Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0217 – Estructuras Abstractas de Datos y Algoritmos para Ingeniería
III ciclo 2024

Proyecto Final

4 En Linea

Link del repositorio de GitHub

Integrantes:
Isaí González S - C03457
Mario Fabian Rocha Morales - B96561
Jr Ruiz Sánchez - B97026
Justin Jimenez Jimenez - B94037

Profesora: Ing. Karen Dayana Tobar Parra,

3 de marzo de 2025

Índice

1. Introducción

El presente documento describe el desarrollo de un juego de **4 en Raya** implementado en **C++**, en el marco del curso *IE0217 – Estructuras Abstractas de Datos y Algoritmos para Ingeniería*. Este proyecto tiene como objetivo aplicar estructuras de datos avanzadas y desarrollar habilidades en programación orientada a objetos, modularidad y optimización de algoritmos.

El juego consiste en un tablero de 7 columnas por 6 filas donde los jugadores colocan fichas con el objetivo de conectar cuatro en línea. Se han implementado tres modalidades:

- **Jugador vs. Jugador**: Dos jugadores alternan turnos.
- **Jugador vs. Bot Aleatorio**: Un bot coloca fichas en columnas aleatorias.
- **Jugador vs. Bot Inteligente**: Un bot que analiza el tablero y selecciona la columna más conveniente.

El proyecto incluye una interfaz gráfica con menú y tablero, y un sistema de persistencia para guardar partida.

2. Objetivos

Los objetivos principales del proyecto incluyen:

- Implementar la lógica del juego de manera eficiente con estructuras dinámicas.
- Desarrollar un bot con distintos niveles de inteligencia.
- Diseñar una interfaz gráfica interactiva.
- Implementar persistencia de datos para guardar y reanudar partidas.
- Evaluar y optimizar el desempeño de los algoritmos utilizados.

3. Clases y función main

3.1. Clase `Matrix6x7`

La clase `Matrix6x7` es la responsable de manejar la estructura interna del tablero, representado por una matriz de 6 filas y 7 columnas. A continuación, se detalla el comportamiento de cada método:

3.1.1. `fill()`

- **Propósito:** Reinicializa la matriz, poniendo todas las celdas a 0.
- **Funcionamiento:** Recorre cada fila (utilizando un `for-each`) y asigna 0 a cada celda. Esto asegura que el tablero esté "limpio" antes de iniciar o reiniciar una partida.

3.1.2. `get_table()`

- **Propósito:** Proporcionar acceso de solo lectura al estado actual de la matriz.
- **Funcionamiento:** Retorna una referencia constante al vector bidimensional que representa el tablero, evitando modificaciones accidentales desde el exterior.

3.1.3. `insert(int col, int player)`

- **Propósito:** Insertar la ficha de un jugador en la columna especificada.
- **Funcionamiento:**
  - Verifica que la celda superior de la columna (fila 0) esté vacía (valor 0).
  - Si es posible, coloca la ficha del jugador (representado por un entero) en dicha posición y retorna `true`.
  - Si la columna ya está ocupada, retorna `false`.

3.1.4. `gravity(int col)`

- **Propósito:** Simular la gravedad en la columna, haciendo que la ficha caiga hasta la posición correcta.
- **Funcionamiento:**
  - Recorre la columna desde la fila superior hasta la penúltima.
  - Si encuentra una ficha que tenga una celda vacía inmediatamente debajo, intercambia ambas celdas.
  - Retorna la nueva posición (fila) de la ficha, lo que ayuda a identificar cuándo la ficha ha llegado a su posición final.

3.1.5. `explore(int R, int C, int dR, int dC)`

- **Propósito:** Desde una posición dada, explora en la dirección definida por los incrementos `dR` y `dC` para buscar secuencias de fichas consecutivas.
- **Funcionamiento:**
  - Limpia el vector auxiliar `path`.
  - Comienza en la posición `(R, C)` y, mientras se mantenga dentro de los límites de la matriz, almacena el valor de cada celda en `path`.
  - Una vez completada la exploración, comprueba si los primeros cuatro elementos son iguales y distintos de 0, indicando una secuencia ganadora. De ser así, retorna `true`; de lo contrario, `false`.

