Entrega de trabajos del primer parcial

Barrera Pérez Carlos Tonatihu
Profesor: Genaro Juárez Martínez
Computing Selected Topics
Grupo: 3CM8

10 de noviembre de 2018

Índice

1. Máquina de Turing

1.1. Introducción

La máquina de Turing, presentada por Alan Turing en 1936 en On computable numbers, with an application to the Entscheidungsproblems, es el modelo matemático de un dispositivo que se comporta como un autómata finito y que dispone de una cinta de longitud infinita en la que se pueden leer, escribir o borrar símbolos. Existen otras versiones con varias cintas, deterministas o no, etc., pero todas son equivalentes (respecto a los lenguajes que aceptan).

Uno de los teoremas más importantes sobre las máquinas de Turing es que pueden simular el comportamiento de una computadora (almacenamiento y unidad de control). Por ello, si un problema no puede ser resuelto por una de estas máquinas, entonces tampoco puede ser resuelto por una computadora (problema indecidible, NP) [**?**].

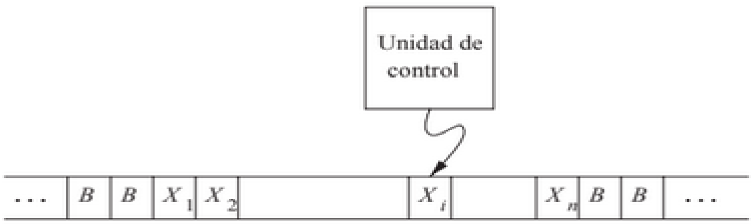

Figura 1: Una máquina de Turing

De manera formal la máquina de Turing se define de la siguiente manera:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

donde:

Q es el conjunto finito de estados de la unidad de control.

Σ es el conjunto finito de símbolos de entrada

Γ es el conjunto completo de símbolos de cinta; Σ siempre es un subconjunto de Γ

δ Es la función de transición. Los argumentos de $\delta(q, X)$ son un estado q y un símbolo de cinta X. El valor de $\delta(q, X)$, si está definido, es (p, Y, D) donde:

1. p es el siguiente estado de Q
2. Y es el símbolo de Γ, que se escribe en la casilla que señala la cabeza y que sustituye a cualquier símbolo que se encontrara en ella.
3. D es una dirección y puede ser L o R, lo que nos indica la dirección en que la cabeza se mueve, 'izquierda' (L) o 'derecha' (R), respectivamente

q_0 es el estado inicial, un elemento de Q, en el que inicialmente se encuentra la unidad de control.

B es el símbolo espacio en blanco. Este símbolo pertenece a *Gamma* pero no a *Sigma*; es decir, no es un símbolo de entrada.

F es el conjunto de los estados finales, un subconjunto de Q

1.2. Planteamiento de la práctica

La elaboración de este programa consistió en elaborar un máquina de Turing capaz de duplicar la cantidad de unos en una cadena de unos, es decir, si la cadena que se ingresa es 11 entonces la cadena de salida sera 1111.

Es por esto que la máquina sólo aceptara unos en la entrada mientras que los símbolos de la cinta incluirán a la X y la Y. De esta forma la máquina de Turing para este problema se define como [**?**]:

$$M = (\{q_0, q_1, q_2, q_3\}, \{1\}, \{1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

donde δ se especifica en la siguiente tabla:

	Símbolo			
Estado	1	X	Y	B
0	(1, X, R)	-	(3, 1, R)	-
1	(1, 1, R)	-	(1, Y, R)	(1, Y, L)
2	(2, 1, L)	(0, 1, R)	(2, Y, L)	-
3	-	-	(3, 1, R)	-

EL funcionamiento de esta máquina se puede entender mejor con el diagrama de la figura 2

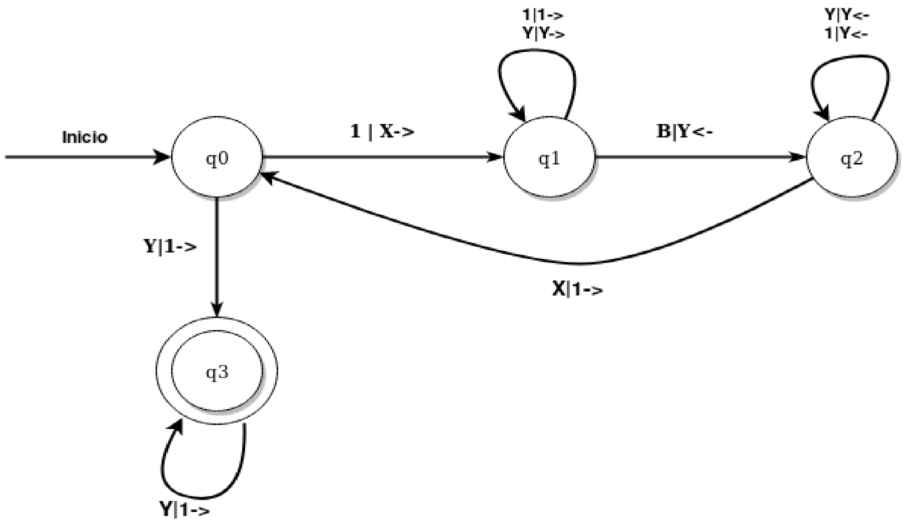

Figura 2: Representación gráfica de las transiciones de la máquina de Turing

El funcionamiento del programa empieza cuando se inserta una cadena de unos y la máquina empezara a trabajar, además de generar una cadena final se muestra una animación del como se están realizando las transiciones en la cinta de la máquina, por otro lado se imprime en consola el historial de movimientos que se hacen, este historial también se guarda en un archivo de texto.

1.3. Desarrollo

El código de este programa fue realizado en Python 3.7 y se utilizo la biblioteca tkinter para la parte gráfica.

Archivo: turing.py Esta clase es la que modela la máquina de Turing, los parámetros importantes son: el estado inicial, los estados finales, la cadena de entrada y la función de transición.

```python
class MaquinaTuring:
    """MaquinaTuring"""
    def __init__(self, estado_inicial, estados_finales, cadena, transiciones):
        self.transiciones = transiciones
        self.inicial = estado_inicial
        self.finales = estados_finales
        self.cinta = list(cadena)
        self.estado_actual = self.inicial
        self.apuntador = 0
        self.blanco = "B"
```

```
        self.direccion = None

    def consumir(self):
        """Toma un simbolo de la cinta y lo evalua en la funcion de
        transicion"""
        if len(self.cinta) - 1 < self.apuntador:
            caracter = 'B'
        else:
            caracter = self.cinta[self.apuntador]
        tupla = (self.estado_actual, caracter)
        if tupla in self.transiciones:
            siguiente = self.transiciones[tupla]
            if len(self.cinta) - 1 < self.apuntador:
                self.cinta.append(self.blanco)
            if self.apuntador < 0:
                self.cinta.insert(0, self.blanco)
            self.cinta[self.apuntador] = siguiente[1]
            if siguiente[2] == "R":
                self.apuntador += 1
            else:
                self.apuntador -= 1

            self.estado_actual = siguiente[0]
            self.direccion = siguiente[2]
            return True
        else:
            return False

    def es_final(self):
        """Metodo para comprobar si nos encontramos en un estado final"""
        if self.estado_actual in self.finales:
            if len(self.cinta) - 1 < self.apuntador or self.apuntador < 0:
                return True
        return False
```

Archivo: diagrama.py Este archivo implementa la clase MaquinaTuring.py y se declaran los parámetros que se pasaran a este archivo así como la captura de la cadena y la escritura del registro de transiciones en consola y en archivo de texto, sin olvidar la animación de dichas transiciones.

```
import tkinter as tk
import time
from tkinter import font as tkfont
from turing import MaquinaTuring

# Tabla de transiciones que modela el automata
```

```python
transiciones = {
            ("q0", "1"): ("q1", "X", "R"),
            ("q0", "Y"): ("q3", "1", "R"),
            ("q1", "1"): ("q1", "1", "R"),
            ("q1", "Y"): ("q1", "Y", "R"),
            ("q1", "B"): ("q2", "Y", "L"),
            ("q2", "1"): ("q2", "1", "L"),
            ("q2", "X"): ("q0", "1", "R"),
            ("q2", "Y"): ("q2", "Y", "L"),
            ("q3", "Y"): ("q3", "1", "R"),
            }

entrada = input("Ingrese la cadena de unos: ")
maquina = MaquinaTuring("q0", "q3", entrada, transiciones)

# Configuracion de la ventana
gui = tk.Tk()
gui.geometry("600x400+100+100")
gui.title("Maquina de Turing")
c = tk.Canvas(gui, width=600, height=400)
c.pack()
bold_font = tkfont.Font(family="Arial", size=24)

# Principales componentes que se animan
control = c.create_rectangle(150, 100, 200, 150, fill="lightblue")
flecha = c.create_line(175, 150, 175, 175, arrow=tk.LAST, width=3)
texto = c.create_text(165, 200, text=''.join(maquina.cinta), font=bold_font,
                      anchor=tk.W)
estado = c.create_text(160, 125, text=maquina.estado_actual, font=bold_font,
                       anchor=tk.W)

archivo = open("salida.txt", "w+")
# Mientras no llegues a un estado final continua
while not maquina.es_final():
    print('Cadena: {}'.format(''.join(maquina.cinta)))
    print('Estado actual: {}, apuntador: {}'.format(maquina.estado_actual,
          maquina.apuntador+1))

    archivo.write('Cadena: {}\n'.format(''.join(maquina.cinta)))
    archivo.write('Estado actual: {}, apuntador: {}\n'
                  .format(maquina.estado_actual, maquina.apuntador+1))
    if not maquina.consumir():
        print('*' * 20)
```

```
        archivo.write('*' * 20)
        archivo.write('\n')
        break
    print('Siguiente estado: {}'.format(maquina.estado_actual))
    print('*'*20)

    archivo.write('Siguiente estado: {}\n'.format(maquina.estado_actual))
    archivo.write('*' * 20)
    archivo.write('\n')

    gui.update()
    time.sleep(1)
    c.itemconfigure(texto, text=''.join(maquina.cinta), anchor=tk.W)
    c.itemconfigure(estado, text=maquina.estado_actual)
    if maquina.direccion == 'R':
        c.move(control, 19, 0)
        c.move(flecha, 19, 0)
        c.move(estado, 19, 0)
    else:
        c.move(control, -19, 0)
        c.move(estado, -19, 0)
        c.move(flecha, -19, 0)

print('Cadena final: {}'.format(''.join(maquina.cinta)))
archivo.write('Cadena final: {}\n'.format(''.join(maquina.cinta)))
archivo.close()
gui.mainloop()
```

1.4. Pruebas

El siguiente ejemplo es la cadena con un solo uno.

