UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MANIZALES

ANÁLISIS Y DISEÑO DE ALGORITMOS

MANUAL TÉCNICO DEL PROYECTO FINAL

Presentado a
Luz Enith Guerrero Mendietai

Presentado por
Manuela Zuluaga Cardona
Santiago Hoyos Gomez

Fecha de entrega
Diciembre 2023

1. **Estrategia de solución:** La estrategia general del código es calcular diversas matrices de probabilidades basadas en secuencias de estados para varios canales. Aquí hay un resumen de las funciones clave y su propósito:
    - **Generación de muestras: generar_muestras(numCanales):** Esta función genera todas las posibles combinaciones de estados para los canales dados. Utiliza la función format para obtener representaciones binarias de todas las combinaciones posibles de estados (0 o 1) para cada canal. El resultado es una lista muestrasBase que contiene todas las combinaciones posibles.

```
muestrasBase = []
#Genera las 2^n posibilidades de n canales
def generar_muestras(numCanales):
    for i in range(2 ** numCanales):
        num = format(i, f'0{numCanales}b')
        muestrasBase.append(num)
```

      Por ejemplo, si numCanales es 2, se generan las combinaciones: ['00', '01', '10', '11'].

    - **Generación de estados aleatorios: generar_estados_aleatorios(num_canales, num_muestras):** Esta función genera estados aleatorios para cada canal en cada muestra. Utiliza un diccionario llamado estados_generados, donde cada canal tiene una lista de estados aleatorios asociados.

```
def generar_estados_aleatorios(num_canales, num_muestras):
    estados_posibles = ["0", "1"]  # Estados posibles
    #Genera las entradas en el diccionario, con las letras del alfabeto secuancialmente y
    estados_generados = {f"{chr(65 + i)}": [] for i in range(num_canales)}
    # print("Estados generados en aleatorios para diccionario",estados_generados)
```

      Este diccionario se utiliza posteriormente para calcular varias matrices de probabilidades.

    - **Cálculo de Matriz de Probabilidades para Estados Actuales en Canales: calcular_matriz_EstadoCanalF(estados_canales, numMuestras):** Esta función calcula la matriz de probabilidades para los estados actuales en cada canal. También guarda los índices de los estados. La matriz resultante se llama EstadoCanalF:

```python
# Función para calcular la matriz de probabilidades para los 1 en los canales y guardar los indexs
def calcular_matriz_EstadoCanalF(estados_canales,numMuestras):
    #Almacena los indices de los estados, para reutilizarlos en el calculo de las otras matrices de estados
    indexEstados= {muestra: [] for muestra in muestrasBase}
    EstadoCanalF = {estados: {canal: 0 for canal in estados_canales.keys()} for estados in muestrasBase}
    print("Matriz Inicial",EstadoCanalF)

    # Itera por cada una de las muestras
    for time in range(numMuestras):
        # Añade el index, en el que un estado posible aparece en la muestra
        temp = getEstado(estados_canales,time)
        indexEstados[temp].append(time)
        # print("Estoy en la iteracion",time+1)
        # Itera entre los canales en caso de no tratarse de la ultima muestra
        # Se usa el -1 para que se ajuste a la forma en la que se manejan los indexs (desde el cero hasta numMuestras-1)
        if(time!=(numMuestras-1)):
            # print("Ha entrado")
            for canal in estados_canales.keys():
                # print("Evaluando si hay un",canal ,"en UNO justo despues, para la muestra",muestra)
                # print("EL valor del canal ",canal,"en t+1 es:",estados_canales[canal][time+1])
                proxEstadoCanal= estados_canales[canal][time+1]
                EstadoCanalF[temp][canal] += int(proxEstadoCanal)

    # print("Matriz valores despues de iterar",EstadoCanalF)
    # print("Lista de ocurrencias",indexEstados)
```

```python
    # Calcular las probabilidades dividiendo por el número de coincidencias del estado actual
    for estado in EstadoCanalF:
        for canal in EstadoCanalF[estado]:
            estadoCanal=EstadoCanalF[estado][canal]
            sumEstado= len(indexEstados[estado])
            #Por si el divisor es 0
            if(sumEstado==0):
                EstadoCanalF[estado][canal]="Indefinido"
            # Por si el dividendo es 0
            elif (estadoCanal==0):
                EstadoCanalF[estado][canal]=str(0)
            # Por si la dicision es exacta
            elif(estadoCanal%sumEstado==0):
                EstadoCanalF[estado][canal]=str(int(estadoCanal/sumEstado))
            # Por si la division no es exacta
            else:
                # Descomentar si se desea fracciones simplificadas
                # estadoCanal,sumEstado=simpFraccion(estadoCanal,sumEstado)

                EstadoCanalF[estado][canal] =""+str(estadoCanal)+"/"+str(sumEstado)
            # EstadoCanalF[estado][canal] =Fraction(int(EstadoCanalF[estado][canal])/len(indexEstados[estado]))

    print("Matriz valores despues de operar",EstadoCanalF)

    return EstadoCanalF, indexEstados
```

- **Generación de Matriz de Probabilidades Entre Estados Consecutivos: generar_matriz_EstadoEstadoF(indexEstados):** Esta función genera la matriz de probabilidades entre estados consecutivos. La matriz resultante se llama EstadoEstadoF:

```python
def generar_matriz_EstadoEstadoF(indexEstados):
    # Inicializar la matriz de probabilidades previas con ceros
    EstadoEstadoF = {estado: {estado: 0 for estado in muestrasBase} for estado in muestrasBase}
    print("Matriz inicial", EstadoEstadoF)

    #Iteracion n*n de estados
    for estado in muestrasBase:
        for estadoNext in muestrasBase:
            for index in indexEstados[estado]:
                # print("actual",index)
                for indexNext in indexEstados[estadoNext]:
                    # print("next",indexNext)
                    if index+1 ==indexNext:
                        EstadoEstadoF[estado][estadoNext]+=1

    print("Matriz despues de iterar", EstadoEstadoF)
```

```python
    # Calcular las probabilidades dividiendo por el número de coincidencias del estado actual
    for estado in EstadoEstadoF:
        for estadoNext in EstadoEstadoF[estado]:
            # print("valores",estado,"y",estadoNext,"son",EstadoEstadoF[estado][estadoNext])
            estadoMuestra=EstadoEstadoF[estado][estadoNext]
            # print("estadoMuestra",estadoMuestra)
            sumEstado= len(indexEstados[estado])
            # print("sumEstado",sumEstado)
            #Por si el divisor es 0
            if(sumEstado==0):
                EstadoEstadoF[estado][estadoNext]="Indefinido"
            # Por si el dividendo es 0
            elif (estadoMuestra==0):
                EstadoEstadoF[estado][estadoNext]=str(0)
            # Por si la dicision es exacta
            elif(estadoMuestra%sumEstado==0):
                EstadoEstadoF[estado][estadoNext]=str(int(estadoMuestra/sumEstado))
            # Por si la division no es exacta
            else:
                # Descomentar si se desea fracciones simplificadas
                # estadoMuestra,sumEstado=simpFraccion(estadoMuestra,sumEstado)
                EstadoEstadoF[estado][estadoNext] =""+str(estadoMuestra)+"/"+str(sumEstado)
            # EstadoCanalF[estado][canal] =Fraction(int(EstadoCanalF[estado][canal])/len(indexEstados[estado]))

    print("Matriz valores despues de operar",EstadoEstadoF)

    return EstadoEstadoF
```

