

Universidad de Concepción

Facultad de Ingeniería

Departamento de Ingeniería Eléctrica

Taller de aplicación TIC II

549305-1

Profesor Vincenzo Alexo Caro Fuentes

MiniProyecto 1

Alumnos:

Lissette Martin Carrasco

Javiera Mora Morales

Tomás Troncoso Enríquez

Concepción, 26 de abril del 2024

Introducción

En este informe, se detalla la realización de dos actividades relacionadas con la implementación de juegos interactivos utilizando placa Arduino y comunicación serial entre el lenguaje de Arduino con Python. La primera actividad consiste en la creación del juego "Whack-a-Mole"trabajada en Arduino IDE, mientras que la segunda implica la modificación del juego de Conway "The Game of Life"para integrar condiciones climáticas y eventos de guerra.

Objetivos

- Implementar el juego "Whack-a-Mole" utilizando una placa Arduino y módulos LEDs, botones, buzzer.
- Implementar el juego de la vida de Conway en Python y vincularlo mediante comunicación serial con Arduino.
- Modificar el juego de Conway para incluir nuevas reglas relacionadas con condiciones climáticas y eventos de guerra.
- Desarrollar una interfaz gráfica en Python para visualizar la cuadrícula del juego y controlar eventos de guerra mediante botones.

Metodología

Para la primera actividad, se siguió un enfoque paso a paso que implicaba la conexión de LEDs como "topos" y botones como martillos a una placa Arduino. Se desarrolló un programa en Arduino que generaba una secuencia aleatoria de "topos" iluminados y monitoreaba el estado de los botones para detectar golpes y una logica de juego. Se implementaron condiciones de término del juego y se actualizó la puntuación en tiempo real mediante el monitor serial.

Para la segunda actividad, se modificó el juego de Conway para integrar nuevas reglas relacionadas con condiciones climáticas y eventos de guerra. Se utilizó Arduino para enviar datos de temperatura y recibir señales de botones para eventos de guerra. Se implementaron reglas adicionales, como la reducción de la vida de las células según las condiciones climáticas y la interacción con eventos de guerra. Se creó una interfaz gráfica en Qt Designer para luego ser trabajada en Python con PyQt6 para visualizar la cuadrícula del juego, mostrar la temperatura y controlar los eventos de guerra mediante botones.

Índice general

Actividad n°1. Whack-a-mole!

Enunciado

Esta primera actividad consiste en implementar el juego interactivo "Whack-A-Mole" utilizando una placa Arduino, LEDs como "topos" y botones como martillos. El juego consistirá en golpear los "topos" iluminados en el momento justo para ganar puntos, definiendo distintos niveles de dificultad asociados a los tiempos de duración del encendido de los LEDS y el tiempo total para cada ronda del juego.

Ítem 1.1

Para implementar su juego, consideren los siguientes módulos de su kit de sensores:

- 3-6 módulos LEDs (KY-009, KY-011, KY-016 o KY-029).
- 3-6 botones (KY-004, KY-023, KY-040 u otros).
- 1 buzzer pasivo (KY-006) o activo (KY-012).

Con los elementos anteriores, preparen un circuito de tal manera que cada LED quede asociado con único botón, y con ello desarrollar un programa en Arduino que tenga en cuenta los siguientes puntos:

- El programa iniciará seleccionando una dificultad inicial, ya sea definida por código o ingresada manualmente desde el monitor serial.
- Luego, el código debe generar y reproducir una secuencia aleatoria distinta para el encendido de los LEDs que representen los "topos.ª iluminar en cada ronda, utilizando para ello la función `random()`.
- A la par con la reproducción de la secuencia, el código deberá monitorear el estado de los botones para detectar cuando un jugador golpea un "topoïluminado, contabilizando el puntaje apropiado y reproduciendo un tono o melodía por un buzzer dependiendo si el golpe es exitoso o errado.
- Para definir una condición de término del juego, el jugador comenzará con un total de 50 puntos. En caso de un golpe exitoso, se sumarán 10 puntos. En caso de un golpe errado, se restarán 5 puntos. En caso de avanzar de nivel, se sumarán 20 puntos y se aumentará la dificultad del juego. En el momento que el puntaje llegue a 0, el juego se termina.
- Finalmente, una vez el juego se termina, el código debe reproducir una melodía por el buzzer y reiniciar el juego, restaurando tanto la dificultad como el puntaje del jugador. El monitor serial deberá mostrar y actualizar en todo momento la puntuación del jugador, el nivel de dificultad, el número de golpes exitosos y el número de golpes errados.

1.1. Esquemático del circuito

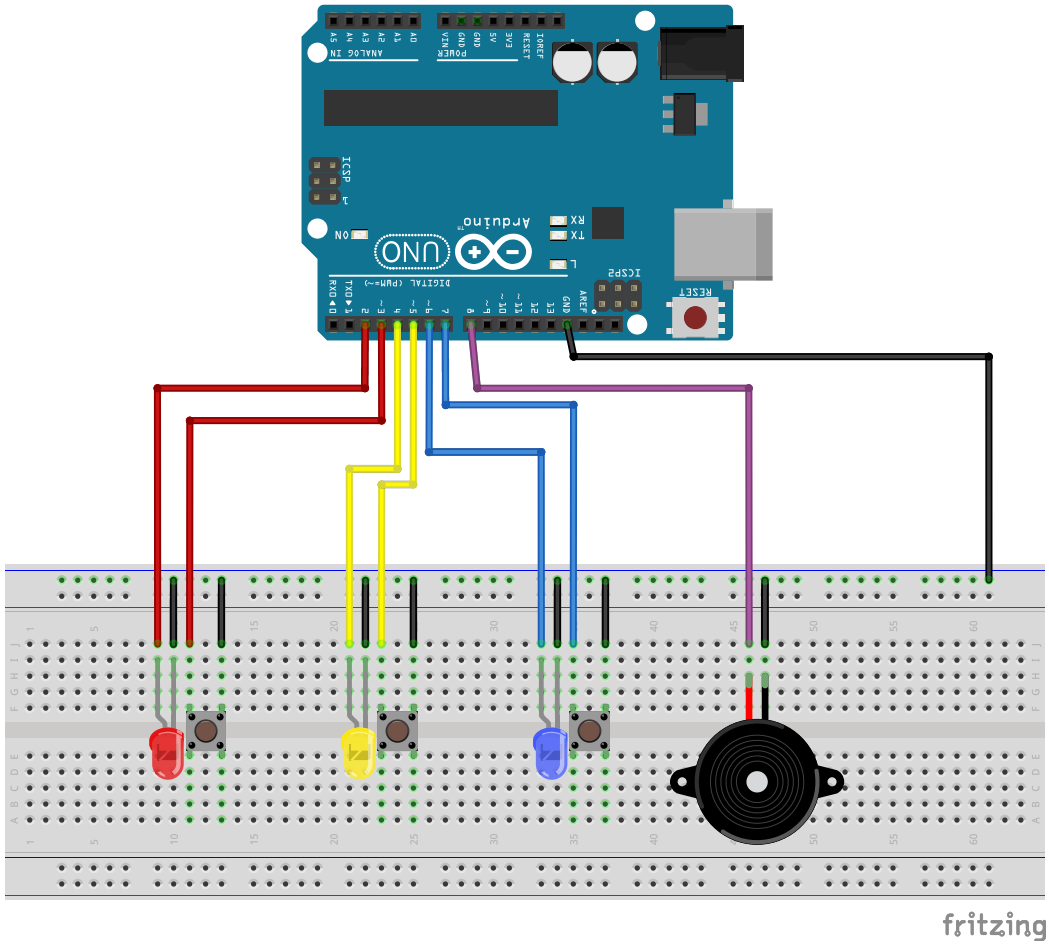

Figura 1.1: Esquemático del circuito empleado en la actividad de emular el juego Whack-a-mole!

El circuito del sistema consiste de 3 LEDs de colores diferentes, cada uno asociado con un botón del mismo color, y un passive buzzer para reproducir los indicadores de sonido del juego.

1.1.1. Código empleado

El código proporcionado implementa de manera efectiva el juego del "Whack-a-mole!" en el que los jugadores deben golpear los botones correspondientes a LEDs que se encienden de manera intermitente y completamente aleatoria. A continuación, se detallaran los aspectos más relevantes del código adjunto.

Función clearSerialBuffer(): Esta función limpia el búfer del puerto serial, asegurándose de que no quede ningún dato sin procesar.

Definición de Constantes y Variables:

- Se definen los pines asociados a 3 módulos LEDs, 3 botones y un zumbador implementados en la recreación de este juego.
- Declaramos las condiciones iniciales del juego tales como el puntaje inicial, el tiempo de cada ronda, la duración en la que los LEDs se mantienen encendidos en principio.