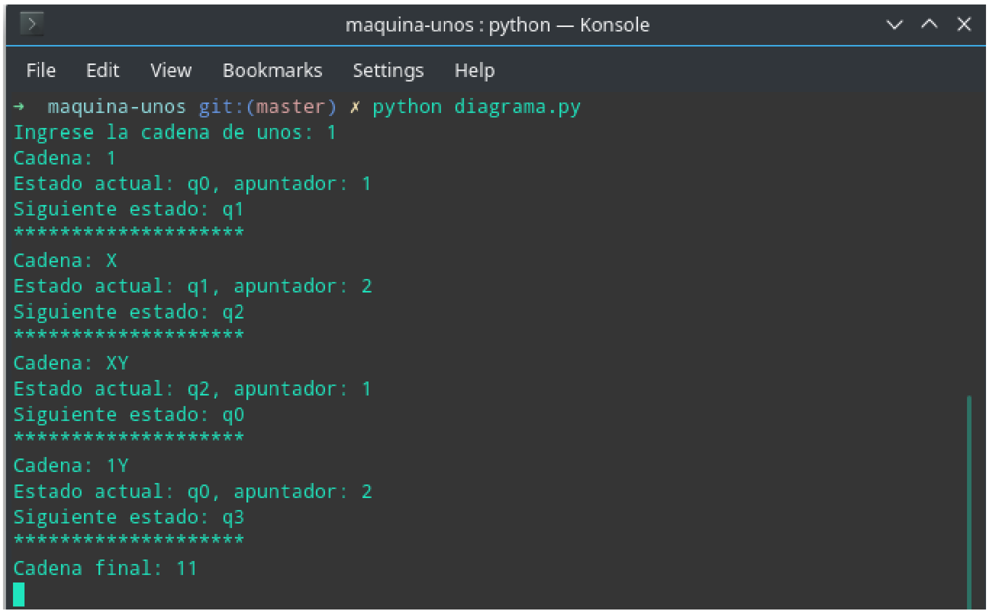

Figura 3: Salida en consola

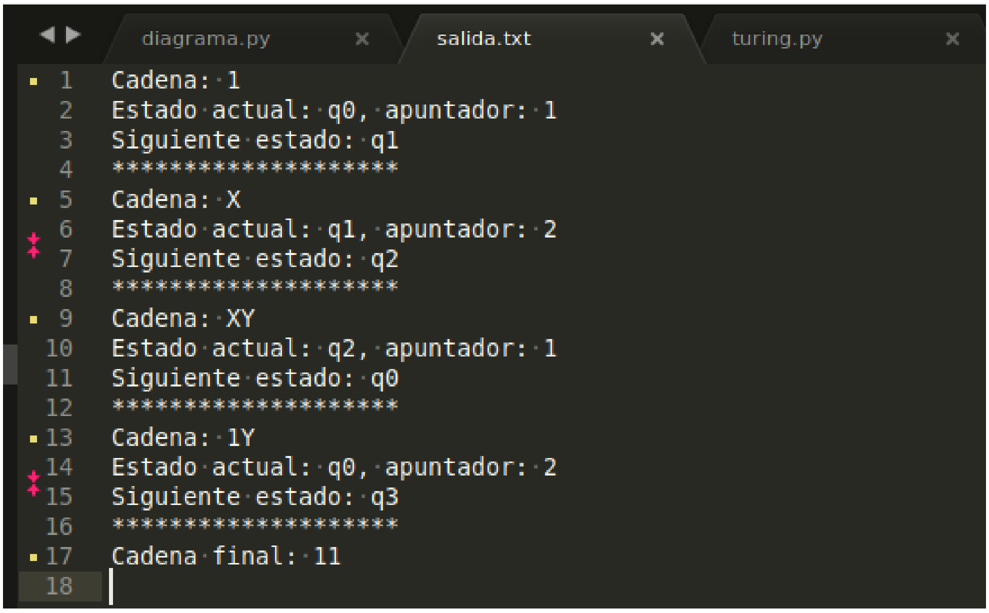

Figura 4: Registro de transiciones

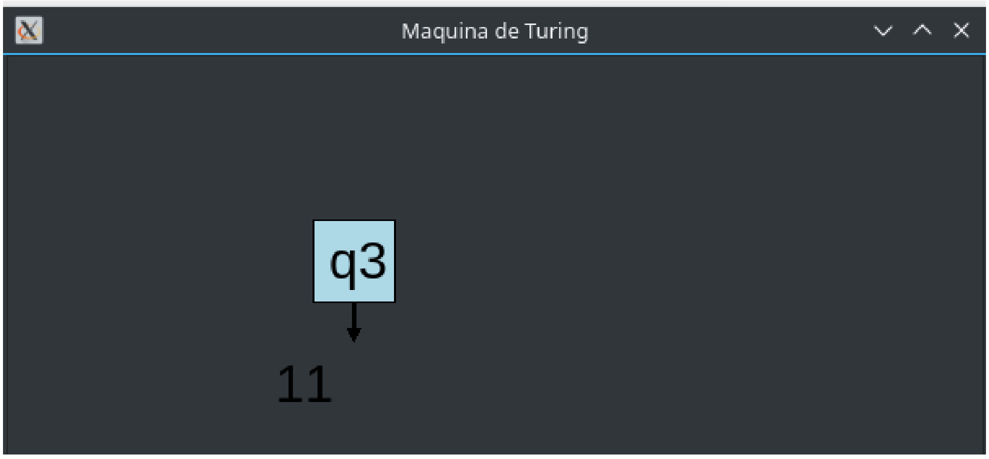

Figura 5: Representación gráfica de las transiciones de la máquina de Turing

En este ejemplo se inserta una cadena con dos unos.

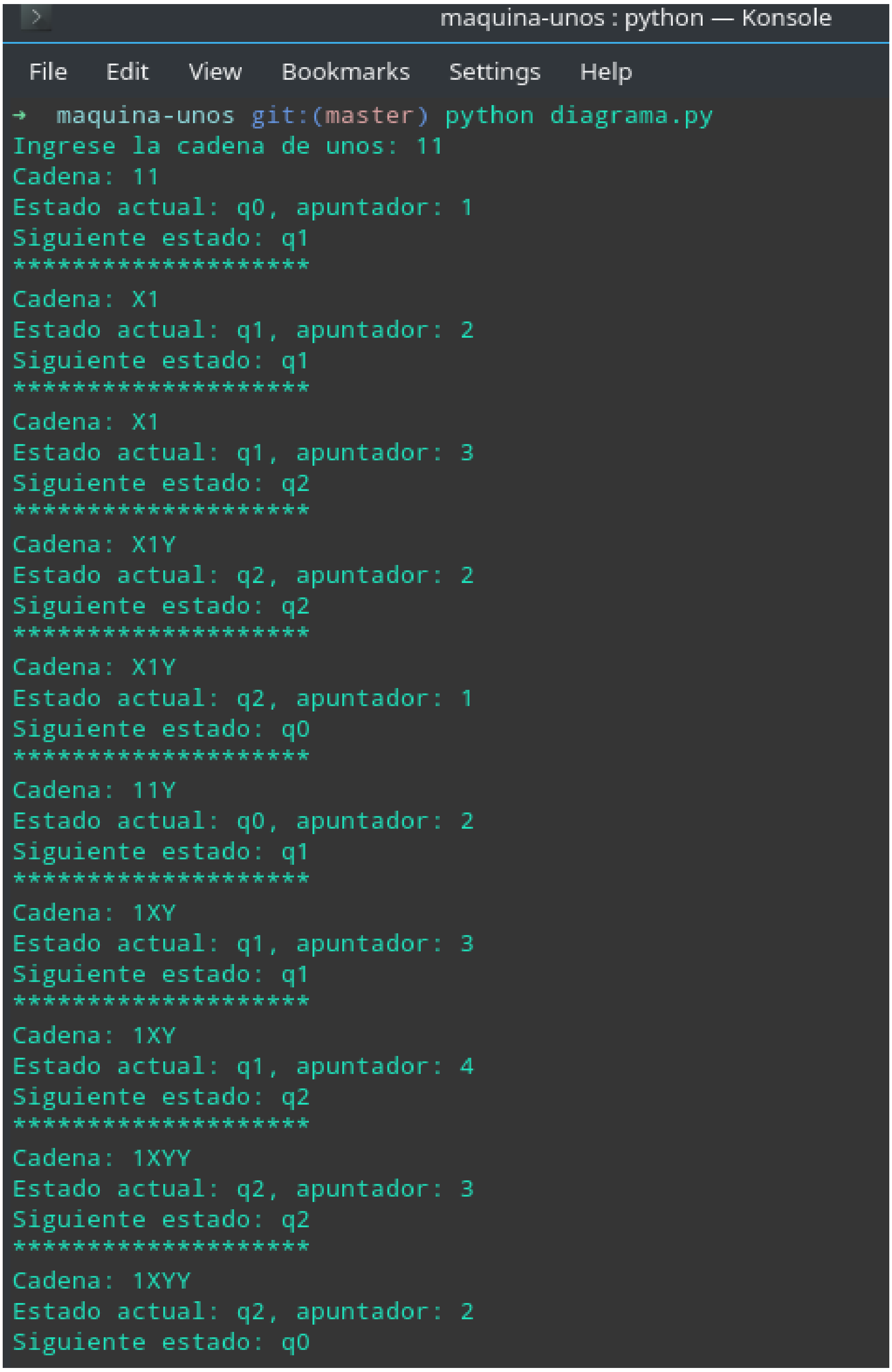

Figura 6: Salida en consola parte uno

```
********************
Cadena: 1XYY
Estado actual: q2, apuntador: 3
Siguiente estado: q2
********************
Cadena: 1XYY
Estado actual: q2, apuntador: 2
Siguiente estado: q0
********************
Cadena: 11YY
Estado actual: q0, apuntador: 3
Siguiente estado: q3
********************
Cadena: 111Y
Estado actual: q3, apuntador: 4
Siguiente estado: q3
********************
Cadena final: 1111
```

