

## Documentación de Proyecto de Grafos



Análisis y Diseño de Algoritmos

Profesor: Luz Enith Guerrero Mendieta

Miembros de Proyecto:  
Bryan Mauricio Gonzalez Giraldo  
Jhair Alexander Peña Aguirre

Universidad de Caldas

2024

## Especificación de Requisitos del Software (SRS)

### Requisitos Funcionales

La aplicación desarrollada fue hecha con el fin de cumplir tres aspectos principales, todos relacionados al análisis y manejo de grafos: Su creación y edición, su visualización y la ejecución de algoritmos de análisis de probabilidades.

#### Creación y Edición de Grafos:

El sistema permite la creación manual de grafos, donde los usuarios pueden agregar y eliminar nodos y aristas según sus necesidades. Además, el sistema también podrá generar grafos de manera automática basándose en parámetros que son ingresados por el usuario, como los que podrían ser la cantidad de nodos, o si el grafo es dirigido, ponderado, conexo o completo, lo que ofrece una gran flexibilidad y adaptabilidad a las necesidades del usuario.

#### Visualización de Grafos:

El sistema permitirá la visualización del grafo creado ya sea de manera gráfica o ver la información de estos grafos mediante tablas, los usuarios tendrán la opción de guardar, importar y exportar grafos en diferentes formatos, lo que facilita la gestión y el intercambio de datos. Los usuarios podrán visualizar nodos y aristas con diversos atributos, como etiquetas, colores, valores y direcciones. Esta funcionalidad mejora la comprensión y el análisis de los grafos.

#### Análisis de Grafos:

El aspecto más importante de esta aplicación va ser la ejecución de algoritmos capaces de analizar el estado que tiene el grafo actualmente, estos dos algoritmos solo tienen dos fines en concreto, uno de ellas es analizar el algoritmo y detectar si el grafo presente es bipartito y las diferentes componentes que tiene el grafo actualmente; el otro análisis va más enfocado al análisis de estados y probabilidades en el cual se buscará identificar cual partición del sistema es el que genera la menor pérdida.

### Requisitos No Funcionales

#### Usabilidad:

La interfaz de usuario debe de ser intuitiva y fácil de usar. Esto significa que los usuarios podrán navegar y utilizar el sistema sin dificultades. Además, se proporcionará una guía de usuario y una documentación adecuada para ayudar a los usuarios a entender y utilizar mejor el sistema.

#### Rendimiento:

El sistema se diseñó para manejar grafos de tamaño mediano a grande. Esto significa que el sistema será capaz de procesar y analizar estos grafos con tiempos de respuesta aceptables, asegurando así una experiencia de usuario fluida.

#### **Portabilidad:**

El sistema será compatible con los sistemas operativos más comunes, incluyendo Windows, macOS y Linux. Esto garantiza que el mayor número posible de usuarios pueda acceder y utilizar el sistema.

#### **Seguridad:**

Para prevenir inyecciones de código, el sistema validará todas las entradas de usuario. Además, todas las comunicaciones se realizan a través de HTTPS para garantizar la seguridad de los datos transmitidos.

### **Capa de Presentación (Front-End)**

La capa de presentación está desarrollada con “**Streamlit**” y está encargada de la interacción con el usuario.

#### **Tecnologías Utilizadas:**

- **Streamlit:** Para construir la interfaz gráfica de usuario (GUI).

#### **Estructura de las Vistas:**

- **Página Principal:** Permite al usuario crear y editar grafos.
  - Menú de opciones: Crear grafo, cargar grafo, guardar grafo.
  - Área de trabajo: Visualización interactiva del grafo.
- **Página de Configuración:** Permite configurar las propiedades del grafo a crear.
- **Página de Ejecución:** Permite seleccionar y ejecutar algoritmos sobre el grafo creado.

#### **Interacción del Usuario:**

- **Menús Contextuales:** Para agregar, eliminar y renombrar nodos y aristas.
- **Formularios:** Para ingresar parámetros de generación automática de grafos.

### **Capa de Lógica de Negocio (Back-End)**

La lógica de negocio se implementa en Python y se encarga del procesamiento y manipulación de los grafos.

#### **Tecnologías Utilizadas:**

- **Python:** Lenguaje de programación principal.
- **NetworkX:** Para la creación y manipulación de grafos.

- **Pandas:** Para la creación de tablas y exportación de archivos Excel.
- **NumPy:** Para cálculos numéricos y operaciones con matrices.
- **SciPy:** Para cálculos de distribución de probabilidades.
- **PyAutoGUI:** Para capturas de pantalla y exportación.

### **API y Servicios Utilizados:**

- El sistema no utiliza servicios externos o APIs REST, ya que es un monolito y todo el procesamiento se realiza dentro de la misma aplicación.

### **Lógica de Procesamiento:**

- **Creación de Grafos:** Funciones para crear grafos manualmente o automáticamente según los parámetros del usuario.
- **Algoritmos de Grafos:** Implementación de algoritmos para determinar si un grafo es bipartito, identificar componentes conexas y analizar la desconexión de grafos.

### **Componentes del Sistema**

1. **Front-End:**
  - **Interfaz de Usuario:** Desarrollada con **Streamlit**, permite la interacción con el usuario.
  - **Visualización de Grafos:** Utiliza “**streamlit-agraph**” para mostrar grafos de forma interactiva, el cual está basado en el manejo de grafos de “**viz.js**”.
2. **Back-End:**
  - **Manejo de Grafos:** Utiliza NetworkX para la creación y manipulación de grafos.
  - **Procesamiento de Datos:** Utiliza NumPy y SciPy para cálculos y operaciones numéricas.
  - **Exportación de Datos:** Utiliza Pandas para exportar datos a archivos Excel.
  - **Capturas de Pantalla:** Utiliza PyAutoGUI para capturas de grafos.

### **Comunicación entre Componentes**

La comunicación entre el front-end y el back-end se realiza de manera interna dentro del monolito, utilizando funciones y métodos de Python.

### **Gestión de Solicitudes y Respuestas:**

- **Interacción Directa:** Streamlit maneja la interacción directa con los componentes de back-end. Cuando un usuario realiza una acción en la interfaz (e.g., crear un grafo, ejecutar un algoritmo), Streamlit llama a las funciones correspondientes en el back-end.
- **Actualización de la Interfaz:** Los resultados del procesamiento en el back-end (e.g., resultados de algoritmos, visualización de grafos) se devuelven al front-end y se actualizan en la interfaz de usuario en tiempo real.

## Diseño de Algoritmos:

### 1. Generación de tabla de distribución de probabilidades:

#### - Descripción General del Algoritmo:

En este conjunto de algoritmos se buscará generar una tabla de distribución de probabilidades basado en un conjunto de estados que se ingresa al sistema, este conjunto de datos se presenta de la siguiente manera:

E.Futuro			A		E.Futuro			B		E.Futuro			C	
E.Actual			0	1	E.Actual			0	1	E.Actual			0	1
A	B	C			A	B	C			A	B	C		
0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	1	0	1	0	0	1	0	1	0	0	0	1
0	1	0	0	1	0	1	0	1	0	0	1	0	0	1
1	1	0	0	1	1	1	0	0	1	1	1	0	1	0
0	0	1	0	1	0	0	1	0	1	0	0	1	1	0
1	0	1	0	1	1	0	1	1	0	1	0	1	0	1
0	1	1	0	1	0	1	1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	1	0	1	1	1	1	1	0

Los estados se representan de dos formas, el estado cuando está en el presente y el estado cuando este va estar en el futuro, por lo cual los datos representarán los valores futuros que tendrá cada estado, los valores en las entradas representan probabilidades, siendo el valor de “1” un 100% y “0” un 0%, por lo cual en este ejemplo el valor futuro cuando los estados valen (1,0,0) es (0,1,1). Para representar estas tablas en el sistema se ingresará de la siguiente forma:

"A": {  
 (0, 0, 0): 0,  
 (1, 0, 0): 0,  
 (0, 1, 0): 1,  
 (1, 1, 0): 1,  
 (0, 0, 1): 1,  
 (1, 0, 1): 1,  
 (0, 1, 1): 1,  
 (1, 1, 1): 1,  
 },

"B": {  
 (0, 0, 0): 1,  
 (1, 0, 0): 1,  
 (0, 1, 0): 1,  
 (1, 1, 0): 1,  
 (0, 0, 1): 1,  
 (1, 0, 1): 0,  
 (0, 1, 1): 1,  
 (1, 1, 1): 0,  
 },

"C": {  
 (0, 0, 0): 0,  
 (1, 0, 0): 1,  
 (0, 1, 0): 1,  
 (1, 1, 0): 0,  
 (0, 0, 1): 0,  
 (1, 0, 1): 1,  
 (0, 1, 1): 1,  
 (1, 1, 1): 0,  
 },

Para ahorrar la cantidad de datos que se cargan solo se tomará el valor futuro que tendrá el estado, en vez de usar los dos, esto facilitará también la identificación del estado futuro completo que tendrá el sistema.

La clave será el nombre que tendrá el estado, y como es que se representarán en el sistema y cómo es que se ingresarán en la interfaz.

Se buscará que este conjunto de datos se una mediante producto tensor para así obtener el sistema completo de probabilidades. El sistema completo se visualizará de la siguiente forma:

	A	0	1	0	1	0	1	0	1
	B	0	0	1	1	0	0	1	1
E. Actual	C	0	0	0	0	1	1	1	1
A	B	C							
0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0
1	1	0	0	1	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1
0	1	1	0	0	0	1	0	0	0
1	1	1	0	0	1	0	0	0	0

#### - Análisis de Complejidad:

Para cumplir con este problema se usarán dos funciones:

- **'generated\_state\_transitions'**: Este algoritmo está encargado de detectar los estados futuros completos de cada estado presente completo. Este algoritmo recibe como datos el conjunto de estados mostrados anteriormente y retorna un conjunto con todas las transiciones y una lista con los estados presentes en el sistema.
- **'generarTablaDistribuida'**: Algoritmo encargado de generar la tabla del sistema completo basado en la información obtenida en el algoritmo **'generate\_state\_transitions'**, por lo cual la entrada de datos en este algoritmo será el conjunto de transiciones.

#### Algoritmo diseñado:

**'generated\_state\_transitions'**: Algoritmo para generar una lista de adyacencia entre los estados presentes y los estados futuros.

Python

```
generate_state_transitions(subconjuntos):
    estados = lista de las claves de subconjuntos
    transiciones = diccionario vacío
```

```
    keys = unión de todos los conjuntos en los valores de subconjuntos
```

```

for clave en keys:
    transiciones[clave] = tupla(subconjuntos[subconjunto].obtener(clave, 0) para
        cada subconjunto en subconjuntos)

return transiciones, estados

```

Debido a que en el algoritmo es uno iterativo, donde se recorre todos los estados del sistema y todas posibles combinaciones que puedan tener estos estados, por ende en cuanto a la complejidad podemos decir que estas dado por estas iteraciones.

**n:** Cantidad de estados del sistema.

**m:** Cantidad de combinaciones del sistema.

#### **Complejidad temporal:**

$T(n, m) = O(n \times m)$ , podemos que esta será la complejidad debido a que se hacen iteraciones a ambas variables.

#### **Complejidad Espacial:**

1. **Almacenamiento del diccionario de transiciones:** El diccionario de transiciones tendrá **n** entradas, una por cada clave original. Cada entrada es una tupla de longitud **m**. Por lo tanto, la complejidad espacial es  $O(n \times m)$ .
2. **Espacio adicional para las tuplas generadas:** Durante la construcción del diccionario, se generan temporalmente las tuplas de longitud **m**. Sin embargo, estas son de corta duración y se desechan inmediatamente después de ser utilizadas para construir el diccionario de transiciones.

La complejidad espacial dominante es el almacenamiento del diccionario de transiciones, que es:

$$E(n, m) = O(n \times m).$$

Por lo cual podemos decir que las complejidades espacial y temporal de este algoritmo en específico es de:

**n:** Cantidad de estados en el sistema.

**m:** Cantidad de combinaciones presentes en el sistema.

$$T(n, m) = O(n \times m)$$

$$E(n, m) = O(n \times m)$$

### Algoritmo diseñado:

**'generarTablaDistribuida':** Algoritmo para generar la tabla de distribución de probabilidades completa.

Python

```
generarTablaDistribucion(int conjuntos(n)){
    combinaciones[m] = sort(conjuntos[n])
    indices = Crear un diccionario vacio con las llaves
    matriz = Se crea una matriz de (n+1) x (n+1), con todos los elementos siendo
    igual a 0, y es (n+1), debido a que tambien se incluyen las combinaciones a
    cada columna y a cada fila.

    for llave y valor en los elementos de conjuntos:
        i = indices[llave]
        j = indices[llave], si no existe asignar un -1 a j
        if (j <> -1) then
            matriz[i+1][j+1] = 1
        end if
    return matriz
}
```

Debido a que el algoritmo también es uno iterativo y se tiene que recorrer toda la matriz para obtener todos los resultados de la tabla, el código no puede omitir ninguna iteración de llenado en la tabla.

**n:** Siendo la cantidad de combinaciones que tiene el conjunto de transiciones que se generaron en '**generated\_state\_transitions**'.

### Complejidad temporal:

**T(n):**  $O(n)$ , esto es debido a que el código hace iteraciones sobre este diccionario para así obtener tanto la clave como el valor. Como es solo una iteración podemos asumir que la complejidad temporal de este sistema es lineal.

### Complejidad espacial:

Utilizamos un diccionario para mapear cada llave única a su índice correspondiente en la matriz. El espacio ocupado por este mapeo será también  $O(n)$ , ya que hay  $n$  llaves únicas en total.

**E(n):**  $O(n)$

Por lo cual podemos decir que las complejidades espacial y temporal de este algoritmo en específico es de:

**n:** Cantidad de estados en el sistema.

**m:** Cantidad de combinaciones presentes en el sistema.

$$T(n, m) = O(n)$$

$$E(n, m) = O(n)$$

- **Justificación de la estrategia:**

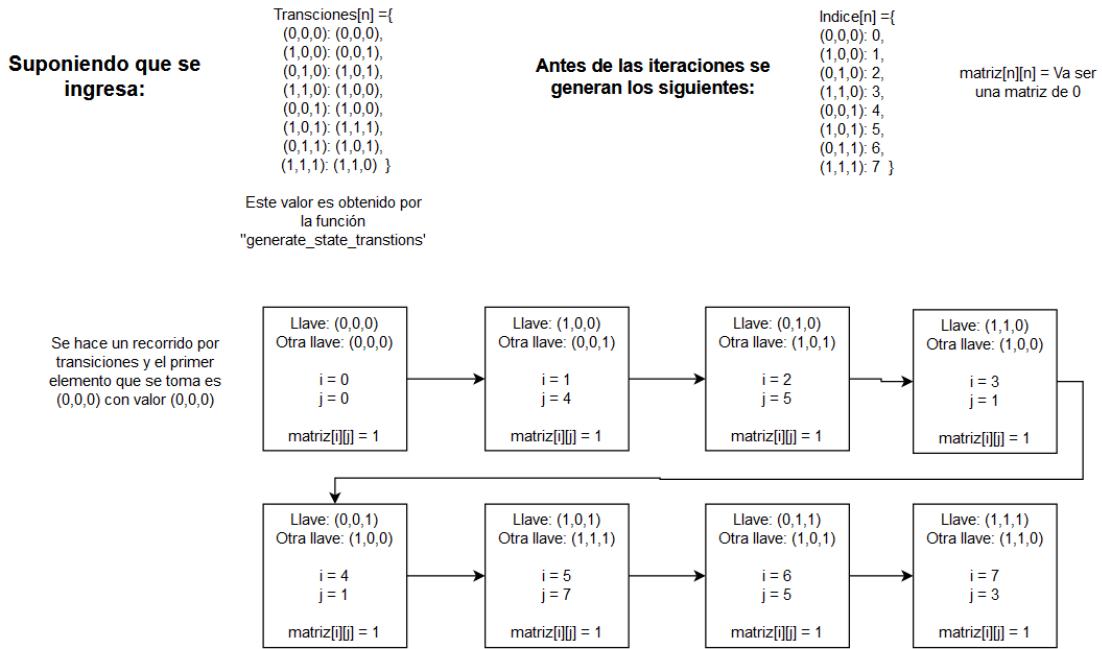
La razón del enfoque iterativo que se usa en este problema es ahorrarnos la necesidad de calcular la multiplicación de múltiples matrices, ya que si usa una fuerza bruta la complejidad de trabajo que puede efectuar el sistema puede aumentar exponencialmente mientras más cantidad de estados estén presentes en el sistema.

El poder detectar las adyacencias que tendrá cada estado presente con respecto al futuro ayuda a reducir la cantidad de trabajo si generamos la tabla de probabilidades de inmediato con fuerza bruta, se presentarán una gran cantidad de ciclos anidados que puede aumentar la complejidad de trabajo.

Si se hacía este trabajo con backtracking, también se hubiera podido evidenciar una complejidad de trabajo exponencial, en este contexto se presentaría una complejidad de  $O(2^n)$ , esto es debido a que es necesario hacer rellenar toda la matriz resultante.

- **Diagramas:**

**Diagrama para ‘generarTablaDistribuida’:** Se puede evidenciar que la cantidad de instancias que se crean se genera un número pequeño.