Además de definir variables de conteo, tanto como los golpes totales, los aciertos y fallos al presionar los botones.

Configuración Inicial (setup()):
Esta sección inicializa los pines de la placa Arduino, configurando los pines de los LEDs como salidas, los pines de los botones como entradas con resistencias de pull-up internas y el pin del zumbador como salida. También inicia la comunicación serial a una velocidad de 9600 baudios.

Bucle Principal (loop()):
El bucle principal comienza con la limpieza del búfer del puerto serial para evitar problemas de lectura. Posterior a ello se aplica `while (!Serial.available)`, esto hace que el código espere una respuesta en consola, de forma que se le solicita al jugador ingresar la dificultad del juego permitida de 1 a 9 en principio, esto mediante la linea `char entrada= Serial.read();` y `int deficultad=atoi(& entrada)` condicionada a que el puerto serial este disponible o bien limpio `if(Serial.available())`. En el caso de que esto se cumpla y se ingresen los datos de manera correcta, se valida la entrada, en caso contrario, se le asignará el nivel 11 de dificultad por defecto, desplegando así el mensaje "$\backslash n \backslash r$ `Dije que ingresaras un numero entre 1 y 9, por pesado te la pondre en 11`$\backslash r \backslash n \ \backslash r \backslash n$". Donde se aplicará una nueva linea seguido de un retorno de carro con tal de asegurar que el texto se muestre correctamente en nuestra terminal. Esto fue especialmente útil para el formateo del texto además de mantener un orden para el usuario al momento de ingresar los datos en consola

Cálculo de Tiempos y asignación de puntaje:

- La variable **tEncendido** calcula el tiempo en que cada LED estará encendido en milisegundos. Se calcula dividiendo 3000 (un valor arbitrario) entre la raíz cuadrada de la dificultad. Esto significa que a medida que aumenta la dificultad, el tiempo de encendido de los LEDs disminuirá durante cada ronda superada.
- La variable **tRonda** multiplicará a `tEncendido` por 10, esto determinará cuánto tiempo durará una ronda completa del juego.
- La variable **delayBotones** calcula el tiempo de espera entre la presión de un botón y el encendido del siguiente LED. Se establece entonces en aproximadamente un tercio del tiempo de encendido de los LEDs por conveniencia.

Junto a esto se define el ciclo principal del código `while(puntaje >0)`, que permite la ejecución repetida del juego mientras el jugar mantenga su puntaje sobre 0.

```
while(puntaje > 0) {
    int tEncendido = 3000/sqrt(dificultad);
    int tRonda =   10*tEncendido;
    int delayBotones = 0.33*tEncendido;
```

Generación de Rondas de Juego

En esta sección del código, se generan las rondas de juego. Donde se controla además el encendido y apagado de los LEDs de forma aleatoria, utilizando la función `random(1,4)`, que determina al principio de cada ronda cual boton se va a utilizar. Ademas, se definen las variables `oldValn`, que almacenan el estado del boton al principio del ciclo, y las variables `vBn`, las que van a almacenar si el botón fue presionado o no.

```
unsigned long tInicial = millis(); //Cuando empieza el Loop de la ronda
    while ((millis() - tInicial) < tRonda) {

        int cBoton = random(1, 4); //Elije uno de los 3 leds
        int cLed   = 2*cBoton; // Elije uno de los 3 botones
        int oldVal1 = digitalRead(3); //Define el estado actual del boton a usar
        int oldVal2 = digitalRead(5);
        int oldVal3 = digitalRead(7);

        bool vB1 = false; // La variable que nos va a decir si el boton fue
    presionado correctamente.
        bool vB2 = false;
        bool vB3 = false;


      digitalWrite(cLed, HIGH);
      unsigned long tBoton = millis(); //registra cuanto tiempo tiene que estar
    prendido el boton
```

Una vez definida todas esas variables, se enciende el LED seleccionado para la ronda en particular. Luego, procedemos al `while` correspondiente a cada intervalo para presionar el boton:

```
      while ((millis()-tBoton) < tEncendido){  // Corresponde a cada iteracion
    del juego
        int val1 = digitalRead(3);
        int val2 = digitalRead(5);
        int val3 = digitalRead(7);

        int bApretado = 0;
```

Donde las variables `Valn` son las que registran si el botón es apretado o no dentro del tiempo asignado mientras que esta prendido.

Luego, el codigo revisa si alguno de los botones fue presionado o no comparando si existe un cambio de estado en ellos, y de ser así, reedefine la variable `bApretado` con el valor del boton presionado:

```
            if ((val1==LOW && oldVal1==HIGH) || (val1==HIGH && oldVal1==LOW )    )
    {
          vB1 = !vB1  ;
          bApretado = 1;
        }
        if ((val2==LOW && oldVal2==HIGH) || (val2==HIGH && oldVal2==LOW )    ) {
          vB2 = !vB2  ;
          bApretado = 2;
        }
        if ((val3==LOW && oldVal3==HIGH) || (val3==HIGH && oldVal3==LOW )  ) {
          vB3 = !vB3  ;
          bApretado = 3;
        }
```

Luego de eso, se va a utilizar el valor modificado de `bApretado` para corroborar si el boton fue apretado de forma exitosa o no.

- **Condición exitosa; Botón presionado correctamente**
    - Primeramente se evalúa si al menos uno de los botones `(vB1, vB2 o vB3)` ha sido presionado y si el botón presionado coincide con el LED encendido `(cBoton == bApretado)`.
    - Si ambas condiciones se cumplen, significa que el jugador ha presionado el botón correcto.

- De forma que se aumenta el puntaje del jugador en 10 puntos (puntaje = puntaje + 10). Y se aumenta el puntaje total (pTotal) en 10 puntos. Además se incrementa el contador de golpes exitosos (golpes) en 1.
- Luego se emite un tono positivo a través del zumbador `(tone(buzzerPin, 1000))`. Se espera 200 milisegundos (delay(200)) para posterior detener el tono del zumbador `(noTone(buzzerPin))`.
- Finalmente se apaga el LED correspondiente `(digitalWrite(cLed, LOW))` y se sale del bucle actual con un break.

```
if (((vB1 == true)||(vB2 == true)||(vB3 == true))&&(cBoton == bApretado)) {
    puntaje = puntaje + 10 ;
    pTotal = pTotal + 10;
    golpes = golpes + 1;
    tone(buzzerPin, 1000); // Tono afirmativo
    delay(200);
    noTone(buzzerPin);
    digitalWrite(cLed, LOW);
    break;
}
```

- **Condición incorrecta; Botón incorrecto presionado**

- Se evalúa si al menos uno de los botones ha sido presionado y si el botón presionado no coincide con el LED encendido. Si ambas condiciones se cumplen, significa que el jugador ha presionado un botón incorrecto.
- Se procede a reducir el puntaje del jugador en 5 puntos `(puntaje = puntaje - 5)`. Incrementando el contador de golpes fallidos (fallos) en 1.
- Se emite un tono negativo a través del zumbador y se espera 300 milisegundos hasta detener el tono del zumbador.
- Luego se apaga el LED correspondiente y se sale del bucle actual con break.

```
 if (((vB1 == true)||(vB2 == true)||(vB3 == true))&&(cBoton!=bApretado)) {
    //apretaron el boton equivocado
    puntaje = puntaje - 5 ;
    fallos = fallos + 1;
    tone(buzzerPin, 150); // Tono Negativo
    delay(300);
    noTone(buzzerPin);
    digitalWrite(cLed, LOW);
    break;
}
```

- **Condición inconclusa; Ningún botón es presionado**

- Una vez que se sale del `while` del boton encendido, Se evalúa que ninguno de los botones fueron presionados, a través de la condicion `(vB1 == false)&&(vB2 == false)&&(vB3 == false)`.
- Cumplidas las condiciones se emite un tono muy característico a través del zumbador, indicándole que no presiono ningún botón en el tiempo correspondiente, descontando así al puntaje del jugador 5 puntos.

```
digitalWrite(cLed, LOW);
if (((vB1 == false)&&(vB2 == false)&&(vB3 == false))){
    tone(buzzerPin, 150);
    delay(300);
    noTone(buzzerPin);
    delay(200);
    tone(buzzerPin, 100);
    delay(150);
    noTone(buzzerPin);
    puntaje = puntaje - 5;
}
```

Una vez terminada la ronda, se revisa al interior si el puntaje sigue siendo mayor a 0, y de ser así se genera una melodia de tono ascendente, se aumenta la dificultad por 1 punto y se imprime n consola el mensaje `Felicitaciones, pasaste de nivel`

```
   if ( puntaje > 0){
     puntaje = puntaje + 20;
     pTotal = pTotal + 20;
     Serial.print("\r\n \r\n Felicitaciones, pasaste de nivel");
     tone(buzzerPin, 550);
     delay(300);
     tone(buzzerPin, 750);
     delay(300);
     noTone(buzzerPin);
     delay(200);
     tone(buzzerPin, 1000);
     delay(150);
     noTone(buzzerPin);
   }
  dificultad = dificultad + 1;
```