Figura 7: Salida en consola parte dos

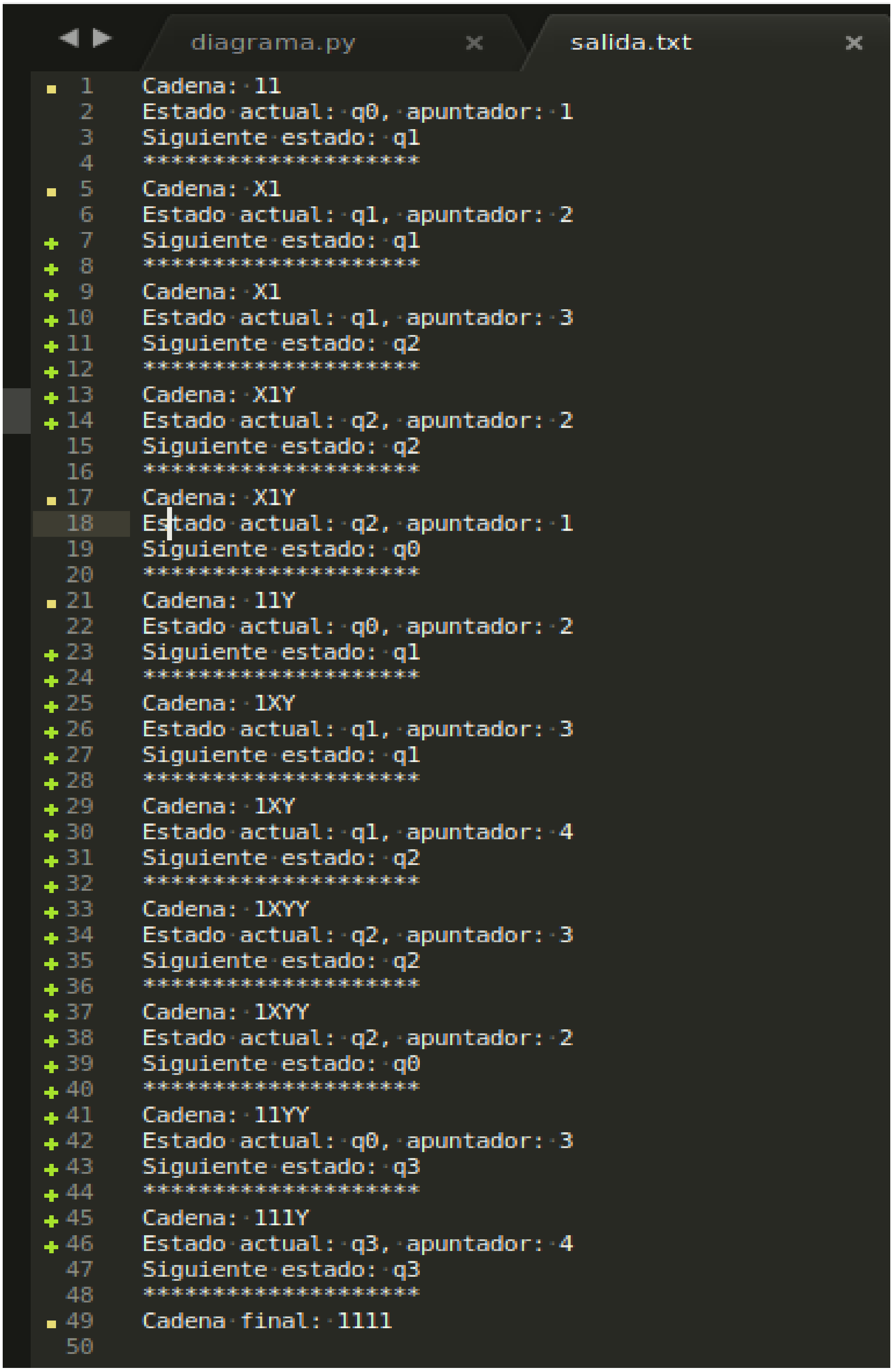

Figura 8: Registro de transiciones

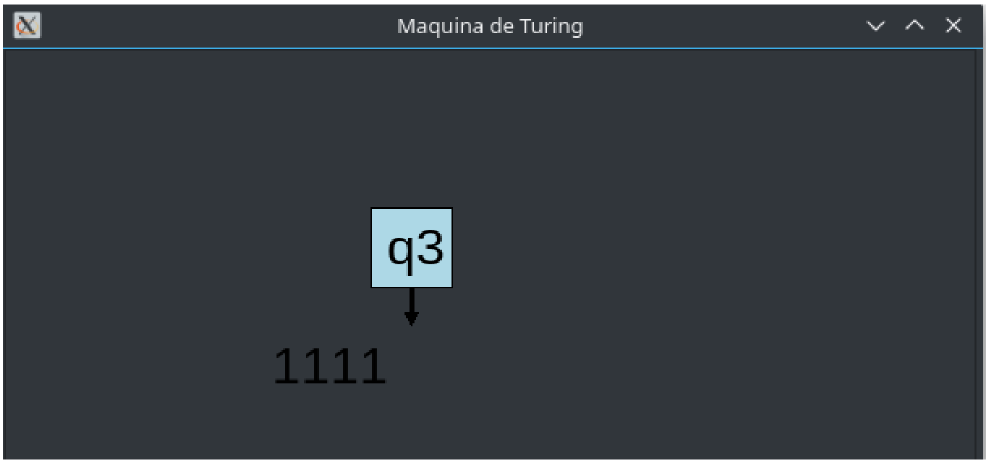

Figura 9: Representación gráfica de las transiciones de la máquina de Turing

1.5. Conclusiones

El hacer este programa probó lo útil y poderosa que es la máquina de Turing en la solución de problemas, ya que como es evidente si un problema no se puede solucionar usando una máquina de Turing entonces es no computable y a pesar que este problema de duplicar una cadena no es difícil el implementar una máquina de Turing resulta en una solución bastante practica y fácil de implementar.

2. Autómata celular (Regla de life y difusión)

Los autómatas celulares(AC) surgen en la década de 1940 con John Von Neumann, que intentaba modelar una máquina que fuera capaz de auto-replicarse, llegando así a un modelo matemático de dicha maquina con reglas complicadas sobre una red rectangular. Inicialmente fueron interpretados como conjunto de células que crecían, se reproducían y morían a medida que pasaba el tiempo. A esta similitud con el crecimiento de las células se le debe su nombre.[**?**]

Un autómata celular se caracteriza por contar con los siguientes elementos:

1. Arreglo regular. Ya sea un plano de dos dimensiones o un espacio n-dimensional, este es el espacio de evoluciones, y cada división homogénea del arreglo es llamada célula.

2. Conjunto de estados. Es finito y cada elemento o célula del arreglo toma un valor de este conjunto de estados. También se denomina alfabeto. Puede ser expresado en valores o colores.

3. configuración inicial. Consiste en asignar un estado a cada una de las células del espacio de evolución inicial del sistema.

4. Vecindades. Define el conjunto contiguo de células y posición relativa respecto a cada una de ellas. A cada vecindad diferente corresponde un elemento del conjunto de estados.

5. Función local. Es la regla de evolución que determina el comportamiento del A. C. Se conforma de una célula central y sus vecindades. Define como debe cambiar de estado cada célula dependiendo de los estados anteriores de sus vecindades. Puede ser una expresión algebraica o un grupo de ecuaciones.

2.1. Introducción

2.2. Planteamiento de la práctica

Este programa implementar la simulación de un autómata celular. Los puntos importantes a señalar son que cuenta con una interfaz gráfica para el usuario en la cual aparecen los unos (célula viva) y ceros (célula muerta) que son el principal elemento en este autómata celular y que son representados como pequeños cuadros que cambian su tamaño de acuerdo a la población que se tenga. Las características de este simulador son las siguientes:

- Permitir seleccionar el tamaño de la población de la matriz de unos y ceros.
- Permitir seleccionar la regla que se utilizara en cada iteración de la simulación.
- Se podrá elegir la distribución de unos que habrá en la matriz.
- Se podrá cambiar los colores de los ceros y los unos de la simulación.
- Permitir guardar la matriz que se esta trabajando en un archivo de texto.
- Se mostrara el cambio de unos que hay a lo largo de cada iteración.

El objetivo que se tiene es mostrar una matriz de hasta 1000 por 1000 para poder observar un comportamiento que nos proporcione información.

2.3. Desarrollo

Archivo: gol.py En este archivo se encuentra la clase que controla todo el juego de la vida, desde el como se muestra en pantalla a como trabaja. El código fue desarrollado en python 3 y se utilizo la biblioteca tkinter.

```python
from tkinter import Tk, Canvas, Frame, Button, Entry, Label,
    Scale
from tkinter import BOTH, TOP, LEFT, HORIZONTAL
import numpy as np
from tkcolorpicker import askcolor

class Ventana(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
        # Elementos interfaz
        self.ceros = "white"
        self.unos = "black"
        self.regla = [2, 3, 3, 3]
        self.e1 = None
        self.e2 = None
        self.contador = 0
        self.colorBtn1 = None
        self.colorBtn2 = None
        self.barra = None
        self.canvas = None
        # variables del juego de la vida
        self.pausa = True
```

```python
        self.tam = 100
        self.tam_cuadro = 10
        self.distribucion = .5
        self.cuadritos = np.zeros(shape=(self.tam, self.tam), dtype=int)
        self.celulas = np.random.randint(2, size=(self.tam, self.tam), dtype=int)
        self.historia_x = list()
        self.historia_y = list()
        self.tiempo = 0
        # Historial de unos
        archivo = open("grafica.txt", "w")
        archivo.close()
        self.initUI()

    def iniciar(self):
        archivo = open("grafica.txt", "w")
        archivo.close()
        self.canvas.delete('all')
        self.update_idletasks()
        self.pausa = True
        self.contador = 0
        self.tiempo = 0
        self.tam = int(self.e2.get())
        self.tam_cuadro = 0
        while self.tam_cuadro*self.tam < 1000:
            self.tam_cuadro += 1
        if self.tam_cuadro*self.tam > 1000:
            self.tam_cuadro -= 1
        print(self.tam_cuadro)
        self.distribucion = self.barra.get()/100
        self.celulas = np.random.choice([1, 0], size=(self.tam, self.tam), p=[self.distribucion, 1-self.distribucion])
        self.cuadritos = np.zeros(shape=(self.tam, self.tam), dtype=int)
        texto = self.e1.get().split(",")
        self.regla[0] = int(texto[0])
        self.regla[1] = int(texto[1])
        self.regla[2] = int(texto[2])
        self.regla[3] = int(texto[3])
        self.contar_unos()
        self.re_dibujar()


    def contar_unos(self):
        for i in range(self.tam):
            for j in range(self.tam):
                if self.celulas[i, j] == 1:
                    self.contador += 1
```

```
    def pulsar_cuadrito(self, event):
        item = self.canvas.find_closest(event.x, event.y)[0]
        x, y = np.where(self.cuadritos==item)
        if self.canvas.itemcget(item, "fill") == self.unos:
            self.canvas.itemconfig(item, fill=self.ceros)
            self.celulas[x[0]][y[0]] = 0
        else:
            self.canvas.itemconfig(item, fill=self.unos)
            self.celulas[x[0]][y[0]] = 1


    def re_dibujar(self):
        print("REDIBUJAR")
        for i in range(self.tam):
            for j in range(self.tam):
                if self.celulas[i, j] == 0:
                    self.cuadritos[i, j] = self.canvas.create_rectangle(0 + (j * self.tam_cuadro),
                                                                        0 + (i * self.tam_cuadro),
                                                                        self.tam_cuadro + (j * self.tam_cuadro),
                                                                        self.tam_cuadro + (i * self.tam_cuadro),
                                                                        fill=self.ceros, width=0, tag="btncuadrito")
                else:
                    self.cuadritos[i, j] = self.canvas.create_rectangle(0 + (j * self.tam_cuadro),
                                                                        0 + (i * self.tam_cuadro),
                                                                        self.tam_cuadro + (j * self.tam_cuadro),
                                                                        self.tam_cuadro + (i * self.tam_cuadro),
                                                                        fill=self.unos, width=0, tag="btncuadrito")
                    self.contador += 1

        self.canvas.tag_bind("btncuadrito", "<Button-1>", self.pulsar_cuadrito)
        self.update()


    def initUI(self):
        self.parent.title("Layout Test")
        self.pack(fill = BOTH, expand = 1)

```

```
        self.canvas = Canvas(self, relief ='raised', width = 1200, height = 800)
        self.canvas.pack(side = LEFT)

        Label(self, text="Regla:").pack(side=TOP)
        self.e1 = Entry(self, fg="black", bg="white")
        self.e1.insert(10, "2,3,3,3")
        self.e1.pack(side=TOP)

        Label(self, text="Tamano:").pack(side=TOP)
        self.e2 = Entry(self, fg="black", bg="white")
        self.e2.insert(10, "100")
        self.e2.pack(side=TOP)

        Label(self, text="Porcentaje de unos").pack(side=TOP)
        self.barra = Scale(self, from_=0, to=100, orient=HORIZONTAL, tickinterval=50)
        self.barra.set(50)
        self.barra.pack(side=TOP)

        btnIniciar = Button(self, text="Iniciar/Reiniciar", command=self.iniciar)
        btnIniciar.pack(side=TOP)

        button1 = Button(self, text="Pausa/Reanudar", command=self.empezar_dentener)
        button1.pack(side = TOP)

        self.colorBtn1 = Button(self, text="Selecciona el color de unos", command=self.getColorUnos, bg=self.unos)
        self.colorBtn1.pack(side = TOP)

        self.colorBtn2 = Button(self, text="Selecciona el color de ceros", command=self.getColorCeros, bg=self.ceros)
        self.colorBtn2.pack(side=TOP)

        btnSave = Button(self, text="Guardar", command=self.guardar)
        btnSave.pack(side=TOP)


    def actualizar_color_matriz(self):
        for i in range(self.tam):
            for j in range(self.tam):
                if self.celulas[i][j] == 0:
                    self.canvas.itemconfig(self.cuadritos[i][j], fill=self.ceros)
                else:
```

```python
                    self.canvas.itemconfig(self.cuadritos[i][j], fill=self.unos)

        self.update_idletasks()

    def getColorUnos(self):
        color = askcolor()
        if not color[1] == None:
            self.unos = color[1]
            self.colorBtn1.configure(bg=self.unos)
            self.actualizar_color_matriz()


    def getColorCeros(self):
        color = askcolor()
        if not color[1] == None:
            self.ceros = color[1]
            self.colorBtn2.configure(bg=self.ceros)
            self.actualizar_color_matriz()

    def guardar(self):
        # np.savetxt("matriz.txt", self.celulas, fmt="%d")
        archivo = open("matriz.txt", 'a')
        archivo.write("tiempo={}\n".format(self.tiempo))
        for i in range(self.tam):
            for j in range(self.tam):
                archivo.write("{} ".format(self.celulas[i, j]))
            archivo.write("\n")

        archivo.write("\n")
        archivo.close()


    def cargar(self):
        self.celulas = np.loadtxt("prueba.txt", dtype=int)
        self.canvas.delete('all')
        self.tam = self.celulas.shape[0]
        #self.celulas = np.random.randint(2, size=(self.tam, self.tam), dtype=int)
        self.cuadritos = np.zeros(shape=(self.tam, self.tam), dtype=int)
        print(self.celulas)
        print(self.tam)
        self.canvas.configure(width=self.tam, height=self.tam)
        # self.contar_unos()
        self.re_dibujar()
        self.update_idletasks()
        self.update()