### - Manejo de casos especiales y límites:

#### Valores Extremos:

**Valores Nulos o Vacíos:** Este algoritmo tiene que recibir unos valores de entrada en específico para poder calcular de una forma adecuada las probabilidades del sistema completo, es por eso que si se llegan a ingresar valores no válidos o nulos, el algoritmo simplemente no ejecutará ni dará ningún resultado.

Cómo se gestionan las entradas que no cumplen con los requisitos esperados. Si se llegan a ingresar entradas de combinaciones incorrectas el programa ya está preparado para no validar esta combinación, y dejando esa probabilidad como una de 0.

#### Pruebas y Validación del Algoritmo:

- Definir conjuntos de datos pequeños y medir el tiempo de ejecución del algoritmo para revisar que tan eficiente es la solución incluso en escenarios básicos.

Vamos a utilizar los mismos datos que usamos anteriormente, esto corresponde a una red de 3 nodos:

```
"A": {
  (0, 0, 0): 0,
  (1, 0, 0): 0,
  (0, 1, 0): 1,
  (1, 1, 0): 1,
  (0, 0, 1): 1,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 1,
  },
  "B": {
    (0, 0, 0): 1,
    (1, 0, 0): 1,
    (0, 1, 0): 1,
    (0, 0, 1): 1,
    (1, 0, 1): 1,
    (0, 1, 1): 1,
    (1, 1, 0): 1,
    (1, 1, 1): 1,
    },
  "C": {
    (1, 1, 0): 1,
    (0, 0, 1): 1,
    (1, 0, 1): 0,
    (0, 1, 1): 1,
    (1, 1, 1): 0,
    },
  }
```

$(0, 0, 0): 0,$	$(1, 1, 0): 0,$	$(0, 1, 1): 1,$
$(1, 0, 0): 1,$	$(0, 0, 1): 0,$	$(1, 1, 1): 0,$
$(0, 1, 0): 1,$	$(1, 0, 1): 1,$	$\},$

Con estos datos nos deben de dar los mismos resultados que planteamos anteriormente:

			A	0	1	0	1	0	1	0	1
			B	0	0	1	1	0	0	1	1
			C	0	0	0	0	1	1	1	1
A	B	C									
0	0	0		1	0	0	0	0	0	0	0
1	0	0		0	0	0	0	1	0	0	0
0	1	0		0	0	0	0	0	1	0	0
1	1	0		0	1	0	0	0	0	0	0
0	0	1		0	1	0	0	0	0	0	0
1	0	1		0	0	0	0	0	0	0	1
0	1	1		0	0	0	0	0	1	0	0
1	1	1		0	0	0	1	0	0	0	0

### Datos en código:

<b>"A": {</b>	<b>"B": {</b>	<b>"C": {</b>
$(0, 0, 0): 0,$	$(0, 0, 0): 0,$	$(0, 0, 0): 0,$
$(1, 0, 0): 0,$	$(1, 0, 0): 0,$	$(1, 0, 0): 1,$
$(0, 1, 0): 1,$	$(0, 1, 0): 0,$	$(0, 1, 0): 1,$
$(1, 1, 0): 1,$	$(1, 1, 0): 0,$	$(1, 1, 0): 0,$
$(0, 0, 1): 1,$	$(0, 0, 1): 0,$	$(0, 0, 1): 0,$
$(1, 0, 1): 1,$	$(1, 0, 1): 1,$	$(1, 0, 1): 1,$
$(0, 1, 1): 1,$	$(0, 1, 1): 0,$	$(0, 1, 1): 1,$
$(1, 1, 1): 1,$	$(1, 1, 1): 1,$	$(1, 1, 1): 0,$
<b>},</b>	<b>},</b>	<b>},</b>

Solo con estos datos ya podremos ejecutar el algoritmo, nos da la siguiente tabla de distribución, vemos que las adyacencias están colocadas de una manera adecuada, y el tiempo obtenido es de 0.000953 segundos:

Selecciona un subconjunto:

- Tres
- Cuatro
- Cinco
- Seis

Tiempo de generación de tabla de distribución: `0.00095367431640625`

$$P(ABC^{t+1} | ABC^t)$$

	0	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')
0	0 0 0	1	0	0	0	0	0	0	0
1	0 0 1	0	0	0	0	1	0	0	0
2	0 1 0	0	0	0	0	0	1	0	0
3	0 1 1	0	0	0	0	0	1	0	0
4	1 0 0	0	1	0	0	0	0	0	0
5	1 0 1	0	0	0	0	0	0	0	1
6	1 1 0	0	0	0	0	1	0	0	0
7	1 1 1	0	0	0	0	0	0	1	0

- 2) Para conjuntos de datos más grandes evaluar cómo la solución escala en términos de tiempo de ejecución y uso de recursos

Ahora se usará una red de seis estados:

Se evidencia que la cantidad de datos aumenta significativamente a comparación de una red de 3 estados.

Con estas entradas se generará los siguientes resultados:

Como se ve se retornará una tabla (que no se ve completa en la imagen debido a lo grande que puede llegar a ser esta tabla), mientras que el tiempo aumentó en 0.00316 segundos.

Selecciona un subconjunto:

- Tres
- Cuatro
- Cinco
- Seis

Tiempo de generación de tabla de distribución: 0.003160238265991211

		$P(ABCDEF^{t+1}   ABCDEF^t)$											
		(0', 0', 0', 0', 0', 0')	(0', 0', 0', 0', 0', 1')	(0', 0', 0', 0', 1', 0')	(0', 0', 0', 0', 1', 1')	(0', 0', 0', 1', 0', 0')	(0', 0', 0', 1', 0', 1')	(0', 0', 0', 1', 1', 0')	(0', 0', 0', 1', 1', 1')	(0', 0', 1', 0', 0', 0')	(0', 0', 1', 0', 0', 1')	(0', 0', 1', 0', 1', 0')	(0', 0', 1', 0', 1', 1')
0		0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	1	0	0	0	0	0	0	
2	0	0	0	0	1	0	1	0	0	0	0	0	
3	0	0	0	0	1	1	0	0	0	0	0	0	
4	0	0	0	1	0	0	0	0	0	0	0	0	
5	0	0	0	1	0	1	0	0	0	0	0	0	
6	0	0	0	1	1	0	0	0	0	0	0	0	
7	0	0	0	1	1	1	0	0	0	0	0	0	
8	0	0	1	0	0	0	0	0	0	0	0	0	
9	0	0	1	0	0	1	0	0	0	0	0	0	

## 2. Generación de subsistemas:

### Descripción del algoritmo:

Este algoritmo busca en base a la distribución probabilidades, la creación de subsistemas en base al sistema más grande, estos subsistemas son tomando una cantidad de estados tanto futuros como presentes, y un valor inicial como por ejemplo (1,0,0). Por lo cual las entradas de este algoritmo van a ser los mismo subconjuntos que se aplicaron en la anterior estrategia, con otra cantidad de datos más, como lo son los estados presentes que va a tomar el algoritmo, los estados futuros y el valor que tendrá el estado presente.

E.Futuro			A			E.Futuro			B			E.Futuro			C		
E.Actual			0 1			E.Actual			0 1			E.Actual			0 1		
A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0
1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0
1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	0	0
0	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	1	0	1	0	1	1	0	0	1	0	1	0	1	0
0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0	1	0
1	1	1	0	1	0	1	1	1	0	0	1	1	1	0	1	0	0

Valores de los subconjuntos.

También se ingresarán los valores de estados futuros, como estamos usando un sistema de 3 estados, podemos tomar por ejemplo la B y la C.

También se tienen que ingresar los valores de los estados presentes, como podrías ser el estado A, junto al valor que tendrá este estado presente, podemos de usar de ejemplo el 1.

En este caso el subsistema que se busca obtener será el siguiente:

$$P(BC^{(t+1)} | A = 1)$$

Lo que se busca con este problema es generar la tabla que represente este subsistema, utilizando la marginalización de tablas. Entonces usando el ejemplo que tenemos, solo tomaríamos las tablas de B y C, que son los estados futuros ingresados.

E.Futuro			B	
E.Actual			0	1
A	B	C		
0	0	0	0	1
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	0	1

E.Futuro			C	
E.Actual			0	1
A	B	C		
0	0	0	1	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

A las tablas se les aplica marginalización eliminando las columnas que no estén presentes en los estados presentes ingresados. Por ejemplo en la tabla de B se marginaliza las columnas de B y C, ya que A es el único estado presente que se ingresó. y a los valores estados que queden igual se sumarán y se dividirán entre dos.

#### Marginalización de C:

E. Futuro			B	
E.Actual			0	1
A	B			
0	0		1	0
1	0		1	0
0	1		1	0
1	1		1	0
0	0		1	0
1	0		0	1
0	1		1	0
1	1		0	1

E. Futuro			B	
E.Actual			0	1
A	B			
0	0		1	0
1	0		0,5	0,5
0	1		1	0
1	1		0,5	0,5

#### Marginalización de B:

E. Futuro	B	
E. Actual	0	1
A		
0	1	0
1	0,5	0,5
0	1	0
1	0,5	0,5

E. Futuro	B	
E. Actual	0	1
A		
0	1	0
1	0,5	0,5

Como el valor que se ingreso es A = 1, solo se toma esa fila.

E. Futuro	B	
E. Actual	0	1
A		
1	0,5	0,5

Aplicamos la misma lógica para la tabla de C y nos dará el siguiente resultado.

E. Futuro	C	
E. Actual	0	1
A		
1	0,5	0,5

Teniendo ya las tablas ya marginalizadas, ahora se hace producto tensor para obtener el subsistema que se busca.

Tabla que representa el subsistema:  $P(BC^{(t+1)} | A = 1)$

E. Futuro	C		C	
E. Actual	"00"	"10"	"01"	"11"
A				
1	0,25	0,25	0,25	0,25

### Análisis de Complejidad:

Para cumplir con este problema se usarán varias funciones:

- '**generarDistribucionProbabilidades
- '**porcentajeDistribuido****

- ‘generarTabla’: Algoritmo encargado de crear la tabla de distribución, una vez se hayan hecho las marginalizaciones.

### Algoritmo Diseñado:

#### ‘porcentajeDistribuido’

Python

```
def porcentajeDistribuido(self, tabla, posiciones, num):
    nueva_tabla = [tabla[0]]
    tabla_filtrada = [
        fila
        for fila in tabla[1:]
        if all(fila[0][pos] == num[i] for i, pos in enumerate(posiciones))
    ]

    valores = [0, 0]
    for fila in tabla_filtrada:
        valor1 = fila[1]
        valor2 = fila[2]
        valores[0] += valor1
        valores[1] += valor2

    valores = [valor / len(tabla_filtrada) for valor in valores]

    nueva_fila = [num, *valores]
    nueva_tabla.append(nueva_fila)

    return nueva_tabla
```

### Complejidad temporal:

Este algoritmo también tendrá un enfoque iterativo, donde se recorra la tabla de valores ingresados y se buscarán las llaves similares que tienen entre ellas una vez se hayan marginalizado, por lo cual:

**n:** El número de combinaciones posibles en el sistema.

**m:** Número de posiciones a verificar en cada lista.

En el algoritmo se presente dos ciclos principales en los cuales, el primero para la ‘**tabla\_filtrada**’

- Aquí, `tabla[1:]` tiene  $m$  filas (donde  $m$  es el número de filas en `tabla` menos uno).
- La condición de filtrado `all(fila[0][pos] == num[i] for i, pos in enumerate(posiciones))` tiene una complejidad de  $O(n)$  en el peor caso, donde  $n$  es el número de elementos en `posiciones`.

Por lo tanto, la complejidad del marginalización es  **$O(m * n)$** .

### Complejidad espacial:

- ‘tabla\_filtrada’ es una lista que puede contener hasta m filas, donde m es el número de filas en tabla menos uno (ya que excluimos la primera fila).
- Cada fila ocupa un espacio constante porque su tamaño no depende de m o n (asumiendo que las filas tienen un tamaño fijo).
- El resto de sentencias tienen una complejidad espacial de **O(1)** debido a que

Por lo cual el espacio para la tabla filtrada es de **O(m)**.

Por lo cual podemos asumir que la función “**porcentajeDistribuido**” el análisis de complejidad es de la siguiente forma:

$$T(n, m) = O(n * m)$$

$$E(n, m) = O(m)$$

### ‘generarTabla’

Este algoritmo es el encargado de una vez se haya hecho las marginalizaciones en cada una de las tablas ingresadas mediante un producto tensor, este algoritmo usará la lógica de backtracking para hacer el llenado de la tabla.

```
Unset
def generarTabla(
    self, distribucionProbabilidades, num, i=0, binario="", nuevo_valor=1, memo =
None):
    if memo is None: memo = {}
    key = (i, binario, nuevo_valor)
    if key in memo: return memo[key]

    if i == len(distribucionProbabilidades):
        binario = "0" * (len(distribucionProbabilidades) - len(binario)) + binario
        nueva_tupla = tuple(int(bit) for bit in binario)
        result = [[nueva_tupla], [nuevo_valor]]
    else:
        tabla1 = self.generarTabla(distribucionProbabilidades, num, i + 1,
                                    binario + "0", nuevo_valor * distribucionProbabilidades[i][1][2],memo)

        tabla2 = self.generarTabla(distribucionProbabilidades, num, i + 1,
                                    binario + "1", nuevo_valor * distribucionProbabilidades[i][1][1],memo)
        result = [tabla1[0] + tabla2[0], tabla1[1] + tabla2[1]]

    memo[key] = result
    return result
```

### Análisis de complejidad

## Complejidad Temporal

1. Caso base de la recursión: Cuando  $i$  es igual a la longitud de **distribucionProbabilidades**, se genera una tupla binaria y se devuelve una lista que contiene esta tupla y **nuevo\_valor**. Esta operación es de complejidad constante **O(1)**.
2. Recursión y combinación de tablas: En cada llamada recursiva, el método se llama a sí mismo dos veces, incrementando  $i$  en 1. Esto significa que la recursión genera un árbol binario completo con una profundidad de  $\text{len}(\text{distribucionProbabilidades})$ , resultando en  $2^n$  hojas (donde  $n$  es  $\text{len}(\text{distribucionProbabilidades})$ ).

Combinar tabla1 y tabla2 implica concatenar listas, cada una de las cuales puede contener hasta  **$2^{n-1}$**  elementos en el peor de los casos, lo que resulta en una operación  **$O(2^n)$** .

**n:** La longitud de **distribucionProbabilidades**.

Por lo cual la complejidad temporal del algoritmo se da en  **$O(2^n)$** .

## Complejidad espacial

Espacio para binario y nuevo\_valor:

- El parámetro binario crece en longitud con cada llamada recursiva, pero su tamaño máximo es  $n$  (la longitud de **distribucionProbabilidades**). Poseyendo una complejidad de  **$O(n)$**

## Espacio para las listas combinadas:

- Las combinaciones de las tablas tabla1 y tabla2 en cada nivel de la recursión también resultan en listas de tamaño creciente, hasta llegar a  $2^n$  en el nivel más alto.

En resumen, la complejidad espacial total del algoritmo es:  **$O(2^n)$**

Entonces la complejidad temporal y espacial que tiene este algoritmo es de:

$$T(n) = O(2^n)$$

$$E(n) = O(2^n)$$

La complejidad  **$O(2^n)$**  en este código se debe a la naturaleza del problema que intenta resolver: generar todas las combinaciones posibles de una secuencia de bits (binario) de longitud  $n$  y sus correspondientes valores. Este problema intrínsecamente tiene  $2^n$  combinaciones posibles, por lo que la complejidad no se puede evitar si todas las combinaciones deben ser generadas y almacenadas, sin embargo el memorizado que se aplica ayuda a restarle complejidad al ejercicio.

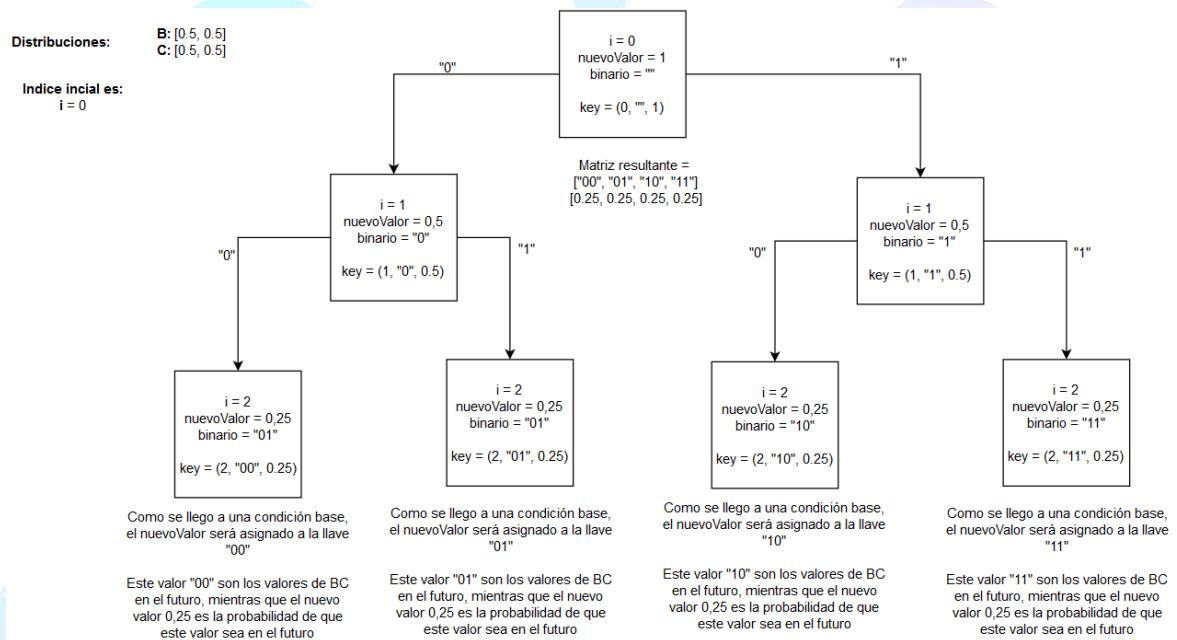
## Justificación de la estrategia

La estrategia de memorización (memoization) usada en la implementación de **generarTabla** se justifica en comparación con otras estrategias debido a los siguientes factores:

Evitar la producción de subproblemas ya que problemas de combinatoria, especialmente en la generación de todas las combinaciones posibles, se encuentran muchos subproblemas que se resuelven repetidamente y con la memorización se almacena los resultados de estos subproblemas, evitando cálculos redundantes y reduciendo el tiempo de ejecución global.