Y con esto termina el loop `while` del juego, se vuelve al incio y de tener un puntaje mayor a 0, se vuelve a jugar una ronda. En el caso de que el puntaje del juego sea 0 o menos, se da por terminado el juego y se reproduce la melodía de Ámong Usá través del buzzer. Ademas, se muestran las estadisticas del juego por consola:

```
   Serial.print("\n\r \n\r Fin de la linea soldado! tus resultados fueron: \r\n"
   );
  Serial.print("\n\r Puntaje: ");
  Serial.print(pTotal);
  Serial.print("\n\r Dificultad final: ");
  Serial.print(dificultad);
  Serial.print("\n\r Aciertos a los topos: ");
  Serial.print(golpes);
  Serial.print("\n\r Golpes equivocados: ");
  Serial.println(fallos);
```

Por ultimo, se reinician los valores de todas las variables que registran las estadísticas del juego a su valor inicial y se reproduce el mensaje en puerto serial `Lamentablemente no puedo programar al arduino para que detenga el loop principal, así que tendras que jugar denuevo, en aproximadamente 10 segundos` . Y despues de dichos 10 segundos el juego vuelve a empezar desde el inicio.

A modo de resumen entonces tenemos que durante el juego, se generan rondas donde se iluminan de forma aleatoria 3 módulos LEDs, de forma que el jugador debe presionar el botón asociado a dicho led para aumentar su puntaje inicial. Y el juego se ejecutará en un bucle mientras que el puntaje del jugador sea mayor que cero. Se actualiza el puntaje y la dificultad según el rendimiento del jugador. Después de cada ronda exitosa, el puntaje y la dificultad del juego aumentan. Y al finalizar el tiempo final se proporciona retroalimentación auditiva al jugador y

se muestra un mensaje por el puerto serial para informarle sobre su éxito. Luego el juego vuelve a reiniciarse y realiza el mismo proceso anteriormente descrito.

1.1.2. Comentarios y desafíos

Una de las dificultades que experimentamos al abordar esta actividad fue plantear la electrónica y la gestión del espacio en la protoboard de todos los componentes que requeríamos, además de la implementación de nuevos botones pulsadores que con anterioridad no habíamos utilizado, lo que nos complico un tanto la asociación botón LED en un comienzo. Sin embargo, gracias a la guía del profesor logramos implementar una electrónica correcta e intuitiva para los usuarios empleando por supuesto las herramientas y facilidades que ofrece Arduino al implementar una resistencia de PULL-UP interna en cada botón.

Otra de las dificultades que enfrentamos fue generar una estructura clara para nuestro código, pues para funcionar correctamente requería un gran numero de ciclos while incrustados uno dentro del otro, además de un gran numero de condicionales que se evaluaran en cada uno de estos. Además, implementar una dificultad progresiva sin que terminara en números negativos para los tiempos parecía inicialmente una tarea dificultosa, pero al recurrir al uso de la función raíz cuadrada, que corresponde a una estrategia que se utilizaba en juegos antiguos, resulto ser una solución bastante elegante al problema que permite tener una dificultad que incrementa de forma indefinida.

A modo de desafío, y bajo otras circunstancias hubiera sido factible implementar una mayor cantidad de "topos" de forma que en niveles más altos, aparecieren LEDs de forma simultánea y completamente aleatoria, desafiando así la capacidad del jugador para mantenerse al tanto de múltiples objetivos a la vez y tomar decisiones rápidas. Además, a futuro también se podría implementar que el jugador necesite obtener un puntaje mínimo por ronda, talque en lugar de simplemente jugar hasta que un cierto instante de tiempo, se establezca un puntaje objetivo que el jugador debe alcanzar para no perder, y que aumente con la dificultad, proporcionando un desafío creciente.

1.1.3. Evidencias

A continuación se deja constancia gráfica del correcto funcionamiento de la actividad desarrollada si accede al siguiente link.

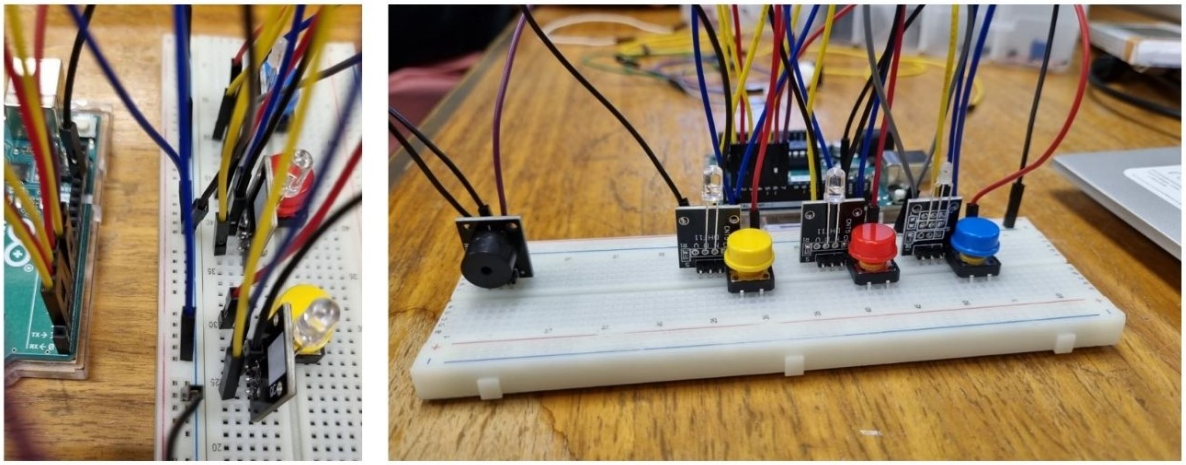

Figura 1.2: Circuito empleado en la actividad de emular el juego Whack-a-mole!

Actividad n°2. The Game of Life

Enunciado

The Game of Life fue creado por Conway como un modelo matemático de un sistema celular con el fin de explorar la idea de la vida y la muerte en un entorno abstracto. Conway buscaba un sistema con reglas simples, pero que pudiera producir comportamientos complejos y variados. Las reglas originales del juego son simples y se aplican a una cuadrícula bidimensional de celdas, cada una de las cuales puede estar viva (1) o muerta (0). Dependiendo de la ubicación de cada célula y sus respectivos vecinos, se pueden dar alguno de los siguientes casos:

- Sobrepoblación: Si una célula viva tiene más de tres vecinos vivos, muere debido a la falta de recursos.
- Subpoblación: Si una célula viva tiene menos de dos vecinos vivos, muere por soledad.
- Estabilidad: Una célula viva con dos o tres vecinos vivos permanece viva en la siguiente generación.
- Reproducción: Si una célula muerta tiene exactamente tres vecinos vivos, nace en la siguiente generación debido a la reproducción.

Ítem 2.1

La idea detrás de esta actividad es que sean capaces de programar y alterar externamente una versión modificada de The Game of Life, para lo cual pueden considerar el código base mostrado en el Anexo, donde se implementa el juego para una cuadrícula de 100x100 con sus reglas originales.

De manera inicial, adapten dicho código para implementar el juego en una interfaz gráfica creada con la biblioteca PyQt6, apoyándose de la biblioteca matplotlib para construir la cuadrícula y de QTimers para actualizar el juego en tiempo real, con una tasa de actualización variable a definir por ustedes. Luego, utilizando como intermediario la comunicación por protocolo serial, integren un programa de Arduino que sea capaz de interactuar con su juego de la siguiente forma:

- Cada un intervalo de 10 segundos, el programa de Python debe enviar hacia Arduino la cantidad de células vivas que se encuentran actualmente en la cuadrícula. Luego, definan en Arduino 3 intervalos para evaluar las condiciones de estabilidad, subpoblación y sobrepoblación de las células; vinculando cada estado con un color particular de un módulo LED. En caso de ocurrir un cambio de estado, la transición entre cada color de cada estado debe realizarse de forma paulatina, utilizando las salidas PWM de Arduino.
- Desde Arduino, implementen un botón pulsador que sea capaz de reiniciar desde 0 el juego en Python. Para ello, cada vez que se presione el botón, envíen un carácter o un mensaje hacia Python para que la interfaz reaccione a esta acción.

