Instituto Politecnico Nacional
Escuela Superior de Computo

Materia: Teoría de la Computación
Grupo: 5BM1

Profesor: Genaro Juarez Martínez
Periodo: 2024/02

Practica 02:

Automáta finito no determinista (AFN) detector de cadenas binarias con terminacion «01»

Realizado por:
Carrillo Barreiro José Emiliano

Escuela Superior de Computo
Fecha: 20 de marzo de 2024

Índice

Índice de Listados de C++

Índice de Listados de MATLAB

Índice de figuras

1 Resumen.

En la Práctica 02, nos adentramos en el mundo de los autómatas no deterministas para la detección de cadenas binarias con terminación *01*. Comenzamos implementando un autómata no determinista en un lenguaje de programación C++, para posteriormente hacer la animacion en Matlab. Este autómata se diseña para reconocer cadenas binarias que finalizan con *01*. El programa solicita al usuario que ingrese una cadena binaria y luego evalúa si cumple con la condición especificada por el autómata.

El proceso implica la definición de los estados del autómata, las transiciones entre estos estados y el manejo de la entrada del usuario para determinar si la cadena ingresada es aceptada o rechazada por el autómata. Este enfoque permite comprender cómo funcionan los autómatas no deterministas y cómo se pueden utilizar para resolver problemas específicos, en este caso, la detección de ciertos patrones en cadenas binarias.

2 Introducción.

Este informe detalla la implementación de un autómata no determinista para la detección de cadenas binarias con terminación *01*, utilizando los lenguajes de programación C++ y MATLAB. Nos centraremos en los principios fundamentales de la teoría de autómatas, explorando cómo estos conceptos se aplican en la creación de un autómata capaz de reconocer patrones específicos en cadenas binarias.

El objetivo principal es comprender cómo los autómatas no deterministas pueden ser diseñados e implementados en un entorno computacional utilizando C++ y MATLAB. Para ello, nos sumergiremos en conceptos como estados, transiciones, alfabetos y la estructura de las cadenas binarias. Esta comprensión teórica proporcionará la base necesaria para desarrollar un autómata capaz de identificar cadenas binarias que terminan con *01*.

Al combinar la teoría de autómatas con las capacidades de programación de C++ y MATLAB, no solo buscamos crear un autómata funcional, sino también profundizar en nuestra comprensión de los principios subyacentes que gobiernan su comportamiento. Este enfoque integral nos permitirá no solo resolver problemas específicos relacionados con la detección de patrones en cadenas binarias, sino también sentar las bases para abordar problemas más complejos en el ámbito de la informática teórica.

3 Marco Teorico.

3.1. Teoría de Automátas.

La teoría de autómatas es un campo fundamental en la ciencia de la computación que se ocupa del estudio de dispositivos abstractos de cálculo. Estos dispositivos, conocidos como *máquinas*, son modelos matemáticos que nos permiten entender y analizar el comportamiento de sistemas computacionales. Desde sus primeros pasos en la década de 1930 con los trabajos de Alan Turing, la teoría de autómatas ha evolucionado y se ha ramificado en diversos aspectos que abarcan desde autómatas finitos simples hasta conceptos más complejos como gramáticas formales y problemas computacionales.

El punto de partida de la teoría de autómatas se encuentra en los estudios de Turing sobre las *máquinas de Turing*, dispositivos abstractos capaces de realizar cualquier cálculo computacional. Turing propuso estas máquinas como un modelo universal de computación, estableciendo así los fundamentos teóricos de lo que hoy entendemos como computación. A partir de este trabajo pionero, otros investigadores comenzaron a explorar variantes más simples de las máquinas de Turing, dando lugar a los autómatas finitos.

Los autómatas finitos son modelos computacionales simples que representan sistemas con un número limitado de estados y una capacidad limitada de procesamiento. Estos dispositivos

se utilizan para modelar sistemas de control, reconocer patrones en cadenas de símbolos y resolver problemas de decisión. Además, los autómatas finitos están estrechamente relacionados con las gramáticas formales, ya que ambos se utilizan para describir y generar lenguajes formales.

La teoría de autómatas también abarca el estudio de problemas computacionales y la clasificación de su complejidad. Investigadores como Stephen Cook han contribuido significativamente al campo al desarrollar técnicas para clasificar los problemas en función de su dificultad computacional. Esta clasificación ha llevado a la identificación de problemas que pueden resolverse eficientemente y problemas que son inherentemente difíciles de resolver.[1]

3.2. Alfabetos.

Un alfabeto, representado convencionalmente por el símbolo Σ, es un conjunto finito y no vacío de símbolos. Estos símbolos pueden ser números, letras, caracteres especiales, o cualquier otro tipo de elemento que se utilice para formar cadenas de caracteres. Algunos ejemplos comunes de alfabetos incluyen:

1. $\Sigma = \{0, 1\}$: el alfabeto binario, utilizado en sistemas informáticos para representar datos de manera binaria.
2. $\Sigma = \{a, b, \ldots, z\}$: el conjunto de todas las letras minúsculas del alfabeto latino.
3. El conjunto de todos los caracteres ASCII o el conjunto de todos los caracteres ASCII imprimibles, utilizado en programación y comunicaciones para representar texto y caracteres especiales.[1]

3.3. Cadena de caracteres.

Una cadena de caracteres, también conocida como palabra en algunos contextos, es una secuencia finita de símbolos seleccionados de un alfabeto específico. Por ejemplo, la cadena «01101» es una cadena del alfabeto binario $\Sigma = \{0, 1\}$, mientras que «hola» es una cadena del alfabeto de letras minúsculas del alfabeto latino. Incluso la cadena vacía, representada por ε, es una cadena que puede construirse en cualquier alfabeto.[1]

3.4. Longitud de cadenas.

La longitud de una cadena se refiere al número de símbolos que contiene. Por ejemplo, la cadena «01101» tiene una longitud de 5, mientras que la cadena vacía tiene una longitud de 0. La notación estándar para indicar la longitud de una cadena w es $|w|$. Por lo tanto, $|01101| = 5$ y $|\varepsilon| = 0$.[1]

3.5. Autómatas Finitos No Deterministas (AFN)

Los autómatas finitos no deterministas (AFN) son una clase de autómatas que poseen la capacidad de estar en varios estados simultáneamente. Esta característica se manifiesta en la capacidad del autómata para “conjeturar” sobre su entrada en ciertos momentos durante su procesamiento. Por ejemplo, al buscar determinadas secuencias de caracteres, como palabras clave, dentro de una cadena de texto extensa, el AFN puede “conjeturar” el inicio de una de estas cadenas y emplear una serie de estados para verificar la presencia de la cadena, carácter por carácter.[1]

3.5.1. Aplicaciones de los AFN

Un ejemplo claro de la aplicación de los AFN se encuentra en la búsqueda de patrones en texto, como la búsqueda de palabras clave o la validación de formatos específicos. Además, los AFN se utilizan en el análisis léxico de compiladores, donde se emplean para reconocer tokens en un programa fuente.

