

# Chess Game Report

Prepared for

Dr/ Mayar Ayman

Professor, Faculty of Computers And

Data Science, Alexandria University

By

Name	ID
Mesk Khaled	23011537
Asmaa Mahmoud	23011194
Judy Ahmed	23011244
Basmala Moataz	23012111
Adham Ashraf	23011220
Omar Kamel	23011380

December 2024

## Introduction:

Chess is an immortal, internationally popular strategy game that challenges players to think critically and outmaneuver their opponents. Played on an 8x8 board with 16 pieces per side, the objective is to checkmate the opponent's king.

Chess is a game combining logic, creativity, and flexibility that has grown from an amateur game to a competitive and culturally important activity around the globe. Advances in technology, especially concerning artificial intelligence and algorithms, have changed the way chess study and play are approached.

The following report discusses the making of a chess game using Python, including graphical user interface development and AI techniques in order to extend the opportunities for gameplay and strategic analysis.

## Problem Description:

Chess is a complex strategy game with countless possible moves and outcomes. While the rules are straightforward, mastering the game requires deep strategic thinking and tactical foresight. Many beginners find it challenging to understand the game dynamics and make informed decisions, particularly against skilled opponents.

For that purpose, a project will be developed-a chess program, written in Python, combined with the use of an interactive graphical interface and incorporating elements of AI. This will simulate the game of chess, through which the user can engage in playing chess against a genetically driven algorithm opponent for move valuation, which keeps the user interaction both smooth and challenging at the same time.

### *a) Program functionality:*

The chess program will:

1. Provide an intuitive graphical user interface (GUI) for players to interact with the chessboard and move pieces.
2. Enforce the official rules of chess, including legal moves, check, checkmate, and stalemate conditions.
3. Incorporate an AI opponent for single-player mode, using a genetic algorithm to evaluate and select the best moves for the black player.
4. Highlight possible moves and alert the player when the game ends (e.g., checkmate or stalemate).

### *b) Inputs of the program:*

1. Player moves: The user selects a piece and its target square through the GUI.
2. AI parameters: Internal configurations such as the number of generations, population size, and mutation rate for the genetic algorithm.

### *c) Outputs of the program:*

1. Visual updates: Real-time rendering of the chessboard and pieces after each move.
2. Notifications: Alerts for game events such as invalid moves, check, checkmate, or stalemate.
3. AI move: The program computes and displays the AI's selected move.
4. Endgame messages: Display of the final game state (e.g., "Checkmate!").

## Role of Each Member:

### *1) Genetic Algorithm Development:*

**Judy, Mesk,** and **Omar** collaborated on designing and implementing the genetic algorithm. They worked on creating a robust AI opponent by developing the algorithm's logic for move evaluation and optimization.

### *2) Graphical User Interface (GUI) Development:*

**Basmala, Asmaa,** and **Adham** were responsible for designing and implementing the GUI. They created an intuitive and visually appealing interface for the chessboard, enabling user interaction and move selection.

### *3) Testing and Integration:*

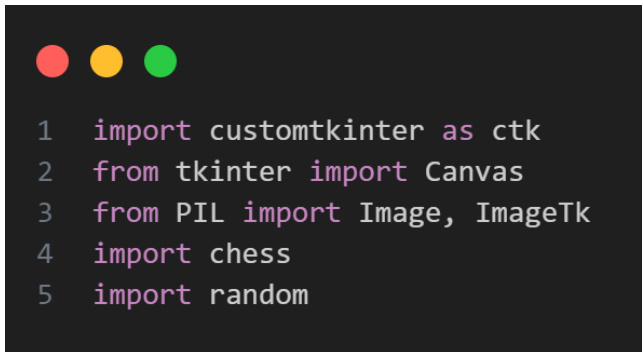
**Mesk** ensured the functionality and reliability of the program by testing the code thoroughly. Additionally, she managed the integration of the genetic algorithm and GUI components to produce a cohesive and functional final product.