La recursión sin memorización tiene una complejidad exponencial debido a la naturaleza del problema y con memorización, cada subproblema se resuelve una sola vez. Aunque el número de subproblemas únicos es todavía exponencial, la cantidad de trabajo se reduce significativamente, por último esta estrategia puede ser más eficiente en términos de uso de pila porque evita el uso de la recursión profunda.

### Diagramas:



### Valores Extremos:

**Valores Nulos o Vacíos:** La manera en cómo este algoritmo puede evitar el ingreso de valores nulos o no válidos es mediante la interfaz gráfica, donde el programa solo le dará la opción a los usuarios de seleccionar las únicas opciones válidas en el sistema (Hablando específicamente de los valores estados futuros, presentes y el valor actual que tendrán estos valores presentes).

En cuanto al ingreso de los subconjuntos, el programa solo tomará estas entradas como unas no válidas, y no ejecutará el algoritmo.

### Pruebas y Validación del Algoritmo:

- 3) Definir conjuntos de datos pequeños y medir el tiempo de ejecución del algoritmo para revisar que tan eficiente es la solución incluso en escenarios básicos.

Los resultados que nos deben de dar al ingresar los siguientes datos:

**Entrada:**

```
"A": {
  (0, 0, 0): 0,
  (1, 0, 0): 0,
  (0, 1, 0): 1,
  (1, 1, 0): 1,
  (0, 0, 1): 1,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 1,
},
"B": {
  (0, 0, 0): 1,
  (1, 0, 0): 1,
  (0, 1, 0): 1,
  (1, 1, 0): 1,
  (0, 0, 1): 1,
  (1, 0, 1): 0,
  (0, 1, 1): 1,
  (1, 1, 1): 0,
},
"C": {
  (0, 0, 0): 0,
  (1, 0, 0): 1,
  (0, 1, 0): 1,
  (1, 1, 0): 0,
  (0, 0, 1): 0,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 0,
},
```

**Estados futuros = B, C**

**Estados presentes = A, con valor de 1.**

**Los resultados que debemos obtener:**

E. Futuro	C		C	
E. Actual	"00"	"10"	"01"	"11"
A				
1	0,25	0,25	0,25	0,25

Como se ingresan los datos al programa:

```
"A": {
  (0, 0, 0): 0,
  (1, 0, 0): 0,
  (0, 1, 0): 1,
  (1, 1, 0): 1,
  (0, 0, 1): 1,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 1,
},
"B": {
  (0, 0, 0): 0,
  (1, 0, 0): 0,
  (0, 1, 0): 0,
  (1, 1, 0): 0,
  (0, 0, 1): 0,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 0,
},
"C": {
  (0, 0, 0): 0,
  (1, 0, 0): 1,
  (0, 1, 0): 1,
  (1, 1, 0): 0,
  (0, 0, 1): 0,
  (1, 0, 1): 1,
  (0, 1, 1): 1,
  (1, 1, 1): 0,
},
```

Así es como es que se ingresan los valores al programa.

Estados presentes:	Estados futuros:	Selecciona el valor presente:	Generar distribución
<input type="button" value="A x"/>	<input type="button" value="B x"/> <input type="button" value="C x"/>	<input type="button" value="(1,)"/>	

**Tabla resultante:** Como se ve la tabla es exactamente igual a como se había calculado anteriormente

P(BC <sup>t+1</sup>   A <sup>t</sup> = (1,))	
['A'] \ ['B', 'C']	('0', '0')
0	1

**Si usamos más estados el tiempo de ejecución es el siguiente:**

Estados presentes:		Estados futuros:			Selecciona el valor presente:					
<b>A</b> x	<b>B</b> x	x	v	<b>B</b> x	<b>C</b> x	<b>A</b> x	x	v	(1, 0)	v
<b>P(ABC<sup>t+1</sup>   AB<sup>t</sup> = (1, 0))</b>										
Tiempo de ejecución de generacion de subconjunto: <b>0.00098562240600058594</b>										
	[A', B'] \ [A', B', C']	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')	
0	1	0	0	0.25	0	0.25	0	0.25	0	0.25

**Si seguimos aumentando la cantidad de datos, esta vez con una red de 6 estados.**

**Y este sería un resultado con sus tiempos.**

Estados presentes:

A ✕ E ✕ D ✕ C ✕ B ✕
✖️
▼

Estados futuros:

F ✕ E ✕ C ✕ B ✕
✖️
▼

Selecciona el valor presente:

(0, 0, 0, 1, 1, 0)
✖️
▼

Generar distribución

$P(BCEF^{t+1} | ABCDEF^t = (0, 0, 0, 1, 1, 0))$

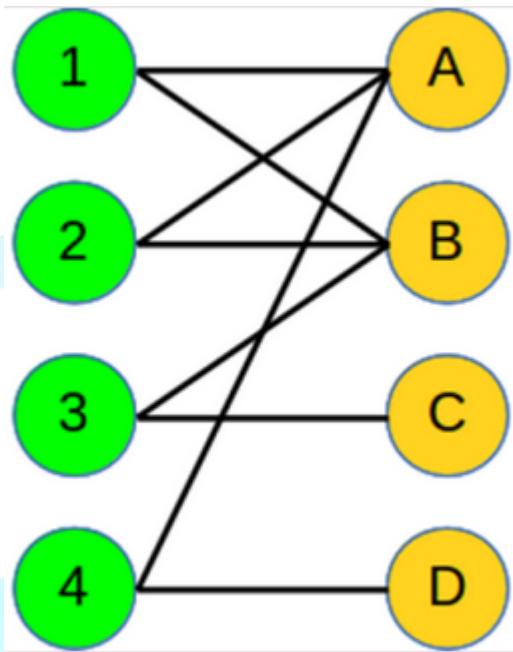
  

Tiempo de ejecución de generación de subconjunto: 0.000985860824584961

[A, 'B', 'C', 'D', 'E', 'F'] \ [B, 'C', 'E', 'F']		(0, '0', '0', '0')	(0, '0', '0', '1')	(0, '0', '1', '0')	(0, '0', '1', '1')	(0, '1', '0', '0')	(0, '1', '0', '1')	(0, '1', '1', '0')	(0, '1', '1', '1')	(1, '0', '0', '0')	(1, '1', '0', '0')	
0	0	0	0	1	1	0	1	0	0	0	0	0

### **3. Análisis de grafos bipartitos y sus componentes:**

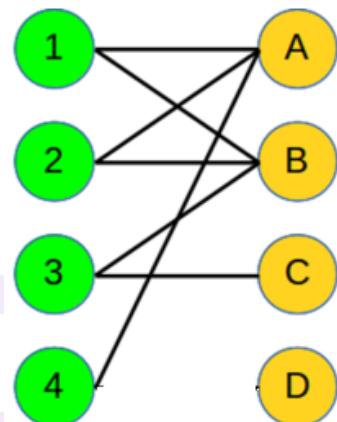
Ya habiendo creado un grafo en el programa, el objetivo de este algoritmo es el análisis de bipartición para este grafo creado, los datos que espera retornar este algoritmo es si un grafo es bipartito o no, y cuántas son las componentes que componen el grafo bipartito.



Un grafo bipartito es un tipo de grafo donde los vértices se pueden dividir en dos conjuntos disjuntos  $U$  y  $V$  tales que cada arista conecta un vértice en  $U$  con uno en  $V$ . No existen aristas que conecten vértices dentro del mismo conjunto. Esto significa que es posible colorear el grafo con solo dos colores, de tal manera que no hay dos vértices adyacentes del mismo color.

En cuanto a las componentes se explican como las partes conexas que tiene cada grafo bipartito, por ejemplo en el caso del ejemplo, solo existen una componente en el sistema, sin embargo si se llega a presentar algún nodo que no tenga alguna arista, este grafo seguirá siendo bipartito pero se ubicará en una componente distinta.

**Primera componente:** {[1, 2, 3, 4][A, B, C]} **Segunda componente :** {[][D]}



**Análisis de complejidad:**

```
Python
def create_adjacency_list(self, nodes, edges):
```

```

adj = {node.id: [] for node in nodes}
for edge in edges:
    adj[edge.source].append(edge.to)
    adj[edge.to].append(edge.source)
return adj

```

### Complejidad temporal:

- La inicialización de adj toma  $O(V)$  tiempo porque estamos iterando sobre cada nodo.
- El bucle que itera sobre edges toma  $O(E)$  tiempo, porque cada arista se procesa una vez.
- En total, la complejidad temporal es  $O(V+E)$ .

### Complejidad espacial:

- Se usa un diccionario para almacenar la lista de adyacencia, lo que ocupa  $O(V+E)$  espacio (para almacenar cada nodo y sus conexiones).
- 1) Definir conjuntos de datos pequeños y medir el tiempo de ejecución del algoritmo para revisar que tan eficiente es la solución incluso en escenarios básicos.

Trabajaremos con un tamaño de entrada de datos que corresponde a una red de 4 nodos.

- Representación del sistema original

Estados presentes:
A ✕ B ✕ C ✕ D ✕
x
▼
A ✕ B ✕ C ✕ D ✕
x
▼
(1, 0, 0, 0)
▼

$P(ABCD^{t+1} | ABCD^t = (1, 0, 0, 0))$

- Primera componente generada

$P(D^{t+1} | t = [])$

	0	1	2
0	$\emptyset \setminus ['D']$	$(0,)$	$(1,)$
1	$\emptyset$	1.0	0.0

- Segunda componente generada

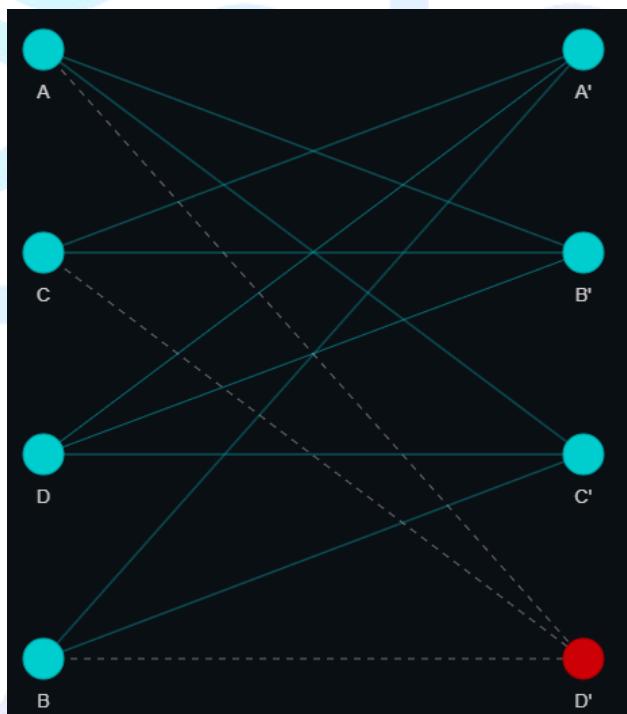
		0	1	2	3	4	5	6
		[A', B', C', D'] \ [A', B', C']	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)
0	[A', B', C', D'] \ [A', B', C']	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)	
1	[1, 0, 0, 0]	0.0	1.0	0.0	0.0	0.0	0.0	

- Medición del tiempo y pérdida del sistema partido respecto al sistema original

Tiempo de ejecución: 0.008 segundos

El emd es: 0.0

- Representación del sistema partido, las aristas puntaagudas representan las aristas que se han eliminado y que generan una pérdida de 0.



- 4) Para conjuntos de datos más grandes evaluar cómo la solución escala en términos de tiempo de ejecución y uso de recursos

Para estas pruebas realizaremos la evaluación de la estrategia 1 para una red de 6 nodos

Estados presentes:      Estados futuros:      Selecciona el valor presente:

A ×	B ×	C ×	D ×	E ×
F ×				

A ×	B ×	C ×	D ×	E ×
F ×				

(1, 0, 0, 0, 1, 0)

$P(ABCDEF^{t+1} | ABCDEF^t = (1, 0, 0, 0, 1, 0))$

- Primera componente generada

$$P(D^{t+1} | t = [] )$$

	0	1	2
0	[] \ ['D']	(0,)	(1,)
1	[]	1.0	0.0

- Segunda componente generada

$$P(EFABC^{t+1} | ABCDEF^t = [1, 0, 0, 0, 1, 0])$$

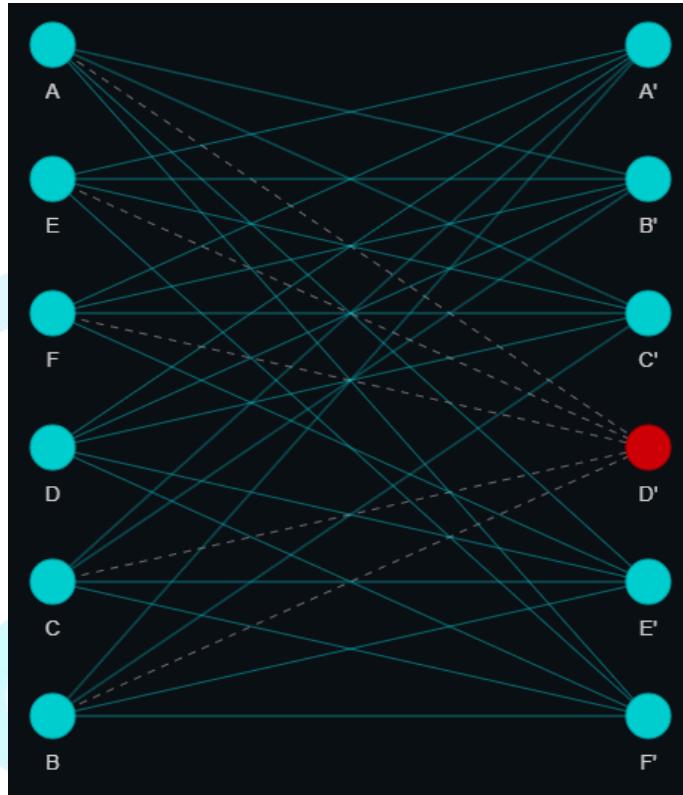
	0	1	2	3	4
0	[A', 'B', 'C', 'D', 'E', 'F'] \ ['E', 'F', 'A', 'B', 'C']	(0, 0, 0, 0, 0)	(0, 0, 0, 0, 1)	(0, 0, 0, 1, 0)	(0,
1	[1, 0, 0, 0, 1, 0]	0.0	1.0	0.0	0.0

- Medición del tiempo y pérdida del sistema partido respecto al sistema original

Tiempo de ejecución: 0.0131 segundos

El emd es: 0.0

- Representación del sistema partido, las aristas puntaagudas representan las aristas que se han eliminado y que generan una pérdida de 0.



- 5) Comparar la eficiencia de las soluciones propuestas con un enfoque de “fuerza bruta”. La solución propuesta debería ser significativamente más eficiente.

La fuerza bruta implica generar todas las posibles combinaciones, a continuación se realizan pruebas entre las estrategias versus la fuerza bruta.