4 Desarrollo de la Practica.

4.1. información del sistema.

La siguiente información fue extraida gracias al comando `systeminfo` en `cmd`:

```
Nombre de host:                            CARBAJE
Nombre del sistema operativo:              Microsoft Windows 11 Home
Version del sistema operativo:             10.0.22631 N/D Compilacion 22631
Fabricante del sistema operativo:          Microsoft Corporation
Configuracion del sistema operativo:       Estacion de trabajo independiente
Tipo de compilacion del sistema operativo: Multiprocessor Free
Propiedad de:                              emi.cruzazul@hotmail.com
Organizacion registrada:
Fecha de instalacion original:             04/03/2024, 8:09:03
Tiempo de arranque del sistema:            12/03/2024, 9:41:56
Fabricante del sistema:                    GIGABYTE
Modelo el sistema:                         G5 KF5
Tipo de sistema:                           x64-based PC
Procesador(es):                            1 Procesadores instalados.
                                             [01]: Intel64 Family 6 Model 186
                                                 Stepping 2 GenuineIntel ~2400 Mhz
Version del BIOS:                          INSYDE Corp. FD06, 03/11/2023
Directorio de Windows:                     C:\Windows
Directorio de sistema:                     C:\Windows\system32
Dispositivo de arranque:                   \Device\HarddiskVolume1
Configuracion regional del sistema:        es;Espanol (internacional)
Idioma de entrada:                         en-us;Ingles (Estados Unidos)
Zona horaria:                              (UTC-06:00) Guadalajara, Ciudad de
    Mexico, Monterrey
Cantidad total de memoria fisica:          16.088 MB
Memoria fisica disponible:                 7.991 MB
Memoria virtual: tamano maximo:            65.240 MB
Memoria virtual: disponible:               53.369 MB
Memoria virtual: en uso:                   11.871 MB
Ubicacion(es) de archivo de paginacion:    C:\pagefile.sys
Dominio:                                   WORKGROUP
Servidor de inicio de sesion:              \\CARBAJE
Revision(es):                              5 revision(es) instaladas.
                                             [01]: KB5034467
                                             [02]: KB5027397
                                             [03]: KB5036212
                                             [04]: KB5034848
                                             [05]: KB5035226
Tarjeta(s) de red:                         3 Tarjetas de interfaz de red
    instaladas.
                                             [01]: Realtek PCIe GbE Family
                                                 Controller
                                                     Nombre de conexion: Ethernet
                                                     Estado:             Medios
                                                         desconectados
                                             [02]: Bluetooth Device (Personal Area
                                                 Network)
                                                     Nombre de conexion: Conexion
                                                         de red Bluetooth
                                                     Estado:             Medios
                                                         desconectados
                                             [03]: Intel(R) Wi-Fi 6E AX211 160MHz
                                                     Nombre de conexion: Wi-Fi
                                                     DHCP habilitado:    Si
```

4.2. Lenguajes usados.

la elección de C++ se basa en su eficiencia y control sobre los recursos, ideal para la implementación eficiente del *AFN*. MATLAB se selecciona por su facilidad de prototipado y potentes herramientas de cálculo numérico y visualización, facilitando el análisis y la comprensión de los resultados. Juntos, estos lenguajes ofrecen una solución integral para diseñar, implementar y analizar el *AFN* de manera eficaz y precisa.

4.2.1. C++

La elección de C++ para la elaboración de la lógica del programa se basa en su eficiencia, control sobre los recursos, flexibilidad y compatibilidad. Estas características hacen que sea una opción sólida y adecuada para implementar un *AFN* de manera óptima y eficiente. A continuación se enlista a detenimiento las ventajas mencionadas:

1. **Eficiencia y rendimiento:** C++ es conocido por ser un lenguaje de programación de alto rendimiento. Esto significa que los programas escritos en C++ tienden a ejecutarse más rápido y a consumir menos recursos que aquellos escritos en lenguajes de más alto nivel, como MATLAB. Dado que estamos trabajando en la implementación de un *AFN*, donde la eficiencia es crucial, el uso de C++ puede garantizar un rendimiento óptimo.
2. **Control sobre los recursos:** C++ proporciona un alto grado de control sobre la gestión de memoria y otros recursos del sistema. Esto es especialmente importante en aplicaciones donde se manejan grandes volúmenes de datos o se realizan operaciones intensivas en términos de recursos. En el contexto del *AFN*, donde podríamos estar trabajando con grandes conjuntos de datos, este control adicional puede ser beneficioso.
3. **Flexibilidad y versatilidad:** C++ es un lenguaje multiparadigma que permite programar en diferentes estilos, como programación orientada a objetos, programación genérica y programación procedural. Esta versatilidad ofrece la posibilidad de diseñar y estructurar el código de manera óptima según las necesidades específicas del problema. En el caso de la implementación de un *AFN*, esta flexibilidad puede ser útil para organizar y modularizar el código de manera eficiente.
4. **Compatibilidad y portabilidad:** C++ es un lenguaje ampliamente utilizado y está disponible en una amplia variedad de plataformas y sistemas operativos. Esto garantiza que el código desarrollado en C++ pueda ejecutarse en diferentes entornos sin mayores modificaciones, lo que aumenta la portabilidad y la interoperabilidad de la solución.