Ítem 2.2

Utilizando como base el Ítem anterior, modifiquen su juego para integrar las siguientes reglas:

- Las células ahora tendrán una vida total de 100 puntos, y se verán enfrentadas a condiciones extremas del clima y una guerra contra un planeta vecino.
- Si una célula viva tiene menos de 2 o más de 3 vecinos vivos, su vida se reduce en 30 puntos en cada iteración del juego.
- Si una célula muerta tiene exactamente 3 vecinos vivos, renace con una vida de 100 puntos.
- Si todas las células mueren, reproduzcan un audio (sin buzzer) representando que se perdió la guerra, utilizando para ello la biblioteca playsound u otra similar. Luego, reinicien el juego.

Condiciones de temperatura:

- Si la temperatura está por debajo del umbral de frío (a definir), aumenta la vida de cada célula en 10 puntos en cada iteración del juego.
- Si la temperatura está por encima del umbral de calor (a definir), reduce la vida de cada célula en 20 puntos en cada iteración del juego.

Condiciones por la guerra:

- Si se activa el botón de la bomba nuclear, elimina todas las células dentro de un rango de 21x21 en una posición aleatoria.
- Si se activa el botón de área de curación, revive las células muertas con 70 puntos de vida y aumenta la vida de las células vivas en 50 puntos (con un máximo de 100) dentro de un rango de 21x21 en una posición aleatoria.

Para implementar las condiciones de temperatura, integren un sensor de temperatura en Arduino que sea capaz de enviar este dato hacia la interfaz en Python, considerando para ello una expresión regular de la forma “a-xyz”, donde “a” es el identificador del mensaje de tipo temperatura, y “xyz” son los caracteres que representan el dato de la temperatura de hasta tres dígitos. Por ejemplo, si se tiene el dato “a-022”, este representa que se está enviando una temperatura de 22 °C.

Para implementar las condiciones de guerra, integren en Arduino dos sensores distintos a los botones pulsadores para que ejecuten la bomba nuclear y la bomba de curación, enviando una señal distinta para cada caso. Para lo anterior, consideren una expresión regular de la forma “b-x”, donde “b” es el identificador del mensaje de tipo bomba, y “x” es el carácter que representa el tipo de bomba utilizada. Por ejemplo, si se tienen los datos “b-1” y “b-2”, el primero puede representar una bomba nuclear y el segunda una bomba de curación.

Adicionalmente, integren en su interfaz gráfica 2 QPushButtons (representando las bombas) y 1 QTextEdit (representando el clima) para implementar de manera redundante las condiciones de guerra y de temperatura. Para el caso de la temperatura, esta debe controlarse mediante un QPushButton adicional que permita intercambiar entre la lectura del sensor y un valor ingresado manualmente por el usuario.

2.1. Esquemático de circuito Ítem 2.1

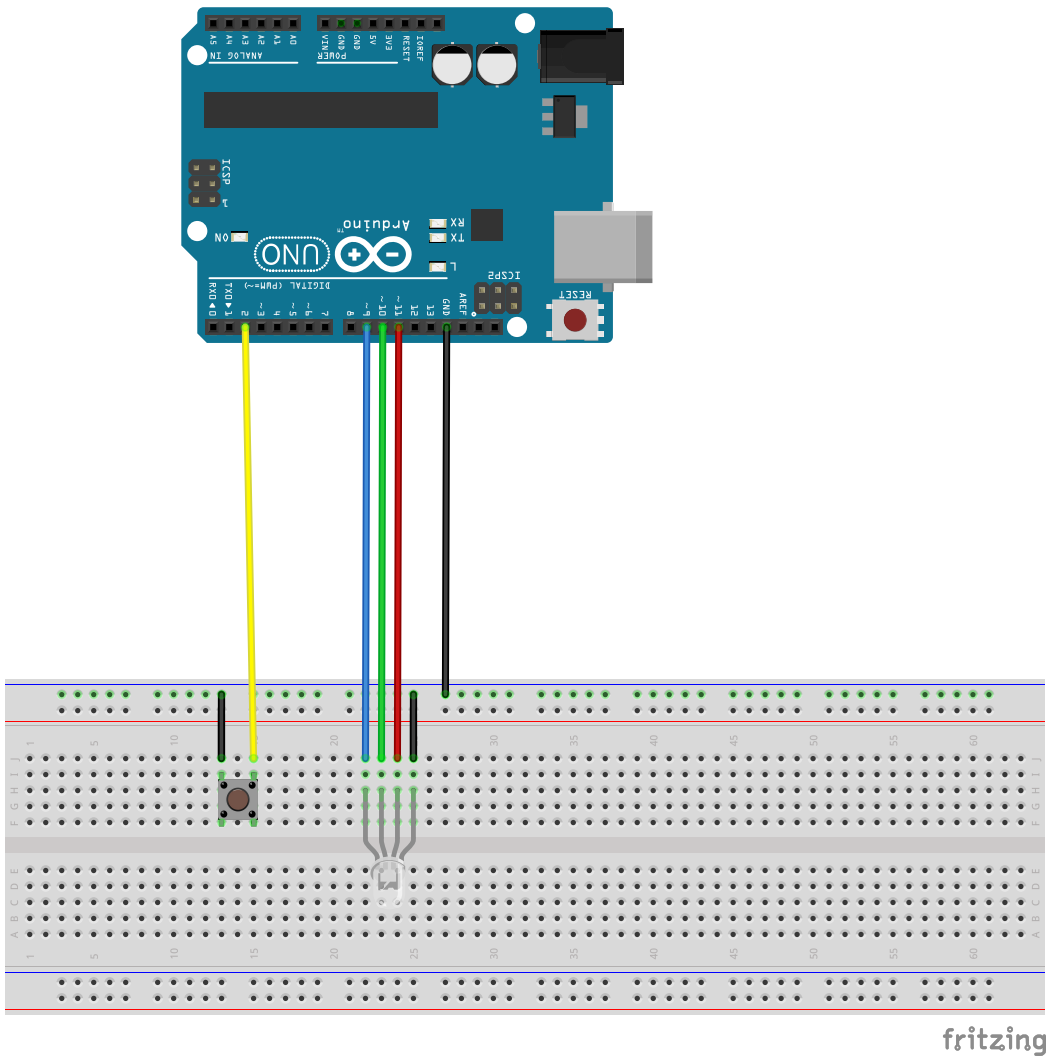

Figura 2.1: Esquemático empleado en la actividad de emular el juego "The game of life"

Para este ítem fue necesario solamente implementar un LED RGB que representara el numero de células que llegaban desde Python al Arduino, y un botón que generara la señal de reseteo que se manda al código de Python.

2.1.1. Código implementado en Python

Se detalla el proceso de desarrollo y la lógica detrás de la implementación en Python del "Juego de la Vida" de Conway. Este juego se ejecuta en una interfaz gráfica creada con PyQt6 y se comunica con un dispositivo Arduino a través de un puerto serial para enviar información sobre el estado del juego. A continuación detallaremos las 3 grandes clases que implementamos en nuestro código.

Implementación del Juego de la Vida en class GolGame

En esta clase se modela el juego de la vida, definiendo así la cuadrícula del juego y las reglas de evolución. La cuadrícula se inicializa aleatoriamente con células vivas y muertas dispuestas en posiciones totalmente randomizadas. Para la lógica de actualización del juego, se empleo la convolución con un kernel predefinido para contar los vecinos vivos de cada célula, para esto fue indispensable la implementación del código entregado por el profesor de asignatura.

```
class GolGame:
    def __init__(self, size):
        self.size = size
        self.grid = np.random.choice([0, 1], size*size, p=[0.8, 0.2]).reshape(
    size, size)

    def update(self):
        kernel = np.array([[1, 1, 1],
                           [1, 0, 1],
                           [1, 1, 1]])
        convolved = convolve2d(self.grid, kernel, mode='same', boundary='fill')
        birth = (convolved == 3) & (self.grid == 0)
        survive = ((convolved == 2) | (convolved == 3)) & (self.grid == 1)
        self.grid[:, :] = 0
        self.grid[birth | survive] = 1
```

Actualización de la interfaz gráfica en class GolWidget(QWidget)

Utilizando PyQt6 para la creación la interfaz gráfica del juego, la clase `GolWidget` es la responsable de mostrar la interfaz gráfica del "Juego de la Vida.[en] una ventana utilizando Matplotlib. A continuación detallaremos el conjunto de métodos y atributos que describen a la clase `GolWidget`.

Método __init__

- `__init__(self, parent=None)`: Es el constructor de la clase. Recibe un argumento opcional `parent`, que por defecto es `None`.
- `super().__init__(parent)`: Llama al constructor de la clase padre `(QWidget)` para inicializar la instancia de `GolWidget`. Esto asegura que se realicen todas las inicializaciones necesarias en la clase padre.
- `self.initUI()`: Llama al método `initUI()` para configurar la interfaz de usuario.