Estados presentes:	Estados futuros:	Selecciona el valor presente:
<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="x"/> <input type="button" value="v"/>	<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="x"/> <input type="button" value="v"/>	<input type="button" value="(1, 0, 0, 0)"/> <input type="button" value="v"/>

$$P(ABCD^t + 1 \mid ABCD^t = (1, 0, 0, 0))$$

- Con fuerza bruta aplicada a una red de 4 nodos

Tiempo de ejecución: 0.0386 segundos

El emd es: 0.0

- Ahora en una red de 4 nodos usando la estrategia 1 tenemos

Tiempo de ejecución: 0.008 segundos

El emd es: 0.0

- Ahora en una red de 4 nodos usando la estrategia 2 tenemos

Valor de perdida: 0

Tiempo de ejecución: 0.007 segundos

- Ahora en una red de 4 nodos usando la estrategia 3 tenemos

Tiempo de ejecución: 0.0095 segundos

El emd es: 0.0

Estados presentes:	Estados futuros:	Selecciona el valor presente:
<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="E x"/> <input type="button" value="F x"/>	<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="E x"/> <input type="button" value="F x"/>	(1, 0, 0, 0, 1, 0)

$$P(ABCDEF^{t+1} | ABCDEF^t = (1, 0, 0, 0, 1, 0))$$

- Con fuerza bruta aplicada a una red de 6 nodos

Tiempo de ejecución: 1.1695 segundos

El emd es: 0.0

- Ahora en una red de 6 nodos usando la estrategia 1 tenemos

Tiempo de ejecución: 0.0135 segundos

El emd es: 0.0

- Ahora en una red de 6 nodos usando la estrategia 2 tenemos

Valor de perdida: 0

Tiempo de ejecución: 0.0385 segundos

- Ahora en una red de 6 nodos usando la estrategia 3 tenemos

Tiempo de ejecución: 0.0131 segundos

El emd es: 0.0

Estados presentes:	Estados futuros:	Selecciona el valor presente:
<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="E x"/> <input type="button" value="F x"/>	<input type="button" value="A x"/> <input type="button" value="B x"/> <input type="button" value="C x"/> <input type="button" value="D x"/> <input type="button" value="E x"/> <input type="button" value="F x"/> <input type="button" value="G x"/> <input type="button" value="H x"/>	(1, 0, 0, 0, 1, 0)

- Con fuerza bruta aplicada a una red de 8 nodos

Tiempo de ejecución: 20.7261 segundos

El emd es: 0.0

- Con la estrategia 1 aplicada a una red de 8 nodos

Tiempo de ejecución: 0.0286 segundos

El emd es: 0.0

- Con la estrategia 2 aplicada a una red de 8 nodos

Valor de perdida: 0

Tiempo de ejecución: 0.0608 segundos

- Con la estrategia 3 aplicada a una red de 8 nodos

Tiempo de ejecución: 0.0368 segundos

El emd es: 0.0

- 6) Verificar que los resultados obtenidos por el algoritmo son correctos y proporcionan la mínima pérdida de información esperada.

Python

```
def is_bipartite(self, node, conjunto, visitados, conjuntos, adj):
    visitados[node] = True
    conjuntos[node] = conjunto
    for vec in adj[node]:
        if not visitados[vec]:
            if not self.is_bipartite(vec, 1 - conjunto, visitados, conjuntos, adj):
                return False
        elif conjuntos[vec] == conjuntos[node]:
            return False
    return True
```

#### Complejidad temporal:

- Esta función utiliza una búsqueda en profundidad (DFS), lo que implica que cada nodo y cada arista son visitados una vez.
- La complejidad temporal es  $O(V+E)$ .

#### Complejidad espacial:

- La profundidad de la recursión puede llegar a ser  $O(V)$  en el peor caso, lo que da una complejidad espacial de  $O(V)$  debido a la pila de recursión y los diccionarios auxiliares visitados y conjuntos.

Python

```
def check_bipartite(self, nodes, edges):
    adj = self.create_adjacency_list(nodes, edges)
    visitados = {node.id: False for node in nodes}
    conjuntos = {node.id: -1 for node in nodes}
```

```

for node in nodes:
    if not visitados[node.id]:
        if not self.is_bipartite(node.id, 0, visitados, conjuntos, adj):
            return False
return True

```

### Complejidad temporal:

- `create_adjacency_list` toma  $O(V+E)$ .
- La inicialización de `visitados` y `conjuntos` toma  $O(V)$ .
- El bucle que llama a `is_bipartite` puede recorrer todos los nodos y aristas, resultando en  $O(V+E)$  por cada llamada a `is_bipartite`.
- En total, la complejidad temporal es  $O(V+E)$ .

### Complejidad espacial:

- Similar a `is_bipartite`, la complejidad espacial es  $O(V+E)$  debido a las estructuras de datos auxiliares y la pila de recursión.

Python

```

def find_connected_components(self, nodes, edges):
    adj = self.create_adjacency_list(nodes, edges)
    visitados = {node.id: False for node in nodes}
    componente = []
    for node in nodes:
        if not visitados[node.id]:
            componenteConjuntos = self.componente(node.id, visitados, adj)
            componente.append(componenteConjuntos)
    return componente

```

### Complejidad temporal:

- `create_adjacency_list` toma  $O(V+E)$ .
- La inicialización de `visitados` toma  $O(V)$ .
- El bucle principal y las llamadas a `componente` cubren todos los nodos y aristas una vez, resultando en  $O(V+E)$  por cada llamada.
- En total, la complejidad temporal es  $O(V+E)$ .

### Complejidad espacial:

- La complejidad espacial es  $O(V+E)$  debido a las estructuras de datos auxiliares utilizadas.

Python

```
def componente(self, node, visitados, adj):
    visitados[node] = True
    conjunto1list = []
    conjunto2list = []
    stack = [(node, 0)]
    while stack:
        node, conjunto = stack.pop()
        if conjunto == 0:
            conjunto1list.append(node)
        else:
            conjunto2list.append(node)
        for vec in adj[node]:
            if not visitados[vec]:
                visitados[vec] = True
                stack.append((vec, 1 - conjunto))
    return conjunto1list, conjunto2list
```

#### Complejidad temporal:

- Esta función también realiza una búsqueda en profundidad utilizando una pila en lugar de recursión.
- Cada nodo y arista se visita una vez, resultando en  $O(V+E)$ .

#### Complejidad espacial:

- La pila puede tener un tamaño máximo de  $O(V)$  y las listas conjunto1list y conjunto2list pueden contener hasta  $V$  nodos, lo que resulta en  $O(V)$ .
- En total, la complejidad espacial es  $O(V)$ .

Python

```

def marcarAristas(self, lista1, lista2, lista11, lista22, optionep, optionef):
    lista2 = [i + "" for i in lista2]

    edges_to_remove = []
    for edge in st.session_state["edges"]:
        source_node = st.session_state["nodes"][edge.source]
        to_node = st.session_state["nodes"][edge.to]

        source_node_color = (
            "#00FFFF"
            if source_node.label in lista1
            else "#FF0000" if source_node.label in lista11 else "#FFFFFF"
        )
        to_node_color = (
            "#00FFFF"
            if to_node.label in lista2
            else "#FF0000" if to_node.label[0] in lista22 else "#FFFFFF"
        )

        source_node.color = source_node_color
        to_node.color = to_node_color

        edge_color = (
            "#00FFFF"
            if (source_node.label in lista1 and to_node.label[0] not in lista22)
            else (
                "#FFFFFF"
                if (source_node.label in lista1 and to_node.label[0] in lista22)
                or (source_node.label in lista11 and to_node.label in lista2)
                else "#FF0000"
            )
        )
        edge.color = edge_color
        edge.dashes = edge_color == "#FFFFFF"

        if source_node.label not in optionep or to_node.label[0] not in optionef:
            edges_to_remove.append(edge)

    st.session_state["edges"] = [
        edge for edge in st.session_state["edges"] if edge not in edges_to_remove
    ]

```

**Complejidad Temporal:**  $O(E \times k)$ , donde  $k$  es el tamaño máximo de las listas proporcionadas.

**Complejidad Espacial:**  $O(E + \text{len}(lista2))$ .

## Diseño y estructura de la primera estrategia:

### 1. generar\_particion\_aleatoria

Esta función genera una partición inicial aleatoria de los estados presentes y futuros.

#### Parámetros:

- estados\_presentes: Lista de estados actuales.
- estados\_futuros: Lista de posibles estados futuros.
- valores\_estados\_presentes: Lista de valores asociados a los estados presentes.

#### Salida:

- Una tupla de seis listas que representan dos particiones (ep1, ef1, vp1, ep2, ef2, vp2).

### 2. calcular\_costo

Esta función calcula el costo de una partición dada.

- **Parámetros:**
  - particion: Tupla con las listas de las dos particiones.
  - subconjunto, original, listaNodos: Datos adicionales necesarios para calcular las distribuciones de probabilidad.
- **Salida:**
  - El costo mínimo entre dos cálculos de distancia de Hamming (emd1, emd2) y las distribuciones de probabilidad r1 y r2.

### 3. generar\_vecino

Esta función genera una nueva partición (vecino) a partir de una partición dada, siguiendo diferentes fases de intercambio de elementos.

- **Parámetros:**
  - particion: Tupla con las listas de la partición actual.
  - fase: Entero que indica en qué fase del intercambio se encuentra.
- **Salida:**
  - Una nueva partición generada mediante intercambios de elementos.

### 4. busqueda\_local\_emd

Esta es la función principal que realiza la búsqueda local utilizando la métrica de Earth Mover's Distance (EMD).

#### Parámetros:

- estados\_presentes, estados\_futuros, valores\_estados\_presentes, subconjunto, listaNodos, original\_system: Datos necesarios para generar particiones y calcular costos.

### **Salida:**

- La mejor partición encontrada, el costo asociado y las distribuciones de probabilidad r1 y r2.

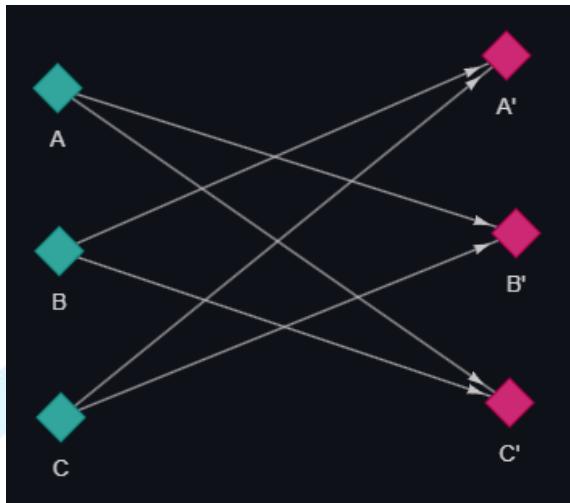
### **Funcionamiento:**

- Se genera una partición inicial aleatoria, en la que inicialmente ep1 se encontrará vacío, mientras que ef1 tendrá 1 elemento, para los demás estados estarán repartidos entre ep2 y ef2.
- Se calcula el costo de esta partición inicial, se evalua si este resultado corresponde a una pérdida de 0 respecto al sistema original, si no es así se añade a un diccionario para ir teniendo referencias de particiones ya generadas y evitar recalcular esas particiones.
- Se intenta mejorar la partición mediante la generación de vecinos. La generación de vecinos se da en 3 fases, en la fase 1, se realizan cambios entre ef1 y ef2, el elemento que estaba en la partición inicial en ef1 pasa a ef2 y el siguiente en ef2 pasa a ef1, la fase inicial una vez se han realizado todos los movimientos entre ef1 y ef2, se vacía ef1 moviendo ese elemento a ef2 y empezamos a realizar intercambio de a un elemento entre ep1 y ep2, una vez culminados los cambios entre ep1 y ep2, pasamos a la fase 3, se añade un elemento a ef1 y ep1, se realizan intercambios ahora entre ep1 con ep2 y ef1 con ef2, estos intercambios son de a un elemento, para cada partición que se genera se evalúa si esa partición es válida, qué particiones no serán válidas, por ejemplo las que son A y A', B y B', y así sucesivamente.
- Se evalúa cada partición de las generadas buscando pérdidas de 0 para terminar la ejecución, caso contrario se añade al diccionario y se continúa hasta que no se observe una mejora significativa o se alcance el número máximo de iteraciones.
- Esto se lleva a cabo máximo hasta alcanzar el máximo de iteraciones que está definido como la multiplicación entre la cantidad entre estados presentes y futuros, adicionado a la suma de las cantidades de estados en estados presentes y estados futuros.

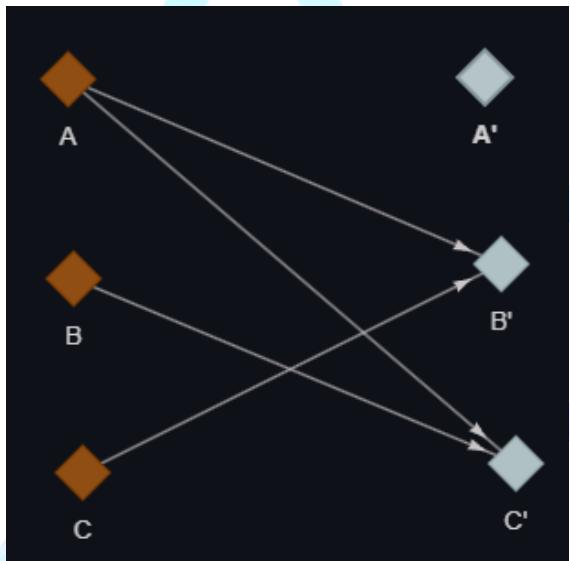
### **A continuación se hace un pequeño seguimiento mediante la interfaz:**

Supongamos que tenemos un sistema original representado con {A, B, C}, en los estados presentes y con {A', B', C'}, en los estados futuros y los valores de los estados presentes es

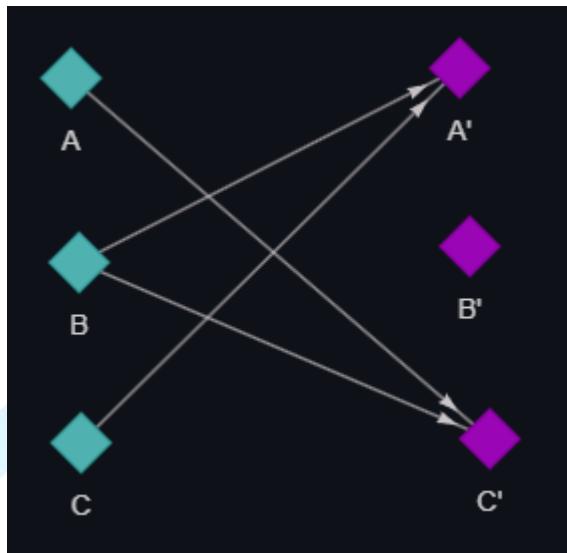
[1, 0, 0].



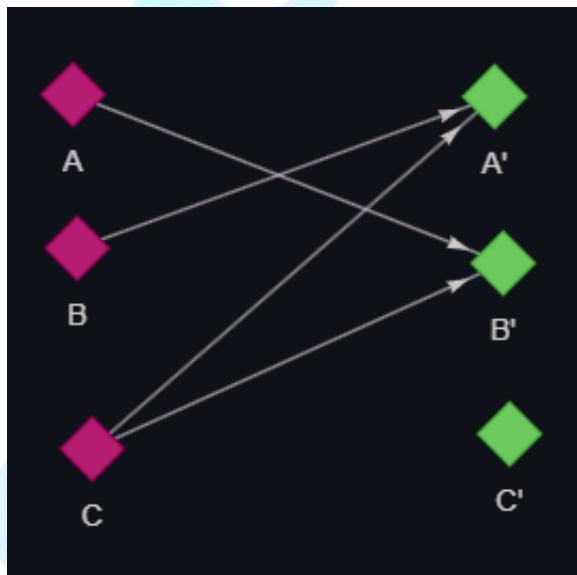
La partición inicial que se genera será la siguiente, con ep1=[ ], ef1=[A'] vp1=[ ], ep2=[A, B, C], ef2=[B', C'] vp2=[1, 0, 0]



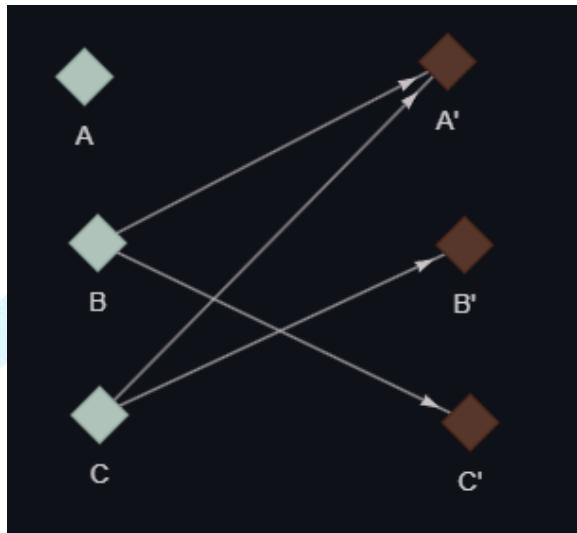
La siguiente partición que se genera será la siguiente, con ep1=[ ], ef1=[B'] vp1=[ ], ep2=[A, B, C], ef2=[A', C'] vp2=[1, 0, 0]



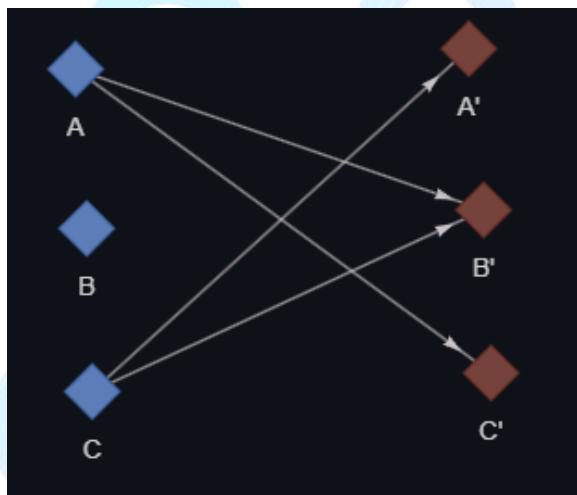
La siguiente partición que se genera será la siguiente, con ep1=[ ], ef1=[C'] vp1=[ ], ep2=[A, B, C], ef2=[B', C'] vp2=[1, 0, 0]



