['best'], W\_degree['good'])), M\_degree['avg']),

min(max(B\_degree['poor'], max(W\_degree['best'], W\_degree['good'])), M\_degree['avg'])

)

)

rating['worst'] = max(

max(

min(max(B\_degree['poor'], W\_degree['best']), M\_degree['huge']),

min(max(B\_degree['poor'], W\_degree['best']), M\_degree['huge'])

),

max(

min(max(B\_degree['poor'], W\_degree['best']), M\_degree['avg']),

min(max(B\_degree['poor'], W\_degree['best']), M\_degree['avg'])

)

)

Return rating

1. Conclusion

The project successfully developed a sophisticated AI-powered Vikings Chess game, leveraging Minimax, Genetic Algorithms, and Fuzzy Logic for strategic gameplay. Despite challenges in implementation and optimization, the integration of these advanced techniques led to a responsive and competitive AI, significantly enhancing the overall gaming experience.

* 1. Conclusion and challenges faced

The Vikings chess game leverages a multi-phase AI strategy to enhance gameplay and decision-making. In the initial phase, the Minimax algorithm provides a robust framework for evaluating potential moves, ensuring optimal decision-making through recursive simulations. The subsequent phase employs Genetic algorithms to refine and evolve strategies, allowing the AI to adapt and optimize its performance over time. Additionally, fuzzy logic is used to assess the player's performance dynamically, offering a nuanced evaluation of gameplay based on timing, move count, and board state.

While the integration of these advanced techniques has significantly improved the game's AI, several challenges were encountered. Implementing the Minimax algorithm required managing a complex game tree and optimizing performance to handle the computational load. The Genetic algorithm, while powerful, posed challenges in tuning parameters and ensuring convergence to effective strategies. Additionally, incorporating fuzzy logic introduced complexity in designing membership functions and evaluation criteria. Despite these obstacles, addressing them head-on led to a more sophisticated and responsive AI system, ultimately enhancing the overall gaming experience.

# References

1. [Game theory — Minimax. This article will be a bit different… | by NerdzLab | Towards Data Science](https://towardsdatascience.com/game-theory-minimax-f84ee6e4ae6e)
2. [Genetic Algorithms - GeeksforGeeks](https://www.geeksforgeeks.org/genetic-algorithms/)
3. [Fuzzy Logic: Definition, Meaning, Examples, and History (investopedia.com)](https://www.investopedia.com/terms/f/fuzzy-logic.asp#:~:text=Fuzzy%20logic%20is%20a%20set,from%20this%20kind%20of%20data.)
4. [The Rules of Nine Mens Morris / Merrills / Mill / Merels (mastersofgames.com)](https://www.mastersofgames.com/rules/morris-rules.htm?srsltid=AfmBOopb_phEilGCsxzu0B5c7vmMENtu1Ra-szEdqMWjEKIaIcSxIxIk)