- **Cálculo de Matriz de Probabilidades Previas en Canales (calcular_matriz_EstadoCanalP(estados_canales, numMuestras, indexEstados):** Esta función calcula la matriz de probabilidades previas para los canales. La matriz resultante se llama EstadoCanalP.

```python
def calcular_matriz_EstadoCanalP(estados_canales,numMuestras,indexEstados):
    EstadoCanalP = {estados: {canal: 0 for canal in estados_canales.keys()} for estados in muestrasBase}
    print("Matriz Inicial",EstadoCanalP)

    # Itera por cada una de las muestras
    for time in range(1,numMuestras):
        temp = getEstado(estados_canales,time)
        # print("Ha entrado")
        for canal in estados_canales.keys():
            prevEstadoCanal= estados_canales[canal][time-1]
            EstadoCanalP[temp][canal] += int(prevEstadoCanal)
```

```
    # Calcular las probabilidades dividiendo por el número de coincidencias del estado actual
    for estado in EstadoCanalP:
        for canal in EstadoCanalP[estado]:
            estadoCanal=EstadoCanalP[estado][canal]
            sumEstado= len(indexEstados[estado])
            #Por si el divisor es 0
            if(sumEstado==0):
                EstadoCanalP[estado][canal]="Indefinido"
            # Por si el dividendo es 0
            elif (estadoCanal==0):
                EstadoCanalP[estado][canal]=str(0)
            # Por si la dicision es exacta
            elif(estadoCanal%sumEstado==0):
                EstadoCanalP[estado][canal]=str(int(estadoCanal/sumEstado))
            # Por si la division no es exacta
            else:
                # Descomentar si se desea fracciones simplificadas
                # estadoCanal,sumEstado=simpFraccion(estadoCanal,sumEstado)

                EstadoCanalP[estado][canal] =""+str(estadoCanal)+"/"+str(sumEstado)
            # EstadoCanalF[estado][canal] =Fraction(int(EstadoCanalF[estado][canal])/len(indexEstados[estado]))

    print("Matriz valores despues de operar",EstadoCanalP)

    return EstadoCanalP
```

- **Generación de Matriz de Probabilidades Entre Estados Consecutivos Anteriores: generar_matriz_EstadoEstadoP(indexEstados):** Esta función genera la matriz de probabilidades entre estados consecutivos anteriores. La matriz resultante se llama EstadoEstadoP

```
def generar_matriz_EstadoEstadoP(indexEstados):
    # Inicializar la matriz de probabilidades previas con ceros
    EstadoEstadoP = {estado: {estado: 0 for estado in muestrasBase} for estado in muestrasBase}
    print("Matriz inicial", EstadoEstadoP)

    #Iteracion n*n de estados
    for estado in muestrasBase:
        for estadoPrev in muestrasBase:
            for index in indexEstados[estado]:
                # print("actual",index)
                for indexPrev in indexEstados[estadoPrev]:
                    # print("next",indexNext)
                    if index-1 ==indexPrev:
                        EstadoEstadoP[estado][estadoPrev]+=1

    print("Matriz despues de iterar", EstadoEstadoP)
```

```
    # Calcular las probabilidades dividiendo por el número de coincidencias del estado actual
    for estado in EstadoEstadoP:
        for estadoPrev in EstadoEstadoP[estado]:
            # print("valores",estado,"y",estadoNext,"son",EstadoEstadoF[estado][estadoNext])
            estadoMuestra=EstadoEstadoP[estado][estadoPrev]
            # print("estadoMuestra",estadoMuestra)
            sumEstado= len(indexEstados[estado])
            # print("sumEstado",sumEstado)
            #Por si el divisor es 0
            if(sumEstado==0):
                EstadoEstadoP[estado][estadoPrev]="Indefinido"
            # Por si el dividendo es 0
            elif (estadoMuestra==0):
                EstadoEstadoP[estado][estadoPrev]=str(0)
            # Por si la dicision es exacta
            elif(estadoMuestra%sumEstado==0):
                EstadoEstadoP[estado][estadoPrev]=str(int(estadoMuestra/sumEstado))
            # Por si la division no es exacta
            else:
                # Descomentar si se desea fracciones simplificadas
                # estadoMuestra,sumEstado=simpFraccion(estadoMuestra,sumEstado)
                EstadoEstadoP[estado][estadoPrev] =""+str(estadoMuestra)+"/"+str(sumEstado)
            # EstadoCanalF[estado][canal] =Fraction(int(EstadoCanalF[estado][canal])/len(indexEstados[estado]))
    print("Matriz valores despues de operar",EstadoEstadoP)
    return EstadoEstadoP
    return EstadoEstadoP
```