```

```

    def empezar_dentener(self):
        print("empezar_detener")
        self.pausa = not self.pausa
        self.animacion()

    def animacion(self):
        if not self.pausa:
            self.historia_y.append(self.contador)
            self.historia_x.append(self.tiempo)
            archivo = open("grafica.txt", "a")
            archivo.write("{},{}\n".format(self.tiempo, self.contador))
            archivo.close()
            nueva_poblacion = self.celulas.copy()
            for i in range(self.tam):
                print(i)
                for j in range(self.tam):
                    vecinos = (self.celulas[i - 1, j - 1] + self.celulas[i - 1, j] + self.celulas[i - 1, (j + 1) % self.tam]
                               + self.celulas[i, (j + 1) % self.tam] + self.celulas[(i + 1) % self.tam, (j + 1) % self.tam]
                               + self.celulas[(i + 1) % self.tam, j] + self.celulas[(i + 1) % self.tam, j - 1] + self.celulas[i, j - 1])
                    if self.celulas[i, j] == 1:
                        if vecinos < self.regla[0] or vecinos > self.regla[1]:
                            nueva_poblacion[i, j] = 0
                            self.canvas.itemconfig(self.cuadritos[i][j], fill=self.ceros)
                            self.contador -= 1
                    else:
                        if vecinos >= self.regla[2] and vecinos <= self.regla[3]:
                            nueva_poblacion[i, j] = 1
                            self.canvas.itemconfig(self.cuadritos[i][j], fill=self.unos)
                            self.contador += 1

            self.celulas[:] = nueva_poblacion[:]
            self.update_idletasks()
            print("Termino")
            self.tiempo += 1
        self.after(1000, self.animacion)

def main():
    root = Tk()
    root.geometry('1360x750+0+0')
```

```
    app = Ventana(root)
    app.mainloop()

main()
```

Archivo: grafica.py En este archivo se encuentra el código que se encarga de graficar la historia de unos a lo largo de la animación, al igual que el archivo anterior se utilizo python 3, sin embargo en la graficación se realizo con la biblioteca matplotlib.

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure('Historial de unos')
fig.suptitle("Historial de unos")
ax1 = fig.add_subplot(1, 1, 1)
def animacion(i):
    info = open("grafica.txt", "r").read()
    lineas = info.split("\n")
    xs = []
    ys = []

    for linea in lineas:
        if len(linea) > 1:
            x,y = linea.split(",")
            xs.append(int(x))
            ys.append(int(y))
    ax1.clear()
    ax1.plot(xs, ys)

ani = animation.FuncAnimation(fig, animacion, interval=1000)