Método initUI

- `self.figure = Figure()`: Crea una nueva figura de Matplotlib, que es el lienzo donde se mostrará el juego de la vida.
- `self.ax = self.figure.add_subplot(111)`: Añade un nuevo subplot a la figura. En este caso, se añade un único subplot con identificador 111, que es un subplot de una fila y una columna para visualizar las dimensiones de la cuadricula.
- `self.canvas = FigureCanvas(self.figure)`: Crea un lienzo (FigureCanvas) para la figura de Matplotlib. Este lienzo es necesario para poder mostrar la figura en la interfaz gráfica.

- `layout = QVBoxLayout()`: Crea un diseño vertical (QVBoxLayout) para organizar los widgets en la interfaz gráfica.
- `layout.addWidget(self.canvas)`: Añade el lienzo al diseño vertical, de forma que agrega a FigureCanvas al diseño, colondo el lienzo en la ventana de la interfaz gráfica.
- `self.setLayout(layout)`: Establece el diseño vertical como el diseño principal de la ventana.
- `self.size = 100`: Define el tamaño del juego de la vida. En este caso, se inicializa con un tamaño de 100x100, como es requerido en el enunciado.
- `self.game = GolGame(self.size)`: Crea una instancia de la clase `GolGame`, que representa el juego de la vida. En otras palabras, hace que la cuadrícula inicial sea generada aleatoriamente por el constructor de `GolGame` .
- `self.im = self.ax.imshow(self.game.grid, cmap='gray')`: Añade una imagen (imshow) al subplot (ax). Esta imagen corresponde a la cuadrícula del juego de la vida. Se utiliza una escala de grises (cmap='gray') para representar las células vivas y muertas.

Método update_game

- `self.game.update()`: Actualiza el estado del juego de la vida invocando el método update() de la instancia de GolGame. Esto hace que el juego avance un paso en su evolución.
- `self.im.set_array(self.game.grid)`: Actualiza la imagen (imshow) en el subplot para reflejar el nuevo estado del juego de la vida.
- `self.canvas.draw()`: Dibuja el lienzo para reflejar los cambios realizados en la figura. Esto actualiza la interfaz gráfica para mostrar el nuevo estado del juego de la vida.

Ventana principal MainWindow(QMainWindow)

La clase `MainWindow` es la ventana principal de la aplicación y coordina el funcionamiento general del juego de la vida, incluyendo la actualización del juego en la interfaz gráfica, la comunicación con Arduino y la gestión de eventos del teclado. En definitiva es la ventana principal de la aplicación y contiene el widget del juego. A continuación se detallan los métodos empleados y sus respectivas funcionalidades.

Método __init__

- `__init__(self)`: Constructor de la clase. Se llama al inicializar un objeto de esta clase.
- `super().__init__()`: Llama al constructor de la clase padre (QMainWindow) para inicializar la ventana principal.
- `self.initUI()`: Llama al método initUI() para configurar la interfaz de usuario y establecer los elementos principales.
- `self.setup_serial()`: Llama al método setup_serial() para inicializar la comunicación serial con Arduino.

Método initUI

En especifico este método además de configurar el título y las dimensiones de la ventana principal, crea un objeto `GolWidget` y lo establece como el widget central de la ventana, el cual describimos con anterioridad y sus funcionalidades respectivas.

```
def initUI(self):
    self.setWindowTitle("Game of Life")
    self.setGeometry(100, 100, 800, 800)

    self.gol_widget = GolWidget(self)
    self.setCentralWidget(self.gol_widget)
```

En esta clase configuramos dos temporizadores asociados a un determinado QTimer correspondiente de PyQt que son `timer` y `timerArdu` , el primero se utilizará para actualizar la interfaz y el otro para enviar datos a Arduino en intervalos regulares. De forma que la interfaz se irá actualizando cada 1 segundo, mientras que por otra parte, se tiene que cada un intervalo de 10 segundos Python enviará hacia Arduino la cantidad de células vivas que se encuentran en la cuadricula, tal como se requirió en el enunciado.

```
    self.timer = QTimer(self)
        self.timer.timeout.connect(self.update_game)
        self.timer.start(100) # 1 segundo

        self.timerArdu = QTimer(self)
        self.timerArdu.timeout.connect(self.LecturaArduino)
        self.timerArdu.start(1000) # 10 segundo
```

Método LecturaArduino

Este método envía información sobre el estado actual del juego de la vida hacia Arduino, de forma que la cantidad de células vivas se calcula y se convierte en una cadena formateada antes de ser enviada a través del puerto serial. Esto permite que Arduino reciba información actualizada sobre el estado del juego y pueda realizar acciones basadas en estos datos, para así ajustar la iluminación del LED, detallaremos con posterioridad el detalle de dicha función cuando abordemos el **código implementado en Arduino**.

```
def LecturaArduino(self):
     try:
        self.alive_cells =  '{:04d}'.format(np.sum(self.gol_widget.game.grid))
        self.arduino.write(str(self.alive_cells).encode()) # Le manda la
    informacion de las celulas vivas a arduino.
        if self.arduino.in_waiting > 0:
            self.mensaje = self.arduino.readline().decode().strip()  #Recibe, lee
     y almacena un mensaje mandado desde arduino
            print(self.mensaje,type(self.mensaje))
```

En definitiva este método se encarga de leer la cantidad de células vivas del juego de la vida desde GolWidget y la envía a Arduino a través de la conexión serial y a su vez espera a recibir mensajes de Arduino. Es por esto mismo que empleamos un bloque **try-except** para manejar posibles errores de comunicación serial. Esto es crucial para garantizar la robustez y la estabilidad de la aplicación, ya que los errores de comunicación pueden ocurrir debido a problemas de conexión o configuración. Al capturar y manejar estas excepciones, la aplicación puede informar

al usuario sobre cualquier problema y continuar funcionando de manera adecuada.

Además de enviar datos y asegurarse que estos sean recepcionandos con éxito desde Python hacia Arduino, el método LecturaArduino también está diseñado para recibir mensajes desde Arduino a Python. En este caso, espera a que Arduino envíe un mensaje de reinicio (indicado por la cadena r"). Si se recibe este mensaje, se llama al método reset_game para reiniciar el juego de la vida. Esta funcionalidad permite una interacción bidireccional entre Python y Arduino.

```
            if self.mensaje == "r":
                self.reset_game()
                self.mensaje = " "
        self.last_update_time = time.time()  # Actualizar el tiempo de la ultima
    actualizacion
    except serial.SerialException as e:
        print(f"Error al enviar datos a Arduino: {e}")
```

Método setup_serial

Inicializa la conexión serial con Arduino en el puerto COM6 a una velocidad de 9600 baudios, utiliza la función **serial.Serial()** del módulo serial para abrir una conexión serial con el dispositivo Arduino. Esta acción crea un objeto Serial que representa la conexión serial. El objeto Serial recién creado se asigna a la variable **self.arduino**, que es un atributo de la instancia de la clase. Esto permite que otros métodos de la clase accedan y utilicen la conexión serial. De manera que se registra el tiempo actual utilizando la función **time.time()** y se almacena en el atributo **last_update_time**. Lo que es útil para llevar un registro del tiempo transcurrido desde la última actualización de la comunicación serial.

```
def setup_serial(self):
    try:
            self.arduino = serial.Serial('COM6', 9600)
            self.last_update_time = time.time()
```

Dado a su diseño en un bloque try-except, podemos manejar posibles errores que puedan ocurrir al intentar abrir el puerto serial. En caso de que ocurra una excepción de tipo **SerialException**, que podría ocurrir si hay un problema con el puerto serial especificado (por ejemplo, si el puerto no está disponible o está siendo utilizado por otro programa, lo cual ocurría bastante), se imprimirá un mensaje de error y el programa Python se cerrará utilizando **sys.exit(1)**.

```
    except serial.SerialException as e:
            print(f"Error al abrir el puerto serial: {e}")
            sys.exit(1)
```

Método update_game

Este metódo llama al método **update_game()** de **GolWidget** para así actualizar el estado del juego en la interfaz gráfica. Después de actualizar el juego, calcula la cantidad de células vivas en la cuadrícula del juego, utilizando la función **np.sum()** de **NumPy** para sumar todos los elementos de la matriz que representa la cuadrícula del juego. Luego, utiliza la cadena de formato ':04d'para formatear este número como una cadena de 4 dígitos con ceros a la izquierda si es necesario. Luego utiliza la conexión serial con Arduino para enviar la cantidad de células vivas como una cadena a través del puerto serial, esto permite que Arduino reciba información

actualizada sobre el estado del juego de la vida y pueda realizar acciones en función de esta información.