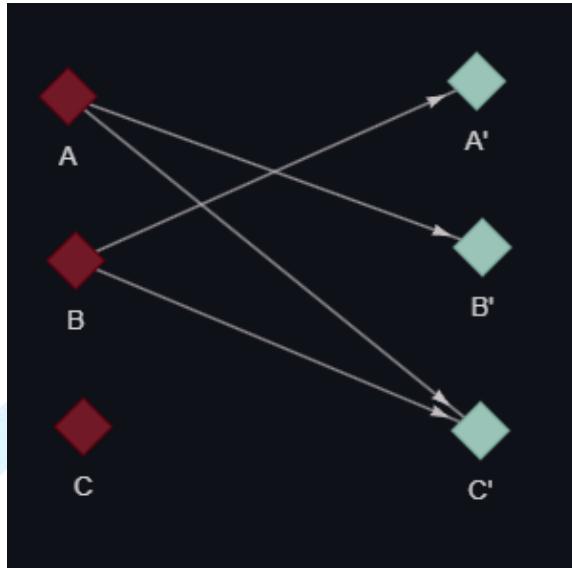
La siguiente partición que se genera será la siguiente, con  $ep1=[A]$ ,  $ef1=[]$   $vp1=[1]$ ,  $ep2=[B, C]$ ,  $ef2=[A', B', C']$   $vp2=[0, 0]$



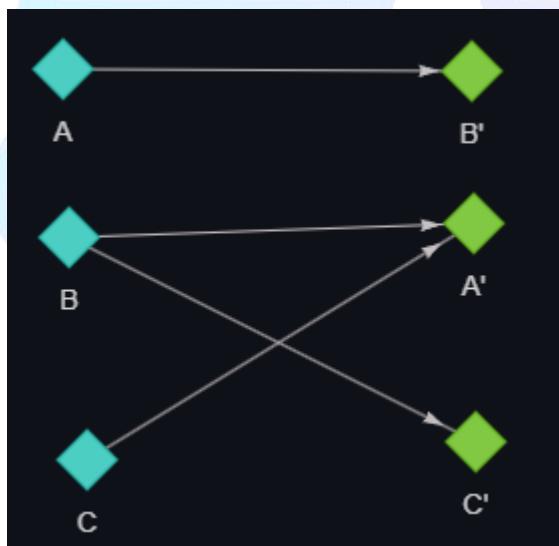
La siguiente partición que se genera será la siguiente, con  $ep1=[B]$ ,  $ef1=[]$   $vp1=[0]$ ,  $ep2=[A, C]$ ,  $ef2=[A', B', C']$   $vp2=[1, 0]$



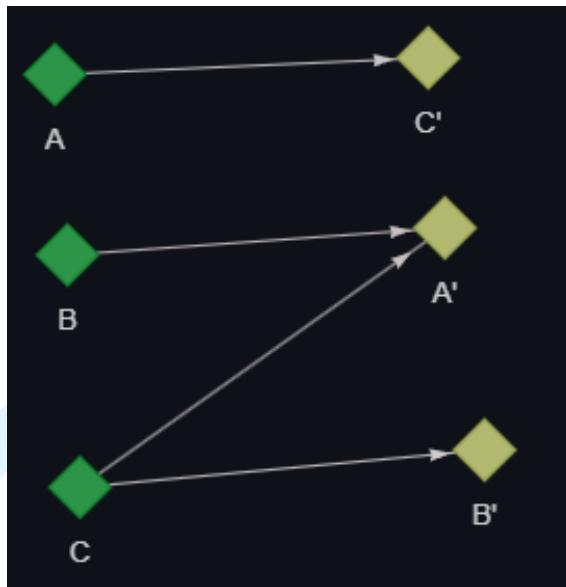
La siguiente partición que se genera será la siguiente, con  $ep1=[C]$ ,  $ef1=[]$   $vp1=[0]$ ,  $ep2=[A, B]$ ,  $ef2=[A', B', C']$   $vp2=[1, 0]$



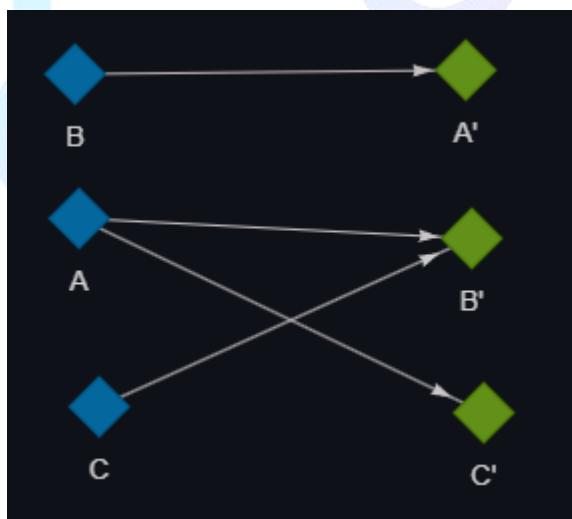
La siguiente partición que se genera será la siguiente, con ep1=[A], ef1=[B'] vp1=[1], ep2=[B, C], ef2=[A', C'] vp2=[0, 0]



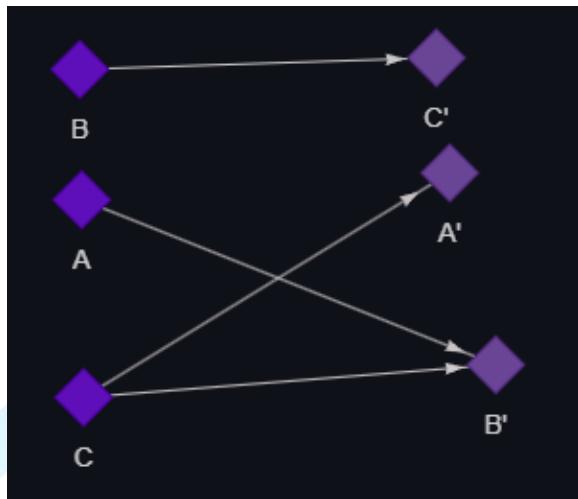
La siguiente partición que se generá será la siguiente, con ep1=[A], ef1=[C'] vp1=[1], ep2=[B, C], ef2=[A', B'] vp2=[0, 0]



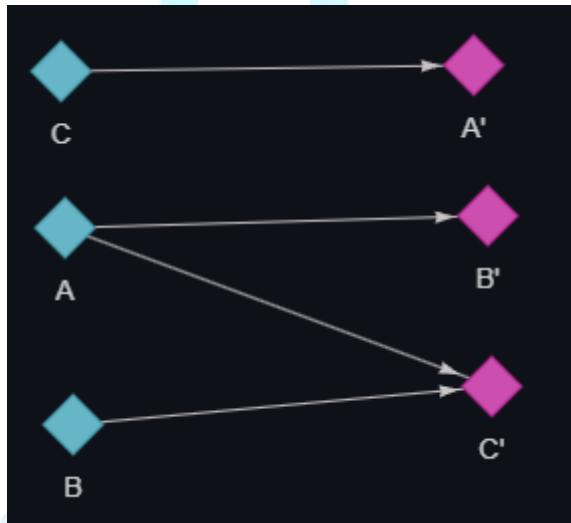
La siguiente partición que se genera será la siguiente, con  $ep1=[B]$ ,  $ef1=[A']$   $vp1=[0]$ ,  $ep2=[A, C]$ ,  $ef2=[B', C']$   $vp2=[0, 0]$



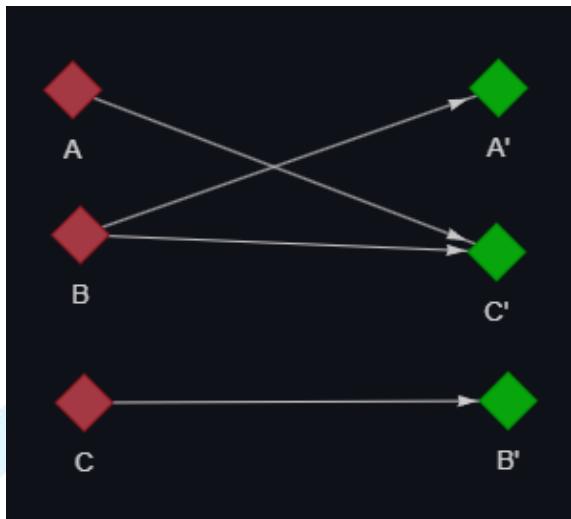
La siguiente partición que se genera será la siguiente, con  $ep1=[B]$ ,  $ef1=[C']$   $vp1=[0]$ ,  $ep2=[A, C]$ ,  $ef2=[A', B']$   $vp2=[0, 0]$



La siguiente partición que se genera será la siguiente, con  $ep1=[C]$ ,  $ef1=[A']$   $vp1=[0]$ ,  $ep2=[A, B]$ ,  $ef2=[B', C']$   $vp2=[0, 0]$



Finalmente la partición que se genera será la siguiente, con  $ep1=[C]$ ,  $ef1=[B']$   $vp1=[0]$ ,  $ep2=[A, B]$ ,  $ef2=[A', C']$   $vp2=[0, 0]$



Python

```

def calcular_costo(self, particion, subconjunto, original, listaNodos):
    ep1, ef1, vp1, ep2, ef2, vp2 = particion

    if len(ep1) > 0:
        ep1, vp1 = zip(*sorted(zip(ep1, vp1)))

    ef1 = sorted(ef1)
    ep2, vp2 = zip(*sorted(zip(ep2, vp2)))
    ef2 = sorted(ef2)

    r1 = P.generarDistribucionProbabilidades(subconjunto, ep1, ef1, vp1, listaNodos)
    r2 = P.generarDistribucionProbabilidades(subconjunto, ep2, ef2, vp2, listaNodos)

    tensor1 = np.tensordot(r2[1][1:], r1[1][1:], axes=0).flatten()
    tensor2 = np.tensordot(r1[1][1:], r2[1][1:], axes=0).flatten()

    emd1 = hamming(original[1][1:], tensor1)
    emd2 = hamming(original[1][1:], tensor2)
    print("Particion: ", particion, "EMD: ", min(emd1, emd2))
    return min(emd1, emd2), r1, r2
  
```

### Complejidad Temporal

Ordenar una lista de tamaño  $n$  tiene una complejidad de  $O(n \log n)$

En este caso, cada lista ( $ep1, ef1, ep2, ef2$ ) se ordena individualmente, por lo que la complejidad combinada es  $O(n \log n)$  para cada lista

$\text{np.tensordot}$  tiene una complejidad  $O(m^{**}2)$  si los vectores de entrada tienen longitud  $m$ .  $\text{flatten}$  tiene una complejidad  $O(m^2)$  ya que aplanar un tensor de tamaño  $m \times m$  es lineal en el número de elementos.

La distancia Hamming entre dos vectores de tamaño  $m$  tiene una complejidad  $O(m)$ .

## Complejidad Espacial

Las listas ordenadas ( $ep_1, ef_1, ep_2, ef_2$ ) ocupan  $O(n)$  espacio cada una.

El espacio total ocupado por las listas es  $O(n)$ .

tensor ocupa  $O(m^2)$  espacio cada uno, ya que son tensores aplanados de tamaño  $m \times m$

La complejidad espacial total es  $O(2n + m^2)$ .

Python

```
def generar_vecino(self, particion, fase):
    ep1, ef1, vp1, ep2, ef2, vp2 = particion

    nuevo_vecino = False

    while not nuevo_vecino:
        if fase < len(ef2): # Fase 1: intercambiar elemento entre ef1 y ef2
            ef1.append(ef2.pop(0))
            ef2.append(ef1.pop(0))
        elif fase < len(ef2) + len(ep2): # Fase 2: intercambiar elementos entre ep1 y
            ep2
                if len(ep1) == 0:
                    ef2.append(ep1.pop(0))
                    ep1.append(ep2.pop(0))
                    vp1.append(vp2.pop(0))
                else:
                    ep1.append(ep2.pop(0))
                    ep2.append(ep1.pop(0))
                    vp1.append(vp2.pop(0))
                    vp2.append(vp1.pop(0))
        else: # Fase 3: ep1 intercambia elemento con ep2 mientras ef1 permanece
            igual
                fase_3_index = fase - len(ef2) - len(ep2)
                if fase_3_index < len(ep2): # intercambiar ep1 y ep2
                    if len(ep1) == 0:
                        ep1.append(ep2.pop(0))
                        vp1.append(vp2.pop(0))
                    else:
                        ep1.append(ep2.pop(0))
                        ep2.append(ep1.pop(0))
                        vp1.append(vp2.pop(0))
                        vp2.append(vp1.pop(0))
                else: # después de intercambiar ep1 y ep2, cambiar ef1 y ef2
                    ef1.append(ef2.pop(0))
                    ef2.append(ef1.pop(0))

# Verificar la condición de evitar ep1 y ef1 conteniendo el mismo elemento
if not (ep1 and ef1 and ep1[0] == ef1[0]):
    nuevo_vecino = True
```

```
return (ep1, ef1, vp1, ep2, ef2, vp2)
```

## Complejidad Temporal

### 1. Inicialización y extracción de elementos:

- La función realiza operaciones como `pop(0)` y `append()`, que tienen complejidad  $O(1)$  cada una para listas.
- Sin embargo, `pop(0)` es  $O(n)$  para listas, ya que todos los elementos tienen que ser desplazados. Este es el principal factor que afecta la complejidad.

### 2. Fases de intercambio:

- Hay tres fases principales:
  - **Fase 1:** Intercambio entre `ef1` y `ef2`. Esto ocurre en  $O(1)$  operaciones de `pop` y `append`.
  - **Fase 2:** Intercambio entre `ep1` y `ep2`, y sus correspondientes valores en `vp1` y `vp2`. Esto también ocurre en  $O(1)$  operaciones por intercambio.
  - **Fase 3:** Intercambio entre `ep1` y `ep2`, con una subsecuente verificación de `ef1` y `ef2`. Nuevamente, esto ocurre en  $O(1)$  operaciones por intercambio.
- Cada intercambio es  $O(1)$ , pero si usamos `pop(0)`, se convierte en  $O(n)$ .

### 3. Verificación de la condición “if not (ep1 and ef1 and ep1[0] == ef1[0])”

Esta verificación tiene una complejidad de  $O(1)$

## Complejidad Espacial

- La función no crea nuevas estructuras de datos significativas que escalen con el tamaño de la entrada.
- Las listas `ep1`, `ef1`, `vp1`, `ep2`, `ef2`, `vp2` se modifican por lo que no hay una sobrecarga adicional significativa en términos de espacio, la complejidad espacial total es  $O(1)$  ya que no se utilizan estructuras adicionales significativas.

Python

```
def generar_particion_aleatoria(
    self, estados_presentes, estados_futuros, valores_estados_presentes
):
    ep1 = list([])
    ef1 = list(estados_futuros[0])
    vp1 = list([])
    ep2 = list(estados_presentes)
    ef2 = list(estados_futuros[1:])
```

```

vp2 = list(valores_estados_presentes)

return (ep1, ef1, vp1, ep2, ef2, vp2)

```

## Complejidad Temporal

### 1. Asignaciones de listas vacías:

- ep1 y vp1 se asignan como listas vacías:  $O(1)$ .

### 2. Asignaciones de sublistas:

- ef1 se asigna como una lista con el primer elemento de estados\_futuros:  $O(1)$  ya que solo toma un elemento.
- ep2 se asigna con todos los elementos de estados\_presentes:  $O(n)$  porque copia todos los elementos.
- ef2 se asigna con todos los elementos de estados\_futuros desde el segundo en adelante:  $O(m)$  porque copia todos los elementos excepto el primero.
- vp2 se asigna con todos los elementos de valores\_estados\_presentes:  $O(n)$  porque copia todos los elementos.

## Complejidad Espacial

### • Espacio para nuevas listas:

- ep1 y vp1 son listas vacías:  $O(1)$ .
- ef1 es una lista con un solo elemento de estados\_futuros:  $O(1)$ .
- ep2 es una copia de estados\_presentes:  $O(n)$ .
- ef2 es una copia de todos los elementos de estados\_futuros excepto el primero:  $O(m)$ .
- vp2 es una copia de valores\_estados\_presentes:  $O(n)$ .