plt.show()
```

2.4. Pruebas

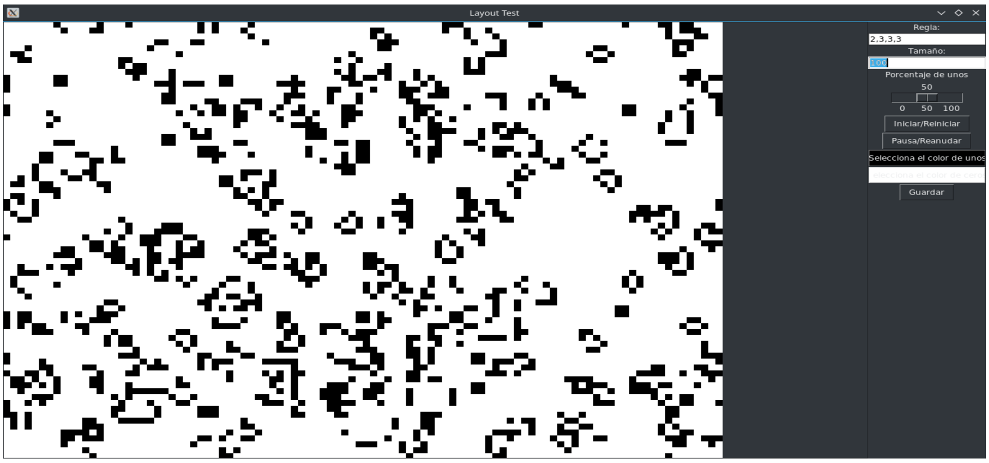

Figura 10: Juego de la vida con la regla 2333

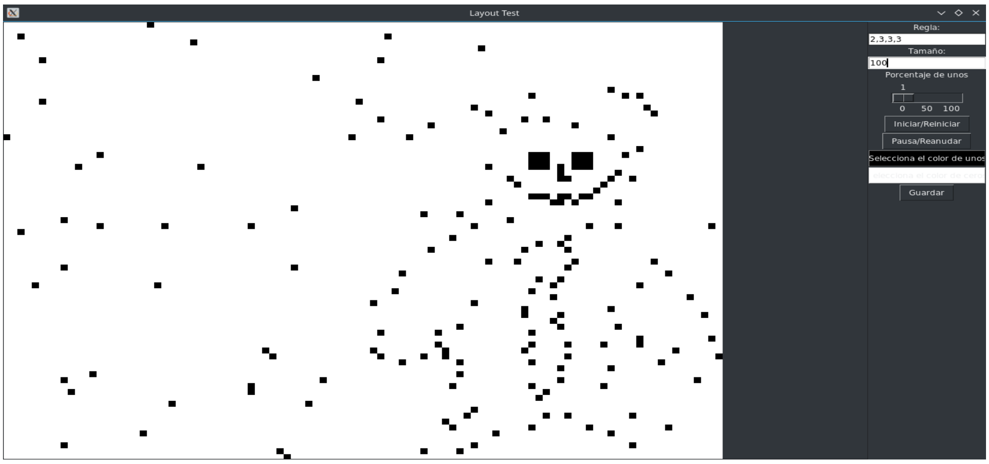

Figura 11: Juego de la vida con células seleccionadas por el usuario

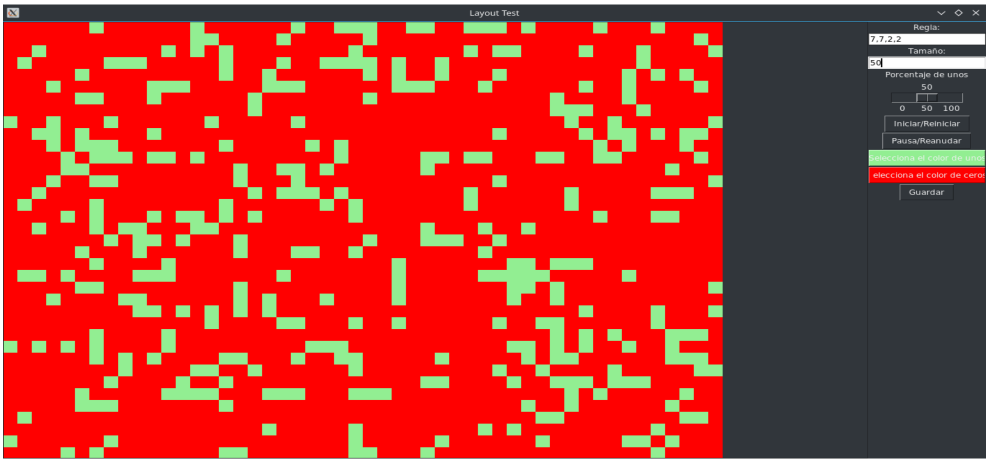

Figura 12: Juego de la vida cambiando los colores y la regla 7722

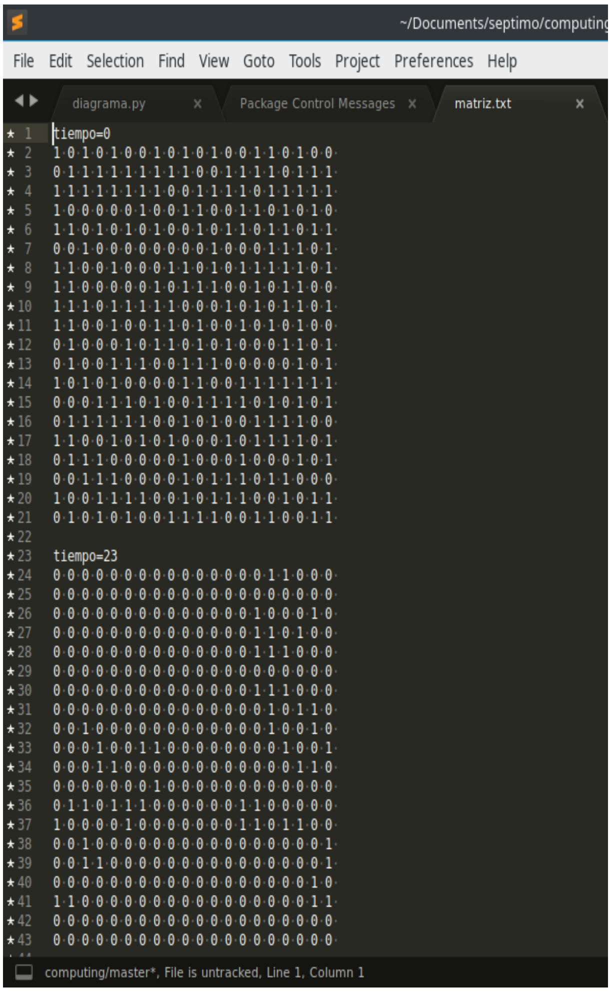

Figura 13: Matriz que se guarda

2.4.1. Análisis de poblaciones

En esta parte se hicieron pruebas con la regla de life y difusión aumentando la densidad de la población de 10 en 10 por ciento hasta llegar al máximo de 90 por ciento debido que en un 100 por ciento no se aprecia nada al igual que en cero, las pruebas de realizaron tras 500 o 1000 generaciones en una matriz de 100 por 100.

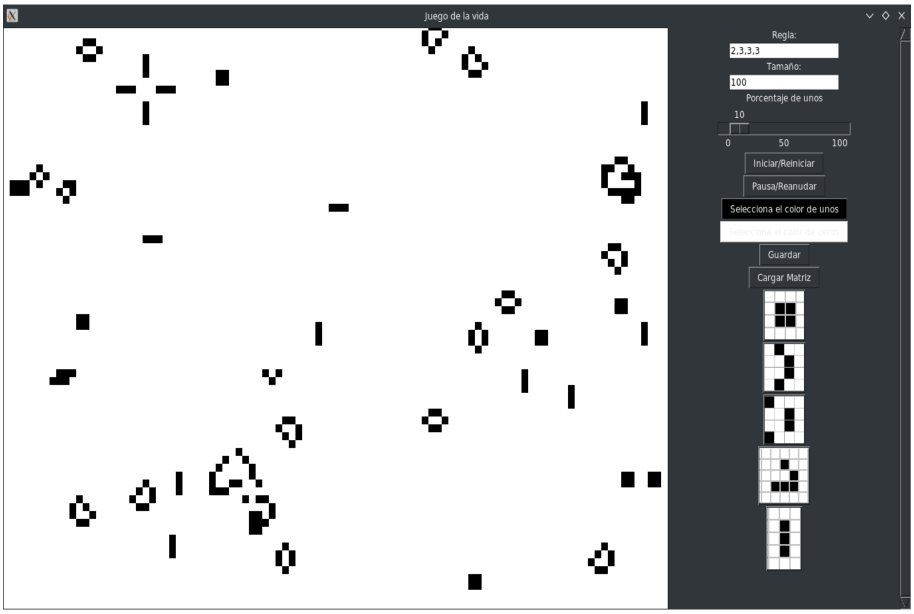

Figura 14: Regla de life con una probabilidad de unos de 10 %

Figura 15: Comportamiento de la población de la simulación anterior

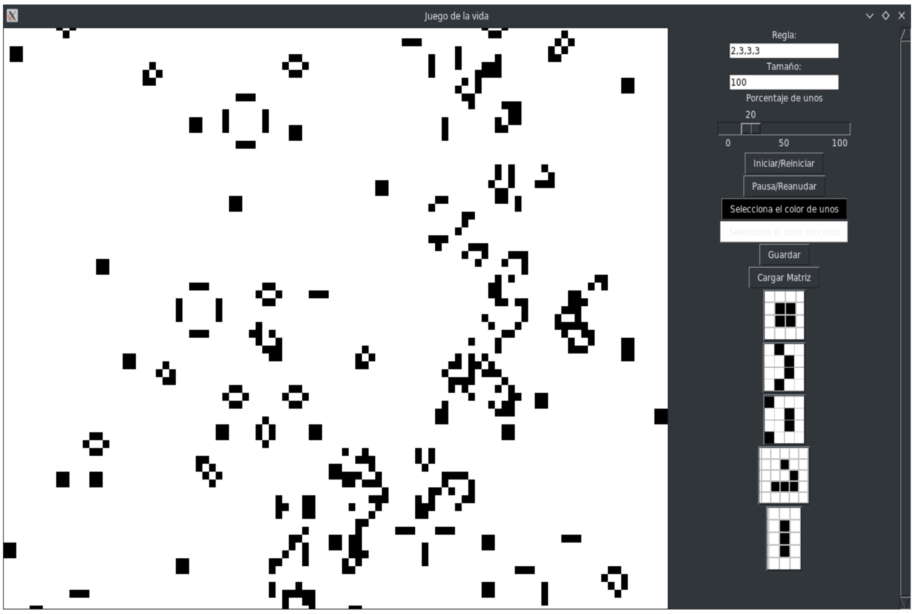

Figura 16: Regla de life con una probabilidad de unos de 20 %

Figura 17: Comportamiento de la población de la simulación anterior

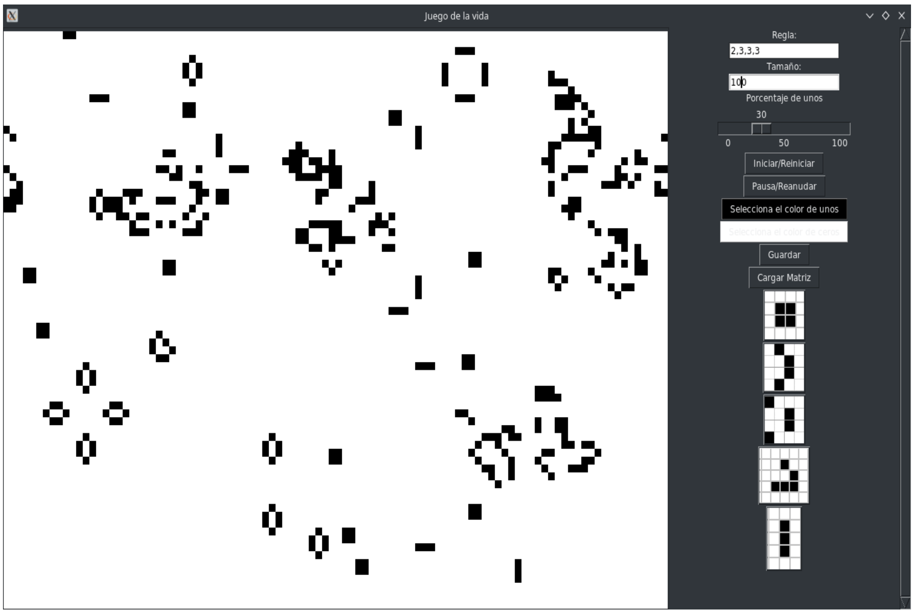

Figura 18: Regla de life con una probabilidad de unos de 30 %

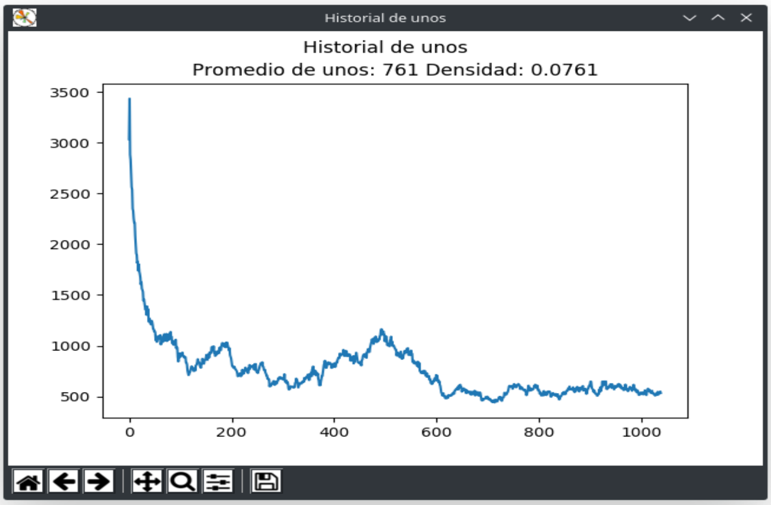

Figura 19: Comportamiento de la población de la simulación anterior

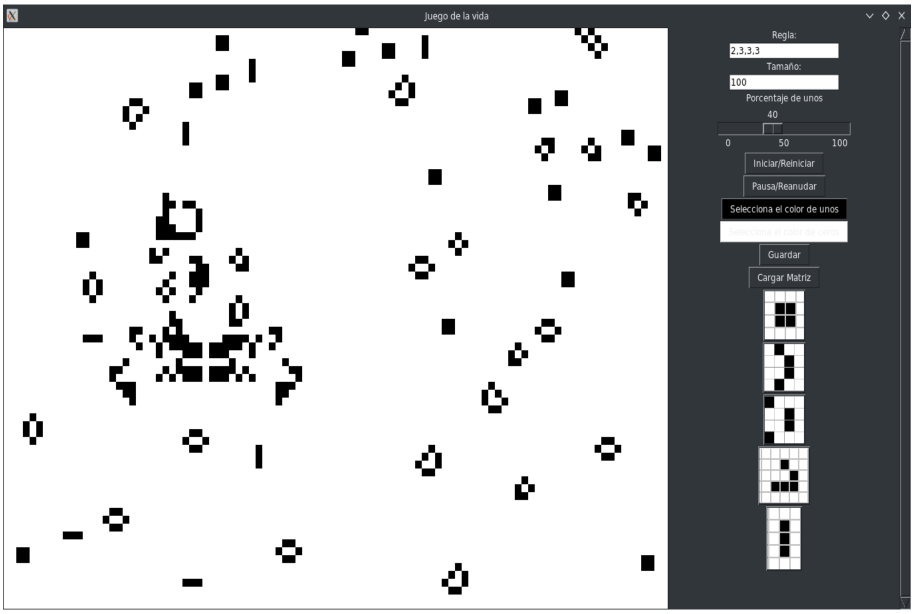

Figura 20: Regla de life con una probabilidad de unos de 40 %

Figura 21: Comportamiento de la población de la simulación anterior

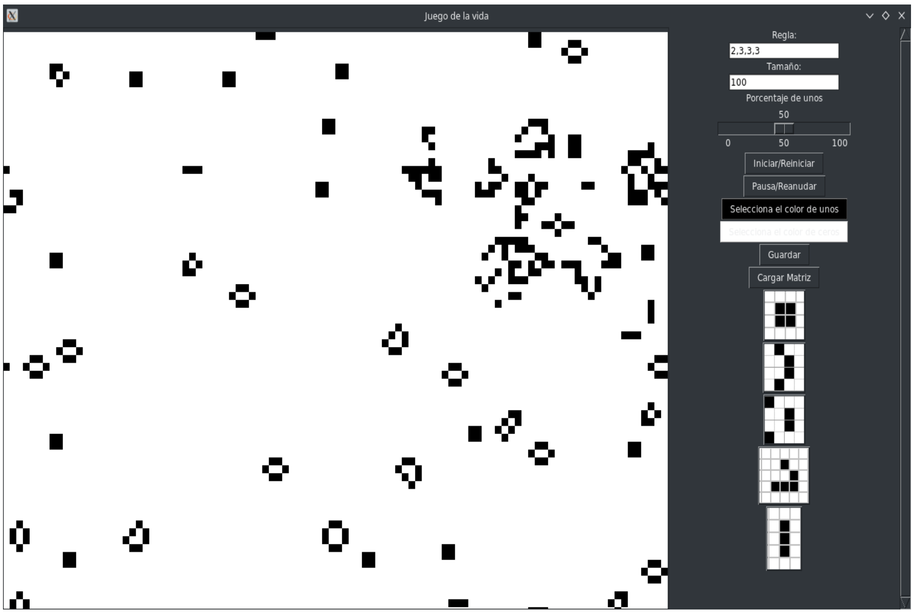