Finalmente se imprime la cantidad de células vivas en la consola para asegurarnos de que los parámetros de sobrepoblación, subpoblación y estabilidad sean correspondientes a lo impuestos por nosotros sean los correctos y que además sean concordantes con los cambios de iluminación del módulo LED en Arduino.

```
def update_game(self):
        self.gol_widget.update_game()

        # Enviar la cantidad de celulas vivas a Arduino cada vez que se actualice el juego
        self.alive_cells = '{:04d}'.format(np.sum(self.gol_widget.game.grid))

        print(self.alive_cells)
```

Método reset_game

Este método genera una nueva cuadricula aleatoria utilizando la función `np.random.choice()` de NumPy, la cual elige valores aleatorios de 0 o 1 con una probabilidad de 0.8 para 0 (célula muerta) y 0.2 para 1 (célula viva). Luego asigna la nueva cuadrícula generada a la cuadrícula de juego del objeto `gol_widget` . Esto se hace mediante la asignación directa del atributo `grid` del objeto `game` contenido en `gol_widget` .

```
def reset_game(self):
        self.gol_widget.game.grid = np.random.choice([0, 1], self.gol_widget.size*self.gol_widget.size, p=[0.8, 0.2]).reshape(self.gol_widget.size,self.gol_widget.size)
```

Finalmente se emplea el método `reshape` de NumPy, que le da a la cuadrícula la forma deseada con `self.gol_widget.size x self.gol_widget.size`. Esto garantiza que la cuadrícula tenga el tamaño correcto y la forma correcta para el juego de la vida una vez se realice el reset.

Método keyPressEvent

Es un **método de evento** proporcionado por **PyQt** que se activa cada vez que se presiona una tecla en el teclado mientras la ventana está activa, de forma que si la tecla presionada es 'r', llama al método **reset_game()** para reiniciar el juego.

```
def keyPressEvent(self, event):
    if event.key() == Qt.Key.Key_R:  # Si se presiona la tecla 'r'
            self.reset_game()
```

2.1.2. Código implementado en Arduino

Primero se definen las variables asociadas a los pines del LED RGB y las del botón pulsador. Luego se inicializan variables para el recuento de células vivas, brillo. Definiendo estas variable procedemos a la configuración del setup y el loop del código implementado en Arduino.

Configuración inicial en la función setup()

Se inicia la comunicación serial a una velocidad de 9600 baudios, se configuran los pines como entrada o salida según corresponda al LED y el botón pulsador. De forma que se lee el estado inicial del botón para establecer el estado previo del botón.

Bucle principal en la función loop()

Se lee la cantidad de células vivas recibidas a través del puerto serial desde un programa en Python, con la precaución de que se lean como un numero de 4 digitos, ya que en ese formato son enviados desde del codigo de python. El codigo encargado de procesar los datos recibidos y almacenarlos en una variable viene dado por:

```
1       if (Serial.available() >= 4) {  // Se asegura de recibir 4 caracteres
2         char buffer[5];
3         Serial.readBytes(buffer, 4);
4         buffer[4] = '\0';
5
6         // Transforma estos 4 caracteres a un numero
7         alive_cells = atoi(buffer);
8
9     }
```

Que básicamente tiene como condición esperar a que se acumulen 4 digitos enviados (que corresponden al primer mensaje), los acumula en un buffer y despues los transforma de un string a un numero utilizando la función `atoi()`.

Luego, se define una estructura de if que dependiendo de la cantidad de células vivas prende el led correspondiente al numero (subpoblado, sobrepoblado o población adecuada), además, se define la variable `u1` con el objetivo de registrar cual luz se prende en cierto ciclo del código y compararla con la que se prendió en el ciclo anterior, que se almacena en `u2`. La estructura de los if para cada color viene dada por:

```
1       if (alive_cells < 870) {
2       u1 = 11;
3       if(u1 != u2) {
4         for (int i = 0; i <= 255; i++) {
5           analogWrite(u1, i);
6           analogWrite(u2, 255 - i);
7           delay(10);
8         }
9       }
10      else{  analogWrite(LED_PIN_3,255);
11                analogWrite(LED_PIN_1,0);
12             analogWrite(LED_PIN_2,0); }
13    }
```

Donde dependiendo del numero de células (en este caso `alive_cells <870`) se determina cual va a ser el LED a encender, luego de eso, si el LED encendido en la ronda anterior es diferente al actual (visto por la condición `u1 != u2`), se realiza un cambio gradual entre el actual y el anterior. Si dicha condición no se cumple, como cuando no ha cambiado el color del

LED, simplemente se llega y fuerza el color correspondiente del LED a su máximo valor.

La siguiente parte del código es la encargada de registrar el cambio del botón de reseteo para mandar la señal al código de python, este se integra con relativa facilidad con el siguiente código:

```
1         if ((buttonState == LOW && lastButtonState == HIGH)) {
2       char message = "r";
3
4       Serial.println("r");  // Add newline character as delimiter
5     }
6     lastButtonState = buttonState;
7     u2 = u1;
```

Donde simplemente al hacer click en el botón se manda el mensaje `'r'` a python, que gatilla el reseteo del programa. Ademas notamos que es en la linea `u2 = u1` donde se almacena cual fue el color del LED en la ronda anterior.

Se verifica si se ha presionado el botón pulsador para reiniciar el juego, enviando un carácter 'r' a través del puerto serial si se cumple la condición.

2.1.3. Comentarios y desafíos

En particular la estructura de la clase **MainWindow** nos permite organizar y coordinar eficientemente las diferentes funcionalidades en el juego de la vida, esto debido a la separación de sus métodos. Es más, en virtud de la herencia de **QMainWindow**, la clase MainWindow obtiene todas las funcionalidades de una ventana principal de PyQt, esto proporciona una base sólida para construir la interfaz de usuario. Otra muy buena razón para definir la clase de esta manera, fue por la inicialización de la interfaz de usuario (initUI), pues dicho método se encarga de configurar la apariencia y la disposición de los widgets en la ventana principal, lo que facilita la comprensión y proporciona una estructura flexible para cambios en el código. Por ejemplo y como explicamos, el método `update_game` se encarga de actualizar el estado del juego de la vida en la interfaz gráfica, además de enviar datos a Arduino en intervalos regulares de 10 segundos. Esta separación de responsabilidades mantiene el código organizado y facilita la implementación de nuevas funcionalidades relacionadas con el juego.

Respeto de la clase antes mencionada, la gestión de la comunicación serial (setup_serial y LecturaArduino) fue relevante, pues al manejarse en métodos separados, nos dio las facilidad de reutilizar el código del primero para luego diseñar el siguiente con un fin muy especifico.

No obstante y debido a que el código que trabajamos utiliza varias bibliotecas y patrones de programación fue complejo comprender cómo interactuaban las diferentes partes del código. Claramente durante la implementación, surgieron errores en la lógica del juego de la vida y sobretodo problemas de comunicación serial con Arduino.

La tarea de configurar la comunicación serial entre Python y Arduino fue compleja, pues requeríamos un mayor conocimiento sobre los puertos seriales y los protocolos de comunicación, por lo que fue necesario depurar la comunicación serial y seccionar los errores.

Esta actividad resultó en un gran desafío debido a la variedad de tecnologías y conceptos involucrados, así como los problemas y errores que surgieron durante el proceso de desarrollo, no obstante, con la suficiente paciencia, conocimiento y experiencia, estas dificultades pudieron superarse para crear una aplicación funcional y efectiva.

2.1.4. Evidencias

A continuación se deja constancia grafica del correcto funcionamiento de la actividad desarrollada si accede al siguiente link.

