FLIPTURN - Un Juego de Estrategia con IA y Habilidades

Autores: Daniel trigo, Julián Gallardo
Carrera: ITI
Rut: 21.374.808-4, 21.836.146-3

Descripción General

Este proyecto es una implementación avanzada del clásico juego de mesa **Reversi (también conocido como Othello)**. Desarrollado en C++ con la librería SFML para su interfaz gráfica (GUI), esta versión ofrece una experiencia de juego rica con inteligencia artificial, modos de juego variados y un sistema de habilidades estratégico.

El desarrollo se ha enfocado en la modularidad del código y la aplicación práctica de algoritmos y estructuras de datos.

Características del Juego

- **Modos de Juego:**
  - **Humano vs. Humano:** Partida local para dos jugadores.
  - **Humano vs. IA:** Desafía a una Inteligencia Artificial integrada.
- **Inteligencia Artificial (IA) Avanzada:**
  - Implementada mediante el algoritmo **Minimax** con **Poda Alpha-Beta** para optimizar la toma de decisiones de la IA, permitiéndole evaluar movimientos futuros.
  - Utiliza **Tablas de Transposición (Zobrist Hashing)** para almacenar y reutilizar evaluaciones de estados del tablero previamente visitados, lo que acelera significativamente la búsqueda de la IA en situaciones repetitivas del juego.

- **Interfaz Gráfica (GUI) Dinámica:**
    - Construida con SFML, con un diseño adaptable que permite una visualización correcta al maximizar la ventana.
    - **Layout Mejorado:** El tablero se posiciona al centro, con paneles laterales dedicados a estadísticas de los jugadores y botones de habilidades.
- **Sistema de Recursos y Habilidades:**
    - Al capturar fichas enemigas, los jugadores ganan **puntos de energía.**
    - La energía se puede gastar para activar habilidades especiales:
        - **Intercambiar Color:** Voltea una ficha del oponente a tu color, con un costo de energía y un uso limitado por partida.
        - **Previsualización:** Permite previsualizar los movimientos y las fichas que se voltearían al mover el ratón sobre el tablero, con un costo de energía al activarse el modo.
        - **Volteo Forzado:** Gasta energía para voltear una ficha específica del oponente sin necesidad de encerrarla.
- **Bonificación por Dominación de Esquinas**: Los jugadores reciben puntos de bonificación al final de la partida por cada esquina del tablero que controlen.
- **Tiempo de respuesta optimizado (IA)**: Se aplicó una profundidad de búsqueda de la Inteligencia artificial creada de 4, para evitar altos tiempos de respuesta en los movimientos. El tiempo promedio de respuesta de la IA es de 500ms (0,5 segundos), esto considerando movimientos en etapas finales del juego, en los cuales la IA por falta de espacios para generar movimientos ocupa más tiempo buscando el mas optimo.

Estructuras de Datos Aplicadas

El diseño del juego aprovecha diversas estructuras de datos para su funcionamiento eficiente:

- **Vectores (`std::vector`):**
    - Utilizados para representar el **tablero de juego** (`std::vector<std::vector<char>>`) y almacenar dinámicamente las **listas de movimientos posibles** (`std::vector<Movimiento>`) o las **fichas a voltear** (`std::vector<std::pair<int, int>>`). Su capacidad de redimensionarse dinámicamente es fundamental para la flexibilidad del juego.
- **Pilas (`std::stack`):**
    - La pila `historial` en la clase `GestorJuego` almacena los **estados previos del tablero**, permitiendo la gestión del flujo del juego y facilitando posibles funcionalidades de "deshacer" (aunque no implementadas explícitamente en la GUI).
- **Mapas (`std::map` - para Tablas de Transposición):**
    - Utilizado en la clase `IA` para implementar la `tabla_transposicion`. Este mapa asocia un `unsigned long long` (el hash Zobrist de un estado del tablero) con una `EntradaTransposicion` (que guarda el valor evaluado de ese estado y la profundidad de la búsqueda). Permite a la IA recordar y reutilizar los resultados de cálculos anteriores, optimizando enormemente el rendimiento del algoritmo Minimax.
- **Árboles (Conceptual - Árbol de Juego):**
    - Aunque no se implementa un objeto `Árbol` explícito, el algoritmo Minimax opera sobre un **árbol de juego** conceptual. Cada nodo es un estado del tablero, y las ramas son los posibles movimientos. La IA "recorre" este árbol para encontrar la mejor jugada.