Figura 22: Regla de life con una probabilidad de unos de 50 %

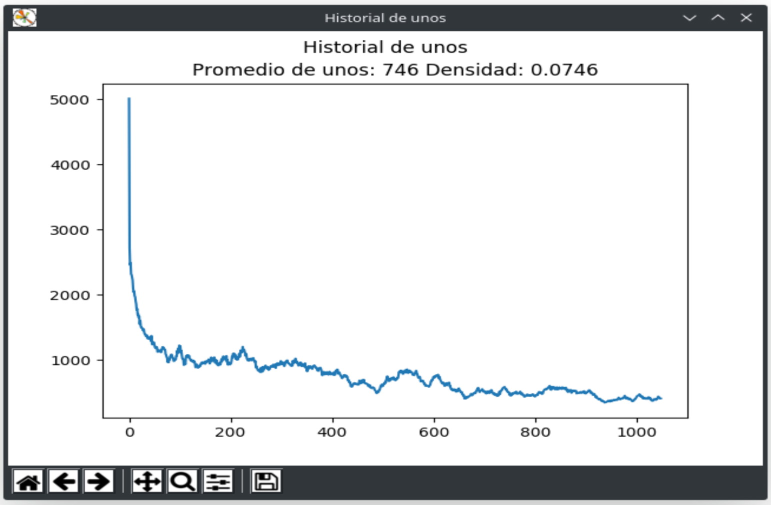

Figura 23: Comportamiento de la población de la simulación anterior

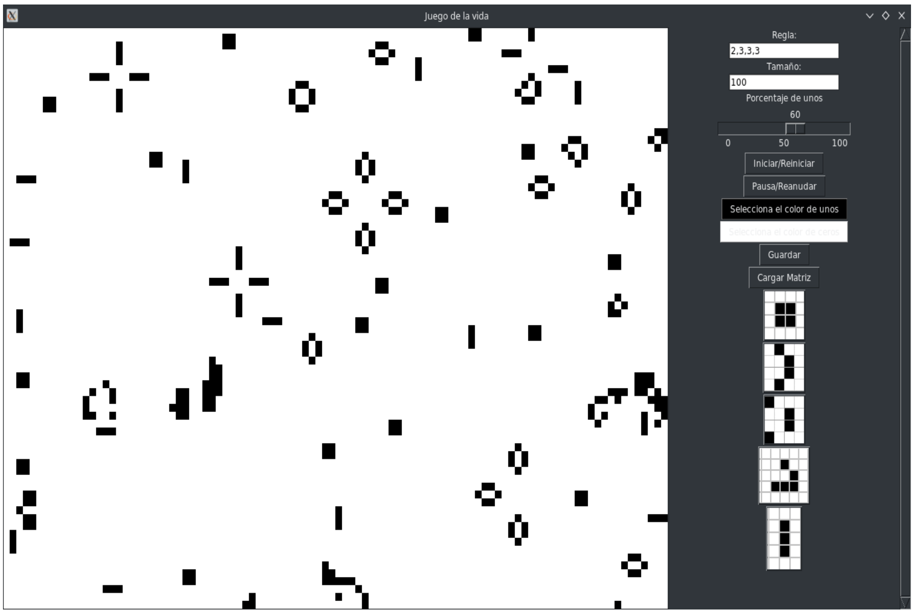

Figura 24: Regla de life con una probabilidad de unos de 60 %

Figura 25: Comportamiento de la población de la simulación anterior

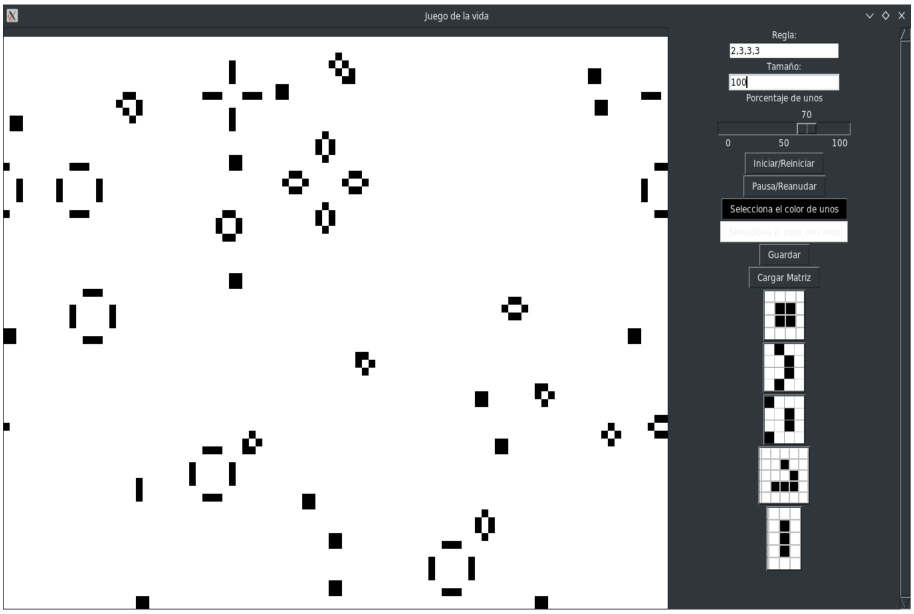

Figura 26: Regla de life con una probabilidad de unos de 70 %

Figura 27: Comportamiento de la población de la simulación anterior

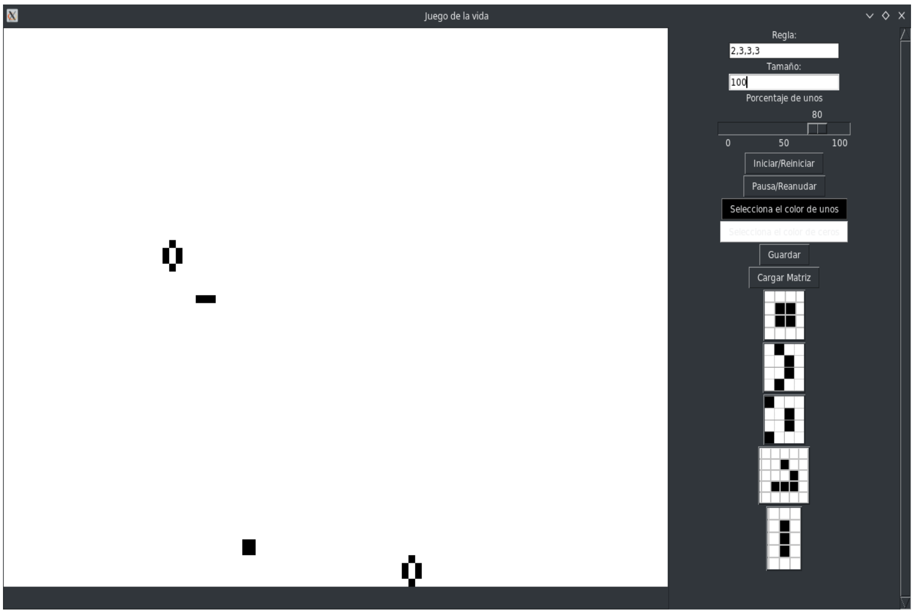

Figura 28: Regla de life con una probabilidad de unos de 80 %

Figura 29: Comportamiento de la población de la simulación anterior

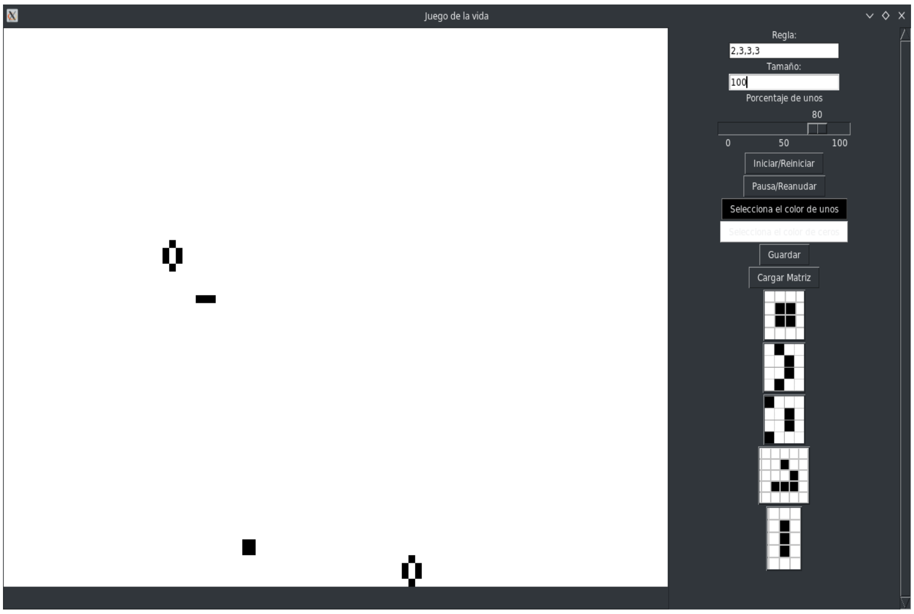

Figura 30: Regla de life con una probabilidad de unos de 90 %

Figura 31: Comportamiento de la población de la simulación anterior

En todas las simulaciones se encuentra un comportamiento similar en el cual la población inicial es muy alta y de manera brusca disminuye y apartir de ahi decrece de manera lenta.

En la figura 26 se aprecia asi que valor de densidad de la población tiende la regla de life, el valor es 0.0352, el resto de simulaciones tiende a un valor similar pero debido a que solo se trabajaron con 1000 generaciones no se logra apreciar por completo.

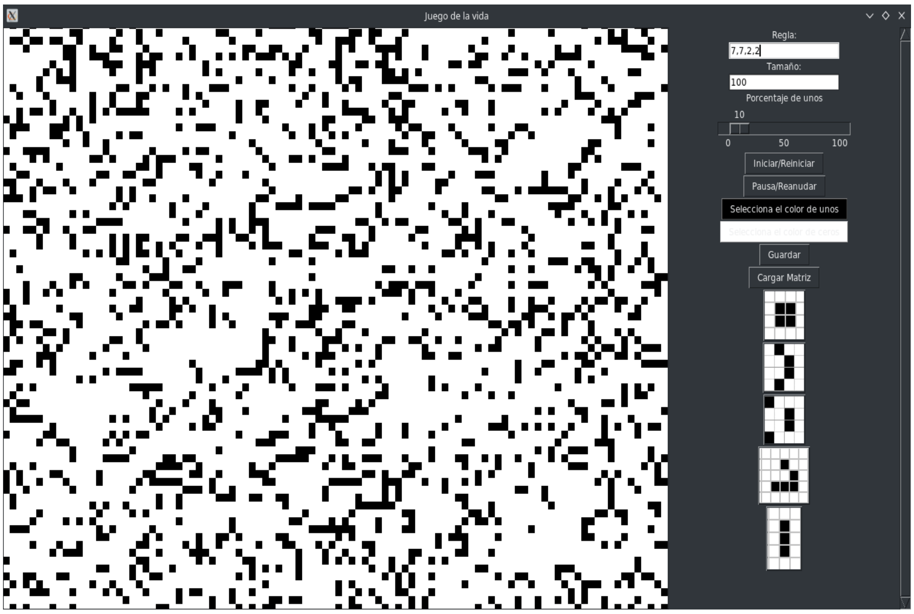

Figura 32: Regla de difusión con una probabilidad de unos de 10 %

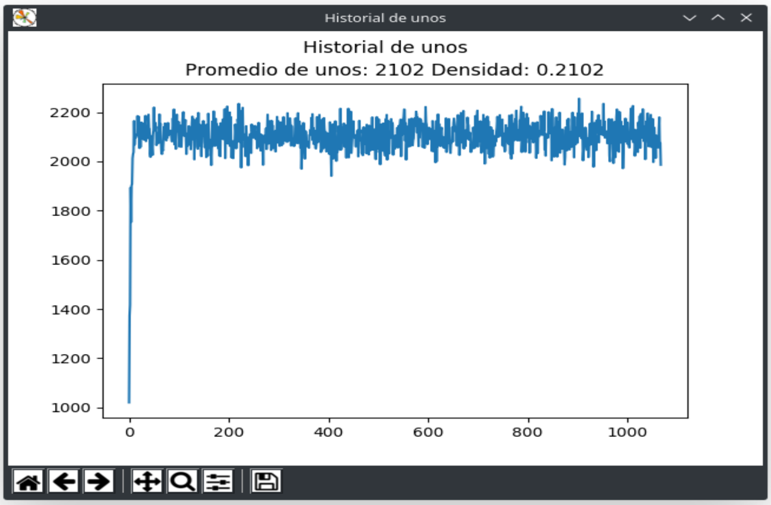

Figura 33: Comportamiento de la población de la simulación anterior