Python

```

def busqueda_local_emd(
    self,
    estados_presentes,
    estados_futuros,
    valores_estados_presentes,
    subconjunto,
    listaNodos,
    original_system,
):
    mejor_particion = self.generar_particion_aleatoria(
        estados_presentes, estados_futuros, valores_estados_presentes
    )
    mejor_costo, r1, r2 = self.calcular_costo(

```

```

        mejor_particion, subconjunto, original_system, listaNodos
    )
max_iteraciones = (len(estados_presentes) * len(estados_futuros)) + (
    len(estados_presentes) + len(estados_futuros)
)
print(max(len(estados_futuros), len(estados_presentes)), max_iteraciones)

iteraciones_sin_mejora = 0
particiones_visitadas = set()
particiones_visitadas.add(tuple(map(tuple, mejor_particion)))

if mejor_costo == 0.0:
    return mejor_particion, mejor_costo, r1, r2

while iteraciones_sin_mejora < max_iteraciones:
    vecino = self.generar_vecino(mejor_particion, iteraciones_sin_mejora)
    vecino_tuple = tuple(map(tuple, vecino))

    if vecino_tuple in particiones_visitadas:
        iteraciones_sin_mejora += 1
        continue

    particiones_visitadas.add(vecino_tuple)
    costo_vecino, r1, r2 = self.calcular_costo(
        vecino, subconjunto, original_system, listaNodos
    )

    if costo_vecino == 0.0:
        return vecino, costo_vecino, r1, r2

    if costo_vecino < mejor_costo:
        mejor_particion = vecino
        mejor_costo = costo_vecino
        iteraciones_sin_mejora = 0
    else:
        iteraciones_sin_mejora += 1

return mejor_particion, mejor_costo, r1, r2

```

## Complejidad Temporal

1. **Generación de Partición Aleatoria:**
  - self.generar\_particion\_aleatoria(estados\_presentes, estados\_futuros, valores\_estados\_presentes):
    - Complejidad:  $O(n+m)$ .
2. **Cálculo de Costo Inicial:**
  - self.calcular\_costo(mejor\_particion, subconjunto, original\_system, listaNodos):
    - La complejidad de calcular\_costo es  $O((n+m)^2)$  como se analizó anteriormente.
3. **Bucle while:**
  - Máximo número de iteraciones:  $O(n \cdot m+n+m)$ .
  - Dentro del bucle:
    - self.generar\_vecino(mejor\_particion, iteraciones\_sin\_mejora):
      - La complejidad puede considerarse  $O(n+m)$  en el peor caso debido a las operaciones de intercambio.
    - vecino\_tuple = tuple(map(tuple, vecino)):
      - Complejidad:  $O(n+m)$ .
    - Verificación en el conjunto particiones\_visitadas:
      - La verificación y adición tienen complejidad amortizada  $O(1)$ .
    - self.calcular\_costo(vecino, subconjunto, original\_system, listaNodos):
      - Complejidad:  $O((n+m)^2)$ .
    - Actualización de variables (mejor\_particion, mejor\_costo, iteraciones\_sin\_mejora):
      - Complejidad:  $O(1)$ .
  - **Número de iteraciones:** En el peor caso, puede iterar hasta max\_iteraciones, que es  $O(n \cdot m+n+m)$  donde  $n$  es el número de estados presentes y  $m$  el número de estados futuros.
  - **Costo por iteración:** En cada iteración, llama a generar\_vecino y calcular\_costo, que tienen una complejidad  $O(1)$  y  $O(n\log n+k+m^2)$  respectivamente.

La complejidad del algoritmo busqueda\_local\_emd es:

- **Tiempo:**  $O((n \cdot m+n+m) \cdot (n\log n+k+m^2))$
- **Espacio:**  $O(m^2)$  debido al almacenamiento de los tensores en cada cálculo de costo.

## Complejidad Espacial

1. **Espacio para Particiones:**
  - mejor\_particion y vecino ocupan  $O(n+m)$ .
2. **Espacio para Conjuntos y Listas:**
  - particiones\_visitadas puede almacenar hasta  $O(n \cdot m+n+m)$  elementos, con cada elemento ocupando  $O(n+m)$  espacio.
  - Complejidad espacial para el conjunto:  $O((n \cdot m+n+m) \cdot (n+m))$ .
3. **Espacio para Resultados Intermedios:**
  - r1 y r2 ocupan espacio de  $O((n+m)^2)$ .
4. **Total Complejidad Espacial:**

- $O((n \cdot m+n+m) \cdot (n+m)+(n+m)^{**2})$ .

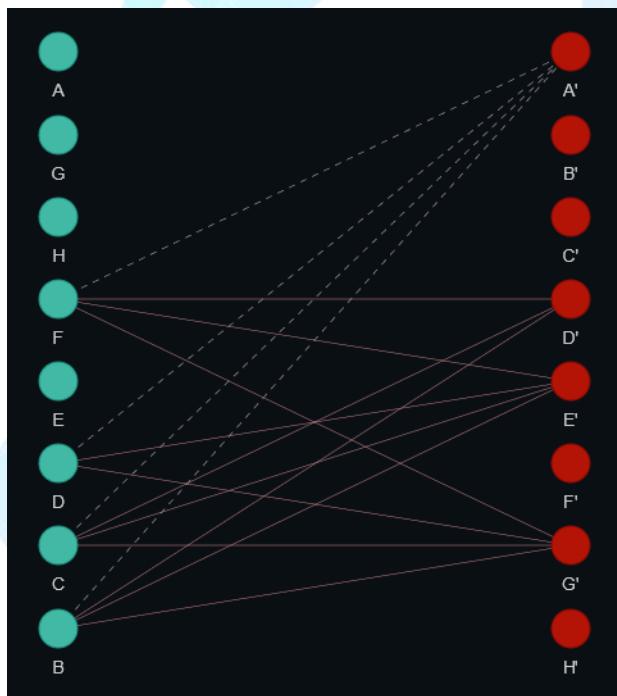
## Diseño y estructura de la segunda estrategia:

### Descripción general del algoritmo:

Con esta estrategia se busca generar la partición del sistema que genere la menor pérdida usando los grafos bipartitos y los sistemas de distribución, el grafo sólo será una representación de los estados del sistema, mientras que las aristas pueden representar una pérdida.

### Ejemplo de cómo se vería una solución para este problema:

Los valores presentes en la derecha son los estados futuros del sistema, mientras que los estados que se presentan en el izquierda son los estados presentes, se puede ver también que no todos los nodos tienen aristas, esto es debido a que los nodos que no tienen aristas no están haciendo parte del sistema ingresado. En este caso el subsistema que se ingresó fue:  $P(ADEF^{(t+1)} | BCDF = 1000)$



Una vez vista esta forma de representar el sistema, se explicará la estrategia utilizada para resolver este problema, y es mediante de nuevo al uso de tablas de probabilidades que se ingresa al sistema.

Volviendo a los 3 estados que hemos estado utilizando para los problemas.

E.Futuro			A		E.Futuro			B		E.Futuro			C	
E.Actual			0	1	E.Actual			0	1	E.Actual			0	1
A	B	C			A	B	C			A	B	C		
0	0	0			1	0		0	1	0	0	0	1	0
1	0	0			1	0		1	0	0	1	0	0	1
0	1	0			0	1		1	0	0	1	0	0	1
1	1	0			0	1		0	1	1	1	0	1	0
0	0	1			0	1		0	1	0	0	1	1	0
1	0	1			0	1		1	0	1	0	0	1	0
0	1	1			0	1		1	0	0	1	1	0	1
1	1	1			0	1		0	1	1	1	1	1	0

Lo que se hará es eliminar las aristas una en una, las aristas siempre tienen un origen y un destino, por lo que lo que se hará va a ser buscar la tabla que de la arista destino, y marginalizar con el nodo origen, y volver a expandir la tabla. Aquí esta un ejemplo para el procedimiento:

Supongamos que se elimina una arista desde A hasta B. Por lo cual tomaremos la tabla de B y marginazlizaremos A.

Se marginaliza como se explicó

E. Futuro			B		E. Futuro			B		E. Futuro			B	
E. Actual			0	1	E. Actual			0	1	E. Actual			0	1
	B	C				B	C				B	C		
	0	0				1	0				0	0	1	0
	0	0				1	0				1	0	0	1
	1	0				1	0				0	1	0	1
	1	0				1	0				1	1	0	1
	0	1				1	0				0	0	1	1
	0	1				0	1				1	0	1	0
	1	1				1	0				0	1	0	1
	1	1				0	1				0	1	1	0

Y luego se vuelve a expandir la tabla y se hace un producto tensor

E.Futuro			A			E. Futuro			B			E.Futuro			C		
E.Actual			0 1			E. Actual			0 1			E.Actual			0 1		
A	B	C				A	B	C				A	B	C			
0	0	0		1	0	0	0	0		1	0	0	0	0		1	0
1	0	0		1	0	1	0	0		1	0	1	0	1		0	1
0	1	0		0	1	0	1	0		1	0	0	1	0		0	1
1	1	0		0	1	1	1	0		1	0	1	1	0		1	0
0	0	1		0	1	0	0	1		0,5	0,5	0	0	1		1	0
1	0	1		0	1	1	0	1		0,5	0,5	1	0	1		0	1
0	1	1		0	1	0	1	1		0,5	0,5	0	1	1		0	1
1	1	1		0	1	1	1	1		0,5	0,5	1	1	1		1	0

Como resultado dará una nueva distribución, esta se comparara con la distribución original y se encontrará el valor de pérdida usando la función de EMD, en el caso de que la diferencia de 0, se eliminará la arista directamente y se guardará la tabla que se generó, sin embargo si se genera una pérdida la arista no se borra y se conserva la tabla anterior a la expansión. Si se genera una partición en ese proceso significa que se generó un resultado.

Las aristas eliminadas se marcarán como punteadas, si no tienen un color indica que no hubo pérdida sin embargo si se genera con color rojo, esa arista si que tuvo una perdida.

### Análisis de complejidad

Este algoritmo usa tres funciones principales:

**'expandirTabla'**: Como su nombre lo dice es el algoritmo encargado de generar la expansión de la tabla de cada estado según sea la situación.

**"generarSubGrafoMinimo"**: Una vez se hayan analizado las aristas y no se haya encontrado alguna solución, el algoritmo hará un recorrido con recursión memorizada para encontrar las aristas que generen la menor pérdida, ya que anteriormente las aristas ya fueron marcadas.

**"estrategia2"**: Este algoritmo es el que recorre las aristas y genere el análisis de expansión y comparación a cada una de estas.

**expandirTabla**: Algoritmo diseñado para marginalizar y expandir tablas para el análisis de pérdida de una arista, este algoritmo recibe dos valores principales los cuales son:

**Tabla** Es el TDM del estado futuro al cual se le está eliminando la arista, el dato se presenta de igual forma que se había mostrado anteriormente y **posición** es el estado presente al cual se va a marginalizar, solo se ingresa como la letra y el algoritmo será capaz de identificar la columna para ese dato.

```

Unset

def expandirTabla(self, tabla, posicion):
    nueva_tabla = [tabla[0]] # Mantener la primera fila de la tabla
original
    indicesVisitados = {}

    for fila in tabla[1:]:
        nuevo_indice = fila[0][:posicion] + fila[0][posicion + 1 :]

        if nuevo_indice not in indicesVisitados:
            nueva_tabla.append(fila)
            indicesVisitados[nuevo_indice] = len(nueva_tabla) - 1
        else:
            index = indicesVisitados[nuevo_indice]
            nueva_tabla[index][1] = (nueva_tabla[index][1] + fila[1]) /
2
            nueva_tabla[index][2] = (nueva_tabla[index][2] + fila[2]) /
2

            nueva_tabla.append(
                [fila[0], nueva_tabla[index][1], nueva_tabla[index][2]])
        )

    return nueva_tabla

```

## Complejidad temporal

Crear una nueva lista **nueva\_tabla** con la primera fila de tabla tiene una complejidad de **O(1)**. Inicializar un diccionario vacío **indicesVisitados** también tiene una complejidad de **O(1)**.

Iteramos sobre  $n-1$  filas (del índice 1 al  $n-1$ ), lo que implica una complejidad de **O(n)** ya que recorremos todas las filas excepto la primera. Esta operación toma una cadena **fila[0]** y crea una nueva cadena excluyendo el carácter en la posición **posición**. Asumamos que la longitud de **fila[0]** es **m**. Entonces, esta operación de rebanado y concatenación tiene una complejidad de **O(m)**.

Comprobar si **nuevo\_indice** está en **indicesVisitados** tiene una complejidad amortiguada de **O(1)** debido a la naturaleza de las búsquedas en diccionarios. Si **nuevo\_indice** no está en el diccionario, se agrega una nueva entrada a **nueva\_tabla** y al diccionario, cada uno con una complejidad de **O(1)**. Si **nuevo\_indice** ya está en el diccionario, actualizamos los valores correspondientes en **nueva\_tabla** y calculamos el promedio, lo cual también tiene una complejidad de **O(1)**.

Por lo tanto podemos asumir que las dos complejidades más importantes de este algoritmo son **O(n) + O(m)**, lo cual nos da como resultado una complejidad temporal de:

**n**: Número de filas de la tabla

**m:** Número de columnas de la tabla

$$T(n, m) = O(n + m)$$

### Complejidad espacial:

El análisis de complejidad espacial considera cuánta memoria adicional se necesita.

#### 1. nueva\_tabla:

- La nueva tabla puede llegar a tener hasta **2n** entradas en el peor caso (donde cada fila resulta en un nuevo índice). Esto implica una complejidad espacial de **O(n)**.

#### 2. indicesVisitados:

- El diccionario indicesVisitados almacena claves de longitud  $m-1$  y valores enteros, que es también de complejidad **O(n)**.

#### 3. Variables auxiliares:

- Variables como nuevo\_index y index tienen una complejidad espacial constante **O(1)**.

Sumando todos estos factores, la complejidad espacial total es: **O(n)**

Por lo cual en resumen para este algoritmo es la complejidad es de:

$$T(n, m) = O(n + m)$$

$$E(n, m) = O(m)$$

### generarSubgrafoMinimo:

Este grafo es el encargado de que cuando se han analizado todas las aristas y en ella no se ha encontrado ninguna partición diferente de 0 en el sistema, se encargará de encontrar la arista que aunque genere pérdida será la menor pérdida posible. Para este algoritmo se necesitan varias cosas para garantizar su correcto funcionamiento.

- La lista de aristas debe de estar ordenada del menor al mayor peso, el algoritmo antes de ejecutar esta función se encargará de que esto se cumpla usando quicksort, es importante que las aristas estén de menor a mayor debido a que se está buscando las aristas con menor peso, y esto facilitará el trabajo.
- Las aristas como están ordenadas de menor a mayor, deben de tener un peso, el cual de nuevo es garantizado por el algoritmo cuando hizo el análisis de las aristas, cada arista tendrá asignado el valor de pérdida EMD.

Python

```
def generarSubGrafoMinimo(self, i, nodes, edges, solucion, listaSolucion,
solucionActual, listaSolucionActual, nodedict, numComponentes, memo):
    # Verificar si ya hemos resuelto este subproblema
    estado_actual = (i, tuple(listaSolucion), solucion)
```

```

    if estado_actual in memo:
        return memo[estado_actual]

    # Filtrar las aristas que no están en la lista de soluciones
    copiaEdges = [edge for edge in edges if edge not in listaSolucion]
    componentes = self.find_connected_components(nodes, copiaEdges)

    # Si es bipartito y hemos encontrado más componentes
    if len(componentes) == numComponentes + 1:
        if solucion <= solucionActual or solucionActual == -1:
            memo[estado_actual] = (listaSolucion, solucion)
            return listaSolucion, solucion

    # Si hemos procesado todos los bordes
    if i >= len(edges):
        return listaSolucionActual, solucionActual

    # Si es bipartito y tenemos el mismo número de componentes
    if len(componentes) == numComponentes:
        copynodedict = nodedict.copy()
        copiaSolucion = solucion

        if edges[i].to not in nodedict:
            copiaSolucion += float(edges[i].label)
            copynodedict[edges[i].to] = float(edges[i].label)
        else:
            copiaSolucion -= copynodedict[edges[i].to]
            copynodedict[edges[i].to] += float(edges[i].label)
            copiaSolucion += copynodedict[edges[i].to]

        copiaListaSolucion = listaSolucion + [edges[i]]

        if copiaSolucion <= solucionActual or solucionActual == -1:
            # Explorar la rama sin incluir la arista actual
            listaSolucionActual, solucionActual =
self.generarSubGrafoMinimo(
                i + 1, nodes, edges, solucion, listaSolucion,
solucionActual,
                listaSolucionActual, nodedict, numComponentes, memo
            )
            # Explorar la rama incluyendo la arista actual
            listaSolucionActual, solucionActual =
self.generarSubGrafoMinimo(
                i + 1, nodes, edges, copiaSolucion, copiaListaSolucion,
solucionActual,
                listaSolucionActual, copynodedict, numComponentes, memo
            )

```

```

# Memorizar el resultado antes de devolver
memo[estado_actual] = (listaSolucionActual, solucionActual)
return listaSolucionActual, solucionActual

```

### Complejidad temporal:

La verificación y la búsqueda en el diccionario memo tienen una complejidad de **O(1)** debido a la naturaleza del diccionario. Filtrar aristas tiene una complejidad de **O(e)**, donde e es el número de aristas en edges. Esto se debe a que estamos iterando sobre todas las aristas y realizando una operación de búsqueda en listaSolucion, que tiene una longitud de **O(e)** en el peor de los casos.

Las listas copiaEdges y copiaListaSolucion, así como el diccionario copynodedict, tienen una complejidad espacial de **O(e)** en el peor de los casos. Dado que se realizan copias en cada llamada recursiva, el espacio total utilizado es proporcional al número de llamadas recursivas activas en la pila, lo que es **O(e)** en profundidad.