- **Función General: allOps(estados_canales, numMuestras):** La función allOps utiliza todas las funciones anteriores para realizar todas las operaciones y devolver las matrices de probabilidades finales.

```
def allOps(estados_canales,numMuestras):
    print("\n*****MODO TODOS*****")
    #Almacena los indices de los estados, para reutilizarlos en el calculo de las otras matrices de estados
    indexEstados = {muestra: [] for muestra in muestrasBase}
    EstadoCanalF = {estados: {canal: 0 for canal in estados_canales.keys()} for estados in muestrasBase}
    EstadoEstadoF = {estado: {estado: 0 for estado in muestrasBase} for estado in muestrasBase}
    EstadoCanalP = {estados: {canal: 0 for canal in estados_canales.keys()} for estados in muestrasBase}
    EstadoEstadoP = {estados: {estado: 0 for estado in muestrasBase} for estados in muestrasBase}

    print("\nMatriz valores iniciales")
    print("EstadoCanalF:",EstadoCanalF)
    print("EstadoCanalP:",EstadoCanalP)
    print("EstadoEstadoF:",EstadoEstadoF)
    print("EstadoEstadoP:",EstadoEstadoP)

    # Itera por cada una de las muestras
    for time in range(numMuestras):
        # Añade el index, en el que un estado posible aparece en la muestra
        actual = getEstado(estados_canales,time)
        indexEstados[actual].append(time)
        # print("Estoy en la iteracion",time+1)

        # print("EL estado actual es", actual)
```

```
# print("EL estado actual es", actual)
if(time==0):
    proxEstado = getEstado(estados_canales,time+1)
    EstadoEstadoF[actual][proxEstado] += 1
    for canal in estados_canales.keys():
        proxEstadoCanal = estados_canales[canal][time+1]
        EstadoCanalF[actual][canal] += int(proxEstadoCanal)

elif(time==numMuestras-1):
    prevEstado=getEstado(estados_canales,time-1)
    EstadoEstadoP[actual][prevEstado] += 1
    for canal in estados_canales.keys():
        prevEstadoCanal= estados_canales[canal][time-1]
        EstadoCanalP[actual][canal] += int(prevEstadoCanal)
else:
    proxEstado=getEstado(estados_canales,time+1)
    EstadoEstadoF[actual][proxEstado] += 1
    prevEstado=getEstado(estados_canales,time-1)
    EstadoEstadoP[actual][prevEstado] += 1

    for canal in estados_canales.keys():
        proxEstadoCanal= estados_canales[canal][time+1]
        EstadoCanalF[actual][canal] += int(proxEstadoCanal)
        prevEstadoCanal= estados_canales[canal][time-1]
        EstadoCanalP[actual][canal] += int(prevEstadoCanal)
```

```
print("\nMatriz valores conteo")
print("Indexes",indexEstados)
print("EstadoCanalF:",EstadoCanalF)
print("\nEstadoCanalP:",EstadoCanalP)
print("\nEstadoEstadoF:",EstadoEstadoF)
print("\nEstadoEstadoP:",EstadoEstadoP)

# Calcular las probabilidades dividiendo por el número de coincidencias del estado actual
canales= list(estados_canales.keys())
for estadoi in muestrasBase:
    # print("\nEvaluando para estado actual",estadoi)
    sumEstado=len(indexEstados[estadoi])
    # print("con apariciones en",indexEstados[estadoi])
    for canal in canales:
        # print("Evaluando para el canal",canal)
        estadoEvalF=EstadoCanalF[estadoi][canal]
        # print("estado en proximo",estadoEvalF)
        estadoEvalP=EstadoCanalP[estadoi][canal]
        # print("estado en previo",estadoEvalP)
```

```
            if(sumEstado==0):
                # print("Entro por divisor 0")
                EstadoCanalF[estadoi][canal]="Indefinido"
                EstadoCanalP[estadoi][canal]="Indefinido"
            # Por si el dividendo es 0
            elif(estadoEvalF==0 and estadoEvalP==0):
                # print("Entro por dividendo 0  (1)")
                EstadoCanalF[estadoi][canal]=str(0)
                # print("Entro por dividendo 0  (2)")
                EstadoCanalP[estadoi][canal]=str(0)
```

```
    elif(estadoEvalF==0):
        # print("Entro por dividendo 0  (1)")
        EstadoCanalF[estadoi][canal]=str(0)
        EstadoCanalP[estadoi][canal] =""+str(estadoEvalP)+"/"+str(sumEstado)
    elif(estadoEvalP==0):
        # print("Entro por dividendo 0  (2)")
        EstadoCanalP[estadoi][canal]=str(0)
        EstadoCanalF[estadoi][canal] =""+str(estadoEvalF)+"/"+str(sumEstado)
    # Por si la dicision es exacta
    elif(estadoEvalF==sumEstado and estadoEvalP==sumEstado):
        EstadoCanalF[estadoi][canal]=str(int(estadoEvalF/sumEstado))
        EstadoCanalP[estadoi][canal]=str(int(estadoEvalP/sumEstado))
    elif(estadoEvalF==sumEstado):
        EstadoCanalF[estadoi][canal]=str(int(estadoEvalF/sumEstado))
        EstadoCanalP[estadoi][canal] =""+str(estadoEvalP)+"/"+str(sumEstado)
    elif(estadoEvalP==sumEstado):
        EstadoCanalP[estadoi][canal]=str(int(estadoEvalP/sumEstado))
        EstadoCanalF[estadoi][canal] =""+str(estadoEvalF)+"/"+str(sumEstado)
    # Por si la division no es exacta
    else:
        # print("***Estructurar fraccionarios***")
        # Descomentar si se desea fracciones simplificadas
        # estadoMuestra,sumEstado=simpFraccion(estadoMuestra,sumEstado)
```

```
        EstadoCanalF[estadoi][canal] =""+str(estadoEvalF)+"/"+str(sumEstado)
        EstadoCanalP[estadoi][canal] =""+str(estadoEvalP)+"/"+str(sumEstado)
    # print("EstadoCanalF en este caso:",EstadoCanalF[estadoi][canal])
    # print("EstadoCanalP en este caso:",EstadoCanalP[estadoi][canal])


for estadoj in muestrasBase:
    estadoEvalF=EstadoEstadoF[estadoi][estadoj]
    estadoEvalP=EstadoEstadoP[estadoi][estadoj]
```

```
    if(sumEstado==0):
        EstadoEstadoF[estadoi][estadoj]="Indefinido"
        EstadoEstadoP[estadoi][estadoj]="Indefinido"
    # Por si el dividendo es 0
    elif(estadoEvalF==0 and estadoEvalP==0):
        EstadoEstadoF[estadoi][estadoj]=str(0)
        EstadoEstadoP[estadoi][estadoj]=str(0)
    elif(estadoEvalF==0):
        EstadoEstadoF[estadoi][estadoj]=str(0)
        EstadoEstadoP[estadoi][estadoj] =""+str(estadoEvalP)+"/"+str(sumEstado)
    elif(estadoEvalP==0):
        EstadoEstadoP[estadoi][estadoj]=str(0)
        EstadoEstadoF[estadoi][estadoj] =""+str(estadoEvalF)+"/"+str(sumEstado)
    # Por si la dicision es exacta
    elif(estadoEvalF==sumEstado and estadoEvalP==sumEstado):
        EstadoEstadoF[estadoi][estadoj]=str(int(estadoEvalF/sumEstado))
        EstadoEstadoP[estadoi][estadoj]=str(int(estadoEvalP/sumEstado))
    elif(estadoEvalF==sumEstado):
        EstadoEstadoF[estadoi][estadoj]=str(int(estadoEvalF/sumEstado))
        EstadoEstadoP[estadoi][estadoj] =""+str(estadoEvalP)+"/"+str(sumEstado)
    elif(estadoEvalP==sumEstado):
        EstadoEstadoP[estadoi][estadoj]=str(int(
        EstadoEstadoF[estadoi][estadoj] =""+str(estadoEvalF)+"/"+str(sumEstado)
    # Por si la division no es exacta
```

(variable) estadoEvalF: int

```
            else:
                # Descomentar si se desea fracciones simplificadas
                # estadoMuestra,sumEstado=simpFraccion(estadoMuestra,sumEstado)

                EstadoEstadoF[estadoi][estadoj] =""+str(estadoEvalF)+"/"+str(sumEstado)
                EstadoEstadoP[estadoi][estadoj] =""+str(estadoEvalP)+"/"+str(sumEstado)
            # EstadoEstadoF[estadoi][estadoj] =Fraction(int(EstadoEstadoF[estadoi][estadoj])/len(indexEstados[estadoi]))
            # EstadoEstadoP[estadoi][estadoj] =Fraction(int(EstadoEstadoP[estadoi][estadoj])/len(indexEstados[estadoi]))
    # print("\nMatriz valores despues de iterar")
    # print("EstadoCanalF:",EstadoCanalF)
    # print("\nEstadoCanalP:",EstadoCanalP)
    # print("\nEstadoEstadoF:",EstadoEstadoF)
    # print("\nEstadoEstadoP:",EstadoEstadoP)

    return EstadoCanalF, EstadoEstadoF, EstadoCanalP, EstadoEstadoP
```