## Program Implementation:

This chess game program is implemented in Python and consists of two main components: The Genetic Algorithm for the AI and the Graphical User Interface (GUI) for user interaction. Below is a detailed explanation of each part.

### *1. Libraries used and imports:*

- **customtkinter:** Used for creating an interactive and customizable GUI.
- **python-chess:** Provides chessboard representation and move validation.
- **Pillow:** Handles image loading and processing for chess pieces.
- **random:** Generates random sequences for the genetic algorithm.



```
1 import customtkinter as ctk
2 from tkinter import Canvas
3 from PIL import Image, ImageTk
4 import chess
5 import random
```

## 2. Genetic Algorithm class:

### ➤ `evaluate_fitness(self):`

- Determine each move's fitness score for the AI player (the black one) so that the best move for winning can be chosen in the end.

```
1 def evaluate_fitness(self):
2     fitness_scores = {}
3     for move in self.board.legal_moves:
4         self.board.push(move)
5         score = self.calculate_score()
6
7         fitness_scores[move] = score
8         self.board.pop()
9     return fitness_scores
```

### ➤ `calculate_score(self):`

- Based on AI piece safety, it assists the AI player in determining how favorable or unfavorable the current circumstances are. Each piece in the game has its initial score (for example, pawn=1, knight=3, queen=9, etc.), so we can determine whether there are any dangerous spots for AI pieces. The higher the score, the better position for AI pieces.

```
1 def calculate_score(self):
2     score = 0
3     for square in chess.SQUARES:
4         piece = self.board.piece_at(square)
5         if piece:
6             if piece.piece_type == chess.PAWN:
7                 score += 1
8             elif piece.piece_type == chess.KNIGHT:
9                 score += 3
10            elif piece.piece_type == chess.BISHOP:
11                score += 3
12            elif piece.piece_type == chess.ROOK:
13                score += 5
14            elif piece.piece_type == chess.QUEEN:
15                score += 9
16            elif piece.piece_type == chess.KING:
17                score += 100
18            if not self.is_piece_safe(piece, square):
19                score -= 2
20            if square in [chess.D4, chess.E4, chess.D5, chess.E5]:
21                score += 1
22    return score
```

### ➤ `is_piece_safe(self, piece, square):`

- Used to determine whether or not the AI piece is in a risky position.

```
1 def is_piece_safe(self, piece, square):
2     for move in self.board.legal_moves:
3         if move.to_square == square:
4             return False # The piece is under attack
5     return True
6
```

### ➤ `def genetic_algorithm(self, board, generations=20, population_size=10, mutation_rate=0.2, sequence_length=3):`

- Uses a genetic algorithm to find the best move sequence for the black player.
- Initializes a population of random move sequences.
- Evaluates the fitness of each sequence.
- Selects the top sequences and applies crossover and mutation to create the next generation.
- Returns the best move from the top sequence.

```
1 def genetic_algorithm(self, board, generations=20, population_size=10,
2 mutation_rate=0.2, sequence_length=3):
3     self.board = board
4     population = [
5         [random.choice(list(self.board.legal_moves)) for _ in range(sequence_length)]
6         for _ in range(population_size)
7     ]
8     for generation in range(generations):
9         fitness_scores = {(tuple(sequence)): self.evaluate_sequence(sequence) for sequence in population}
10        sorted_population = sorted(population, key=lambda seq: fitness_scores[tuple(seq)], reverse=True)
11        top_sequences = sorted_population[:population_size // 2]
12        children = self.crossover_sequences(top_sequences, population_size - len(top_sequences))
13        for i in range(len(children)):
14            if random.random() < mutation_rate:
15                children[i] = self.mutate_sequence(children[i])
16        population = top_sequences + children
17        best_sequence = max(population, key=lambda seq: self.evaluate_sequence(seq))
18        return best_sequence[0]
19
```

