Chess Game Project Report

Contributors:

- Adrian Florczak

- Gloria Delgado

- Alberto Cuervo

Project Overview

We embarked on creating a chess game application in Python, aiming to develop a game that could be enjoyed both in local multiplayer mode and by playing against a computer opponent. Our primary goal was to deliver an engaging and challenging chess experience.

Initial Ideas & Research

Our team explored several game ideas initially, including Tic Tac Toe and other chess-like games. After considering our resources and the complexity involved, we settled on developing a chess game. Chess provided a well-defined set of rules and a challenging problem to solve, which made it a perfect project for us.

Main Features

Local Multiplayer

Two players can play against each other on the same device, bringing the traditional chess experience to the digital realm.

Playing Against Computer

A single player can challenge a computer-controlled opponent, providing a solo play option with varying levels of difficulty.

Game Interface

We focused on designing a user-friendly interface that would make interacting with the game intuitive and enjoyable. This included a clear visual representation of the chessboard, smooth piece movement animations, and clear indicators for game status and player turns.

Game Experience

Computer Player

To create a challenging computer opponent, we implemented two strategies:

Random Player

Our initial approach was to have the computer make random moves.

- **Challenges**:

- Moves often lacked strategic depth.

- The gameplay felt repetitive and predictable.

- Randomness sometimes led to illogical moves, detracting from the gaming experience.

Alpha-Beta Algorithm

To improve the computer's play, we researched and implemented the alpha-beta pruning algorithm, which is a refined version of the minimax algorithm.

- *Research: We started with the minimax algorithm, which laid the foundation for understanding the more advanced alpha-beta pruning technique.

- Implementation: Integrating the alpha-beta algorithm into our code significantly enhanced the computer's ability to make strategic decisions, making it a more formidable opponent.

Code Description

Our project is organized into several key components:

Pygame Initialization

We used the Pygame library for handling graphics and user input. This setup included initializing the screen, loading images, and setting up fonts and colors.

```
import pygame
import chess
import random
import time
import sys

# Pygame initialization
pygame.init()

# Constants
SCREEN_WIDTH, SCREEN_HEIGHT = 1000, 600  # Increased width for side panel
BOARD_SIZE = 600  # Fixed board size
SQ_SIZE = BOARD_SIZE // 8
FPS = 30
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BUTTON_COLOR = (70, 130, 180)
HOVER_COLOR = (100, 149, 237)
HIGHLIGHT_COLOR = pygame.Color("yellow")
SIDEPANEL_WIDTH = SCREEN_WIDTH - BOARD_SIZE
SIDEPANEL_PADDING = 20
FONT_COLOR = BLACK
```

Board Representation

The chessboard is represented as an 8x8 grid. We used the `chess` library to handle board logic and piece movements.

```
# Main game function
def play_game(player1, player2):
    board = chess.Board()
```

Drawing Functions

We created helper functions to draw the chessboard, pieces, and UI elements.

```
# Helper function to draw board
def draw_board(board, selected_square=None):
    screen.blit(board_image, (0, 0))
    for row in range(8):
        for col in range(8):
            square = chess.square(col, 7 - row)
            piece = board.piece_at(square)
            if selected_square == square:
                pygame.draw.rect(screen, HIGHLIGHT_COLOR, pygame.Rect(col * SQ_SIZE, row * SQ_SIZE, SQ_SIZE, SQ_SIZE), 5)
            if piece:
                piece_color = 'W' if piece.color == chess.WHITE else 'B'
                piece_type = piece.symbol().upper()
                piece_image = pygame.image.load(f"images/{piece_color}{piece_type}.png")
                piece_image = pygame.transform.scale(piece_image, (SQ_SIZE, SQ_SIZE))
                screen.blit(piece_image, pygame.Rect(col * SQ_SIZE, row * SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

```
# Helper function to draw text
def draw_text(text, font, color, surface, x, y):
    textobj = font.render(text, True, color)
    textrect = textobj.get_rect(center=(x, y))
    surface.blit(textobj, textrect)
```

Game Logic

The core game logic manages player turns, move execution, check/checkmate detection, and updates the game state. It is managed by python library called “chess”. We can use it executing in terminal command below:

```
C:\Users\adria>pip install chess
```

Alpha-Beta Algorithm

For the computer opponent, we implemented the alpha-beta pruning algorithm to evaluate possible moves.

```
def alphabeta(board, depth, alpha, beta):
    # Returns a tuple (score, bestmove) for the position at the given depth
    if depth == 0 or board.is_checkmate() or board.is_stalemate() or board.is_fifty_moves():
        return [staticAnalysis5(board), None]
    else:
        if board.turn == chess.WHITE:
            bestmove = None
            for move in board.legal_moves:
                newboard = board.copy()
                newboard.push(move)
                score_and_move = alphabeta(newboard, depth - 1, alpha, beta)
                score = score_and_move[0]
                if score > alpha:  # white maximizes her score
                    alpha = score
                    bestmove = move
                if alpha >= beta:  # alpha-beta cutoff
                    break
            return [alpha, bestmove]
        else:
            bestmove = None
            for move in board.legal_moves:
                newboard = board.copy()
                newboard.push(move)
                score_and_move = alphabeta(newboard, depth - 1, alpha, beta)
                score = score_and_move[0]
                if score < beta:  # black minimizes his score
                    beta = score
                    bestmove = move
                if alpha >= beta:  # alpha-beta cutoff
                    break
            return [beta, bestmove]
```

Technical Challenges and Solutions

Random Player Limitations

The initial random player approach resulted in uninteresting and often illogical gameplay. This was resolved by implementing the more sophisticated alpha-beta algorithm.

Algorithm Integration

Integrating the alpha-beta algorithm posed challenges in terms of complexity and required extensive testing. We iteratively tested and refined the algorithm to ensure it worked effectively within our game.

Testing and Feedback

We conducted thorough testing, including feedback from individuals outside our development team. This feedback was invaluable in identifying and fixing bugs and enhancing the game's overall experience. Testers found the computer opponent to be a worthy challenge, which validated the effectiveness of our alpha-beta implementation.

Live Demonstration

We presented a live demonstration of the game, highlighting its features and showcasing the strategic depth provided by the alpha-beta algorithm, which made the computer a challenging opponent.

Conclusion

We successfully developed a chess game that supports both local multiplayer and playing against a computer opponent. The implementation of the alpha-beta algorithm was a significant achievement, greatly enhancing the computer's gameplay. Overall, we created an engaging and challenging chess experience that met our initial goals.

Presentation Slides

The attached presentation offers additional visual insights into our development process, challenges faced, and solutions implemented.