4.2.2. Matlab

La elección de MATLAB para la elaboración de la graficación del programa se basa en su facilidad de prototipado, sus capacidades avanzadas de cálculo numérico y matemático, sus herramientas de visualización y su integración con otras herramientas. Estas características hacen que sea una opción sólida y eficaz para la representación visual y el análisis de los resultados del *AFN*. A continuación se enlista a detenimiento las ventajas mencionadas:

1. **Facilidad de prototipado y desarrollo rápido:** MATLAB es conocido por su capacidad para el prototipado rápido y el desarrollo eficiente de algoritmos. Proporciona una amplia gama de funciones y herramientas integradas que facilitan la implementación de algoritmos complejos con un código más compacto y legible. Esto es especialmente útil en el contexto de la práctica, donde la experimentación y la iteración rápida son fundamentales para el diseño y la optimización del generador de potencias.
2. **Amplio conjunto de herramientas para cálculos numéricos y matemáticos:** MATLAB ofrece una amplia gama de funciones y herramientas especializadas para realizar cálculos numéricos y matemáticos avanzados. Esto incluye funciones para operaciones con matrices, álgebra lineal, transformadas, optimización y simulación, entre otros. Estas herramientas son fundamentales para la manipulación y el procesamiento de datos en el contexto de la generación de potencias.

3. **Gráficos y visualización avanzados:** MATLAB cuenta con potentes capacidades de gráficos y visualización que facilitan la representación visual de datos y resultados. Esto es especialmente importante en el contexto de la práctica, donde se desea visualizar y analizar los resultados del *AFN*, la animacion del automata de manera grafica. Las capacidades de graficación de MATLAB permiten crear gráficos personalizados y visualizaciones interactivas para explorar y comprender mejor los datos generados por el programa.
4. **Integración con herramientas adicionales:** MATLAB se integra bien con otras herramientas y entornos de desarrollo, lo que facilita la incorporación de funcionalidades adicionales o la conexión con sistemas externos si es necesario. Esto puede ser útil para ampliar la funcionalidad del *AFN* e integrarlo en un flujo de trabajo más amplio, como es la practica final.

4.3. Libreras usadas

Las siguientes bibliotecas proporcionan las herramientas necesarias para implementar eficientemente un *AFN* en C++, abordando aspectos clave como la manipulación de cadenas, la lectura y escritura de archivos, el rendimiento y la optimización del tiempo de ejecución, la manipulación de datos en estructuras dinámicas y el control del formato de salida.

1. **iostream**:
   - Esencial para interactuar con el usuario a través de la entrada y salida estándar, lo que facilita la comunicación con el programa.
2. **string**:
   - Permite la manipulación eficiente de cadenas de caracteres, lo que es fundamental para procesar y generar secuencias de texto en el *AFN*.
3. **fstream**:
   - Necesaria para leer y escribir archivos, lo que facilita la entrada y salida de datos y la posibilidad de almacenar resultados en archivos para su posterior análisis.
4. **cstdlib**:
   - Proporciona funciones de utilidad para realizar operaciones comunes de bajo nivel, como conversiones de tipos y gestión de memoria, que pueden ser útiles en la implementación del *AFN*.
5. **ctime**:
   - Permite el manejo de operaciones relacionadas con el tiempo, como la obtención de la hora actual del sistema, lo que puede ser útil para realizar mediciones de tiempo y optimizar el rendimiento del *AFN*.
6. **vector**:
   - Proporciona una estructura de datos dinámica que puede cambiar de tamaño automáticamente, lo que es útil para almacenar y manipular conjuntos de datos variables en la implementación del *AFN*.
7. **chrono**:
   - Permite medir el tiempo de ejecución de partes específicas del programa, lo que es útil para evaluar el rendimiento del *AFN* y optimizar su eficiencia.
8. **windows.h**:
   - Proporciona acceso a funciones del sistema operativo Windows y manipulación de recursos del sistema, lo que puede ser útil si se desarrolla el *AFN* específicamente para la plataforma Windows.

4.4. Implementación

4.4.1. *PRACTICA03.h*

Este archivo, tambien llamado biblioteca a partir de ahora, tiene solamente una bolque de codigo donde fragmento de código establece las bases para el programa, incluyendo las bibliotecas necesarias y la declaración de las funciones que se utilizarán. La implementación específica de cada función y la lógica del programa se encuentra en otros lugares del código. A continuacion se deja el bloque de codigo de la biblioteca:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <time.h>
#include <chrono>
#include <windows.h>

// Definicion de la estructura de los Estados
struct Estado{
    int id;
    std::vector<std::pair<char, int>> transiciones; // Transiciones: (simbolo, estado_destino)
};

// Definicion de la estructura del automata
struct Automata {
    std::vector<Estado> grafo;
    int numEstados;
    int estadoFinal;
};

//Funcion para realizar la animacion
void animarP02(int&);

//Funcion para solicitar al usuario la animacion
void opcionAnimarP02(int&);

// Funcion para encontrar todas las rutas al estado final usando DFS
std::vector<std::vector<int>> encontrarRutas(const Automata& automata, const std::string& cadena, int i, int posicion, std::vector<int> ruta_actual);

// Funcion para guardar las rutas en un archivo de texto
void guardarRutasEnArchivo(const std::vector<std::vector<int>>&, const std::string&, std::string);

//Funcion para ejecutar la secunecia de funciones del programa
void ejecucion(std::string, Automata);

//Funcion para generar una cadena binaria aleatoria de tamano aleatorio
std::string generarCadena();

//Funcion para solicitar al usuario la cadena
std::string solicitarCadena();

//Funcion para ingresar la opcion:
int ingresarOpcion();

//Funcion para generar el automata y definir sus trancisiones
Automata crearAutomata();

//Funcion para crear los estados del Automata
std::vector<Estado> crearEstado();

//Funcion menu
void menu(int&);

//Funcion main
int main();
```

Listing 1: PRACTICA01.h

4.4.2. *mainP02.cpp*

En este archivo existe únicamente una función. El siguiente fragmento de código inicia la ejecución del programa, llama a una función `menu ()` para presentar al usuario un menú de opciones y devuelve un valor de error al sistema operativo al finalizar la ejecución. La lógica específica del programa, incluyendo la implementación de la función `menu ()`, se encuentra en otros archivos que están incluidos a través del archivo de encabezado `PRACTICA02.h`. A continuación se muestra la función `main ()` en cuestión:

```
#include "PRACTICA02.h"

int main(){
    int error = 0;
    menu(error);
    return error;
}
```

Listing 2: *mainP02.cpp* — `main()`

4.4.3. *menuP02.cpp*

El código en cuestión presenta una implementación que ofrece un menú de opciones al usuario, controlado por la función denominada `menu`. Este menú facilita al usuario la interacción con el programa al proporcionar una serie de opciones claramente definidas.

El programa inicia con un mensaje de bienvenida, seguido por la presentación de las opciones disponibles. Estas opciones incluyen la posibilidad de ingresar una cadena, generar una cadena aleatoria, ejecutar un programa, o finalizar la ejecución del mismo.

La función `menu` estructura un bucle que permite al usuario seleccionar repetidamente las opciones proporcionadas hasta que decide salir del programa. Cada opción elegida por el usuario se maneja mediante un conjunto de instrucciones definidas en un bloque `switch`, lo que permite realizar acciones específicas dependiendo de la elección realizada.

De particular interés es la opción 3, donde se ejecuta un programa específico después de verificar si se ha ingresado una cadena previamente. Esta opción proporciona una funcionalidad adicional al programa, permitiendo su ejecución bajo condiciones definidas.