➤ **def evaluate\_sequence(self, sequence):**

- Instead of rating a single move as in the evaluate\_fitness function(), it rates the fitness score of a sequence of moves. Every move in the sequence on the board is imagined by the AI. Depending on how well or poorly it appears to the AI, it determines a score for the final board position.

```
1 def evaluate_sequence(self, sequence):
2     score = 0
3     pushed_moves = 0
4     try:
5         for move in sequence:
6             if move not in self.board.legal_moves:
7                 break
8             self.board.push(move)
9             pushed_moves += 1
10            score += self.calculate_score()
11    finally:
12        for _ in range(pushed_moves):
13            self.board.pop()
14    return score
15
```

➤ **def crossover\_sequences(self, top\_sequences, num\_children):**

- This function is similar to combining two good designs to create a new one that should be better. Similar to attacking and defending in a game of chess, we can move a piece that uses an attacking strategy while simultaneously defending one. In this case, we combine two strategies, which is far superior.

```
1 def crossover_sequences(self, top_sequences, num_children):
2     children = []
3     for _ in range(num_children):
4         parent1 = random.choice(top_sequences)
5         parent2 = random.choice(top_sequences)
6         crossover_point = random.randint(1, len(parent1) - 1)
7         child = parent1[:crossover_point] + parent2[crossover_point:]
8         children.append(child)
9     return children
10
```

➤ **def mutate\_sequence(self, sequence):**

- We utilize this feature to add variety to the game rather than simply repeating the same tactic for every generation. Take an established move sequence. Select one move at random from the sequence, then swap it out for another legitimate move. There has been a tiny change to the modified sequence. This unpredictability aids the AI in discovering novel alternatives and preventing it from becoming mired in the same tactics.

```
1 def mutate_sequence(self, sequence):
2     mutation_index = random.randint(0, len(sequence) - 1)
3     mutated_sequence = sequence[:]
4     mutated_sequence[mutation_index] = random.choice(list(self.board.
5     legal_moves))
6     return mutated_sequence
```

### 3. GUI class:

#### ➤ `__init__(self):`

- Initializes the main window using `customtkinter`.
- Sets up the board and other GUI components, such as buttons and piece images.
- Calls the `menu()` function to display the initial menu.

```
1 def __init__(self):
2     self.root = ctk.CTk()
3     self.root.title("Chess GUI")
4     self.root.geometry("640x640")
5     self.board = chess.Board()
6     self.squares = [[] for _ in range(8)]
7     self.selected_square = None
8     self.turn = chess.WHITE
9     self.king_in_check_square = None
10    self.genetic_algorithm = GeneticAlgorithm()
11    self.piece_images = {
12        piece: ctk.CTkImage(light_image=Image.open(f"pie
13ce).png"), size=(70, 70))
14        for piece in [
15            "white_knight", "white_pawn", "black_king",
16            "black_pawn", "black_knight",
17            "white_king", "default", "white_rook", "black
18            _rook", "white_bishop",
19            "black_bishop", "white_queen", "black_queen"
20        ]
21    }
22    self.root.resizable(False, False)
23    self.menu()
24    self.root.mainloop()
```

#### ➤ `menu(self):`

- Creates a menu window with options for playing the game or quitting.
- Adds buttons styled with `customtkinter` and places them on a canvas with an image background.
- Assigns button actions, such as opening the game board or quitting the program.

```
1 def menu(self):
2     self.root.geometry("300x350")
3     self.root.title("Chess Game")
4     menu_frame = ctk.CTkFrame(self.root, width=300, height=350,
5 fg_color="black")
6     menu_frame.pack(fill="both", expand=True)
7     canvas = Canvas(menu_frame, width=300, height=350, highligh
8 tthickness=0)
9     canvas.pack(fill="both", expand=True)
10    self.bg_photo = ImageTk.PhotoImage(Image.open("Chess Wallpa
11per.jpeg"))
12    canvas.create_image(0, 0, image=self.bg_photo, anchor="nw")
13    self.overlay_photo = ImageTk.PhotoImage(Image.open("chess.p
14ng"))
15    canvas.create_image(150, 100, image=self.overlay_photo, anc
16hor="center")
17    button_style = {
18        "width": 100, "height": 50, "font": ("PMingLiU-ExtB", 3
190),
20        "text_color": "white", "fg_color": "#141414", "bg_colo
21r": "#141414", "hover_color": "#292927"
22    }
23    ctk.CTkButton(menu_frame, text="Play", command=lambda: [sel
24f.board_gui(), menu_frame.destroy()], **button_style) \
25        .place(relx=0.5, rely=0.5, anchor="center")
26    ctk.CTkButton(menu_frame, text="Quit", command=self.root.de
27stroy, **button_style) \
28        .place(relx=0.5, rely=0.7, anchor="center")
```