- **Función graphMatriz(matriz, opc):** Esta función se encarga de presentar gráficamente las matrices de probabilidades en formato de tabla. Utiliza la biblioteca tabulate para dar formato a las matrices y mostrarlas de manera clara. La variable opc determina si se trata de los estados actuales (opc='+') o futuros (opc='-').

```
def graphMatriz(matriz, opc):
    if opc=="+":
        titulosColumnas=["Estado//T+1"]
    elif opc=="-":
        titulosColumnas=["Estado//T-1"]

    otrosTitulos=matriz[list(matriz.keys())[0]].keys()
    titulosColumnas.extend(otrosTitulos)

    table = [[key] + [matriz[key][column] for column in titulosColumnas[1:]] for key in matriz.keys()]

    print(tabulate(table, titulosColumnas, tablefmt="fancy_grid"))

tiempos={"Actual":dict(),"Futuro":list()}
canalesBase=["a","b","c"]
```

tiempos: Es un diccionario que almacena los valores actuales y futuros de los canales. En este contexto, se utiliza para gestionar los valores ingresados por el usuario.
canalesBase: Es una lista que contiene los identificadores de los canales base. Esta lista se utiliza para validar los canales ingresados por el usuario**.**

- **SampleData:** Es un diccionario que contiene datos de muestra representados por combinaciones binarias. Cada combinación binaria representa un estado, y los valores asociados a esas combinaciones representan probabilidades.

```
sampleData={
    '000': {'000': 1, '001': 0, '010': 0, '011': 0, '100': 0, '101': 0, '110': 0, '111': 0},
    '001': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 1, '101': 0, '110': 0, '111': 0},
    '010': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 0, '101': 1, '110': 0, '111': 0},
    '011': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 0, '101': 1, '110': 0, '111': 0},
    '100': {'000': 0, '001': 1, '010': 0, '011': 0, '100': 0, '101': 0, '110': 0, '111': 0},
    '101': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 0, '101': 0, '110': 0, '111': 1},
    '110': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 1, '101': 0, '110': 0, '111': 0},
    '111': {'000': 0, '001': 0, '010': 0, '011': 0, '100': 0, '101': 0, '110': 1, '111': 0}
    }
```

- **Función menuDivision(data=sampleData, canales=canalesBase):** Esta función implementa un menú interactivo para que el usuario introduzca particiones de canales en diferentes momentos. Utiliza las funciones graphMatriz, printDataDivision, y final para mostrar datos y realizar cálculos basados en las elecciones del usuario.

```
def menuDivision(data=sampleData,canales=canalesBase):
    #Corregir para listas vacias
    print("\n\nLa matriz original es")
    graphMatriz(data,"+")
    print("\n**Menu de particiones**")
    print(tiempos)
    print("Los canales disponibles para realizar las particiones son: ", canales)
    for tiempo in tiempos:
        print("\nIngrese los canales de la particion {}".format(tiempo)+" Ejemplo: a,b,c o b,c o a,c")
        part=input().lower()
        part= part.split(",")
        for canal in part:
            if canal not in canales:
                print("**El canal {} no existe**".format(canal))
                menuDivision(sampleData,canalesBase)
            if tiempo == "Actual":
                while True:
                    print("ingrese el valor actual para el canal {}".format(canal))
                    val=input()
                    if val not in ["0","1"]:
                        print("\nEl valor debe ser 0 o 1")
                    else:
                        tiempos[tiempo][canal]=val
                        break
            else:
                tiempos[tiempo].append(canal)
    print("\n\nTiempos iniciales: ",tiempos)
```

```
    print("\nDesea realizar comprobaciones de distancia entre particiones(s/n)?")
    op=input()
    if op=="s":
        final(data,canales)
    else:
        printDataDivision(data,canales,list(tiempos["Actual"].keys()),tiempos["Futuro"])
```

- **Función final(data, canales):** Realiza cálculos finales, calcula la matriz original y evalúa distintas combinaciones de particiones futuras y actuales. Llama a checkEmd para comparar la distancia EMD entre la distribución original y las resultantes.

```python
def final(data,canales):
    biPartActuales, biPartFuturo= allBiParticiones(tiempos)
    #calculo de original
    print("***Se mirara el original**")
    original=printDataDivision(data,canales,list(tiempos["Actual"].keys()),tiempos["Futuro"])
    print("\n\n")
    matFinal={}
    for i in range(len(biPartActuales)):
        for j in range(len(biPartFuturo)):
            # print("\n\nSe evaluara, con las siguientes partciones")
            # print("Particion para el actual",biPartActuales[i])
            # print("Particion para el futuro",biPartFuturo[j])
            for ii in range(len(biPartActuales[i])):
                referencias=[]
                preStr=""
                for jj in range(len(biPartFuturo[j])):
                    if (biPartActuales[i][ii]==[]) and (biPartFuturo[j][jj]==[]):
                        continue
                    print("Para el actual",biPartActuales[i][ii])
                    preStr+=str(biPartActuales[i][ii])
                    print("Para el futuro",biPartFuturo[j][jj])
                    preStr+=str(biPartFuturo[j][jj])
                    referencia=printDataDivision(data,canales,biPartActuales[i][ii],biPartFuturo[j][jj])
                    # print("La referencia dio:",referencia,"\n\n")
                    referencias.append(referencia)
                    preStr+=" // "
                preStr=preStr[:-3]
```

```python
                # print("***Referencias***")
                # print(len(referencias))
                # print(referencias)
                temp=[]
                for element in referencias:
                    preList=[]
                    for key in element:
                        for subkey in element[key]:
                            preList.append(element[key][subkey])
                    temp.append(preList)
                # print("Datos de temp:", temp)
                matFinal[preStr]=multpMatriz(temp)
                # print("Tras Multiplicar:",matFinal[preStr])

                checkEmd(original.copy(),matFinal.copy(),preStr)
```

- **Función checkEmd(original, matFinal, preStr):** Calcula y muestra la distancia Earth Mover's Distance (EMD) entre dos distribuciones. Utiliza la biblioteca SciPy.

```python
def checkEmd(original,matFinal,preStr):
    # print("Original:",original[list(original.keys())[0]])
    # print("Matfinal:",matFinal[preStr])
    if len(list(original[list(original.keys())[0]].keys()))==len(list(matFinal[preStr].keys())):
        print("\nComparacion de distancia con la distrinucion compuesta: ",preStr)
        keysA=list(original[list(original.keys())[0]].keys())
        tempA=[]
        keysB=list(matFinal[preStr].keys())
        tempB=[]
        for i in range(len(keysA)):
            tempA.append(original[list(original.keys())[0]][keysA[i]])
            tempB.append(matFinal[preStr][keysB[i]])

        print("valores previos:",tempA)
        print("valores previos:",tempB)
        # Calculamos la distancia EMD entre las dos distribuciones
        distancia_emd = wasserstein_distance(tempA, tempB)

        # Mostramos el resultado
        print(f'Distancia EMD: {distancia_emd}')
        print("\n\n\n")
```