La recursión genera un árbol binario de profundidad e (el número de aristas), y en el peor de los casos, exploramos todas las combinaciones posibles de inclusión/exclusión de aristas. Esto da una complejidad de **(2^e)**.

Por lo cual la complejidad temporal para este algoritmo es de:

$T(e) = O(2^e)$  en el peor de los casos, sin embargo también hay que considerar la poda y el memorizado, el cual puede reducir la carga de trabajo de una manera más significativa. Sin embargo puede pasar a ser **O(e)** si se desarrolla correctamente.

### Complejidad espacial:

La memoización almacena resultados para cada subproblema único, cada una con un estado de **O(e)** tamaño. Esto resulta en una complejidad espacial de **O(e)**.

Las listas copiaEdges y copiaListaSolucion, así como el diccionario copynodedict, tienen una complejidad espacial de **O(e)** en el peor de los casos. Dado que se realizan copias en cada llamada recursiva, el espacio total utilizado es proporcional al número de llamadas recursivas activas en la pila, lo que es **O(e)** en profundidad.

Por lo tanto la complejidad de este algoritmo llega a ser de:

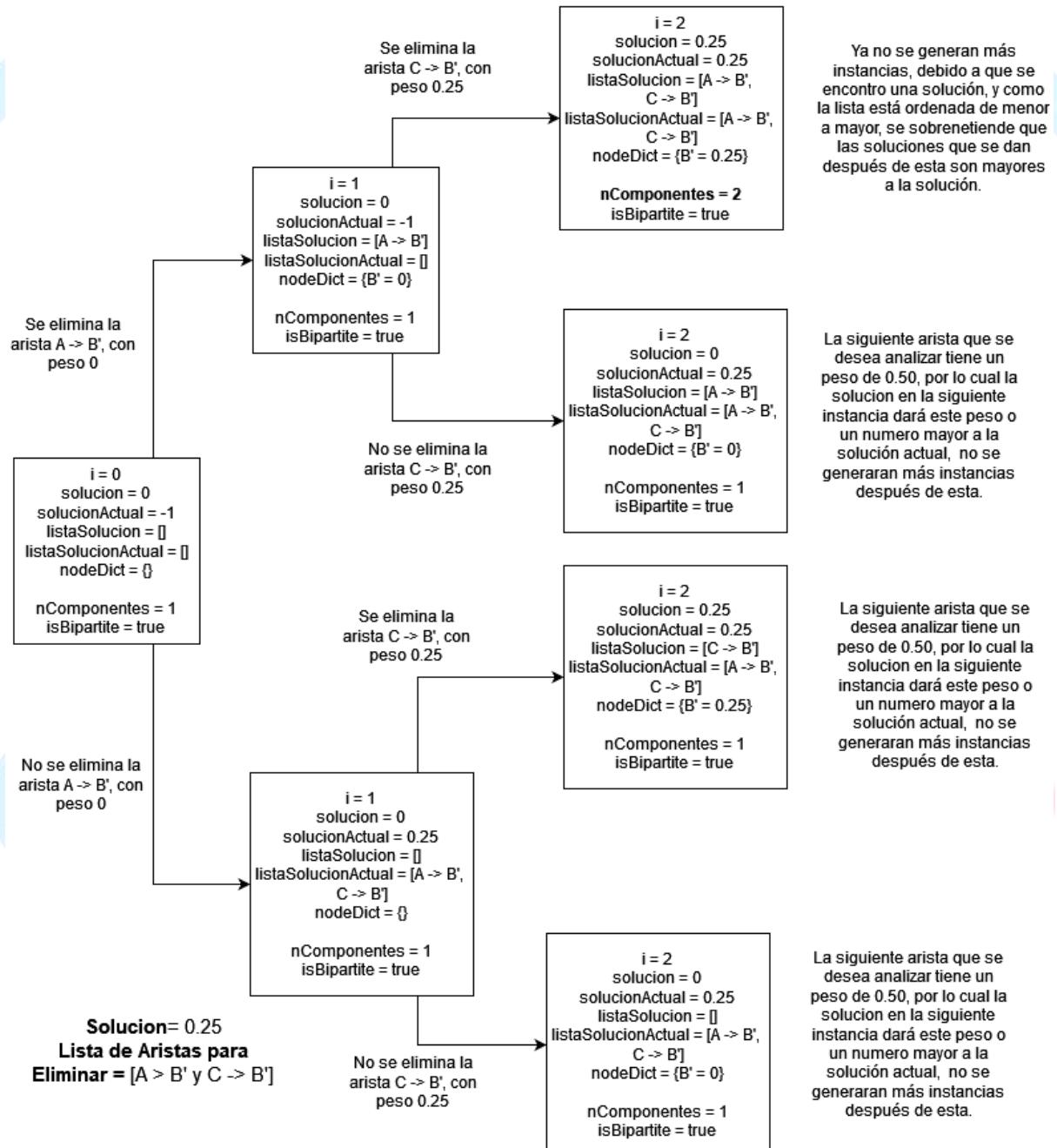
**e**: Número de aristas del grafo.

$$T(e) = O(e)$$

$$E(e) = O(e)$$

**Diagramas:** Se puede evidenciar que no se recorren todas las aristas totalmente, si no que cuando se encuentra una solución el algoritmo se encargará de que si la pérdida es más grande que la solución actual, evitar generar más instancias

Lista de Aristas ordenada = [ A -> B' con peso 0, C -> B' con peso 0.25, A -> C' con peso 0.50, B -> A' con peso 0.50, B -> C' con peso 0.50, C -> A' con peso 0.50]



Ahora se probará el funcionamiento con una red de 3 nodos donde tomamos ABC presentes y ABC futuros y un valor de 100:

Tenemos el sistema original de la siguiente manera:

$$P(ABC^{t+1} | ABC^t = (1, 0, 0))$$

Tiempo de ejecución de generación de subconjunto: 0.0

	$[A, B, C] \setminus [A, B, C]$	(0', 0', 0')	(0', 0', 1')	(0', 1', 0')	(0', 1', 1')	(1', 0', 0')	(1', 0', 1')	(1', 1', 0')	(1', 1', 1')
0	1 0 0	0	1	0	0	0	0	0	0

Esta arista se marca como 0.5, así se hará con las siguientes:

Se elimina la arista de B -----> A

A

	Llave	(1')	(0')
0	0 0 0 0	0.5	0.5
1	1 0 0 0	0.5	0.5
2	0 1 0 0	0.5	0.5
3	1 1 0 0	0.5	0.5
4	0 0 0 1	1	0
5	1 0 0 1	1	0
6	0 1 0 1	1	0
7	1 1 0 1	1	0

0.5

	$[A, B, C] \setminus [A, B, C]$	(0', 0', 0')	(0', 0', 1')	(0', 1', 0')	(0', 1', 1')	(1', 0', 0')	(1', 0', 1')	(1', 1', 0')	(1', 1', 1')
0	1 0 0	0	0.5	0	0	0	0.5	0	0

Se elimina la arista de C -----> A

A

	Llave	(1')	(0')
0	0 0 0 0	0.5	0.5
1	1 0 0 0	0.5	0.5
2	0 1 0 0	1	0
3	1 1 0 0	1	0
4	0 0 0 1	0.5	0.5
5	1 0 0 1	0.5	0.5
6	0 1 0 1	1	0
7	1 1 0 1	1	0

0.5

	$[A, B, C] \setminus [A, B, C]$	(0', 0', 0')	(0', 0', 1')	(0', 1', 0')	(0', 1', 1')	(1', 0', 0')	(1', 0', 1')	(1', 1', 0')	(1', 1', 1')
0	1 0 0	0	0.5	0	0	0	0.5	0	0

Esta arista tiene un valor de 0, por ende ya se elimina.

**B**

	Llave	('1')	('0')
0	0 0 0 0	0	1
1	1 0 0 0	0	1
2	0 1 0 0	0	1
3	1 1 0 0	0	1
4	0 0 1 0	0.5	0.5
5	1 0 1 0	0.5	0.5
6	0 1 1 0	0.5	0.5
7	1 1 1 0	0.5	0.5

0

	[A', B', C'] \ [A', B', C']	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')
0	1 0 0 0	0	1	0	0	0	0	0	0

**B**

	Llave	('1')	('0')
0	0 0 0 0	0.25	0.75
1	1 0 0 0	0.25	0.75
2	0 1 0 0	0.25	0.75
3	1 1 0 0	0.25	0.75
4	0 0 1 0	0.25	0.75
5	1 0 1 0	0.25	0.75
6	0 1 1 0	0.25	0.75
7	1 1 1 0	0.25	0.75

0.25

	[A', B', C'] \ [A', B', C']	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')
0	1 0 0 0	0	0.75	0	0.25	0	0	0	0

Se elimina la arista de A -----> C

C

	Llave	('1')	('0')
0	0 0 0 0	0.5	0.5
1	1 0 0 0	0.5	0.5
2	0 1 0 0	0.5	0.5
3	1 1 0 0	0.5	0.5
4	0 0 1 0	0.5	0.5
5	1 0 1 0	0.5	0.5
6	0 1 1 0	0.5	0.5
7	1 1 1 0	0.5	0.5

0.5

	[A', 'B', 'C'] \ [A', 'B', 'C']	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')
0	1 0 0 0	0.5	0.5	0	0	0	0	0	0

C

	Llave	('1')	('0')
0	0 0 0 0	0.5	0.5
1	1 0 0 0	0.5	0.5
2	0 1 0 0	0.5	0.5
3	1 1 0 0	0.5	0.5
4	0 0 1 0	0.5	0.5
5	1 0 1 0	0.5	0.5
6	0 1 1 0	0.5	0.5
7	1 1 1 0	0.5	0.5

0.5

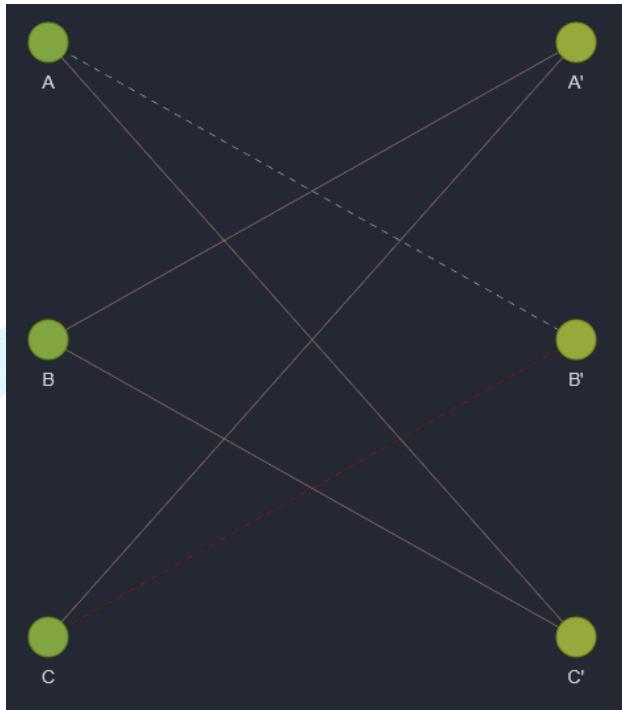
	[A', 'B', 'C'] \ [A', 'B', 'C']	('0', '0', '0')	('0', '0', '1')	('0', '1', '0')	('0', '1', '1')	('1', '0', '0')	('1', '0', '1')	('1', '1', '0')	('1', '1', '1')
0	1 0 0 0	0.5	0.5	0	0	0	0	0	0

Los resultados nos darán de la siguiente forma:

Valor de perdida: 0.25

Tiempo de ejecución: 0.011 segundos

El grafo quedará de esta otra forma:



### Diseño y estructura de la tercera estrategia:

#### 1. calcular\_costo

Esta función calcula el costo de una partición dada usando la métrica de distancia de Hamming.

##### Parámetros:

- `particion`: Tupla con las listas de las dos particiones.
- `subconjunto`, `original`, `listaNodos`: Datos adicionales necesarios para calcular las distribuciones de probabilidad.

##### Salida:

- El costo mínimo entre dos cálculos de distancia de Hamming (`emd1`, `emd2`) y las distribuciones de probabilidad `r1` y `r2`.

##### Funcionamiento:

- Ordena las listas de la partición.
- Genera distribuciones de probabilidad `r1` y `r2` para ambas particiones.
- Calcula los tensores y las distancias de Hamming.
- Retorna el menor costo y las distribuciones de probabilidad.

#### 2. generate\_random\_partition

Esta función genera una partición aleatoria de los estados presentes y futuros.

- **Parámetros:**
  - estados\_presentes, estados\_futuros, valores\_estados\_presentes: Listas de estados y sus valores.
- **Salida:**
  - Una tupla que representa una partición aleatoria.
- **Funcionamiento:**
  - Selecciona al azar uno de los tres tipos de partición: "empty\_ef", "empty\_ep", "full".
  - Genera la partición correspondiente.

### **3. metropolis\_update**

Esta función realiza una actualización de Metropolis para una réplica.

- **Parámetros:**
  - replica: Diccionario que contiene información sobre la réplica actual.
  - estados\_presentes, estados\_futuros, valores\_estados\_presentes, subconjunto, original\_system, listaNodos: Datos necesarios para generar particiones y calcular costos.
- **Salida:**
  - Ninguna, pero actualiza el diccionario de la réplica si se acepta la nueva partición.
- **Funcionamiento:**
  - Genera una nueva partición aleatoria.
  - Calcula el costo de la nueva partición.
  - Decide si acepta la nueva partición usando el criterio de Metropolis.

### **4. replica\_exchange**

Esta función realiza un intercambio de réplicas.

#### **Parámetros:**

- replicas: Lista de diccionarios, cada uno representando una réplica.

#### **Salida:**

- Ninguna, pero intercambia las particiones y pérdidas entre réplicas adyacentes si se cumplen ciertas condiciones.

#### **Funcionamiento:**

- Itera sobre las réplicas adyacentes.
- Calcula el cambio en la función objetivo.
- Decide si intercambiar las particiones y pérdidas basándose en el criterio de Metropolis.

Python

```

def generate_random_partition(
    self, estados_presentes, estados_futuros, valores_estados_presentes
):
    n_ep = len(estados_presentes)
    n_ef = len(estados_futuros)
    partition_type = random.choice(["empty_ef", "empty_ep", "full"])

    if partition_type == "empty_ef" and n_ef > 0:
        i = random.randint(0, n_ef - 1)
        partition = (
            [],
            [estados_futuros[i]],
            [],
            estados_presentes,
            [ef for ef in estados_futuros if ef != estados_futuros[i]],
            valores_estados_presentes,
        )
    elif partition_type == "empty_ep" and n_ep > 0:
        i = random.randint(0, n_ep - 1)
        partition = (
            [estados_presentes[i]],
            [],
            [valores_estados_presentes[i]],
            [ep for ep in estados_presentes if ep != estados_presentes[i]],
            estados_futuros,
            [valores_estados_presentes[j] for j in range(n_ep) if j != i],
        )
    else:
        percentage = random.uniform(0.1, 0.4)
        num_ep1 = max(1, int(n_ep * percentage))
        num_ef1 = max(1, int(n_ef * percentage))

        ep_indices = random.sample(range(n_ep), num_ep1)
        ef_indices = random.sample(range(n_ef), num_ef1)

        ep1 = [estados_presentes[i] for i in ep_indices]
        ef1 = [estados_futuros[i] for i in ef_indices]

        vp1 = [valores_estados_presentes[i] for i in ep_indices]
        ep2 = [estados_presentes[i] for i in range(n_ep) if i not in ep_indices]
        ef2 = [estados_futuros[i] for i in range(n_ef) if i not in ef_indices]
        vp2 = [
            valores_estados_presentes[i] for i in range(n_ep) if i not in ep_indices
        ]

```

```

partition = (ep1, ef1, vp1, ep2, ef2, vp2)

return partition

```

## Complejidad Temporal

1. **Inicialización y Elección de Partición:**
  - La inicialización de  $n_{ep}$  y  $n_{ef}$  es  $O(1)$ .
  - La elección aleatoria de  $partition\_type$  es  $O(1)$ .
2. **Rama empty\_ef:**
  - $random.randint(0, n_{ef} - 1)$  es  $O(1)$ .
  - La generación de  $partition$  implica:
    - Crear una lista de un solo elemento:  $O(1)$ .
    - Copiar la lista  $estados\_presentes$ :  $O(n)$ .
    - Crear una lista por comprensión excluyendo un elemento:  $O(m)$ .
  - Complejidad total para esta rama:  $O(n+m)$ .
3. **Rama empty\_ep:**
  - $random.randint(0, n_{ep} - 1)$  es  $O(1)$ .
  - La generación de  $partition$  implica:
    - Crear una lista de un solo elemento:  $O(1)$ .
    - Crear una lista por comprensión excluyendo un elemento:  $O(n)$ .
    - Copiar la lista  $estados\_futuros$ :  $O(m)$ .
    - Crear una lista por comprensión excluyendo un elemento:  $O(n)$ .
  - Complejidad total para esta rama:  $O(n+m)$ .
4. **Rama full:**
  - $random.uniform(0.1, 0.2)$  es  $O(1)$ .
  - $max(1, int(n_{ep} * percentage))$  y  $max(1, int(n_{ef} * percentage))$  son  $O(1)$ .
  - $random.sample(range(n_{ep}), num_{ep1})$  y  $random.sample(range(n_{ef}), num_{ef1})$  son  $O(n)$  y  $O(m)$  respectivamente.
  - La generación de listas por comprensión para  $ep1, ef1, vp1, ep2, ef2, vp2$ :
    - Cada una de estas operaciones es  $O(n)$  o  $O(m)$  dependiendo del tamaño de la lista.
  - Complejidad total para esta rama:  $O(n+m)$ .