```
void menu(int& error) {
    bool repetir = true;
    int opcion = 1;
    std::string cadena = "";
    Automata automata = crearAutomata();


    while(repetir){
        std::cout<<"BIENVENIDO A LA PRACTICA 01."<<std::endl;
        std::cout<<"OPCIONES A REALIZAR"<<std::endl;
        std::cout<<"1- Ingresar cadena."<<std::endl;
        std::cout<<"2- Generar cadena random."<<std::endl;
        std::cout<<"3- Ejecutar Programa."<<std::endl;
        std::cout<<"0- Salir del Programa."<<std::endl;

        opcion = ingresarOpcion();

        switch (opcion) {
        case 0:
            std::cout<<"Gracias Por usar el programa..."<<std::endl;
            repetir = false;
            break;
        case 1:
            cadena = solicitarCadena();
            break;
        case 2:
            cadena = generarCadena();
            break;
        case 3:
            if (!cadena.empty()) {
```

```
31                  ejecucion(cadena, automata);
32                  opcionAnimarP02(error);
33              }else {
34                  std::cout<<"Favor de ingresar (generar) una cadena binaria al
                        programa..."<<std::endl;
35              }
36              break;
37          default:
38              std::cout<<"Favor de indicar una opcion valida...."<<std::endl;
39          }
40      }
41  }
```

Listing 3: Archivo: *menuP02.cpp* — Funcion: `menu ()`

Esta función `ingresarOpcion` proporciona una forma de interactuar con el usuario, permitiéndole seleccionar una opción deseada del menú mediante la entrada de un número entero en la consola. El valor ingresado por el usuario se utiliza posteriormente en otras partes del programa para determinar la acción a realizar. A continuación se anexa el bloqued de codigo que alude a la función mencionada:

```
1  int ingresarOpcion(){
2      int opcion= 5;
3      std::cout << "Ingrese la opcion deseada: ";
4      std::cin >> opcion;
5
6      return opcion;
7  }
```

Listing 4: Archivo: *menuP02.cpp* — Funcion: `ingresarOpcion ()`

La función `ejecucion` es fundamental en el programa, ya que inicia un cronómetro para medir el tiempo de ejecución, utiliza una función llamada `encontrarRutas` para generar rutas posibles al estado final para una cadena dada y un autómata dado, detiene el cronómetro después de generar las rutas, calcula el tiempo transcurrido en milisegundos, convierte este tiempo en minutos, segundos y milisegundos, lo muestra en la salida estándar y guarda las rutas generadas en un archivo llamado

```
1  void ejecucion(std::string cadena, Automata automata){
2      // Iniciar el cronometro
3      auto start = std::chrono::high_resolution_clock::now();
4
5      //Generar el vector de rutas posibles al estado final para la cadena
6      auto rutas = encontrarRutas(automata, cadena, 0, 0, {});
7
8      // Detener el cronometro
9      auto stop = std::chrono::high_resolution_clock::now();
10
11     //Calcular el tiempo transcurrido
12     auto duration =
           std::chrono::duration_cast<std::chrono::milliseconds>(stop-start);
13
14     // Calcular minutos, segundos y milisegundos
15     auto milliseconds = duration.count() % 1000;
16     auto seconds = (duration.count() / 1000) % 60;
17     auto minutes = (duration.count() / (1000 * 60)) % 60;
18
19     //Mostrar iteraciones y tiempo cronometro
20     std::cout << "Tiempo transcurrido: " << duration.count() << " milisegundos" <<
           std::endl;
21
22     guardarRutasEnArchivo(rutas, "rutas.txt", cadena);
23 }
```

Listing 5: Archivo: *menuP02.cpp* — Funcion: *ejecucion()*

Esta función `medirTiempo` calcula el tiempo transcurrido entre un punto de inicio y un punto de finalización utilizando la librería `<chrono>` de C++. Aquí está el bloque de codigo de la función:

```
1      void medirTiempo(std::chrono::high_resolution_clock::time_point start) {
2          auto stop = std::chrono::high_resolution_clock::now();
```

```
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
        auto milliseconds = duration.count() % 1000;
        auto seconds = (duration.count() / 1000) % 60;
        auto minutes = (duration.count() / (1000 * 60)) % 60;

        std::cout << "Tiempo transcurrido: " << minutes << " minutos, " << seconds << " segundos, " << milliseconds << " milisegundos" << std::endl;
    }
```

Listing 6: Archivo: *menuP02.cpp* — Funcion *medirTiempo()*

4.4.4. solicitudesP02.cpp

El archivo solicitudesP02.cpp presenta una serie de funciones diseñadas para abordar tareas específicas en el contexto de la manipulación de cadenas binarias. Estas funciones reflejan un enfoque hacia la interacción con el usuario y la generación de datos de forma dinámica. Desde la solicitud de cadenas binarias válidas hasta la generación de cadenas aleatorias, el archivo busca facilitar la manipulación y procesamiento de este tipo de datos. Asimismo, la inclusión de una función para guardar rutas en archivos subraya la importancia de almacenar resultados y facilitar su posterior análisis o visualización, brindando así una solución integral para el manejo de información en el contexto de las operaciones con cadenas binarias.

Esta función solicitarCadena solicita al usuario que ingrese una cadena binaria válida, verificando que esta esté compuesta únicamente por caracteres '0' y '1'. Utiliza un bucle do-while para continuar solicitando al usuario la cadena hasta que esta sea válida. Dentro del bucle, se muestra un mensaje pidiendo al usuario que ingrese la cadena binaria, luego se verifica cada carácter de la cadena ingresada. Si se encuentra algún carácter que no sea '0' o '1', se establece la variable de verificación en falso, se muestra un mensaje de error y se rompe el bucle. Finalmente, cuando se obtiene una cadena válida, se retorna dicha cadena.