2.2. Esquemático de circuito Ítem 2.2

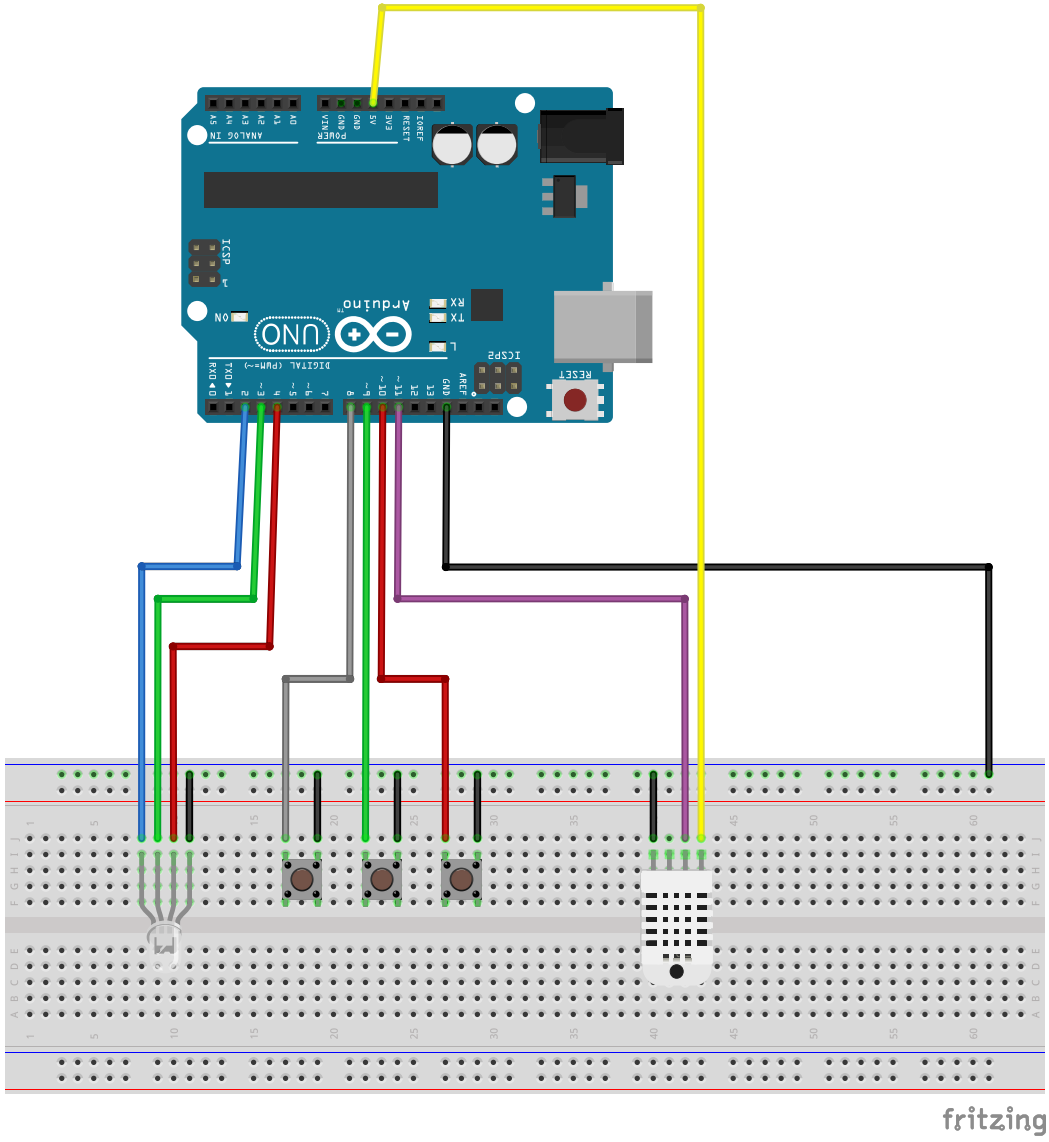

Figura 2.2: Esquemático del circuito empleado en la actividad de emplear "The game of Life" modificado

Para este setup, modificamos el del apartado 2.1 agregando 2 botones más, que se encarguen de las funciones bomba de muerte y curación, además del sensor de temperatura.

2.2.1. Código implementado en Python

Antes de explicar las partes del código, cabe destacar que corresponde a una versión modificada del anterior, por lo que vamos a centrar la discusión del apartado en qué cosas se agregaron y modificaron.

Funciones extra:

Antes de definir las clases que se utilizaran en la interfaz, definimos unas cuantas funciones generales que tienen como objetivo asistir a la ejecución del código principal. Estas son:

- La función `NumerosABooleano(arr)`, que tiene como objetivo tomar un arreglo de números y transformarlo a un conjunto de unos y ceros dependiendo si el numero es mayor a 0 o no.
- La función `FuncionBomba(matriz, bomba, x, y)`, que tiene como objetivo tomar una submatriz de la matriz `matriz` centrada en `(x,y)` y reemplazarla por la matriz `bomba`, que para los propósitos de nuestro código es una matriz de ceros de 21x21.
- La función `FuncionCura(matriz, bomba, x, y)`, que tiene como función hacer lo mismo que la `FuncionBomba`, pero en vez de aplicar directamente una matriz de ceros, aplica las condiciones de los valores entregados para la bomba de curación sobre la submatriz seleccionada.

GolGame modificado:

Tenemos que la versión modificada del juego Golgame esta dada por:

```
1
2     def __init__(self, size):
3         self.size = size
4         self.grid = np.random.choice([0, 100], size*size, p=[0.8, 0.2]).reshape(
    size, size)
5         self.temp = 10
6         self.RecibeTemp = True
7
8     def update(self):
9         kernel = np.array([[1, 1, 1],
10                           [1, 0, 1],
11                           [1, 1, 1]])
12        convolved = convolve2d(NumerosABooleano(self.grid), kernel, mode='same',
    boundary='fill')
13        birth = (convolved == 3) & (self.grid == 0)
14        survive = ((convolved == 2) | (convolved == 3)) & (NumerosABooleano(self.
    grid) == 1)
15        stress = ((convolved < 2) | (convolved > 3)) & (NumerosABooleano(self.
    grid) == 1)
16        self.grid[stress] = self.grid[stress] - 30
17        self.grid[birth] = 100
18        self.grid[survive] =  self.grid[survive]
19        self.grid[self.grid < 0] = 0
20        self.grid[self.grid > 100] = 100
21        if (self.temp  > 20):
22            self.grid[survive] =  self.grid[survive] + 10
23        elif (self.temp  < 20):
24            self.grid[survive] = self.grid[survive] - 50
```

En este caso, el cambio mas significativo es pasar de usar un booleano para los valores de las células a utilizar un numero del 0 al 100. Para eso, al inicializar el juego empezamos las células con un valor 100 en vez de 1, y luego, al definir la variable `convolved` para determinar el futuro de cada célula, utilizamos la función `NumerosABooleano()` talque podamos aplicar nuestras condiciones como en el ítem 2.1 sin ninguna dificultad.

Luego, agregamos todos los cambios de valores tal cual se nos dieron en las instrucciones del enunciado, que vienen dados por:

- La condición de `stress`, que produce que en vez de matar a todas las células cada ronda y después recontar las que sobreviven, se le reste 30 puntos de vida a todas las células que se encuentran en condiciones donde en el juego base morirían.

- la condición de que si alguno de los valores de la grilla son mayores o menores a 100 que se redefinan al limite correspondiente
- Agregamos la variable `temp`, con el objetivo de aumentar o disminuir la vida de las células si la temperatura se encuentra por sobre o debajo del umbral.

Clase GolWidget

Esta función, que es la responsable de generar el gráfico del juego fue dejada básicamente igual, con el único cambio realizado siendo que se cambio el mapa de color del gráfico de una escala de grises al `'viridis'`, para una visualización mas sencilla.

Clase Ui_MainWindow

Esta es la clase donde mas modificaciones se realizaron, ya que se implementó un buen número de nuevas funcionalidades con respecto al ítem 2.1. Antes que nada es conveniente mostrar la forma de la nueva interfaz y explicar parte por parte lo que hace cada elemento:

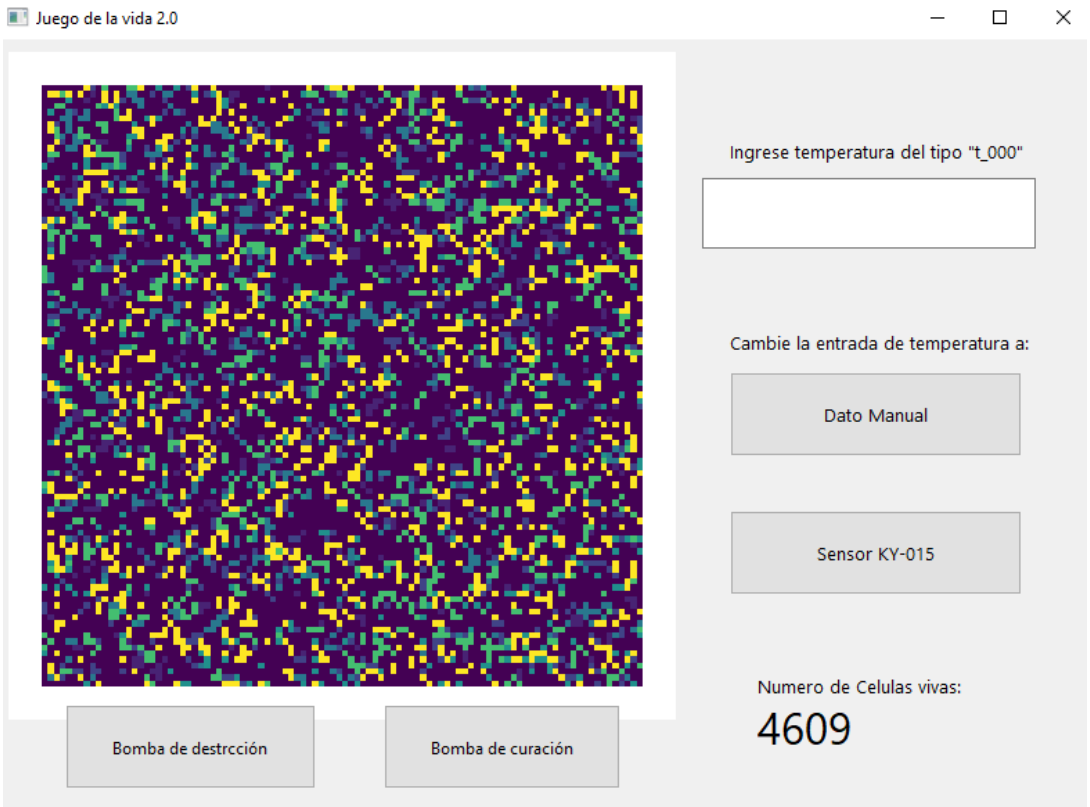

Figura 2.3: Interfaz gráfica de el juego de la vida modificado

Tenemos que los elementos nuevos de la función `setupUi` vienen dados por:

- Los 4 nuevos botones, que se crearon utilizando la función `QPushButton` que tienen como objetivo llamar a diferentes funciones dentro de `Ui_MainWindow` . Su estructura de código es la misma entre todos, por lo que basta con mostrar uno de ellos:

```
                self.pushButton_b = QtWidgets.QPushButton(parent=self.
centralwidget)
        self.pushButton_b.setGeometry(QtCore.QRect(50, 480, 181, 61))
        font = QtGui.QFont()
        font.setPointSize(9)
        self.pushButton_b.setFont(font)
        self.pushButton_b.setObjectName("pushButton")
        self.pushButton_b.clicked.connect(self.bomb_game)

```

  Los atributos definen básicamente donde se ubica espacialmente el botón y que función llama (en este caso `self.bomb_game`).

- El recuadro de texto donde ingresamos la temperatura, que se crea utilizando la función `QtextEdit`, con su código de la forma:

```
                self.textEdit = QtWidgets.QTextEdit(parent=self.
centralwidget)
        self.textEdit.setGeometry(QtCore.QRect(510, 100, 241, 51))
        self.textEdit.setObjectName("textEdit")
        dataT = self.textEdit.toPlainText()
        print(dataT)

```

  Donde la información se almacena en `dataT` utilizando la función `toPlainText()` .

- Las lineas de texto que no están asociadas a los botones, incluyendo el numero de células que se actualiza. Para esto se utilizo la función `QLabel`, que permite agregar un recuadro de texto sobre la interfaz, con codigo de la siguiente estructura:

```
            self.label = QtWidgets.QLabel(parent=self.centralwidget)
        self.label.setGeometry(QtCore.QRect(530, 70, 311, 20))
        font = QtGui.QFont()
        font.setPointSize(10)
        self.label.setFont(font)
        self.label.setObjectName("label")


```

  Donde notamos que el asociado al numero de células se actualiza dinámicamente al momento de llamar a la función `update_game( )` , que va a ser explicado posteriormente.

Notar que la función `setupUi` todavía es la que llama los timers del arduino y la actualización del juego.

Además se agrego la función `retranslateUi` , que tiene como objetivo simplemente agregar el texto a todos los elementos de la interfaz, utilizando la siguiente estructura:

```
    MainWindow.setWindowTitle(_translate("MainWindow", "Juego de la vida 2.0"))
```

Ahora, tenemos que las modificaciones realizadas a la función que recibe información al arduino vienen dadas por:

```
    if self.arduino.in_waiting > 0:
                self.mensaje = self.arduino.readline().decode().strip()  #Recibe, lee y almacena un mensaje mandado desde arduino
                print(self.mensaje,type(self.mensaje))
                if self.mensaje == "r":
                    self.reset_game()
                    self.mensaje = " "
                elif self.mensaje == "b_m":
                    self.bomb_game()
                    self.mensaje = " "
                elif self.mensaje == "b_c":
                    self.heal_game()
                    self.mensaje = " "
                elif self.mensaje[0] == "t" and self.gol_widget.game.RecibeTemp == True :
                    self.gol_widget.game.temp =  (int(self.mensaje[2:])) #'{:03d}'.format
                    self.mensaje = " "
```

Que es básicamente igual al del ítem 2.1 en estructura, solo que tiene la capacidad de recibir un numero mas amplio de datos, y gatillar otro conjunto de funciones que explicaremos a continuación.

Primero que nada, la función `reset_game` es completamente idéntica a la anterior, pero ahora tenemos las funciones `bomb_game` y `heal_game` , que simplemente lo que hacen es aplicar las funciones `FuncionBomba` y `FuncionCura` centradas en una casilla aleatoria de la grilla del juego, a través del siguiente código:

```
    def bomb_game(self): #Funcion que aplica la bomba de muerte:
        xrand =  np.random.randint(1, 101)
        yrand =  np.random.randint(1, 101)
        bomba = np.zeros((21, 21))
        self.gol_widget.game.grid = FuncionBomba(self.gol_widget.game.grid, bomba , xrand , yrand)
```

La siguiente es la condición que lee que se envió una señal de temperatura, y que además se asegura que el botón de ingreso por sensor (de la interfaz) sea el que esta activado, y aprovecha de actualizar el valor de la variable `temp` en la clase `GolGame` .

También se modifico la función `update_game()` talque actualizará el valor del contador del numero de células de la interfaz, y que aplicara la condición de reseteo si el numero de células vivas llega a ser 0, esta función quedo como:

```
        def update_game(self): #Funcion que actualiza el juego dentro de la interfaz principal e imprime el numero de celulas:
        self.gol_widget.update_game()
        self.alive_cells = '{:04d}'.format(np.sum(NumerosABooleano(self.gol_widget.game.grid))) # Este actualiza el numero de celulas vivas en la interfaz:
        self.number_label.setText(self.alive_cells)
        if self.alive_cells == 0:
            print("Se acabo el Juego, reiniciando:")
            self.reset_game()
```

Como últimos detalles, mencionamos que se agregaron las siguientes funciones:

```
    def LeeryEnviarT(self):
        dataT = self.textEdit.toPlainText()
        self.gol_widget.game.temp = int(dataT[2:])
        print("temperatura definida como:",dataT[2:])
        self.gol_widget.game.RecibeTemp = False

    def MandarTdeArduino(self):
        self.gol_widget.game.RecibeTemp = True
```

Que son las encargadas de conectar y desconectar el arduino como método de cambio de la temperatura, además de `LeeryEnviarT(self)` ser responsable de leer la casilla de texto y redefinir la variable `t` .

Notamos que los botones de la interfaz están cada uno de ellos conectados a las diferentes funciones que definimos anteriormente.

2.2.2. Código implementado en Arduino

El código implementado en el arduino recibió aun menos modificaciones que el de python, ya que simplemente agregamos un código similar para la activación de reseteo de la parte 2.1 asociado a las bombas de vida y muerte.

El elemento nuevo significativo del código corresponde al envió de la señal de temperatura, que viene dado por la siguiente estructura:

```
float t = dht.readTemperature();
int tInt = int(t);
String tStr = "t_0" + String(tInt); // La convertimos en algo enviable a python

// Comprobamos si ha habido algun error en la lectura
 //Serial.println(t);
  if (tBucle - tTemperatura >= 20000) {
    // Your data to sen

    tTemperatura = tBucle; // Update last send time
   Serial.println(tStr);
    if ( isnan(t)) {
   Serial.println("Error obteniendo los datos del sensor DHT11"); return; }
}
```

Donde básicamente implementamos la lectura del sensor DTH a una variable t, la que luego es transformada en un numero entero de 2 dígitos y enviada como un `string` a Python en intervalos de 20 segundos, la que se actualiza si esta conectada a la lectura por Arduino.

2.2.3. Comentarios y desafíos

El desarrollo de esta parte del problema, si bien mas largo y engorroso, resulto mas sencillo que el de la parte 2.1, ya que una vez teníamos una estructura funcional para la interfaz y el traspaso de datos mutuos entre arduino y python, el resto fue realizar modificaciones progresivas y graduales, sin mayores dificultades. Lo mas interesante fue que la suma de vida por temperatura a las células alteraba considerablemente la dinámica, ya que en los casos donde se le sumaba vida a las células, estas tendían a formar estructuras a lo largo de la grilla, mientras que cuando se restaba vida por temperatura la dinámica era mucho mas acelerada y las células no lograban formar ninguna estructura estable. Estas 2 dinámicas se pueden observar en el vídeo anexo.

2.2.4. Evidencias

A continuación se deja constancia gráfica del correcto funcionamiento de la actividad desarrollada si accede al siguiente link.