- **Función printDataDivision(data, canales, listActual, listFuturo):** Filtra las entradas de datos basadas en las particiones actuales y futuras proporcionadas. Devuelve el resultado de operProbs que realiza operaciones con probabilidades.

```python
def printDataDivision(data,canales,listActual,listFuturo):
    actCoincidencias=[]
    elmCoincidencias=[]
    futCoincidencias={}

    #Por los identificadores de canales en actual
    for i in range(len(listActual)):
        lAct=listActual[i]
        #Valor de de referencia en el canal
        canalEstAct=tiempos["Actual"][lAct]
        if not(canalEstAct==None):
            indexT=canales.index(lAct)
            # Si es la primera coincidencia
            if i==0:
                #Por cada entrada en la matriz
                for entry in data:
                    if entry[indexT]==canalEstAct:
                        # print("Entradas que parcialmente coinciden:",entry)
                        actCoincidencias.append(entry)
            else:
                # print("Estado actCoincidencias:",actCoincidencias)
                for entry in actCoincidencias:
                    # print("Se esta verificando que la entrada {} tenga un {} en la posicion {}".format(entry,canalEstAct,ind
                    if not(entry[indexT]==canalEstAct):
                        #print("No cumplio")
                        elmCoincidencias.append(entry)
```

```
#Ciclo para eliminar valores invalidos
for el in elmCoincidencias:
    if el in actCoincidencias:
        actCoincidencias.remove(el)

# print("Entradas que coinciden:",actCoincidencias)

#Por los identificadores de canales en futuro
for entry in actCoincidencias:
    for i in range(len(listFuturo)):
        futCoincidencias[listFuturo[i]]={"0":[],"1":[]}
        for value in data[entry]:
            indexEval=canales.index(listFuturo[i])
            for bin in ["0","1"]:
                if value[indexEval]==bin:
                    futCoincidencias[listFuturo[i]][bin].append(value)

# print("valores por entrada:",futCoincidencias)

return operProbs(actCoincidencias,futCoincidencias,data)
```

- **Función auxOperProbs(data, actList, dataEntrada):** Calcula la probabilidad para un canal específico en un estado dado.

```
def auxOperProbs(data,actList,dataEntrada):
    cero=[]
    uno=[]

    for act in actList:
        # print("act:",act)
        for fut in dataEntrada:
            # print("fut:",fut)
            for value in dataEntrada[fut]:
                # print("value:",value)
                valItem=data[act][value]
                if type(valItem)==str:
                    valItem=eval(valItem)
                if fut=="0":
                    cero.append(valItem)
                else:
                    uno.append(valItem)
    #print("cero:",cero)
    #print("uno:",uno)

    cero=sum(cero*4)/len(cero)
    uno=sum(uno*4)/len(uno)
    return [cero,uno]
```

- **Función multpMatriz(listas):** Multiplica matrices representadas como listas.

```python
def multpMatriz(listas):
    # Inicializamos el diccionario para almacenar los resultados
    resultados = {}
    # Calculamos el número total de combinaciones posibles
    num_combinaciones = 1
    for lista in listas:
        num_combinaciones *= len(lista)
    # print("num_combinaciones:",num_combinaciones)
    # Generamos todas las combinaciones de índices manualmente
    for i in range(num_combinaciones):
        # print("ha entrado")
        clave = ''
        resultado = 1
        # Calcular los índices para cada lista
        for j in range(len(listas)):
            # print("j:",j)
            lista = listas[j]
            # print("lista:",lista)
            indice = (i // (len(lista) ** j)) % len(lista)
            # print("indice:",indice)
            clave += str(indice)
            # print("clave:",clave)
            resultado *= lista[indice]
        resultados[clave] = resultado
    return resultados
```

- **Función operProbs(actlist, futList, data):** Calcula las probabilidades de las particiones futuras dados los datos actuales.

```python
def operProbs(actlist,futList,data):
    matFinal={}
    auxStr=str(tiempos['Actual'])
    # print(auxStr)
    probsEntrada={}
    for entrada in futList:
        probsEntrada[entrada]=auxOperProbs(data,actlist,futList[entrada])
    # print("debug:",probsEntrada)
    listProbsEntrada=[]
    for key in probsEntrada:
        listProbsEntrada.append(probsEntrada[key])
    # print("zzz",listProbsEntrada)
    if len(listProbsEntrada)==1:
        matFinal[auxStr]={"0":listProbsEntrada[0][0],"1":listProbsEntrada[0][1]}
        # print(matFinal)
        graphMatriz(matFinal,"+")
    else:
        matFinal[auxStr]=multpMatriz(listProbsEntrada)
        # print(matFinal)
        graphMatriz(matFinal,"+")
    return matFinal
```

- **Función menu():** Inicia el programa llamando a menuDivision con datos de muestra y canales base predefinidos.

```
def menu():
    menuDivision(sampleData,canalesBase)

if __name__ == "__main__":
    menu()
```

Librerías necesarias para la funcionalidad:

1. Numpy:

Numpy es una biblioteca fundamental en Python para la computación científica. Proporciona soporte para arreglos multidimensionales, funciones matemáticas de alto rendimiento y herramientas para trabajar con estos arreglos. En el código proporcionado, se podría utilizar Numpy para realizar operaciones matriciales y cálculos numéricos eficientes, pero su importación no está presente en el código suministrado.

2. Random:

La biblioteca random es parte de la biblioteca estándar de Python y proporciona funciones para generar números pseudoaleatorios. En el código proporcionado, se usa para realizar operaciones de mezcla aleatoria de estados globales en la función dividir_y_calcular_probabilidades. La función shuffle de esta biblioteca se utiliza para reorganizar aleatoriamente la lista de estados globales.

3. Biparts:

No se proporciona información específica sobre la biblioteca biparts en el código suministrado, pero probablemente sea una biblioteca personalizada creada para el problema específico que estás abordando. Podría contener funciones relacionadas con la generación de bi particiones.

4. Scipy:

Scipy es otra biblioteca esencial para la computación científica en Python. Proporciona funcionalidades adicionales a Numpy, incluidas herramientas para la optimización, álgebra lineal, integración, interpolación y más. En el código, la función wasserstein_distance de Scipy se utiliza para calcular la distancia de Wasserstein entre dos distribuciones. Esta métrica es útil para comparar distribuciones de probabilidad.

5. Tabulate:
Tabulate es una biblioteca externa que facilita la presentación tabular de datos en forma de tablas. En el código suministrado, se utiliza para formatear y presentar las matrices de probabilidades de manera organizada y legible en la función graphMatriz.

6. Math:
La biblioteca math es una biblioteca estándar en Python y proporciona funciones matemáticas básicas. En el código suministrado, no se utiliza directamente, pero podría usarse para funciones matemáticas más generales si es necesario en otras partes del código.