```
std::string solicitarCadena() {
    std::string cadena;
    bool verificacion;

    do{
        std::cout << "Ingrese una cadena binaria): ";
        std::cin >> cadena;

        for (char c : cadena)
        {
            verificacion = true;
            if(c < '0' || c > '1'){
                verificacion = false;

                std::cout << "La cadena ingresada esta compuesta de otros caracteres ademas de 0s y 1s."<<std::endl;
                std::cout << "Intentelo nuevamente."<<std::endl;
                break;
            }
        }

    }while(!verificacion);

    return cadena;
}
```

Listing 7: Archivo: *solicitudesP02.cpp* — Funcion `ingresarPotencia()`

Esta función generarCadena crea y devuelve una cadena binaria aleatoria. Primero, se utiliza la función `srand` para inicializar la semilla de generación de números aleatorios, utilizando el tiempo actual como semilla. Luego, se genera aleatoriamente una longitud para la cadena entre 1 y 100 caracteres. Se crea una cadena inicializada con '0' de longitud igual a la longitud aleatoria generada. Posteriormente, se itera sobre cada posición de la cadena y se asigna un valor aleatorio ('0' o '1') a cada carácter utilizando la función `rand`. Finalmente, se imprime en la consola la longitud y la cadena generada, y se retorna esta cadena.

A continuacion se deja el bloque de codigo en cuestión:

```
std::string generarCadena() {
    srand(time(NULL));
    int longitud = rand() % (100 - 1 + 1) + 1;
    std::string cadena(longitud,'0');

    for(int i = 0 ; i < cadena.size() ; i++){
        cadena[i] = rand() % ('1' - '0' +1) + '0';
    }
    std::cout<<"La cadena generade fue de longitud: "<<longitud<<" denotada como:
        "<<cadena<<std::endl;

    return cadena;
}
```

Listing 8: Archivo: *solicitudesP02.cpp* — Funcion `randomPotencia()`

La función guardarRutasEnArchivo se encarga de guardar las rutas encontradas en dos archivos: uno de texto y otro CSV. Recibe como argumentos un vector de vectores de enteros que representan las rutas, el nombre del archivo de texto donde se guardarán las rutas, y una cadena binaria. La función primero abre los archivos de texto y CSV para escritura. Si ambos archivos se abren correctamente, verifica si el vector de rutas no está vacío. Si hay rutas disponibles, escribe en el archivo de texto un encabezado indicando la cadena binaria y luego itera sobre cada ruta, escribiendo cada una en el archivo de texto. Además, para el archivo CSV, escribe la cadena binaria junto con las rutas en un formato específico. Si no se encontraron rutas, se imprime un mensaje indicando que no se encontraron rutas en el archivo de texto. Finalmente, independientemente del resultado, se cierran ambos archivos. Si no se pueden abrir los archivos para escritura, se muestra un mensaje de error.

```
void guardarRutasEnArchivo(const std::vector<std::vector<int>>& rutas, const
    std::string& nombreArchivo, std::string cadena) {
    std::ofstream archivo(nombreArchivo);
    std::ofstream archivo2("rutaGraficar.csv");
    if (archivo.is_open() && archivo2.is_open()) {
        if (!rutas.empty()) {
            archivo << "Se encontraron las siguientes rutas para la cadea
                <<"<<cadena<<">>:\n";
            for(auto& ruta : rutas) {
                archivo << "Ruta: ";
                for(int i = ruta.size() - 1; i >= 0; --i) {
                    archivo << 'Q' << ruta[i];
                    archivo2 << cadena[ruta.size()-i-1] << ',' <<
                        ruta[i]<<std::endl;
                    if(i != 0) archivo << " -> ";
                }
                archivo << "\n";
            }
            std::cout << "Se han guardado las rutas en el archivo 'rutas.txt'.\n";
        } else {
            std::cout << "No se encontraron rutas."<<std::endl;
            archivo << "No se encontraron rutas.\n";
        }
        archivo.close();
        archivo2.close();
    } else {
        std::cout << "No se pudo abrir el archivo para escribir las rutas.\n";
    }

}
}
```

Listing 9: Archivo: *solicitudesP02.cpp* — Funcion `guardarRutasEnArchivo`

La funcion opcionAnimarP02 permite al usuario decidir si desea graficar el resultado obtenido. Primero, inicializa una variable *opcion* con el valor predeterminado *Y* (sí). Luego, muestra un mensaje preguntando al usuario si desea graficar el resultado, permitiendo que elija entre *Y* o *N*. Dependiendo de la opción seleccionada (*Y* o *y* para sí), se llama a la función animarP02. Esta función no está definida en el código proporcionado y probablemente esté relacionada con la animación del resultado, lo que implica que la

función opcionAnimarP02 facilita la interacción del usuario para iniciar dicha animación si así lo desea. Se deja el bloque de codigo que engloba a la función en cuestion:

```
void opcionAnimarP02(int& error) {
    char opcion = 'Y';
    std::cout<<"Desea Graficar el resultado(Y[y]/N[n]): ";
    std::cin>>opcion;
    if (opcion=='Y' || opcion=='y')
        animarP02(error);
}
```

Listing 10: Archivo: *solicitudesP02.cpp* — Funcion `opcionAnimarP02`

4.4.5. *Generacion de la estructura: Automata*

Este fragmento de código define una función llamada crearAutomata, la cual genera un autómata y define sus transiciones. En primer lugar, se crea un objeto de tipo *Automata* llamado *nuevo*. Luego, se asigna al atributo *grafo* de este objeto el resultado de llamar a la función crearAutomata, que se encarga de generar y definir los estados del autómata. Posteriormente, se establece el número de estados del autómata como 3 y se define el estado final como el estado número 2. Finalmente, se retorna el objeto *nuevo*, que representa el autómata creado.