3.1.6. `exploreDirections(int col)`

- **Propósito:** Verificar, para cada celda no vacía de una columna, si existe alguna dirección en la que se forme una línea ganadora.
- **Funcionamiento:**
  - Recorre cada fila de la columna especificada.
  - Para cada celda con un ficha, llama a `explore` en varias direcciones: vertical, horizontal (ambas direcciones) y diagonales.
  - Si alguna de las exploraciones encuentra cuatro fichas consecutivas, retorna `true`.

3.2. Clase `Bot`

La clase `Bot` implementa la inteligencia artificial del juego. Utiliza una copia interna del tablero para simular movimientos y evaluar jugadas. A continuación, se explica cada método:

3.2.1. `updateMatrix(const Matrix6x7 &board)`

- **Propósito:** Sincronizar la copia interna del estado del tablero con el objeto real de la clase `Matrix6x7`.
- **Funcionamiento:** Llama al método `get_table()` del tablero y actualiza la variable `board_copy`. También actualiza los índices máximos de filas y columnas.

3.2.2. `gravity(int col)`

- **Propósito:** Determinar la fila donde una ficha se posicionará al caer en la columna `col`.
- **Funcionamiento:** Recorre la columna desde la parte inferior hacia arriba y retorna la primera fila vacía.

3.2.3. `scanTable()`

- **Propósito:** Contar las fichas consecutivas del mismo jugador a partir de una posición, en una dirección específica.
- **Funcionamiento:**
  - Inicializa un contador (`count`) en 1, contando la ficha en la posición `(R, C)`.
  - Realiza dos exploraciones:
    - **Hacia adelante:** Aumenta la posición usando `dR` y `dC` hasta 3 pasos.
    - **Hacia atrás:** Decrementa la posición en la misma dirección.
  - Durante cada exploración, si la celda tiene el valor del jugador se incrementa `count`. Si la celda está vacía, se incrementa un contador de espacios vacíos.
  - Si se pasa el parámetro `checkPattern` y se cumple una condición estratégica (por ejemplo, 3 fichas seguidas con al menos un espacio vacío), retorna `true`. En caso contrario, retorna `true` si `count` es 4 o más.

3.2.4. `exploreBot(int R, int C, int player, bool checkPattern)`

- **Propósito:** Desde una posición dada, examinar en cuatro direcciones principales si existe una condición ganadora o un patrón estratégico.
- **Funcionamiento:**
  - Define un arreglo con las direcciones (horizontal, vertical y dos diagonales).
  - Para cada dirección, llama a `scanTable` y, si se detecta una condición favorable, retorna `true`.

3.2.5. `botMove()`

- **Propósito:** Determinar la jugada óptima para el bot en el modo 1 vs AI.
- **Funcionamiento:**

- Recolecta todas las columnas válidas (aquellas cuya celda superior está vacía).
- Primero, simula cada movimiento en la copia del tablero para verificar si la jugada resulta en victoria inmediata para el bot.
- Si no se puede ganar de forma inmediata, simula los movimientos del oponente para detectar posibles victorias y bloquearlas.
- Si aún no se detecta ninguna jugada crítica, evalúa patrones estratégicos que puedan conducir a una victoria en el futuro.
- Como última estrategia, prioriza las columnas centrales (ya que ofrecen mayor conectividad) y, en caso de no haber una estrategia definida, elige aleatoriamente una columna válida.

3.2.6. `randomColumn()`

- **Propósito:** Seleccionar aleatoriamente una columna válida para el modo 1 vs Bot.
- **Funcionamiento:** Revisa todas las columnas y almacena aquellas en las que se puede insertar una ficha (donde la celda superior es 0). Luego retorna una columna aleatoria de esta lista.