Diseño de Clases y Documentación del Código

El proyecto está modularizado en varias clases, cada una con una responsabilidad clara.

1. ConfiguracionJuego.h

- **Propósito:** Centraliza todas las **constantes y enumeraciones globales** del juego y la GUI. Esto incluye el tamaño del tablero, los caracteres de las fichas, los estados del juego y de la GUI, modos de jugador, dimensiones de la ventana, y los costos de energía de las habilidades.
- **Contiene:** `TAMANO_TABLERO`, `JUGADOR_X`, `JUGADOR_O`, `EstadoJuego`, `ModoJugador`, constantes de dimensiones de la GUI (`ANCHO_VENTANA`, `ALTO_VENTANA`, `ANCHO_PANEL_LATERAL`, etc.), costos de habilidades (`COSTO_INTERCAMBIO_COLOR`, `COSTO_FICHA_BOMBA`, etc.), `EstadoGUI`, y `EstadoHabilidad`.
- **Relación:** Es el archivo más básico; todos los demás archivos `.h` lo incluyen para acceder a estas definiciones fundamentales, asegurando consistencia en todo el proyecto.

2. Movimiento.h

- **Propósito**: Define una estructura simple (`struct Movimiento`) para representar un movimiento en el tablero, con sus coordenadas de fila y columna.
- **Contiene:** `fila`, `columna` (miembros).
- **Relación:** Utilizado por `Tablero` para devolver movimientos y por `IA` y `GestorJuego` para manipular y procesar movimientos.

3. TablaTransposicion.h y TablaTransposicion.cpp

- **Propósito:** Implementa el **hashing Zobrist** y la estructura de la **tabla de transposición** para optimizar la IA.
- **Contiene:**
  - `TABLA_ZOBRIST`: Un array tridimensional externo para el hashing.
  - `inicializar_tabla_zobrist()`: Inicializa la tabla al inicio del programa.
  - `EntradaTransposicion`: Estructura para almacenar valores en la tabla de transposición.
- **Relación:** `Tablero` llama a `calcular_hash_zobrist()` para obtener el hash de su estado actual. `IA` utiliza esta tabla para almacenar y recuperar estados, evitando cálculos repetidos.

4. `Tablero.h` y `Tablero.cpp`

- **Propósito:** Representa el **estado actual del tablero de Reversi** y gestiona toda la lógica inherente al tablero.
- **Contiene:**
    - `tablero`: Un `std::vector<std::vector<char>>` que almacena el estado de las fichas.
    - `hash_actual`: Almacena el hash Zobrist del estado actual.
    - `inicializar_tablero()`: Configura el tablero al inicio de una partida.
    - `calcular_hash_zobrist()`: Calcula el hash del tablero actual.
    - `obtener_fichas_a_voltear(fila, columna, jugador)`: Determina y devuelve las fichas a voltear.
    - `es_movimiento_valido(fila, columna, jugador)`: Verifica la legalidad de un movimiento.
    - `realizar_movimiento(fila, columna, jugador)`: Ejecuta un movimiento y voltea las fichas correspondientes.
    - `obtener_movimientos_posibles(jugador)`: Genera una lista de movimientos válidos para un jugador.
    - `contar_fichas(jugador)`: Cuenta las fichas de un jugador.
    - `es_juego_terminado(jugador_x, jugador_o)`: Determina si la partida ha finalizado.
- **Relación:** `GestorJuego` contiene una instancia de `Tablero` y la manipula. `IA` también interactúa con `Tablero` a través de `GestorJuego` para simular jugadas.

5. `IA.h` y `IA.cpp`

- **Propósito:** Encapsula la **Inteligencia Artificial** del oponente. Contiene el algoritmo **Minimax** con poda Alpha-Beta y la lógica para evaluar estados del tablero y tomar decisiones.
- **Contiene:**
    - `profundidad_maxima`: La profundidad de búsqueda de la IA.
    - `tabla_transposicion`: El `std::map` para las tablas de transposición.
    - `encontrar_mejor_movimiento(GestorJuego& gestor_juego, jugador_ia)`: Método principal para que la IA elija su mejor jugada.
    - `evaluar_tablero(const GestorJuego& gestor_juego, jugador_ia)`: Función heurística para puntuar un estado del tablero, considerando fichas, esquinas, movilidad y energía.
    - `minimax(GestorJuego gestor_juego_estado, profundidad, es_maximizador, alfa, beta, jugador_actual)`: Implementación recursiva del algoritmo Minimax.
- **Relación:** `GestorJuego` crea una instancia de `IA` y le solicita el siguiente movimiento.