```
//Funcion para generar el automata y definir sus trancisiones
Automata crearAutomata\@() {
    Automata nuevo;
    nuevo.grafo = crearEstado\@();
    nuevo.numEstados = 3;
    nuevo.estadoFinal = 2;

    return nuevo;
}
```

Listing 11: Archivo: *automataP02.cpp* — `crearAutomata`

Claro, la función crearEstado genera y define los estados de un autómata. Se crean tres estados: *estadoQ0, estadoQ1 y estadoQ2*, cada uno con sus transiciones definidas. Estos estados se almacenan en un vector llamado `grafo`, que representa el grafo del autómata, y luego se retorna este vector.

```
//Funcion para generar el automata y definir sus trancisiones
std::vector<Estado> crearEstado(){
    Estado estadoQ0 = {0, {{'0', 0}, {'0', 1}, {'1', 0}}};
    Estado estadoQ1 = {1, {{'1', 2}}};
    Estado estadoQ2 = {2, {}};

    std::vector<Estado> grafo = {estadoQ0, estadoQ1, estadoQ2};

    return grafo;
}
```

Listing 12: Archivo: *automataP02.cpp* — `crearEstado`

4.4.6. *Logica del problema: rutasP02.cpp*

El archivo *rutasP02.cpp* incluye unicamente a la función, llamada encontrarRutas, encargada de buscar todas las posibles rutas en un autómata dada una cadena de entrada. Recibe como argumentos el autómata (*Automata*), la cadena de entrada (*cadena*), un índice de iteración (i), la posición actual en el autómata (*posicion*), y un vector que representa la ruta actual (*ruta_actual*).

El algoritmo comienza verificando si se ha alcanzado el final de la cadena de entrada (*cadena*) mediante la comparación del índice i con el tamaño de la cadena. Si se ha llegado al final de la cadena y la posición actual en el autómata es el estado final, se añade la ruta actual al vector de rutas (*rutas*).

Después, itera sobre las transiciones del estado actual en el autómata. Si el símbolo en la posición actual de la cadena coincide con el símbolo de alguna transición, se realiza una llamada recursiva a la función

para explorar la siguiente transición. Se concatenan las nuevas rutas encontradas con la ruta actual y se añaden al vector de rutas.

Finalmente, la función retorna el vector de rutas encontrado.

```cpp
#include "PRACTICA02.h"

std::vector<std::vector<int>> encontrarRutas(const Automata& automata, const
    std::string& cadena, int i, int posicion, std::vector<int> ruta_actual){
    std::vector<std::vector<int>> rutas;

    if(i >= cadena.size()){ // Si hemos alcanzado el final de la cadena
        if(automata.estadoFinal == posicion) {
            ruta_actual.push_back(posicion);
            rutas.push_back(ruta_actual);
        }
        return rutas;
    }

    for(auto transicion : automata.grafo[posicion].transiciones){
        if(cadena[i] == transicion.first){
            // Llamada recursiva para explorar la transicion
            auto nuevas_rutas = encontrarRutas(automata, cadena, i+1,
                transicion.second, ruta_actual);
            for(auto& ruta : nuevas_rutas) {
                ruta.push_back(posicion);
                rutas.push_back(ruta);
            }
        }
    }

    return rutas;
}
```

Listing 13: Archivo: *P02.cpp* — Funcion `encontrarRutas`

Se detalla a lujo de detalle, esta funcion, la funcion principal para la realizacion de esta practica:

1. **Parámetros de entrada:**
   - `std::ofstream& archivo1`: Referencia a un archivo de salida donde se escribirán las combinaciones generadas.
   - `std::ofstream& archivo2`: Referencia a otro archivo de salida donde se escribirán los datos relacionados con las combinaciones.
   - `int len`: Longitud de las combinaciones a generar.
   - `int& i`: Variable que lleva la cuenta del número de combinaciones generadas.
   - `char simbolo0`: Símbolo cero del alfabeto.
   - `char simbolo1`: Símbolo uno del alfabeto.
   - `std::string prefix = '' ''`: Prefijo de la combinación actual (se inicializa como una cadena vacía).
   - `int unos = 0`: Número de unos en la combinación actual (se inicializa como cero).
2. **Generación de combinaciones:**
   - La función utiliza recursión para generar todas las combinaciones posibles de longitud `len`.
   - En cada iteración, se agrega uno de los dos símbolos (`simbolo0` o `simbolo1`) al prefijo de la combinación actual.
   - Se escriben las combinaciones y los datos relacionados en los archivos de salida proporcionados.
3. **Conteo de unos:**
   - Se lleva un conteo de los unos en la combinación actual para escribir los datos correspondientes en el segundo archivo de salida.

4.4.7. *graficarP02.cpp*

El archivo *graficarP02.cpp* incluye unicamente a la función `graficar ()`, la cual ejecuta un script de MATLAB para generar un gráfico. Primero, define la ruta del script y luego la ejecuta utilizando la función `system ()`. Se verifica si la ejecución fue exitosa y se imprime un mensaje correspondiente. Después, se espera 15 segundos antes de iniciar la graficación en MATLAB utilizando la función `Sleep ()` de la biblioteca `windows.h`. Esta función facilita la integración de gráficos generados en MATLAB en el flujo de trabajo del programa de C++. A continuacion el bloque de codigo de la funcion en cuestión:

```cpp
#include "PRACTICA01.h"
#include <windows.h> // Incluir la biblioteca windows.h

void graficar(int& error){
    std::string ruta = "MATLAB -r run('graficoMatlab.m')";

    error = system(ruta.c_str());

    // Verificar si la ejecucion fue exitosa
    if (error == 0) {
        std::cout << "La ruta se ejecuto correctamente." << std::endl;
    } else {
        std::cerr << "Error al ejecutar la ruta." << std::endl;
    }

    // Esperar un tiempo adicional
    std::cout << "Esperando 10 segundos antes de cerrar MATLAB..." <<
        std::endl;
    Sleep(10000); // Espera 10000 milisegundos (equivalente a 10 segundos)
}
```

Listing 14: Archivo: *P02.cpp* — Funcion `graficar()`

4.4.8. *graficoMatlab.m*

Este script de MATLAB configura y traza dos subgráficos: uno que muestra la cantidad de puntos por cadena y otro que muestra el logaritmo de la cantidad de puntos por cadena. Se deja el codigo del script en cuestión:

```
% Configurar el trazado
scatter_options = '.';

% Inicializar variables para el trazado
figure;

% Nombre del archivo y longitud del bloque
nombre_archivo = 'salidaPractica1.csv';
block_size = 1e6; % Por ejemplo, lee el archivo en bloques de 1 millon de
    tuplas