3.3. Clase `SaveLoad`

La clase `SaveLoad` se encarga de la persistencia de la partida. Permite guardar movimientos, cargar una partida guardada y reiniciar el estado. Se explica a continuación cada método con mayor detalle:

3.3.1. `setGameType(const string &tipo)`

- **Propósito:** Establecer el tipo de partida (por ejemplo, "1 vs 1", "1 vs Bot", "1 vs AI") y escribirlo en el archivo de guardado.
- **Funcionamiento:**
  - Asigna el valor recibido al atributo `gameType`.
  - Abre el archivo (con nombre especificado en `filename`) en modo `trunc`, lo que elimina el contenido anterior.
  - Escribe el tipo de partida en la primera línea del archivo.

3.3.2. `save(int columna, int jugador)`

- **Propósito:** Registrar cada movimiento tanto en memoria como en un archivo.
- **Funcionamiento:**
  - **Validaciones:** Se asegura de que la columna se encuentre entre 0 y 6 y que el identificador del jugador sea 0 o 1.
  - **Empaquetado en 4 bits:**
    - El método empaqueta la información del movimiento en un único entero usando operaciones de bits.

- El bit 3 se utiliza para representar el jugador y los bits 0 a 2 para la columna. Esto se realiza mediante la siguiente operación:

  ```
  move = (jugador << 3) | (columna & 0x07)
  ```

- Este empaquetado compacta la información, facilitando su almacenamiento y posterior lectura.

- **Guardado en el archivo:**
  - El movimiento empaquetado se añade al vector `movimientos`.
  - Se abre el archivo en modo `append` para no sobrescribir los movimientos previos.
  - El movimiento se escribe en el archivo en formato hexadecimal, seguido de un espacio. Esto permite una representación compacta y legible de la secuencia de movimientos.

3.3.3. `load()`

- **Propósito:** Recuperar una partida previamente guardada.
- **Funcionamiento:**
  - Borra la secuencia actual de movimientos almacenada en el vector `movimientos`.
  - Abre el archivo en modo lectura.
  - Lee la primera línea para recuperar el tipo de partida y la asigna a `gameType`.
  - Lee el resto del archivo, interpretando cada movimiento en formato hexadecimal y añadiéndolo al vector.
  - Para cada movimiento, se extraen los datos mediante operaciones de bits:
    - Se obtiene el identificador del jugador desplazando el entero 3 bits a la derecha.
    - Se obtiene la columna aplicando una máscara con `0x07`.
  - Se ajusta el valor del jugador (por ejemplo, transformando 0 en 1 y 1 en -1) para adecuarlo a la lógica del juego.
  - Se aplican los movimientos en la matriz de juego invocando sucesivamente los métodos `insert` y `gravity`. Se llama a `gravity` múltiples veces para simular el efecto de la ficha cayendo a la posición final.

3.3.4. `restart()`

- **Propósito:** Reiniciar la partida eliminando todos los movimientos guardados.
- **Funcionamiento:**
  - Limpia el vector `movimientos`.
  - Abre el archivo en modo `trunc` para borrar todo su contenido, preparando el entorno para una nueva partida.

3.3.5. `getGameType()` y `getMovimientos()`

- **Propósito:** Proveer métodos *getter* para acceder de forma segura al tipo de partida y a la secuencia de movimientos.
- **Funcionamiento:** Cada método retorna una referencia constante al atributo correspondiente, evitando modificaciones accidentales desde fuera de la clase.

3.4. Clase `GameWindow`

La clase `GameWindow` administra la interfaz gráfica y la interacción del usuario con el tablero. Se integran funcionalidades de dibujo, animación y comunicación con el sistema de guardado/carga. A continuación, se detalla cada método:

3.4.1. `get_game()`

- **Propósito:** Proveer acceso al objeto `Matrix6x7` que contiene el estado actual del tablero.
- **Funcionamiento:** Retorna una referencia a la instancia de la matriz de juego, permitiendo a otros métodos consultar o modificar el estado del tablero.