Cada una de estas bibliotecas tiene su función específica y contribuye a la resolución eficiente y efectiva del problema, proporcionando herramientas para operaciones matemáticas, manipulación de datos y presentación de resultados.
La estrategia de solución adoptada para abordar el problema propuesto se basa en una serie de pasos claves que se ejecutan de manera sistemática. En primer lugar, se generan todas las posibles combinaciones de estados para los canales, estableciendo así un espacio de muestra exhaustivo. Posteriormente, se generan estados aleatorios para cada canal en cada muestra, proporcionando datos de entrada realistas y diversificados. Luego, se procede al cálculo de matrices de probabilidades para los estados actuales, siguientes y previos, lo que implica analizar la ocurrencia de patrones y relaciones entre los diferentes estados. Finalmente, se presentan los resultados de manera gráfica utilizando la biblioteca tabulate, facilitando una comprensión visual y detallada de las probabilidades en juego. Esta estrategia se centra en la generación ordenada de datos, el análisis riguroso de probabilidades y la presentación efectiva de resultados para ofrecer una solución completa y estructurada al problema planteado.
La necesidad de las funciones en el código se conecta intrínsecamente con la estrategia de solución adoptada. En primer lugar, la función graphMatriz desempeña un papel crucial al presentar gráficamente las matrices de probabilidades, permitiendo una interpretación clara y organizada de los resultados. Esta función es esencial para la última etapa de la estrategia, donde se busca proporcionar una comprensión visual y detallada de las relaciones probabilísticas. Además, las funciones printDataDivision y final trabajan conjuntamente para realizar cálculos basados en las particiones de canales, conectando directamente con la fase de análisis riguroso de probabilidades. La función printDataDivision filtra y organiza los datos según las particiones proporcionadas, preparándose para ser procesados y evaluados en la etapa final del cálculo de probabilidades.
En segundo lugar, las funciones auxiliares, como auxOperProbs y multpMatriz, desempeñan un papel crucial en el cálculo detallado de

probabilidades. La estrategia de solución adoptada se centra en la generación ordenada de datos, y estas funciones contribuyen significativamente al procesamiento y la manipulación de las probabilidades en diferentes estados y momentos temporales. La necesidad de estas funciones es evidente en la complejidad del problema, donde se requiere un análisis detallado de las probabilidades en juego. En resumen, cada función del código se diseñó estratégicamente para abordar aspectos específicos de la estrategia general, desde la presentación visual hasta el cálculo preciso de probabilidades, proporcionando así una solución completa y estructurada al problema propuesto.

2. **Proceso detallado de solución**

Generación de Todas las Posibles Combinaciones de Estados para los Canales:
En esta etapa, se utiliza la función generar_muestras para generar todas las combinaciones posibles de estados para los canales. Esta función emplea la representación binaria para las combinaciones, donde cada bit en la cadena binaria representa el estado de un canal.
Generación de Estados Aleatorios para Cada Canal en Cada Muestra:
Utilizando la función generar_estados_aleatorios, se crean estados aleatorios para cada canal en cada muestra. Se especifica el número de canales y el número de muestras como parámetros de entrada, y la función devuelve un diccionario que asigna a cada canal una lista de estados aleatorios para todas las muestras.
Cálculo de Matrices de Probabilidades:
Se emplean varias funciones para calcular diferentes matrices de probabilidades:

a. Matriz de Probabilidades para Estados Actuales y Siguientes (calcular_matriz_EstadoCanalF): Se itera a través de las muestras y se acumulan las ocurrencias de combinaciones de estados actuales y siguientes para cada canal. Luego, se calculan las probabilidades dividiendo el número de ocurrencias entre el número total de apariciones del estado actual.

b. Matriz de Probabilidades para Estados Previos (calcular_matriz_EstadoCanalP): Similar a la matriz de estados actuales y siguientes, se itera a través de las muestras, acumulando las ocurrencias de combinaciones de estados actuales y previos para cada canal. Las probabilidades se calculan de manera análoga.

c. Matriz de Transición entre Estados Actuales y Siguientes (generar_matriz_EstadoEstadoF): Se genera una matriz que representa las transiciones entre estados actuales y siguientes. Se cuentan las ocurrencias de estas transiciones y se calculan las probabilidades**.**

d. Matriz de Transición entre Estados Actuales y Previos (generar_matriz_EstadoEstadoP): Similar a la matriz de transición entre estados

actuales y siguientes, se generan probabilidades para las transiciones entre estados actuales y previos.

Presentación Gráfica de Resultados con la Biblioteca tabulate:

Se utiliza la función graphMatriz para presentar gráficamente los resultados. Esta función utiliza la biblioteca tabulate para mostrar las matrices de probabilidades de manera clara y organizada en formato de tabla. La presentación gráfica facilita la interpretación de las relaciones y probabilidades entre estados **y**

```
def graphMatriz(matriz, opc):
    if opc=="+":
        titulosColumnas=["Estado//T+1"]
    elif opc=="-":
        titulosColumnas=["Estado//T-1"]
    otrosTitulos=matriz[list(matriz.keys())[0]].keys()
    titulosColumnas.extend(otrosTitulos)

    table = [[key] + [matriz[key][column] for column in titulosColumnas[1:]] for key in matriz.keys()]

    print(tabulate(table, titulosColumnas, tablefmt="fancy_grid"))
```

Estrategia y Proceso de Partición y Probabilidades:

a. Función graphMatriz(matriz, opc) Esta función se encarga de presentar gráficamente los resultados mediante la biblioteca tabulate. Toma una matriz de probabilidades y una opción (opc) que indica si se trata de los estados actuales (+) o futuros (-). La función construye una tabla que muestra de manera organizada las probabilidades para cada estado y canal, facilitando la interpretación visual de las relaciones y patrones presentes.

b. Función menuDivision(data=sampleData, canales=canalesBase): La función menuDivision es la pieza central del código, guiando al usuario a través del proceso de partición de los canales en estados actuales y futuros. Presenta menús interactivos para seleccionar canales y asignar valores actuales, y permite al usuario realizar comprobaciones de distancia entre particiones. Utiliza funciones como allBiParticiones, printDataDivision, y final para llevar a cabo estos procesos.

c. Función final(data, canales): Esta función se encarga de realizar comparaciones entre las particiones actuales y futuras, calculando la distancia EMD (Earth Mover's Distance) para evaluar la similitud entre las distribuciones. Utiliza funciones auxiliares como printDataDivision y checkEmd para imprimir información detallada y realizar el cálculo de la distancia EMD.

d. Función checkEmd(original, matFinal, preStr): La función checkEmd calcula la distancia EMD entre la distribución original y las distribuciones finales generadas a partir de las particiones actuales y futuras. Utiliza la biblioteca scipy.stats para calcular la distancia y muestra los resultados, proporcionando información crucial sobre la similitud entre las distribuciones.

e. Función printDataDivision(data, canales, listActual, listFuturo): Esta función realiza operaciones para identificar y organizar las coincidencias entre los estados actuales y futuros. Utiliza funciones auxiliares como operProbs para calcular probabilidades y multpMatriz para multiplicar matrices de probabilidades.

f. Función auxOperProbs(data, actList, dataEntrada) La función auxOperProbs realiza operaciones específicas para calcular probabilidades a partir de datos proporcionados. Se utiliza en el proceso de cálculo de matrices de probabilidades y ayuda a organizar la información de manera adecuada.

g. Función multpMatriz(listas): Esta función realiza la multiplicación de matrices, siendo esencial para el cálculo de matrices de probabilidades en la estrategia general. Genera un diccionario con las combinaciones posibles de estados y canales, proporcionando una representación estructurada de los resultados.

h. Función operProbs(actlist, futList, data): La función operProbs realiza operaciones esenciales para calcular matrices de probabilidades a partir de las coincidencias entre estados actuales y futuros. Organiza la información y utiliza funciones auxiliares como multpMatriz para realizar cálculos fundamentales.