## Complejidad Espacial

1. **Espacio para Variables Locales:**
  - $n_{ep}, n_{ef}, partition\_type, i, percentage, num_{ep1}, num_{ef1}, ep\_indices, ef\_indices$ :  $O(1)$  cada una.
2. **Espacio para Particiones:**
  - La partición resultante ( $partition$ ) tiene listas que copian  $estados\_presentes$ ,  $estados\_futuros$ , y  $valores\_estados\_presentes$ :
    - Complejidad espacial es  $O(n+m)$ .

Python

```

def metropolis_update(
    self,
    replica,
    estados_presentes,
    estados_futuros,
    valores_estados_presentes,
    subconjunto,
    original_system,
    listaNodos,
):
    current_partition = replica["partition"]
    current_loss = replica["loss"]
    new_partition = self.generate_random_partition(
        estados_presentes, estados_futuros, valores_estados_presentes
    )
    new_loss, r1, r2 = self.calcular_costo(
        new_partition, subconjunto, original_system, listaNodos
    )
    delta_loss = new_loss - current_loss

    if delta_loss < 0 or random.uniform(0, 1) < np.exp(
        -replica["beta"] * delta_loss
    ):
        replica["partition"] = new_partition
        replica["loss"] = new_loss
        replica["r1"] = r1
        replica["r2"] = r2

```

## Complejidad Temporal

1. **Asignación de Variables Iniciales:**
  - `current_partition = replica["partition"]`:  $O(1)$ .
  - `current_loss = replica["loss"]`:  $O(1)$ .
2. **Generación de una Nueva Partición:**
  - `new_partition = self.generate_random_partition(estados_presentes, estados_futuros, valores_estados_presentes)`:
    - Hemos determinado previamente que esta función tiene una complejidad de  $O(n+m)$ .
3. **Cálculo del Costo de la Nueva Partición:**
  - `new_loss, r1, r2 = self.calcular_costo(new_partition, subconjunto, original_system, listaNodos)`:
    - Segundo el análisis previo, la complejidad de esta función es  $O((n+m)\log(n+m))$ .

4. **Cálculo de delta\_loss:**
  - $\text{delta\_loss} = \text{new\_loss} - \text{current\_loss}$ :  $O(1)$ .
5. **Condición del Metropolis Update:**
  - $\text{random.uniform}(0, 1) < \text{np.exp}(-\text{replica}["\beta"] * \text{delta\_loss})$ :
    - Generar un número aleatorio y calcular una exponencial son  $O(1)$ .
6. **Actualización del Estado de la Réplica** (en el peor caso):
  - $\text{replica}["\text{partition}"] = \text{new\_partition}$ :  $O(n+m)$ .
  - $\text{replica}["\text{loss}"] = \text{new\_loss}$ :  $O(1)$ .
  - $\text{replica}["r1"] = r1$ :  $O(1)$ .
  - $\text{replica}["r2"] = r2$ :  $O(1)$ .

## Complejidad Espacial

1. **Espacio para Variables Locales:**
  - $\text{current\_partition}$ ,  $\text{current\_loss}$ ,  $\text{new\_partition}$ ,  $\text{new\_loss}$ ,  $\text{delta\_loss}$ ,  $r1$ ,  $r2$ :  $O(1)$  cada una.
2. **Espacio para Nuevas Particiones y Cálculos:**
  - $\text{new\_partition}$ :  $O(n+m)$ .
  - $r1$ ,  $r2$ :  $O(n+m)$ .

Python

```
def replica_exchange(self, replicas):
    for i in range(len(replicas) - 1):
        replica1, replica2 = replicas[i], replicas[i + 1]
        delta = (replica2["beta"] - replica1["beta"]) * (
            replica1["loss"] - replica2["loss"])
        if delta < 0 or random.uniform(0, 1) < np.exp(-delta):
            (
                replica1["partition"],
                replica2["partition"],
            ) = (
                replica2["partition"],
                replica1["partition"],
            )
            (
                replica1["loss"],
                replica2["loss"],
            ) = (
                replica2["loss"],
                replica1["loss"],
            )
            (
                replica1["r1"],
                replica2["r1"],
            )
```

```

) = (
    replica2["r1"],
    replica1["r1"],
)
(
    replica1["r2"],
    replica2["r2"],
) = (
    replica2["r2"],
    replica1["r2"],
)
)

```

## Complejidad Temporal

1. **Iteración sobre las Rélicas:**
  - o for i in range(len(replicas) - 1):  $O(\text{replicas})$
2. **Cálculo de delta:**
  - o  $\text{delta} = (\text{replica2}["\text{beta}"] - \text{replica1}["\text{beta}"]) * (\text{replica1}["\text{loss}"] - \text{replica2}["\text{loss}"])$ :  $O(1)$ .
3. **Condición de Intercambio:**
  - o if  $\text{delta} < 0$  or random.uniform(0, 1) < np.exp(-delta):  $O(1)$ .
4. **Intercambio de Rélicas:**
  - o Si se cumple la condición, se intercambian las siguientes propiedades entre replica1 y replica2:
    - partition, loss, r1, r2: Todas estas operaciones son  $O(1)$ .

## Complejidad Espacial

1. **Espacio para Variables Locales:**
  - o i, replica1, replica2, delta:  $O(1)$  cada una.
2. **Espacio para Rélicas:**
  - o Las rélicas pueden contener particiones, costos y distribuciones de probabilidades asociadas. Siendo  $O(n+m)$  donde  $n$  y  $m$  son las dimensiones de las particiones y distribuciones respectivamente.

## Calibrando los parámetros de la estrategia 3

**Valores de temperatura/beta:** es común utilizar un rango de temperaturas inversas (valores beta) que están espaciadas uniformemente entre 0,1 y 1,0. Esto ayuda a garantizar que las rélicas abarquen una amplia gama del espacio estatal, mejorando las capacidades de exploración (QuantStart).

**Número de rélicas:** lo habitual es utilizar entre 4 y 10 rélicas. Este número equilibra el costo computacional con el beneficio de un muestreo mejorado del espacio de estados (QuantStart).

**Intervalo de intercambio:** los intercambios entre réplicas generalmente se intentan cada 5 a 10 iteraciones. Este intervalo ayuda a mantener un buen equilibrio entre exploración y explotación (QuantStart).

**Temperatura (valores beta):** La elección de los valores de temperatura (o valores beta) es fundamental para la eficiencia de REMC. Un enfoque común es utilizar una progresión geométrica o lineal de valores beta entre una temperatura mínima y máxima. Normalmente, el rango de valores beta comienza entre 0,1 y 1,0, lo que garantiza un buen equilibrio entre exploración y explotación en el espacio de búsqueda. Este rango se ha utilizado eficazmente en estudios relacionados con el acoplamiento proteína-ligando y el plegamiento de proteínas (BioMed Central) (BioMed Central).

**Número de réplicas:** El número de réplicas puede variar según la complejidad del problema. Para modelos más simples, alrededor de 5 réplicas podrían ser suficientes, mientras que los sistemas más complejos pueden requerir 10 o más réplicas. El objetivo es tener suficientes réplicas para cubrir una amplia gama de configuraciones y temperaturas, asegurando una exploración exhaustiva del panorama energético (BioMed Central).

**Iteraciones e intervalos de intercambio:** El número de iteraciones a menudo se establece como proporcional al producto del número de estados que se consideran (tanto estados presentes como futuros) y un multiplicador constante (por ejemplo, 10). Esto permite tiempo suficiente para que el sistema explore diferentes configuraciones. Los intervalos de intercambio (la frecuencia con la que las réplicas intercambian información) suelen establecerse entre 5 y 10 iteraciones. Los intercambios frecuentes pueden ayudar a lograr una mejor convergencia al permitir que las configuraciones escapen de los mínimos locales de manera más efectiva (BioMed Central) (BioMed Central).

La calibración de los parámetros se hará empíricamente usando esta configuración inicial:

Estados presentes:	Estados futuros:	Selecciona el valor presente:
A ✕ B ✕ C ✕ D ✕ E ✕ F ✕	A ✕ B ✕ C ✕ D ✕ E ✕ F ✕ G ✕ H ✕	(1, 0, 0, 0, 1, 0)

Parámetros iniciales

```
num_replicas = 4
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Tiempo de ejecución: 0.0562 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0378 segundos

El emd es: 0.0

Tiempo de ejecución: 0.1276 segundos

El emd es: 0.0

Incrementamos en 1 el número de réplicas

```
num_replicas = 5
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Tiempo de ejecución: 0.0422 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0668 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0401 segundos

El emd es: 0.0

Incrementamos en 1 el número de réplicas

```
num_replicas = 6
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Tiempo de ejecución: 0.0798 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0502 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0484 segundos

El emd es: 0.0

Incrementamos en 2 el número de réplicas

```
num_replicas = 8
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Tiempo de ejecución: 0.0431 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0491 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0423 segundos

El emd es: 0.0

Incrementamos en 2 el número de réplicas

```
num_replicas = 10
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Tiempo de ejecución: 0.0511 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0457 segundos

El emd es: 0 . 0

Tiempo de ejecución: 0.0524 segundos

El emd es: 0 . 0

Un buen número de réplicas parece ser 8 ya que encontramos tiempos muy similares a diferencia de otros números de réplicas

8 en el número de réplicas y 6 en el intervalo de cambios

```
num_replicas = 8
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 6
```

Tiempo de ejecución: 0.0571 segundos

El emd es: 0 . 0

Tiempo de ejecución: 0.0434 segundos

El emd es: 0 . 0

Tiempo de ejecución: 0.0474 segundos

El emd es: 0 . 0

8 en el número de réplicas y 7 en el intervalo de cambios

```
num_replicas = 8
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 7
```

Tiempo de ejecución: 0.0526 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0489 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0526 segundos

El emd es: 0.0

8 en el número de réplicas y 8 en el intervalo de cambios

```
num_replicas = 8
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 8
```

Tiempo de ejecución: 0.0599 segundos

El emd es: 0.0

Tiempo de ejecución: 0.042 segundos

El emd es: 0.0

Tiempo de ejecución: 0.0585 segundos

El emd es: 0.0

8 en el número de réplicas y 10 en el intervalo de cambios

```
num_replicas = 8
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 10
```

Tiempo de ejecución: 0.0506 segundos

El emd es: 0 . 0

Tiempo de ejecución: 0.0452 segundos

El emd es: 0 . 0

Tiempo de ejecución: 0.0735 segundos

El emd es: 0 . 0

La mejor calibración obtenida después de muchas pruebas fue 5 en el número de réplicas y 5 en el intervalo de cambios, ya que se obtiene de manera rápida una solución para una red de 5 nodos, como para una red de 8 nodos, con otros parámetros era muy lento para redes pequeñas.

```
num_replicas = 5
beta_values = np.linspace(0.1, 1.0, num_replicas)
num_iterations = (len(optionep) * len(optionef)) * 10
swap_interval = 5
```

Python

```
def fuerza_bruta(
    self,
    estados_presentes,
    estados_futuros,
    valores_estados_presentes,
    subconjunto,
    listaNodos,
    original_system,
):
    best_cost = float("inf")
    best_partition = None
    best_r1 = None
    best_r2 = None

    # Generar todas las combinaciones posibles de particiones
    for ep1_indices in chain.from_iterable(
        combinations(range(len(estados_presentes)), r)
```

```

        for r in range(len(estados_presentes) + 1)
    ):
        for ef1_indices in chain.from_iterable(
            combinations(range(len(estados_futuros)), r)
            for r in range(len(estados_futuros) + 1)
        ):
            ep1 = [estados_presentes[i] for i in ef1_indices]
            ef1 = [estados_futuros[i] for i in ef1_indices]
            vp1 = [valores_estados_presentes[i] for i in ep1]
            ep2 = [
                estados_presentes[i]
                for i in range(len(estados_presentes))
                if i not in ep1
            ]
            ef2 = [
                estados_futuros[i]
                for i in range(len(estados_futuros))
                if i not in ef1
            ]
            vp2 = [
                valores_estados_presentes[i]
                for i in range(len(valores_estados_presentes))
                if i not in ep1
            ]
            if not ep1 and not ef1:
                continue
            if not ep2 and not ef2:
                continue

            particion = (ep1, ef1, vp1, ep2, ef2, vp2)
            costo, r1, r2 = self.calcular_costo(
                particion, subconjunto, original_system, listaNodos
            )

            if costo < best_cost:
                best_cost = costo
                best_partition = particion
                best_r1 = r1
                best_r2 = r2

    return best_partition, best_cost, best_r1, best_r2

```

## Complejidad Temporal

### 1. Generación de combinaciones:

- La función `chain.from_iterable(combinations(range(len(estados_presentes)), r) for r in range(len(estados_presentes) + 1))` genera todas las combinaciones posibles de los índices de `estados_presentes`.
- Para una lista de longitud  $n$ , el número de combinaciones posibles es  $2^{**n}$ .
- De manera similar, para `estados_futuros` de longitud  $m$ , el número de combinaciones posibles es  $2^{**m}$ .

### 2. Iteración sobre combinaciones:

- El primer bucle `for ep1_indices in ...` tiene  $2^{**n}$  iteraciones.
- El segundo bucle `for ef1_indices in ...` tiene  $2^{**m}$  iteraciones.

### 3. Coste del cálculo dentro de los bucles:

- Dentro de los bucles, hay varias operaciones que involucran seleccionar elementos de las listas según los índices generados. Esto tiene una complejidad lineal respecto al tamaño de las listas, es decir,  $O(n)$  y  $O(m)$  respectivamente.
- La función `self.calcular_costo` se llama en cada combinación. Supongamos que esta función tiene una complejidad de  $O(C)$ .

Combinando estos elementos, la complejidad temporal total es  $O(2^{**}(n+m) \cdot (n+m+C))$ . Dado que  $n$  y  $m$  son los tamaños de las listas, y  $C$  es el tiempo necesario para calcular el costo para una partición dada.

## Limitaciones y Perspectivas Futuras:

- Enuncia todos los problemas que se les presentaron, como los resolvieron y específicamente que no pudieron resolver.

Uno de los principales problemas fue el uso de las matrices cuando se trataba de representar las distribuciones de probabilidades, principalmente debido a que no se podía obtener tan eficientemente a qué valor se le asignaba a las llaves (entiéndase las llaves como los valores  $(0,0,0)$ ), sin embargo estos problemas se pudo solucionar con el uso de memorizado y de diccionarios capaces de guardar la información de estos datos.

Otro problema fue con el manejo de matrices o listas en problemas en los que se usaba backtracking, esto es debido a que por su naturaleza recursiva el programa volvía a instancias anteriores y los datos de esas instancias se conservaban, a excepción de las listas, las cuales sin importar siempre se guardaban los cambios, se tuvo que optar por usar librerías de `copy` para guardar estos datos en otro espacio de memoria, cosa que también pudo afectar a la eficiencia espacial.

Otro problema fue con la obtención de pérdidas **EMD** en las 3 estrategias, ya que los valores mientras no fueran 0, no eran los que se esperaban

- ¿Han considerado cómo su diseño algorítmico podría evolucionar para enfrentar más eficientemente el problema?
- Para la estrategia 1, actualmente, la función `generar_vecino` sigue un proceso secuencial para intercambiar elementos entre las particiones. Esto puede ser ineficiente y no explorar suficientemente el espacio de soluciones. Algunas

estrategias que podrían mejorar esta función como utilizar heurísticas basadas en el costo actual para decidir qué elementos intercambiar, en lugar de hacerlo de forma secuencial.

- Para la estrategia 2 consiste en eliminar aristas del grafo de forma iterativa y evaluar el impacto en la distribución de probabilidades resultante. Se utiliza una combinación de técnicas de programación dinámica, ordenamiento y búsqueda recursiva para encontrar la solución óptima o aproximada. Cada vez que se elimina una arista y se calcula la nueva distribución de probabilidad, se generan nuevas tablas de probabilidad. Dependiendo del tamaño del grafo y de las tablas implicadas, esta operación puede ser costosa, especialmente si se realiza en cada iteración del bucle.
- Para la estrategia 3 la función **replica\_exchange** puede mejorarse para intercambiar réplicas más eficientemente como por ejemplo usar selección inteligente de réplicas, en lugar de intentar intercambios secuenciales, aplicar estrategias más sofisticadas para seleccionar pares de réplicas a intercambiar, como seleccionar las réplicas que han tenido la menor cantidad de intercambios exitosos.
- ¿Qué áreas pueden identificar en las que podrían realizar mejoras o refinamientos en términos de eficiencia?

Los cálculos de marginalización y distribución de probabilidades no llegan a ser los más óptimos debido a que tienen que se tiene que hacer un recorrido exhaustivo por todas las tablas para obtener toda la información que se aplican a las estrategias.