3.4.2. `update_board()`

- **Propósito:** Actualizar visualmente el tablero en la interfaz gráfica.
- **Funcionamiento:**
  - Recorre todas las celdas del grid y solicita su redibujado mediante `gtk_widget_queue_draw()`.
  - Se procesan los eventos pendientes de GTK para asegurar que los cambios se reflejen inmediatamente en pantalla.

3.4.3. `on_button_column_clicked(int col)`

- **Propósito:** Gestionar el evento de clic en uno de los botones de columna.
- **Funcionamiento:**
  - Llama a `drop_piece(col, current_player)` para insertar la ficha en la columna seleccionada.
  - Una vez realizado el movimiento, invierte el turno actual cambiando el signo de `current_player`.

3.4.4. `drop_piece(int column, int player)`

- **Propósito:** Insertar la ficha del jugador en la columna indicada, iniciar la animación de caída y guardar el movimiento.
- **Funcionamiento:**
  - Desactiva temporalmente todos los botones para evitar múltiples inserciones mientras se anima la caída.
  - Intenta insertar la ficha en la matriz. Si la inserción es exitosa:
    - Actualiza la visualización del tablero.

- Mapea el valor del jugador (por ejemplo, 1 se guarda como 0 y -1 como 1) y llama al método `save` de `SaveLoad` para registrar el movimiento.
- Inicia un temporizador (`drop_timer_callback`) que simula la caída de la ficha, actualizando su posición hasta llegar a la posición final.

- Si la inserción falla, vuelve a habilitar los botones de columna según corresponda.

3.4.5. `setGameMode(int mode)`

- **Propósito:** Configurar el modo de juego seleccionado ("1 vs 1", "1 vs Bot" o "1 vs AI").
- **Funcionamiento:**
  - Asigna el valor recibido a `game_mode` y actualiza el título de la ventana acorde al modo.
  - Si se selecciona un modo con bot ("1 vs Bot" o "1 vs AI"), crea o reinicializa el objeto `Bot` para gestionar el oponente automatizado.

3.4.6. `performBotMove()`

- **Propósito:** Ejecutar la jugada del bot cuando le corresponde el turno.
- **Funcionamiento:**
  - Actualiza la copia interna del tablero del bot.
  - Dependiendo del modo de juego, selecciona la jugada adecuada:
    - `randomColumn()` para "1 vs Bot" (jugada aleatoria).
    - `botMove()` para "1 vs AI" (jugada estratégica).
  - Si se obtiene una columna válida, inserta la ficha mediante `drop_piece` y alterna el turno.

3.4.7. `setCurrentPlayer(int player)`

- **Propósito:** Permitir la actualización del turno actual, especialmente útil al cargar una partida.
- **Funcionamiento:** Simplemente asigna el valor recibido a la variable `current_player`.

3.4.8. `getSaveLoad()`

- **Propósito:** Proveer acceso al objeto `SaveLoad` para operaciones de guardado y carga.
- **Funcionamiento:** Retorna el puntero a la instancia de `SaveLoad`.

3.4.9. `drop_timer_callback(gpointer data)`

- **Propósito:** Función de callback encargada de la animación de la caída de la ficha.
- **Funcionamiento:**
  - Llama al método `gravity` de la matriz para mover la ficha hacia abajo.
  - Si la ficha ha llegado a la posición final, desactiva la animación y verifica si se ha cumplido la condición ganadora mediante `exploreDirections`.

- En caso de victoria, muestra un mensaje y reinicia el tablero. Si la partida continúa y es turno del bot, programa la jugada del bot en modo idle.

3.5. Clase `MenuWindow`

La clase `MenuWindow` administra el menú principal de la aplicación. Permite al usuario:

- Iniciar una nueva partida seleccionando el modo ("1 vs 1", "1 vs Bot" o "1 vs AI").
- Cargar una partida previamente guardada.