Estas funciones se interconectan de manera integral para implementar la estrategia definida y proporcionar una solución completa al problema de partición y cálculo de probabilidades

3. **Eficiencia de la solución**

La eficiencia de la solución puede depender de varios factores, y mi estimación se basa en una evaluación general del código proporcionado. Aquí hay algunos aspectos a considerar:

Complejidad Temporal:

- La generación de muestras tiene una complejidad de O(2^n), donde n es el número de canales.
- La generación de estados aleatorios tiene una complejidad de O(num_muestras * num_canales).
- El cálculo de matrices de probabilidades tiene una complejidad que depende de num_muestras y el número de canales.

  La generación de muestras presenta una complejidad exponencial de O(2^n), donde n es el número de canales. Este aumento exponencial se debe a que se están generando todas las combinaciones posibles de estados para cada canal, lo que puede resultar en un tiempo de ejecución prohibitivo a medida que aumenta el número de canales. Esta operación puede ser mejorada mediante

estrategias que reduzcan la cantidad de combinaciones generadas o mediante algoritmos más eficientes. La generación de estados aleatorios, con una complejidad de O(num_muestras * num_canales), es más eficiente en comparación y no presenta el mismo desafío exponencial. El cálculo de matrices de probabilidades, aunque no se proporcionan detalles específicos, generalmente dependerá tanto del número de muestras como del número de canales, lo que puede resultar en una complejidad significativa para conjuntos de datos grandes.

Complejidad Espacial:

- La generación de muestras y estados aleatorios requiere espacio proporcional al número de muestras y canales.
- Las matrices de probabilidades también ocupan espacio adicional.

La generación de muestras y estados aleatorios requiere espacio proporcional al número de muestras y canales. A medida que estos valores aumentan, los requisitos de memoria también lo hacen, lo que podría limitar la capacidad de manejar conjuntos de datos grandes en sistemas con recursos limitados. Además, las matrices de probabilidades también contribuyen al requisito de espacio adicional. Se podría explorar la posibilidad de implementar estrategias para reducir la carga espacial, como generar estados de manera más eficiente o almacenar matrices de manera compacta, sin comprometer la calidad de los resultados.

Número de Iteraciones:

- El código itera sobre todas las muestras y estados, lo que puede llevar a un aumento en el tiempo de ejecución, especialmente para conjuntos de datos grandes**.**

El código itera sobre todas las muestras y estados, lo que puede resultar en un tiempo de ejecución prolongado, especialmente para conjuntos de datos grandes. La complejidad de este enfoque puede afectar la eficiencia global del algoritmo. Se podría considerar la optimización de bucles y la aplicación de técnicas de paralelización para distribuir la carga de trabajo y acelerar la ejecución.

Uso de Bibliotecas:

- No parece haber uso de bibliotecas optimizadas para operaciones numéricas o manipulación de datos, lo que podría afectar el rendimiento en comparación con soluciones que aprovechan estas bibliotecas.

La ausencia de bibliotecas optimizadas, como NumPy, para operaciones numéricas y manipulación de datos, podría afectar la eficiencia. Estas bibliotecas están diseñadas para realizar operaciones de manera más rápida y eficiente que las implementaciones personalizadas, por lo que su integración podría mejorar significativamente el rendimiento del código.

Optimización:

- No se han aplicado técnicas avanzadas de optimización, como la paralelización de operaciones, que podrían mejorar la eficiencia en conjuntos de datos grandes.

  La falta de técnicas avanzadas de optimización, como la paralelización de operaciones, representa una oportunidad para mejorar la eficiencia. La paralelización permite realizar múltiples operaciones simultáneamente, lo que puede ser beneficioso en sistemas con múltiples núcleos de procesamiento. Aplicar estrategias de optimización podría reducir el tiempo total de ejecución y mejorar la eficiencia del algoritmo.

Tamaño de Entrada:

- La eficiencia puede verse afectada significativamente por el tamaño de entrada, especialmente el número de canales y muestras.

  La eficiencia se verá afectada por el tamaño de entrada, especialmente el número de canales y muestras. Mientras que la solución puede ser viable para conjuntos de datos moderados, se vuelve crucial explorar enfoques más eficientes para conjuntos de datos más grandes. Estrategias como el muestreo o la reducción de la dimensionalidad podrían ser consideradas para abordar conjuntos de datos extensos sin comprometer la calidad de los resultados.

4. **Dificultades encontradas:** El proceso de desarrollo de software conlleva inherentemente una serie de desafíos y obstáculos que requieren soluciones creativas y efectivas. Las dificultades pueden surgir en diversas etapas, desde la comprensión inicial del problema hasta la implementación y optimización del código. Estos desafíos no solo se limitan a la lógica algorítmica, sino que también abarcan aspectos prácticos como la interacción con el usuario, la manipulación de datos y la eficiencia del programa. En esta sección, explicaremos las dificultades encontradas durante el desarrollo del código, detallando cada obstáculo específico y la estrategia aplicada para superarlo. Estos desafíos, a menudo inevitables pero siempre superables, brindan una visión más completa del proceso de desarrollo de software y subrayan la importancia de la adaptabilidad y la resolución de problemas en este campo dinámico.

   - **Comprensión de la Problemática:**

Dificultad: Al inicio, la comprensión de la problemática presentó un desafío. La naturaleza técnica y específica del problema requería una serie de explicaciones adicionales para obtener una visión clara y precisa de los requisitos y objetivos del proyecto.
Solución: Se abordó esta dificultad mediante una comunicación más detallada y una explicación paso a paso. Proporcionar ejemplos concretos y contextualizar la información ayudó a superar las barreras iniciales en la comprensión de la problemática.

- **Generación de Matrices para Estados Actuales y Futuros:**
  Dificultad: La creación de matrices para representar los estados actuales y futuros de los canales presentó un desafío inicial. La comparación de probabilidades entre estados también resultó compleja al principio.
  Solución: La resolución de esta dificultad implicó un enfoque paso a paso. Se trabajó en el desarrollo de algoritmos específicos para generar y comparar estas matrices, abordando los desafíos asociados con la representación eficiente de la información y la manipulación de probabilidades.