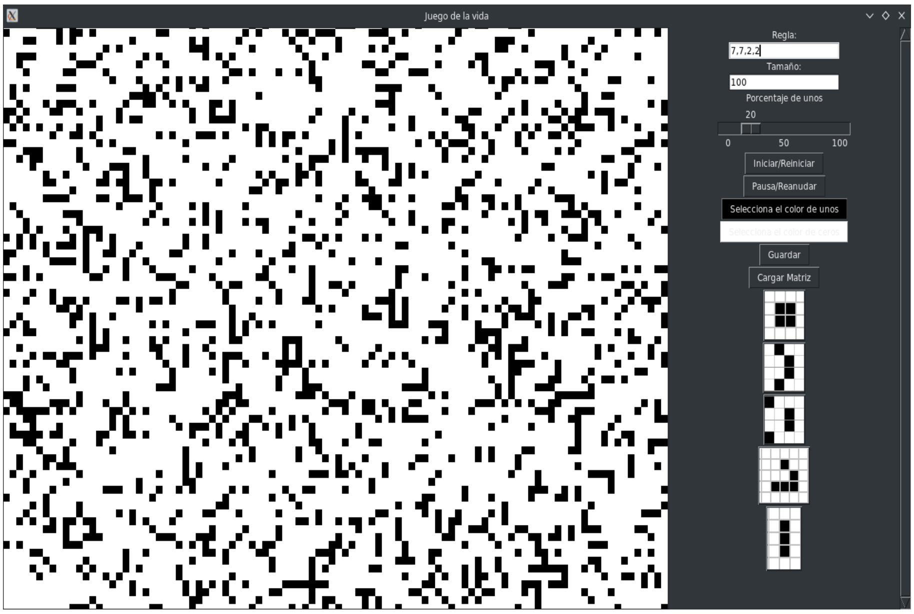

Figura 34: Regla de difusión con una probabilidad de unos de 20 %

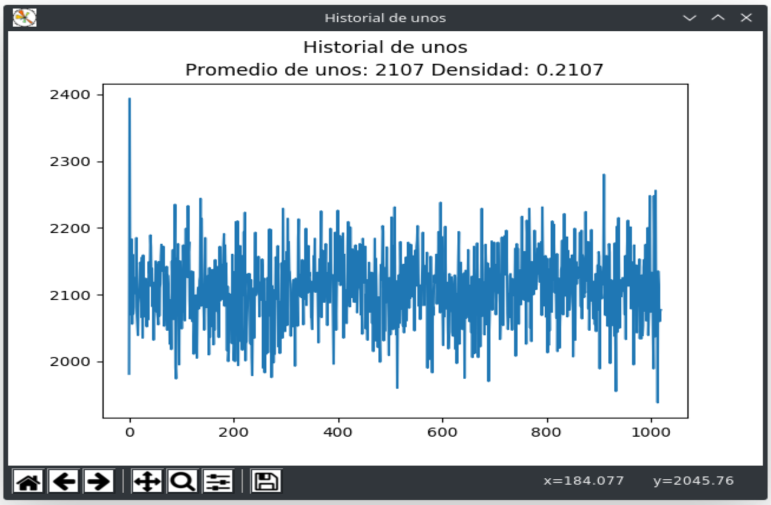

Figura 35: Comportamiento de la población de la simulación anterior

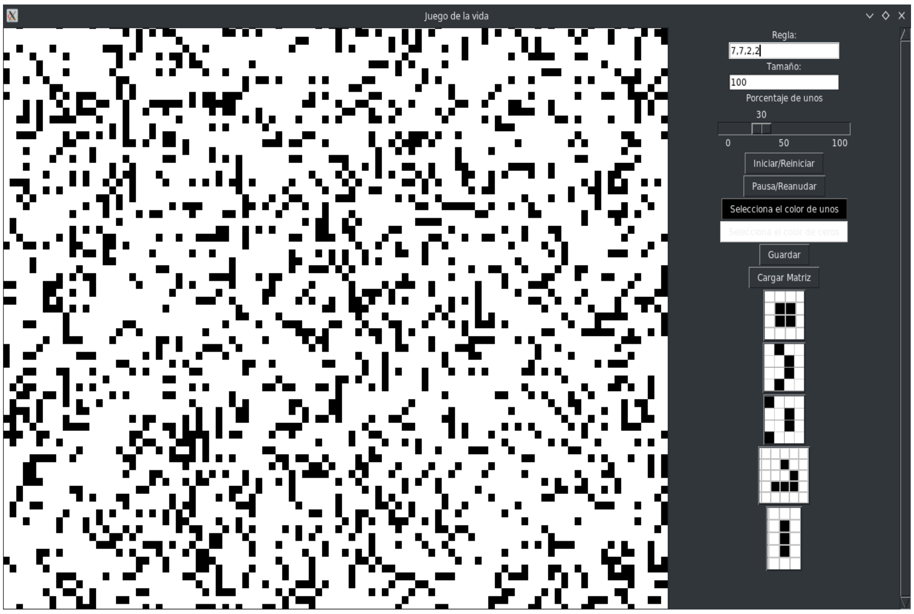

Figura 36: Regla de difusión con una probabilidad de unos de 30 %

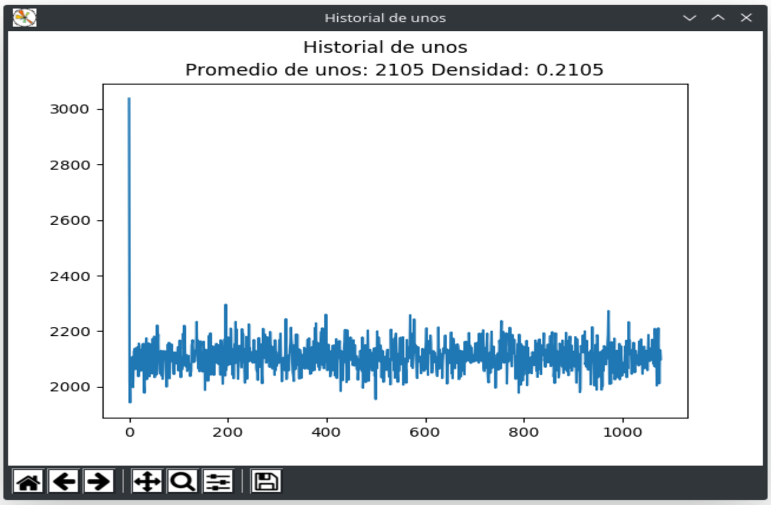

Figura 37: Comportamiento de la población de la simulación anterior

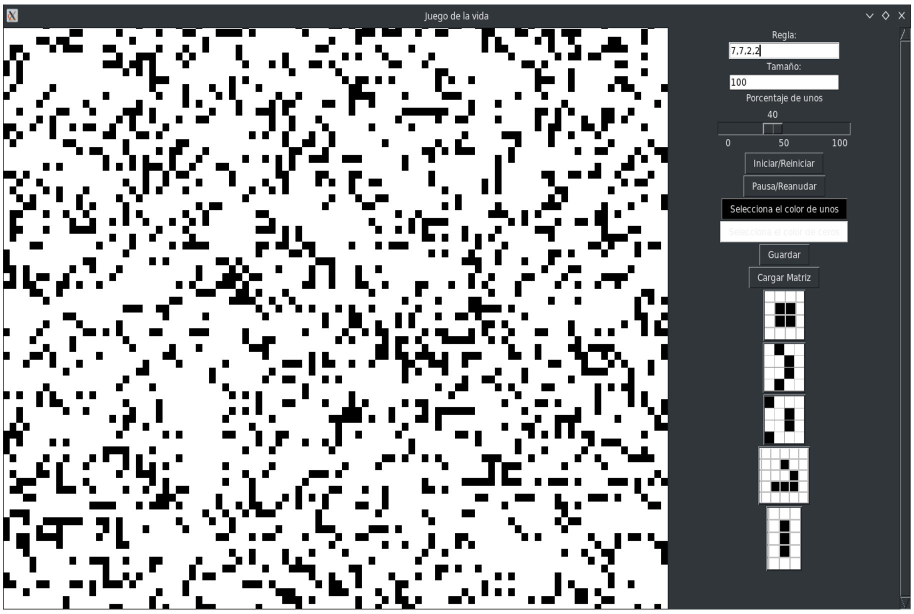

Figura 38: Regla de difusión con una probabilidad de unos de 40 %

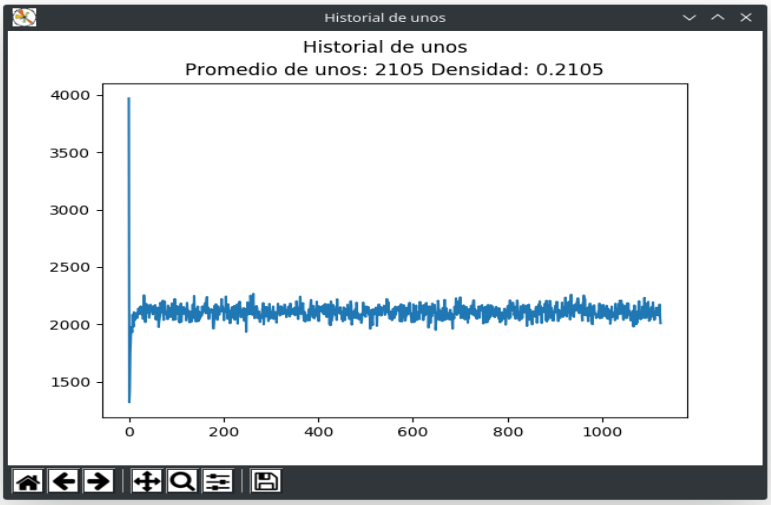

Figura 39: Comportamiento de la población de la simulación anterior

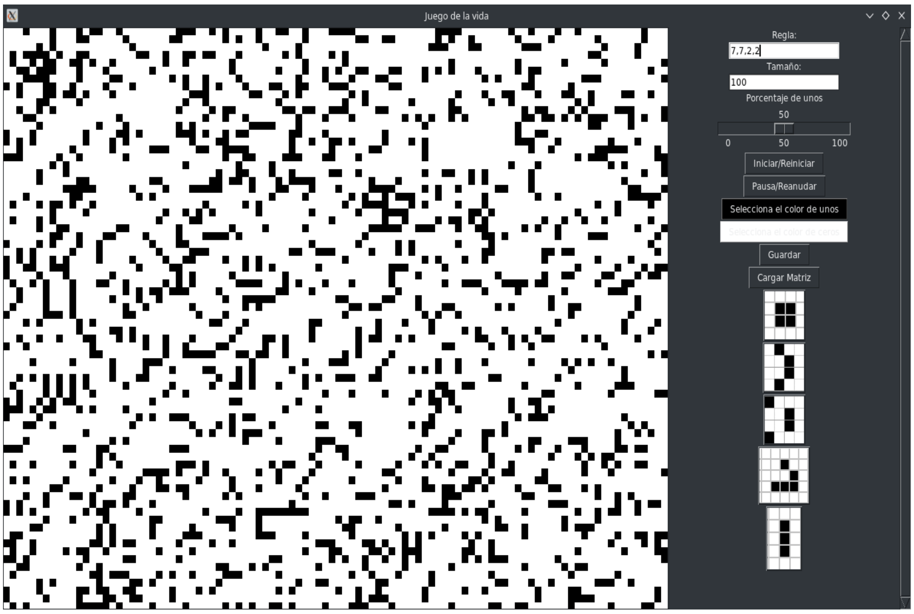

Figura 40: Regla de difusión con una probabilidad de unos de 50 %

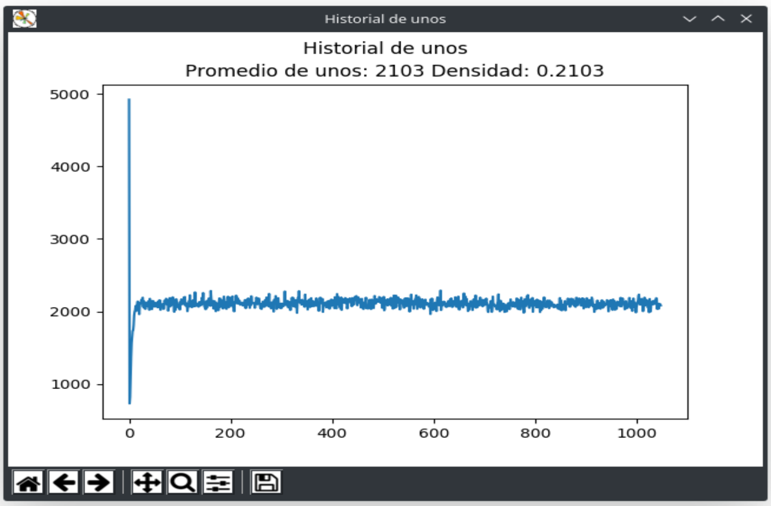

Figura 41: Comportamiento de la población de la simulación anterior

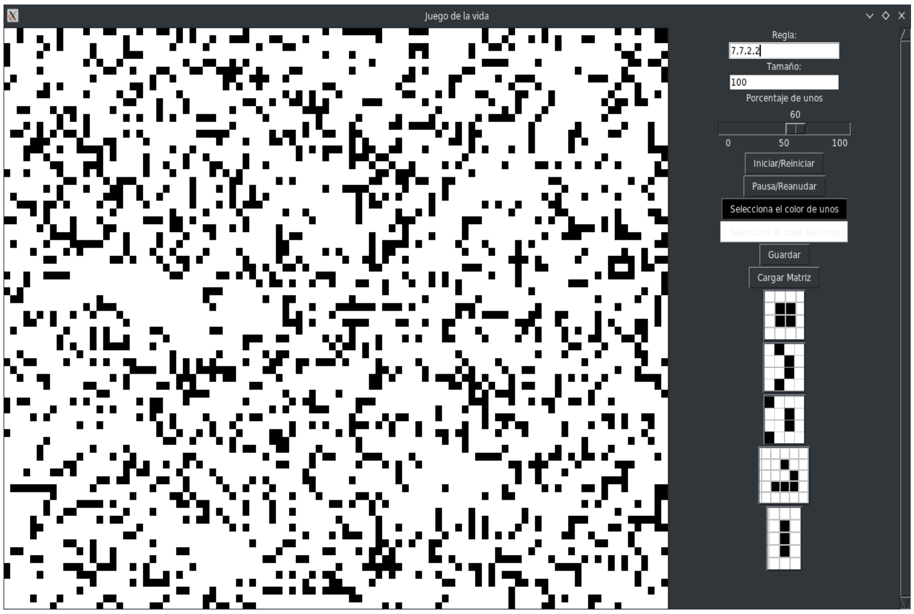

Figura 42: Regla de difusión con una probabilidad de unos de 60 %

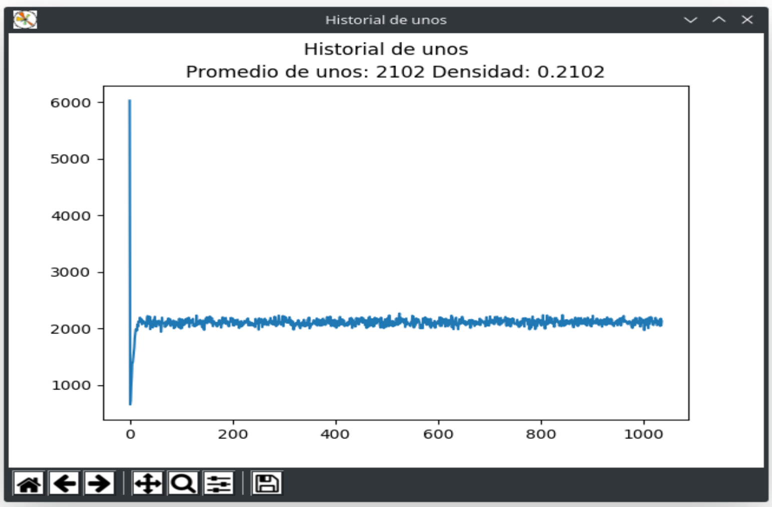

Figura 43: Comportamiento de la población de la simulación anterior

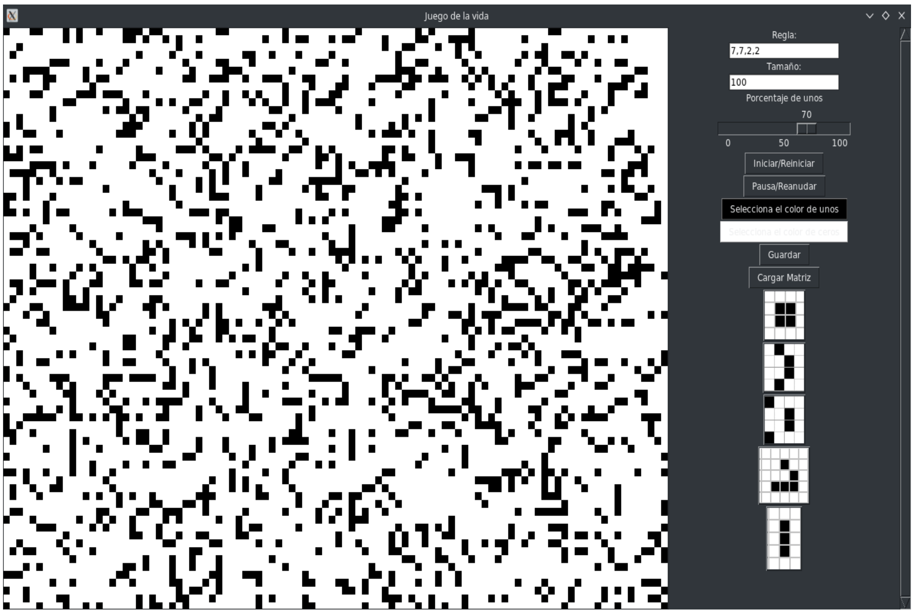

Figura 44: Regla de difusión con una probabilidad de unos de 70 %

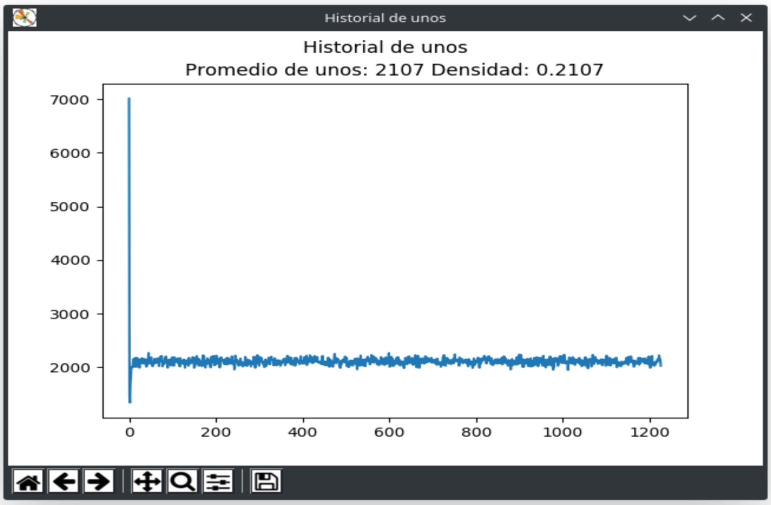

Figura 45: Comportamiento de la población de la simulación anterior

Figura 46: Regla de difusión con una probabilidad de unos de 80 %

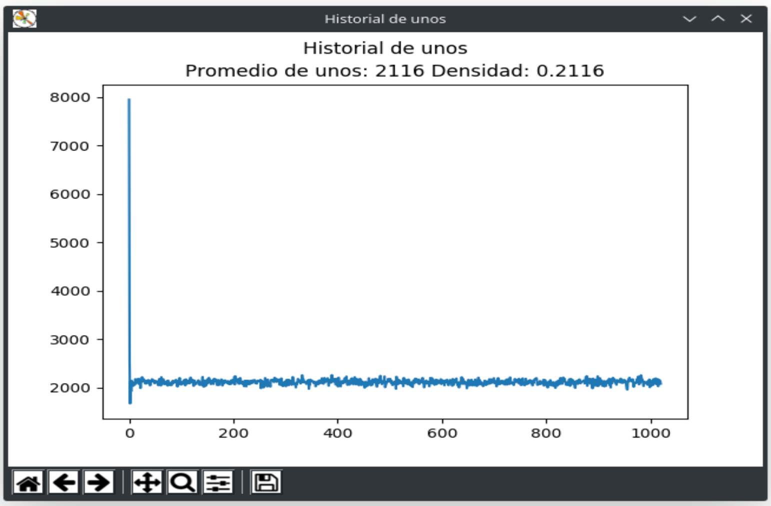

Figura 47: Comportamiento de la población de la simulación anterior

Figura 48: Regla de difusión con una probabilidad de unos de 90 %