El método principal es:

3.5.1. `on_menu_button_clicked(GtkWidget *widget, gpointer data)`

- **Propósito:** Responder al clic en los botones del menú.
- **Funcionamiento:**
  - Determina qué botón fue presionado leyendo su etiqueta.
  - Según la opción:
    - **"Cargar partida":**
      - Crea una nueva instancia de `GameWindow`.
      - Usa un objeto `SaveLoad` para cargar el estado guardado (tipo de partida y movimientos).
      - Ajusta el modo de juego y el turno según la información leída.
    - **"1 vs 1", "1 vs Bot" o "1 vs AI":**
      - Crea una nueva instancia de `GameWindow`.
      - Configura el modo de juego apropiado.
      - Reinicia el archivo de guardado y establece el tipo de partida.
    - Si se selecciona una opción no implementada, muestra un mensaje informativo.

3.6. Función `main`

El `main` es el punto de entrada de la aplicación. Su código es el siguiente:

```
#include <gtk/gtk.h>
#include "gui4connect.hpp"

int main(int argc, char *argv[]){
    gtk_init(&argc, &argv);
    MenuWindow menu;
    gtk_main();
    return 0;
}
```

Explicación:

- **Inicialización de GTK:**

- `gtk_init` prepara la biblioteca GTK con los parámetros de la línea de comandos, lo que es necesario para gestionar la interfaz gráfica.

- **Creación del menú principal:**
  - Se crea una instancia de `MenuWindow`, lo que lanza la ventana del menú donde el usuario puede elegir iniciar o cargar una partida.
- **Bucle principal:**
  - `gtk_main()` inicia el bucle de eventos de GTK, encargado de escuchar y gestionar los eventos (clics, redibujados, etc.) de la interfaz.

4. Desarrollo del Proyecto

4.1. Estructura del Juego

El juego se basa en una matriz de 6×7 representada mediante un vector de vectores para componer la matriz donde se almacenaran los datos de las fichas. Las fichas estan representadas de la siguiente manera:

- 0: Celda vacía.
- 1: Ficha del jugador.
- -1: Ficha del bot.

La implementación se divide en:

- **Matrix6x7**: Estructura del tablero.
- **Bot**: Algoritmos de decisión.
- **GUI**: Representación gráfica.

4.2. Implementación del Bot Inteligente

El bot sigue la siguiente lógica:

1. Analiza el tablero al rededor de la última ficha ingresada.
2. Verificar si puede ganar en la siguiente jugada, es decir, si posee tres dichas alineadas en vertical horizontal o diagonal elegirá la columna para completar 4 en linea.
3. Bloquear jugadas del oponente, es decir, si encuentra 3 fichas seguidas del usuario tira la ficha en la columna que bloquea una 4 ficha del usuario.
4. Buscar jugadas estratégicas para futuras victorias, busca seguir alineando fichas.
5. Priorizar el centro del tablero, ya que suele ser más conveniente para generar jugadas.
6. Si no hay opciones estratégicas, seleccionar una columna válida aleatoria.

Para evaluar el estado del juego, el bot explora cuatro direcciones posibles (horizontal, vertical y diagonales).

4.3. Interfaz Gráfica y Persistencia

El juego incluye una interfaz gráfica implementada con GTK.

Representación del Tablero:

Se dibuja una cuadrícula de 7 columnas por 6 filas donde se colocan las fichas. Cada celda del tablero se representa como un círculo o una imagen en la ventana gráfica. Se utilizan colores distintos para representar fichas del jugador (azul) y fichas del bot (rojo).

Interacción del Usuario:

El usuario selecciona una columna haciendo clic con el mouse sobre el área correspondiente. La interfaz detecta la acción y traduce la posición del clic en la columna correspondiente. Luego, se llama a la función de lógica del juego para validar el movimiento y actualizar el estado del tablero.