6. `GestorJuego.h` y `GestorJuego.cpp`

- **Propósito:** Es el **cerebro central de la lógica del juego**. Mantiene el estado general de la partida, coordina las interacciones entre jugadores y tablero, y gestiona las habilidades, recursos y la puntuación de esquinas.
- **Contiene:**
    - `tablero`: Instancia del tablero de juego.
    - `historial`: Pila de estados del tablero para el historial de movimientos.
    - `jugador_actual`, `estado_juego`, `modo_actual`: Variables de estado de la partida.
    - `energia_jugador_x`, `energia_jugador_o`: Puntos de energía de cada jugador.
    - `puntos_esquina_jugador_x`, `puntos_esquina_jugador_o`: Puntos de bonificación por esquinas.
    - `intentar_mover_jugador(fila, columna)`: Procesa el movimiento de un jugador humano.
    - `procesar_turno_ia()`: Ejecuta el turno de la IA.
    - `actualizar_estado_juego()`: Determina el estado final del juego y al ganador, incluyendo puntos de esquina.
    - `reiniciar_juego()`: Restablece la partida.
    - `cambiar_turno()`: Gestiona el cambio de jugador.
    - `puede_comprar_habilidad(jugador, costo)`: Verifica si hay energía para una habilidad.
    - `gastar_energia(jugador, costo)`, `ganar_energia_por_volteo(jugador, fichas_volteadas)`: Gestionan la energía.
    - `realizar_intercambio_color(...)`, `realizar_volteo_forzado(...)`, `realizar_ficha_bomba(...)`: Implementan la lógica de cada habilidad.
- **Relación:** Es el controlador principal, orquestando las interacciones entre `Tablero`, `IA` y `GUIReversi`.

7. `GUIReversi.h` y `GUIReversi.cpp`

- **Propósito:** La **Interfaz Gráfica de Usuario (GUI)** del juego. Se encarga de todo lo que el usuario ve y con lo que interactúa, usando SFML.
- **Contiene:**
    - Objetos SFML para la ventana (`ventana`), textos (`texto_estado`, `titulo_menu`, etc.), botones (`boton_reiniciar`, `boton_jugar_ia`, `boton_ficha_bomba`, etc.), y la forma de la ficha fantasma (`forma_ficha_fantasma`).
    - Variables de estado de la GUI (`estado_gui`, `estado_habilidad_actual`).
    - `inicializar_graficos()`: Configura la apariencia inicial de los elementos.
    - `ajustar_vista(ancho, alto)`: Adapta la vista de la ventana al redimensionarse.
    - `reposicionar_elementos_gui()`: Recalcula y aplica las posiciones de todos los elementos de la GUI para el layout correcto (paneles laterales, centrado).
    - `ejecutar()`: Bucle principal que maneja eventos, actualiza y renderiza.

- `procesar_eventos()`: Captura y maneja la entrada del usuario.
- `manejar_clic_tablero(fila, columna)`: Procesa los clics en el tablero, activando movimientos o habilidades.
- Métodos `dibujar_...()`: Funciones específicas para dibujar cada componente visual.
- `dibujar_paneles_laterales()`: Dibuja los textos de estadísticas en los paneles laterales.

- **Relación**: Es la capa de presentación. Lee el estado de `GestorJuego` para saber qué dibujar y envía las acciones del usuario de vuelta a `GestorJuego`.

8. `main.cpp`

- **Propósito**: Es el punto de entrada principal del programa.
- **Contiene:**
  - Llama a `inicializar_tabla_zobrist()` al inicio.
  - Crea instancias de `GestorJuego` y `GUIReversi`.
  - Inicia el bucle principal del juego llamando a `gui.ejecutar()`.
- **Relación**: Orquesta el inicio de todos los componentes del juego.