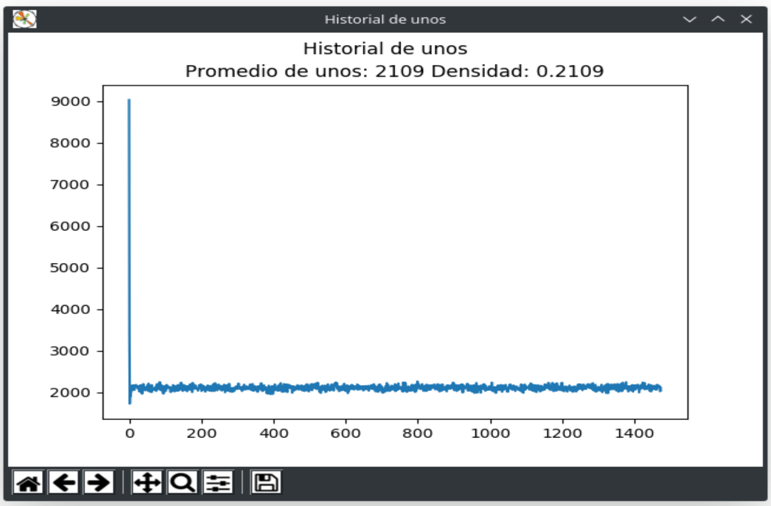

Figura 49: Comportamiento de la población de la simulación anterior

Para la regla de difusión se podria decir que el comportamiento de la población es más estable que la de life debido a que en todas las simulaciones se llego a la densidad de población de 0.21 y el comportamiento de la gráfica es igual en todas las probabilidades de uno que se probaron en donde la población oscila entre un limite mayor y uno menor cerca de los 2000 individuos vivos lo cual es un comportamiento interesante.

2.5. Conclusiones

Al hacer y probar este programa se pudieron solucionar dos cuestiones, la primera es si existe una configuración el la cual el numero de células no se termine, y la respuesta a esto es que exista un oscilador el cual cambia su posición a lo largo del tiempo, otra posible opción es que haya figuras como un bloque formado por 4 cuadros en el cual no hay cambios.

La siguiente cuestión es si existe una configuración en la cual la población crezca indefinidamente. Para lograr esto es indispensable tener un espacio que sea infinito en el cual se puedan propagar las células a través del tiempo, ya que si no se cuenta con esto en algún punto se tendrán tantos elementos vivos que empezaran a morir por sobre población.

Referencias

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introducción a La Teoría De Autómatas, Lenguajes Y Computación.* Addison-Wesley, 2007.

[2] Reyes Gómez, D., *Descripción y Aplicaciones de los Autómatas Celulares.* cinvestav, 2011.