Actualización del Tablero:

Después de cada turno, la interfaz redibuja el tablero para reflejar los cambios en tiempo real. Se verifica si hay un ganador.

Conexión con la Lógica del Juego:

La interfaz gráfica se comunica con el código del juego a través de eventos y funciones específicas. Usa la clase Matrix6x7 para obtener y modificar el estado del tablero. En el modo Jugador vs. Bot, la interfaz espera la decisión del bot y actualiza el tablero en consecuencia.

En cuanto a la persistencia, el juego implementa un sistema que permite guardar y recuperar partidas en curso, garantizando que los jugadores puedan continuar una sesión previa sin perder el progreso. Esta funcionalidad es crucial para mejorar la experiencia del usuario y asegurar la continuidad del juego.

Funcionamiento de la Persistencia: La lógica de persistencia se basa en la escritura y lectura de archivos de texto. Cada vez que el usuario decide cerrar una partida, el estado actual del tablero se almacena en un archivo, junto con información adicional la secuencia en la que se ingresaron las fichas por lo que se guarda tanto el último estado de la matriz como el turno siguiente.

5. Análisis de Resultados

Se evaluó el desempeño del bot en distintos escenarios:

- **Modo aleatorio**: Sin estrategia, pierde fácilmente.
- **Modo inteligente**: Bloquea jugadas clave y busca oportunidades de victoria.
- **Eficiencia**: El bot analiza el tablero sin afectar la fluidez del juego hasta con 3 movimientos de previsión.

Se realizaron múltiples pruebas para validar la lógica, turnos e interfaz gráfica.

A continuación se muestra el menú de la interfaz gráfica con las distintas opciones descritas, además de un modo de juego donde ser observan las fichas y la dinámica de la interfaz en funcionamiento. Por último el mensaje de la condición cuando una de las dos partes ha ganado.