#### ➤ `board_gui(self):`

- Creates and displays the chessboard, including the initial placement of pieces.
- Iterates through each square to create a button representing the square, setting the piece image accordingly.
- Adds a command to handle user interactions, which is passed to `move_piece()`.

```
1 def board_gui(self):
2     self.root.geometry("640x640")
3     board_frame = ctk.CTkFrame(self.root, width=640, height=
4 640)
5     for i in range(8):
6         for j in range(8):
7             fg_color = "white" if (i + j) % 2 == 0 else "gra
8 y"
9             b = ctk.CTkButton(
10                 board_frame, text="", width=80, height=80, fg
11_color=fg_color,
12                 border_color="black", border_width=1, corner
13_radius=0,
14                 image=self.piece_images.get(
15                     self.get_piece(i, j)),
16                 command=lambda row=i, column=j: self.move_pi
17ece(row, column),
18             )
19             b.grid(row=i, column=j)
20             self.squares[i].append(b)
21     board_frame.pack(fill="both", expand=True)
```



➤ **get\_piece(self, row, col):**

- Determines which chess piece is at a given position on the board.
- Converts row and col coordinates to the python-chess coordinate system.
- Returns a string representing the piece (e.g., "white\_pawn", "black\_queen").

```
1 def get_piece(self, row, col):
2     piece = self.board.piece_at(chess.square(col, 7
3     - row))
4     if piece is None:
5         return "default"
6     piece_map = {
7         chess.PAWN: "white_pawn" if piece.color ==
8         chess.WHITE else "black_pawn",
9         chess.KNIGHT: "white_knight" if piece.color
10        == chess.WHITE else "black_knight",
11        chess.BISHOP: "white_bishop" if piece.color
12        == chess.WHITE else "black_bishop",
13        chess.ROOK: "white_rook" if piece.color ==
14        chess.WHITE else "black_rook",
15        chess.QUEEN: "white_queen" if piece.color =
16        = chess.WHITE else "black_queen",
17        chess.KING: "white_king" if piece.color ==
18        chess.WHITE else "black_king",
19    }
20    return piece_map.get(piece.piece_type, "default")
```

➤ **move\_piece(self, row, col):**

- Handles the movement of a piece when a square is clicked.
- If a piece is already selected, it checks the legality of the move, pushes the move if valid, and updates the board.
- Checks for checkmate or stalemate conditions and switches turns accordingly.
- If no piece is selected, it sets the selected square and highlights possible moves.

```
1 def move_piece(self, row, col):
2     if self.selected_square:
3         selected_row, selected_col = self.selected_squ
4         are
5         from_square = chess.square(selected_col, 7 - s
6         elected_row)
7         to_square = chess.square(col, 7 - row)
8         move = chess.Move(from_square, to_square)
9         if move in self.board.legal_moves:
10            self.board.push(move)
11            self.update_board()
12            self.check_king_in_check()
13            if self.board.is_checkmate():
14                self.display_game_over("Checkmate!")
15            elif self.board.is_stalemate():
16                self.display_game_over("Stalemate!")
17            else:
18                self.switch_turn()
19                self.make_black_move()
20                self.selected_square = None
21                self.reset_colors()
22        else:
23            self.selected_square = (row, col)
24            self.highlight_moves(row, col)
```