% Abrir el archivo para lectura
fileID = fopen(nombre_archivo, 'r');

if fileID == -1
    error('No se pudo abrir el archivo.');
end

% Crear subplots
subplot(2, 1, 1);
hold on;

% Leer y graficar los datos en bloques
```

```
while ~feof(fileID)
    % Leer un bloque de datos del archivo
    data = textscan(fileID, '%f%f', block_size, 'Delimiter', ',',
        'HeaderLines', 1);
    if isempty(data{1})
        break; % No hay mas datos para leer, salir del bucle
    end

    % Trazar los datos del bloque actual
    scatter(data{1}, data{2}, scatter_options);
end

% Configurar etiquetas y titulo
xlabel('ID de cadena');
ylabel('Cantidad de puntos en la cadena');
title('Grafico de la cantidad de puntos por cadena');

% Crear subplots para el logaritmo
subplot(2, 1, 2);
hold on;

% Reiniciar la lectura del archivo
frewind(fileID);

% Leer y graficar los datos en bloques con logaritmo
while ~feof(fileID)
    % Leer un bloque de datos del archivo
    data = textscan(fileID, '%f%f', block_size, 'Delimiter', ',',
        'HeaderLines', 1);
    if isempty(data{1})
        break; % No hay mas datos para leer, salir del bucle
    end

    % Aplicar el logaritmo a la cantidad de puntos
    data_log = log(data{2});

    % Trazar los datos del bloque actual con logaritmo
    scatter(data{1}, data_log, scatter_options);
end

% Configurar etiquetas y titulo para el subplot de logaritmo
xlabel('ID de cadena');
ylabel('Log(Cantidad de puntos en la cadena)');
title('Grafico del logaritmo de la cantidad de puntos por cadena');

% Cerrar el archivo
fclose(fileID);

% Mostrar el grafico
hold off;

pause(15);
close all;

pause(5);
exit();
```

Listing 15: Archivo: *graficoMatlab* — Script

Aquí está el resumen de lo que hace:

1. **Configuración del trazado:**
    - Se establece el estilo de los puntos en el gráfico con la variable `scatter_options`.

2. **Inicialización de variables y creación de la figura:**
   - Se inicializan las variables `nombre_archivo` y `block_size`.
   - Se abre el archivo especificado en modo de lectura (`'r'`).
3. **Creación de los subgráficos:**
   - Se crea un subplot con dos filas y una columna.
   - En el primer subplot, se trazan los datos directamente desde el archivo.
   - En el segundo subplot, se trazan los datos con el logaritmo de la cantidad de puntos.
4. **Lectura y trazado de datos en bloques:**
   - Se lee un bloque de datos del archivo y se trazan los puntos en el gráfico.
   - Se aplica el logaritmo a la cantidad de puntos en el segundo subplot.
5. **Configuración de etiquetas y títulos:**
   - Se establecen etiquetas y títulos para ambos subgráficos.
6. **Cierre del archivo y visualización del gráfico:**
   - Se cierra el archivo después de leer todos los datos.
   - Se muestra el gráfico.
7. **Pausa antes de cerrar y salir del script:**
   - Se realiza una pausa de 15 segundos antes de cerrar el gráfico y salir del script.
   - Se cierra el gráfico y se finaliza el script después de una pausa adicional de 5 segundos.

5 Resultados.

A continuación se dejan los resultados dados por el programa con una potencia igual a veinte y ocho:

Figura 1: Ejecucion de la Practica 01 en terminal.

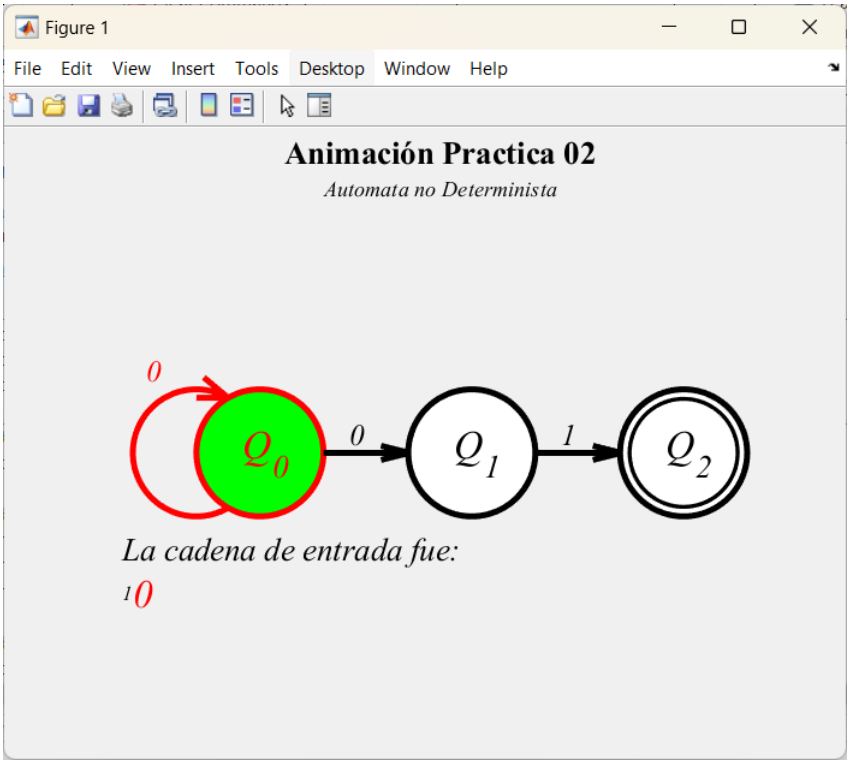

Figura 2: Grafico resultante con potencia = 28

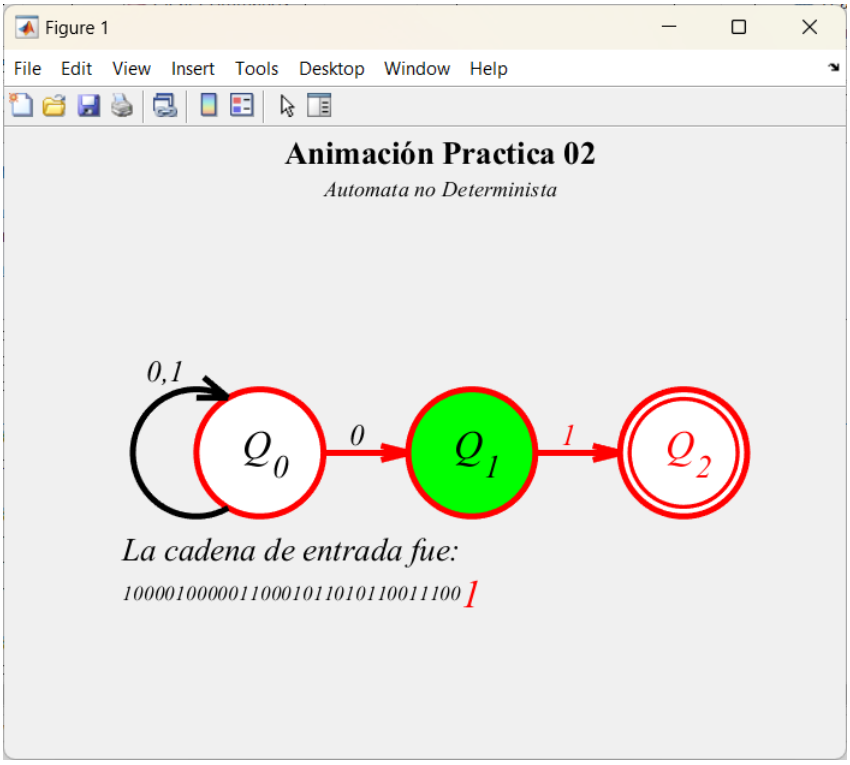

Figura 3: Grafico resultante con potencia = 28

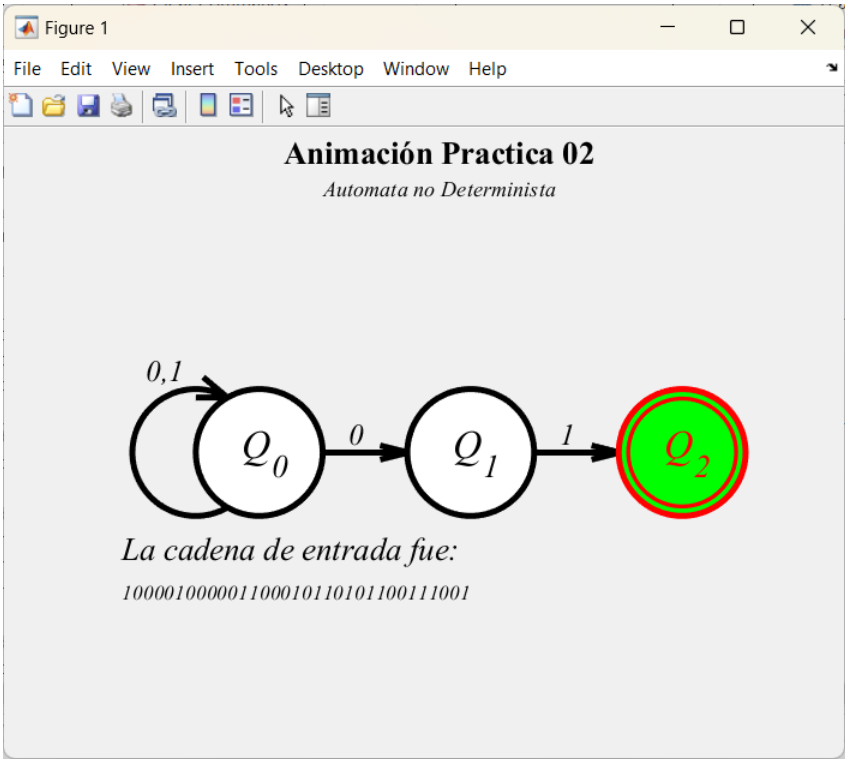

Figura 4: Grafico resultante con potencia = 28

Este es el resultado en el *rutas.txt* con una longitud de setenta y dos:

```
Se encontraron las siguientes rutas para la cadea:
<<01010101110001101000011010001100100010111011111000110010101100001110001101>>:
Ruta: Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 ->
Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 ->
Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 ->
Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 ->
Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 -> Q0 ->
Q0 -> Q0 -> Q0 -> Q0 -> Q1 -> Q2
```

6 Conclusiones

Al completar esta práctica, no solo adquirí conocimientos técnicos en la implementación de algoritmos y la visualización de datos, sino que experimenté un viaje integral que fusionó la teoría y la práctica en el vasto mundo de la informática. A lo largo de este proceso, profundicé en los fundamentos de la teoría de autómatas, aplicando conceptos abstractos en la generación de potencias de un alfabeto utilizando C++. Este ejercicio me permitió no solo entender los principios teóricos detrás de la generación de potencias, sino también apreciar la importancia de la implementación práctica para comprender plenamente su alcance y aplicación en el mundo real.

Además, al explorar la visualización de datos en MATLAB, descubrí cómo transformar números y conceptos abstractos en imágenes claras y significativas. La representación gráfica de los datos generados me proporcionó una nueva perspectiva sobre el comportamiento de los sistemas computacionales, permitiéndome identificar patrones, tendencias y anomalías que de otro modo podrían haber pasado desapercibidos. Esta experiencia me capacitó para comunicar de manera efectiva los resultados de mis análisis y tomar decisiones informadas basadas en la evidencia visual.

Más allá de las habilidades técnicas adquiridas, esta práctica me proporcionó una comprensión más profunda de los conceptos fundamentales que subyacen en el corazón de la computación. Desarrollé una apreciación más amplia de la importancia de la teoría de autómatas en la construcción y el análisis de

sistemas computacionales, así como una comprensión más aguda de cómo la visualización de datos puede amplificar nuestra capacidad para comprender y tomar decisiones en entornos complejos.

Como conclusion final, puedo decir, esta práctica me equipó con las habilidades y el conocimiento necesarios para abordar desafíos más complejos en mi viaje en el mundo de la informática y la ciencia de datos. Al integrar teoría y práctica en un viaje de descubrimiento computacional, avancé hacia un mayor dominio de los principios fundamentales y las herramientas necesarias para enfrentar los desafíos del futuro con confianza y perspicacia.

Referencias

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introducción a la teoría de autómatas, lenguajes y computación*. Madrid: PEARSON EDUCACIÓN S.A., 2007. Formato: 195 x 250 mm. Páginas: 452.