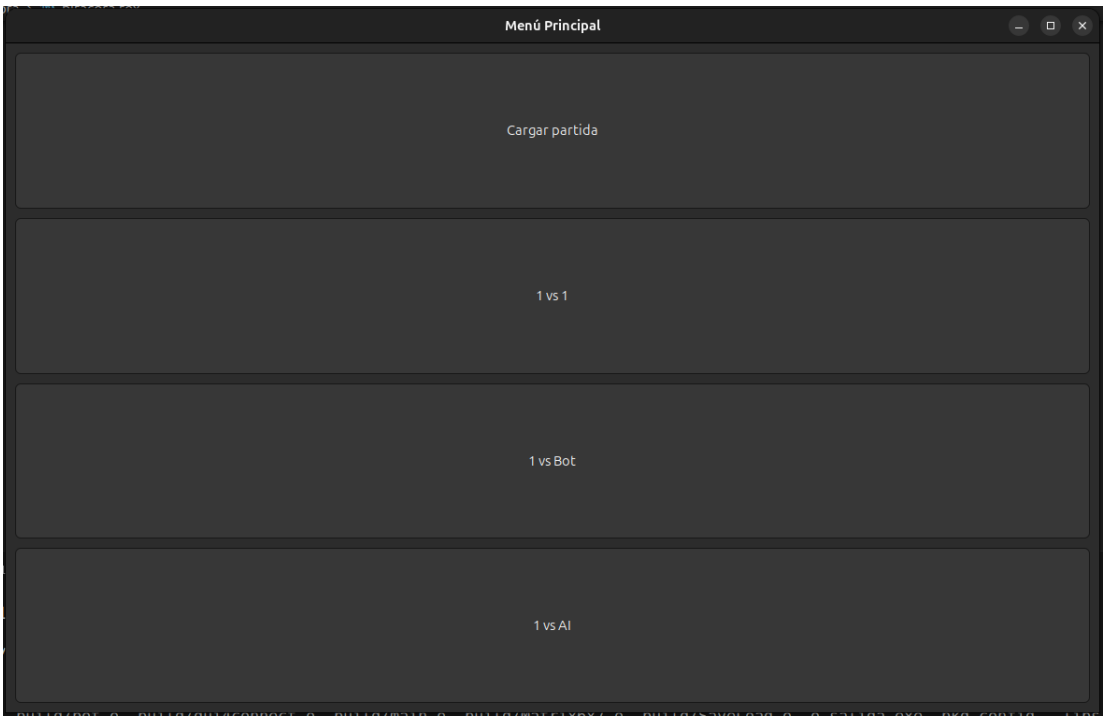

Figura 1: Menú del Juego

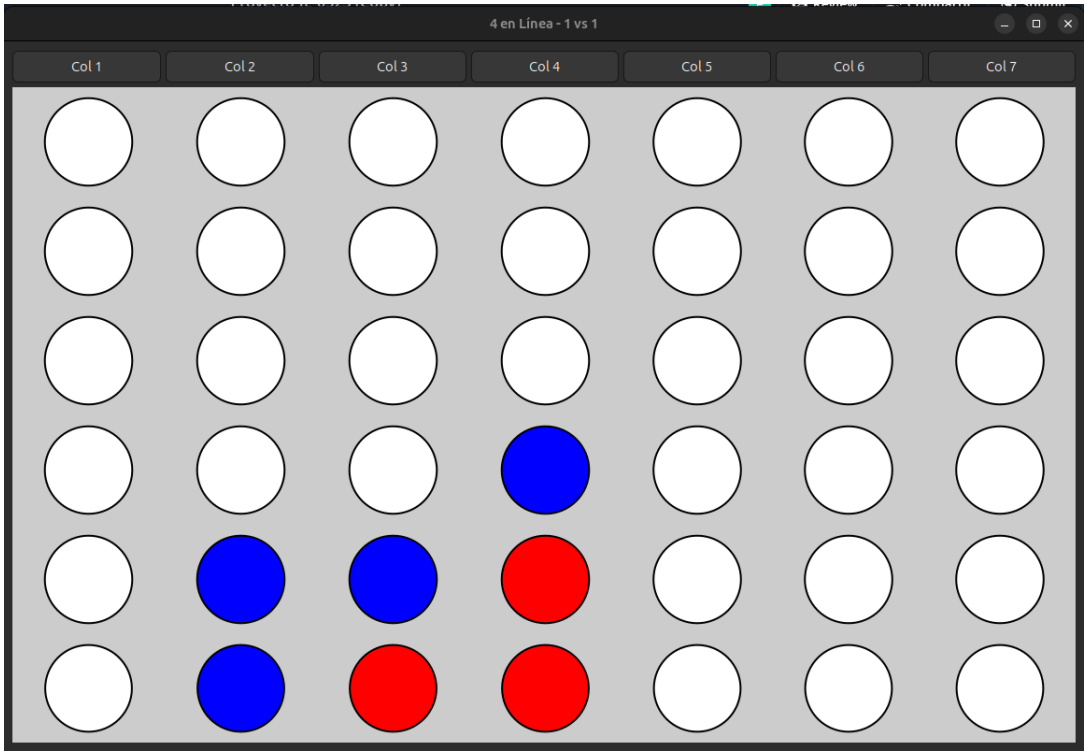

Figura 2: Fichas en posición

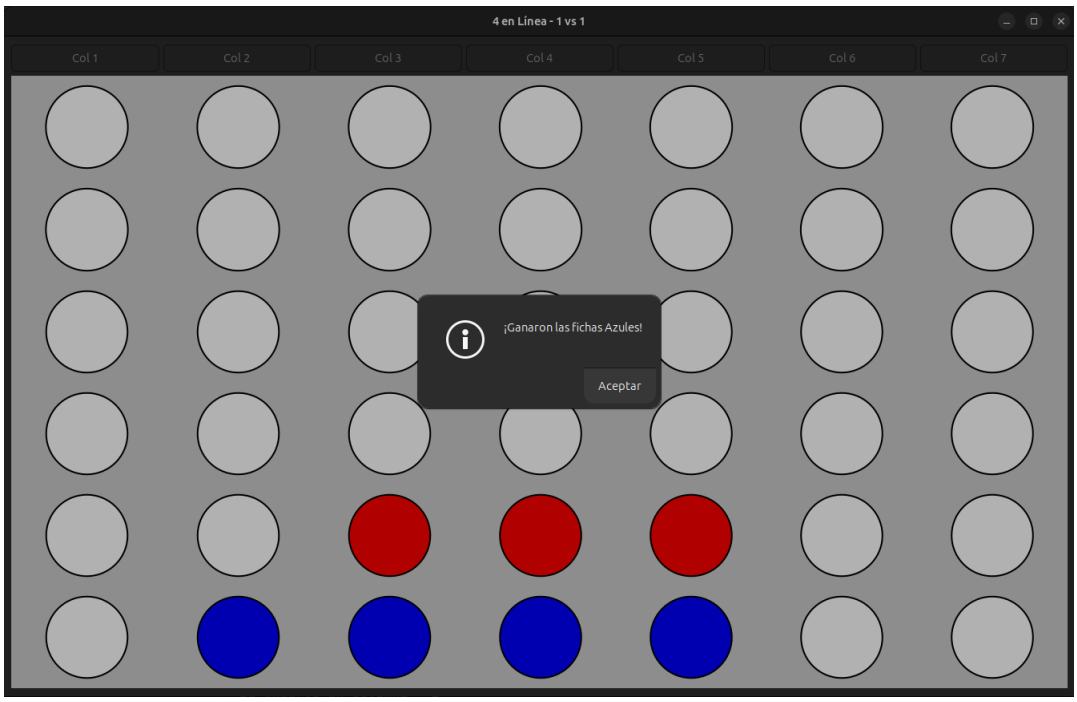

Figura 3: 4 en Linea

Este comportamiento es similar para los demás modos de juego, ya sea versus el bot aleatorio o la IA con predicción. Además como se observa en la imagen 1, la opción de Cargar Partida, recupera el tablero guardado anteriormente.

6. Conclusiones

El desarrollo del juego 4 en Línea permitió la implementación y análisis de diversas estructuras de datos y algoritmos aplicados a un problema práctico. A lo largo del proyecto, se evidenció la importancia de la modularidad en el diseño del código, lo que facilitó la implementación de funcionalidades como la lógica del juego, la inteligencia artificial del bot y la persistencia de datos. La utilización de una estructura de matriz dinámica para representar el tablero garantizó un manejo eficiente de los movimientos y verificaciones de victoria. Además, la separación del código en distintas clases permitió encapsular funcionalidades específicas, como la detección de jugadas ganadoras y la lógica del bot, facilitando la escalabilidad y mantenimiento del código. La interfaz gráfica implementada con GTK proporcionó una experiencia interactiva intuitiva para los usuarios, permitiéndoles jugar en diferentes modos, incluyendo enfrentamientos entre dos jugadores, contra un bot aleatorio o contra un bot con estrategias de decisión más avanzadas.

Uno de los aspectos más destacados del proyecto fue el desarrollo del bot con inteligencia artificial, que incorporó una estrategia de toma de decisiones basada en la exploración del tablero y la detección de jugadas clave. Se implementaron algoritmos que permiten al bot bloquear jugadas del oponente y buscar oportunidades de victoria, aunque se identificaron limitaciones en su capacidad de predicción de jugadas futuras. Este análisis permitió concluir que, si bien el bot implementado logra reaccionar de manera efectiva a jugadas inmediatas, carece de profundidad en la planificación de movimientos futuros, lo que lo hace vulnerable a ciertas estrategias. En cuanto a la persistencia de datos, se logró una implementación funcional que permite guardar y recuperar partidas, garantizando la continuidad del juego