➤ **display\_game\_over(self, message):**

- Creates a popup window displaying a game-over message (e.g., "Checkmate!").
- Disables all buttons on the board and provides an "OK" button to close the popup.

```
1 def display_game_over(self, message):
2     for row in self.squares:
3         for button in row:
4             button.configure(state="disabled")
5     popup = ctk.CTkToplevel(self.root)
6     popup.title("Game Over")
7     popup.geometry("300x150")
8     label = ctk.CTkLabel(popup, text=message, font=
9     ("Garamond", 16))
10    label.pack(pady=20)
11    button = ctk.CTkButton(popup, text="OK", command
12    =popup.destroy)
13    button.pack(pady=10)
```

➤ **switch\_turn(self):**

- Switches the turn between white and black.
- Checks if the game is over when the white player's turn ends (e.g., "Black wins by checkmate!").

```
1 def switch_turn(self):
2     self.turn = chess.BLACK if self.turn == chess.WHITE else
  chess.WHITE
3     if self.turn == chess.WHITE and self.board.is_checkmate
  ():
4         self.display_game_over("Black wins by checkmate!")
```

➤ **check\_king\_in\_check(self):**

- Checks if the king of the current player is in check and highlights its square if true.
- If the king is not in check, it removes the red highlight from the king's square.

```
1 def check_king_in_check(self):
2     if self.board.is_check():
3         if self.turn == chess.WHITE:
4             turn=chess.BLACK
5         else:
6             turn=chess.WHITE
7         king_square = self.board.king(turn)
8         king_row, king_col = divmod(king_square, 8)
9         self.king_in_check_square = (7 - king_row, king
  _col) # Convert to GUI coordinates
10        self.squares[7 - king_row][king_col].configure
  (fg_color="red")
11    else:
12        self.king_in_check_square = None
```

➤ **highlight\_moves(self, row, col):**

- Highlights possible moves for the selected piece.
- Highlights attackable squares in pink and other possible moves in light green.
- Calls reset\_colors() to reset square colors but retain the check highlight.

```
1 def highlight_moves(self, row, col):
2     self.reset_colors()
3     selected_piece = self.get_piece(row, col)
4     if selected_piece == "default":
5         return
6     piece_square = chess.square(col, 7 - row)
7     for move in self.board.legal_moves:
8         if move.from_square == piece_square:
9             to_square = move.to_square
10            to_row, to_col = divmod(to_square,
  8)
11            destination_piece = self.board.pie
  ce_at(to_square)
12            if destination_piece and destinati
  on_piece.color != self.board.piece_at(piece_squar
  e).color:
13                self.squares[7 - to_row][to_co
  l].configure(fg_color="pink")
14            else:
15                self.squares[7 - to_row][to_co
  l].configure(fg_color="light green")
```

➤ **reset\_colors(self):**

- Resets the background color of each square on the board.
- Keeps the red highlight on the king's square if it is in check.

```
1 def reset_colors(self):
2     for i in range(8):
3         for j in range(8):
4             if self.king_in_check_square == (i, j):
5                 continue
6             fg_color = "white" if (i + j) % 2 == 0 else "gray"
7             self.squares[i][j].configure(fg_color=fg_color)
8
```



➤ **update\_board(self):**

- Updates the board GUI to reflect the current board state.
- Loops through all squares and sets the correct piece image.

```
1 def update_board(self):
2     for i in range(8):
3         for j in range(8):
4             self.squares[i][j].configure(
5                 image=self.piece_images.get(self.get_piece(i, j),
6                 self.piece_images["default"])
7             )
```

➤ **make\_black\_move(self):**

- Uses the GeneticAlgorithm class to generate the best move for the black player.
- Applies the move, updates the board, and checks if the king is in check.
- Switches turns after the move.

```
1 def make_black_move(self):
2     if self.turn == chess.BLACK:
3         best_move = self.genetic_algorithm.genetic_algorithm(self.
4         board) # Get the best move using GA
5         self.board.push(best_move)
6         self.update_board()
7         self.check_king_in_check()
8         self.switch_turn()
```