- **Manejo de Entrada de Datos:**
  Dificultad: La entrada de datos se describió mediante números de canales y número de muestras. La interpretación adecuada de esta entrada para generar muestras y estados aleatorios planteó inicialmente un desafío en cuanto a la implementación precisa.
  Solución: Se abordó esta dificultad mediante una mayor clarificación en la interpretación de la entrada. Se desarrollaron funciones específicas para manejar la entrada de manera adecuada, asegurando la coherencia en la generación de muestras y estados aleatorios.

- **Presentación Gráfica de Resultados:**
  Dificultad: La presentación gráfica de resultados utilizando la biblioteca tabulate puede ser compleja debido a la necesidad de organizar y mostrar matrices de probabilidades de manera clara y comprensible.
  Solución: Para superar esta dificultad, se implementó una función llamada graphMatriz. Esta función utiliza la biblioteca tabulate para formatear las matrices de probabilidades en tablas, mejorando la legibilidad y facilitando la interpretación de los resultados.

- **Implementación de la Interfaz Gráfica y Manejo de Archivos:**
  Dificultad: Surgió la necesidad de incorporar una interfaz gráfica para facilitar la interacción con el usuario y permitir cálculos utilizando archivos planos, como xlsx y csv. La integración de esta funcionalidad planteó un desafío debido a la complejidad de la interfaz y la manipulación de archivos.
  Solución: Para superar esta dificultad, se utilizó la biblioteca tkinter, aprovechando sus módulos filedialog y messagebox. Estos componentes

proporcionan funcionalidades para la selección de archivos y la gestión de mensajes, respectivamente. La interfaz gráfica se diseñó de manera intuitiva, permitiendo al usuario cargar archivos de datos y realizar cálculos de manera sencilla. La integración de estas herramientas contribuyó significativamente a la usabilidad y versatilidad de la solución, extendiendo su funcionalidad más allá de la línea de comandos.

- **Entendimiento y Explicación de la Fórmula y Condiciones:**
  Dificultad: Esta radica en la comprensión y explicación adecuada de la fórmula utilizada y las condiciones que determinan su aplicación. La función aplicar_formula presenta condiciones que requieren una interpretación precisa, lo que podría generar confusión al entender en qué situaciones se debe aplicar la fórmula y cuándo no. Además, la lógica condicional involucrada puede resultar compleja, especialmente al tratar con múltiples condiciones que afectan la aplicabilidad de la fórmula.
  Solución: Para abordar esta dificultad, se sugiere proporcionar comentarios detallados dentro del código, explicando cada condición y el propósito detrás de su inclusión. Además, se puede crear documentación específica para la función aplicar_formula, detallando las circunstancias exactas bajo las cuales se aplica la fórmula y proporcionando ejemplos claros. Al agregar comentarios descriptivos y documentación, se facilitará la comprensión y la posterior explicación del funcionamiento de la fórmula y las condiciones a otros desarrolladores que interactúen con el código. Este enfoque mejora la legibilidad y asegura una interpretación más precisa de la lógica de la aplicación de la fórmula en el contexto del problema abordado.

Estas dificultades fueron superadas mediante una combinación de claridad en la comunicación, proporcionando ejemplos concretos y abordando los desafíos algorítmicos de manera incremental. La comprensión mejorada de la problemática y la resolución efectiva de los problemas específicos contribuyeron al desarrollo exitoso de la solución propuesta.

5. Conclusiones generales

Estrategia de Solución:
La estrategia de solución que se delineó para abordar la problemática de la generación y evaluación de estados en un sistema de canales se estructura en un enfoque metódico y detallado. En primer lugar, se reconoció la necesidad imperativa de establecer una representación clara y efectiva de los posibles estados dentro del sistema. La creación de una estructura de datos eficiente, adoptando matrices para modelar las complejas relaciones entre los estados, fue el fundamento esencial. Esta elección permitió una visualización matricial que facilita la interpretación y análisis

de las transiciones entre los estados, sentando las bases para un enfoque analítico sólido.

Propuesta de Solución:
La propuesta de solución se despliega en una serie de etapas cohesionadas que se articulan con la estrategia previamente definida. Comenzando con la generación meticulosa de muestras y estados aleatorios, el código aborda la complejidad del problema al crear combinaciones de estados para cada canal. La modularidad del diseño, con funciones específicas para la generación y evaluación de matrices de probabilidades, mejora la claridad y mantenibilidad del código. La introducción de bibliotecas especializadas, como NumPy para manipulación numérica y Tabulate para presentación visual de datos, añade eficiencia y legibilidad al flujo de trabajo.

Eficiencia de la Solución:
La eficiencia de la solución es un punto de suma importancia en la evaluación global del código. Un análisis detallado destaca varios factores clave. En términos de complejidad temporal, la generación de muestras presenta una complejidad de O(2^n), donde n es el número de canales, mientras que la generación de estados aleatorios es proporcional a la cantidad de muestras y canales (O(num_muestras * num_canales)). El cálculo de matrices de probabilidades tiene su propia complejidad, dependiendo del número de muestras y canales involucrados. En cuanto a complejidad espacial, la necesidad de almacenar muestras y estados aleatorios puede requerir espacio proporcional al tamaño de estos conjuntos, y las matrices de probabilidades también añaden demandas adicionales de memoria. El código itera sobre todas las muestras y estados, lo que puede impactar el tiempo de ejecución, especialmente para conjuntos de datos extensos. La falta de uso de bibliotecas optimizadas para operaciones numéricas o manipulación de datos podría afectar el rendimiento en comparación con soluciones que aprovechan estas bibliotecas. Además, no se aplicaron técnicas avanzadas de optimización, como la paralelización de operaciones, que podría haber mejorado la eficiencia, especialmente para grandes conjuntos de datos. La eficiencia, en última instancia, puede verse significativamente afectada por el tamaño del conjunto de datos, particularmente en términos del número de canales y muestras**.**

Dificultades:
El camino hacia la solución no estuvo exento de desafíos. Inicialmente, la comprensión completa de la problemática presentó dificultades, requiriendo aclaraciones y una exploración más profunda del problema. La creación de matrices para representar estados futuros y actuales también fue un punto de dificultad, donde las comparaciones de probabilidades iniciales generaron incertidumbre. La implementación de una interfaz gráfica y la posibilidad de realizar cálculos utilizando archivos planos, como xlsx y csv, añadieron capas adicionales de complejidad al proyecto. Sin embargo, cada dificultad fue enfrentada con resiliencia y adaptabilidad,

empleando estrategias específicas para superarlas. La claridad en la comunicación y la resolución iterativa de problemas demostraron ser esenciales para avanzar a través de estos obstáculos.

En resumen, la estrategia de solución, la propuesta implementada, las consideraciones de eficiencia y las dificultades encontradas convergen para constituir un enfoque robusto y escalonado para la generación y evaluación de estados en un sistema de canales. La adaptabilidad y creatividad exhibidas en la superación de dificultades resaltan la naturaleza dinámica del desarrollo de software, subrayando la importancia de un enfoque integral y la resolución proactiva de problemas en la resolución de problemas complejos**.**