4. *Snippets of the output:*

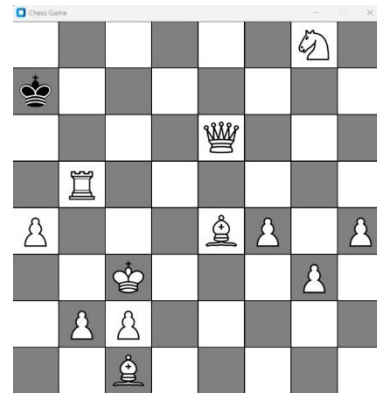
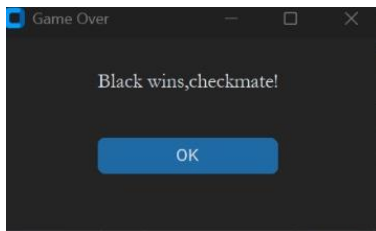
- When you run the code, this window appears.
- Click “Play” to start the game or “Quit” to exit the game.



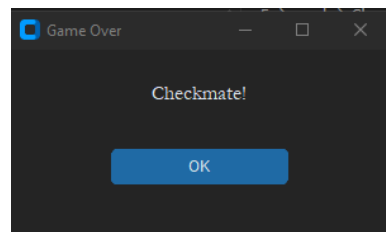
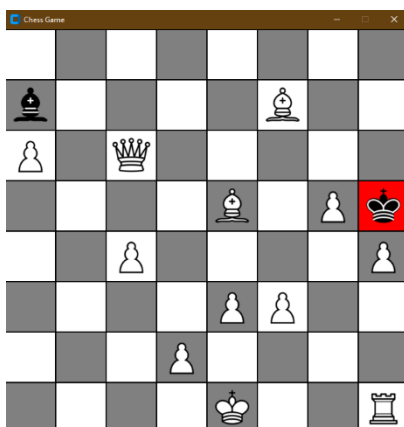
- The chess board appears, start playing and try to defeat the AI.



- AI wins!



- Player wins!



## Conclusion:

This code provides a comprehensive implementation of a chess game that integrates a custom graphical user interface (GUI) with a genetic algorithm for automated gameplay. The **ChessGUI** class builds a user-friendly environment where players can interact with the chessboard, make moves, and play against an AI powered by the **GeneticAlgorithm** class. The genetic algorithm leverages evolutionary principles to simulate decision-making, evaluating potential moves based on a fitness score and optimizing sequences over generations.

### *Key features of the code include:*

- A modern GUI built using customtkinter for an intuitive user experience.
- Integration of the python-chess library to manage game state, legal moves, and checkmate/stalemate conditions.
- A genetic algorithm that iteratively refines sequences of moves, assessing board positions for optimal performance.
- The ability to dynamically display the chessboard and its pieces, facilitating interaction through mouse events.

The code ensures that gameplay is interactive for human players and includes logic for AI opponents capable of learning and adapting to different positions on the board. While the AI's evaluation relies on material and position-based scoring, future enhancements could expand the algorithm to incorporate more advanced strategic considerations.

Overall, this implementation serves as a solid foundation for both learning and expanding the features of chess AI systems. It can be used as a basis for creating more sophisticated chess engines, experimenting with different AI algorithms, or simply playing a game against an automated opponent.

## Sources:

- <https://github.com/mrinoybanerjee/Genetic-ChessAlgorithm/blob/master/chess.py>
- <https://arxiv.org/pdf/1711.08337>
- [https://www.chessprogramming.org/Genetic\\_Programming#Genetic\\_Algorithms](https://www.chessprogramming.org/Genetic_Programming#Genetic_Algorithms)
- <https://youtu.be/EnYui0e73Rs?si=fRhZpA9C9F6vC47D>
- <https://youtu.be/nhT56blfRpE?si=LD-t9474OgQR1s2g>
- <https://youtu.be/OpL0Gcfn4B4?si=Xku1nV9su2BINscg>
- <https://youtu.be/OpL0Gcfn4B4?si=Xku1nV9su2BINscg>