Especificación de requisitos del Software

Requerimientos:

Requerimiento 1: Creación de grafos personalizados

- El sistema debe permitir a los usuarios crear grafos personalizados agregando y eliminando nodos y aristas.
- Los usuarios deben poder definir las características del grafo, como el tipo (dirigido o no dirigido), si es ponderado, y si es completo o conexo.

Requerimiento 2: Visualización de grafos

- El sistema debe proporcionar una visualización gráfica de los grafos creados por los usuarios.
- La visualización debe permitir a los usuarios mover libremente los nodos y las aristas en el área de trabajo.

Requerimiento 3: Ejecución de procesos en grafos

- El sistema debe permitir a los usuarios ejecutar procesos predefinidos en los grafos, como algoritmos de búsqueda o de optimización.
- Los resultados de la ejecución del proceso deben mostrarse claramente en la interfaz de usuario.

Requerimiento 4: Importar y exportar grafos

- El sistema debe permitir a los usuarios importar grafos desde archivos en formatos específicos, como JSON o TXT.
- Los usuarios deben poder exportar los grafos creados en diferentes formatos para su posterior análisis o uso.

Requerimiento 5: Personalización de nodos y aristas

El sistema debe permitir a los usuarios personalizar la apariencia de los nodos y las aristas, incluyendo el color, el tamaño y el estilo.

Requerimiento 6: Gestión de archivos

- El sistema debe proporcionar funcionalidades para guardar y cargar el estado de trabajo actual, lo que incluye los grafos creados y cualquier configuración personalizada.

Casos de Uso:

Caso de Uso 1: Crear un grafo personalizado

Actor: Usuario

Flujo Principal:

- El usuario selecciona la opción de crear un nuevo grafo personalizado.
- El sistema muestra un área de trabajo vacía.
- El usuario agrega nodos y aristas según sus preferencias.

- El usuario define las características del grafo, como su tipo y propiedades adicionales.

Caso de Uso 2: Ejecutar un proceso en un grafo
Actor: Usuario
Flujo Principal:

- El usuario carga un grafo existente o crea uno nuevo.
- El usuario selecciona la opción de ejecutar un proceso en el grafo.
- El sistema muestra una lista de procesos disponibles.
- El usuario elige un proceso y lo ejecuta.
- El sistema procesa el grafo según las especificaciones del proceso seleccionado y muestra los resultados.

Caso de Uso 3: Importar un grafo desde un archivo
Actor: Usuario
Flujo Principal:

- El usuario selecciona la opción de importar un grafo desde un archivo.
- El sistema solicita al usuario que seleccione el archivo a importar.
- El sistema carga el grafo desde el archivo seleccionado y lo muestra en la interfaz de usuario.

Caso de Uso 4: Exportar un grafo a un formato específico
Actor: Usuario
Flujo Principal:

- El usuario selecciona la opción de exportar el grafo actual.
- El sistema presenta al usuario una lista de formatos de archivo disponibles.
- El usuario elige un formato y confirma la exportación.
- El sistema guarda el grafo en el formato especificado y proporciona al usuario un enlace de descarga.

Diseño del Algoritmo

Primera estrategia.
El algoritmo propuesto aborda el problema mediante dos estrategias algorítmicas principales: programación dinámica y búsqueda voraz (greedy). La función **estrategiaUno** utiliza programación dinámica top-down con memoization para generar distribuciones de probabilidad y evitar cálculos innecesarios. La función busqueda_voraz implementa una búsqueda voraz para encontrar la mejor partición que minimice la diferencia en la distribución de probabilidades.

Función: estrategiaUno.

- **Propósito:** Generar la distribución de probabilidades para un estado dado utilizando memoization, con el fin de evitar cálculos innecesarios.
- **Enfoque:** Programación dinámica top-down con memoization, almacenando resultados intermedios en una tabla.

Función: busqueda_voraz.

- **Propósito:** Evaluar diferentes particiones de las listas c1 y c2 para encontrar la partición que minimice la diferencia en la distribución de probabilidades.
- **Enfoque:** Búsqueda voraz, evaluando todas las posibles particiones y seleccionando la que minimice la diferencia calculada mediante la métrica Earth Mover's Distance (EMD).

Análisis de Complejidad

- **Complejidad Temporal de estrategiaUno**
  - Con memoization, cada estado se calcula una sola vez.
  - Si hay S estados y M matrices, la complejidad es $O(S \cdot M)$ en el peor de los casos.
- **Complejidad Temporal de busqueda_voraz**
  - La función itera sobre todas las posibles particiones de c1 y c2
  - Para listas de longitud n, hay O(n^2) posibles particiones.
  - Cada partición evalúa dos llamadas a estrategiaUno y cálculos adicionales.
  - Al estar haciendo uso de la función de estrategiaUno, esto hace que se evite algunos casos en los que la EMD ya ha sido calculada, esto para evitar cálculos repetidos por lo cual la complejidad del algoritmo pasa a ser O(n*m) en el peor de los casos, siendo n el número de elementos del conjunto uno y m el número de elementos del conjunto dos.
- **Complejidad Espacial**:
  - estrategiaUno utiliza una tabla hash para memoization, que en el peor de los casos tiene un tamaño proporcional al número de estados y combinaciones de c1 y c2.
  - La complejidad espacial es O(c1 * c2)

Justificación de la Estrategia

- **Programación Dinámica**:
  - Elegida por su eficiencia al evitar cálculos redundantes mediante memoization.
  - **Ventaja:** Reducción significativa en el tiempo de cómputo al almacenar resultados intermedios.
  - **Desventaja**: Mayor uso de memoria.
- **Búsqueda Voraz**:
  - Elegida por su simplicidad y capacidad de proporcionar soluciones buenas en un tiempo razonable.

- **Ventaja:** Facilidad de implementación y comprensión.
- **Desventaja:** No garantiza encontrar la solución óptima.

Pseudocódigo:

algoritmo estrategiaUno:

```
estrategiaUno(E dict:matrices, list:c1, list:c2, tuple:estadoActual, list;estados)
{
    tabla <- {}
    key <- (c1, c2, estadoActual)
    if key no in tabla then
        tabla[key] <- generarDistribucionProbabilidades(matrices, c1, c2, estadoActual, estados)
    endif
    return tabla[key]
}
```

algoritmo busqueda_voraz:

```
busqueda_voraz(E matrices, estados, distribucionProbabilidadOriginal, c1, c2, estadoActual)
{
    mejor_particion ← []
    menor_diferencia ← inf
    listaParticionesEvaluadas ← []

    for i ← 0 to len(c1) - 1 do
        c1_izq ← c1[:i]
        c1_der ← c1[i:]
        c2_izq ← []
        c2_der ← copia(c2)
        for j ← 0 to len(c2) - 1 do
            c2_izq.add(c2_der.pop(0))
            distribucion_izq ← estrategiaUno(matrices, c1_izq, c2_izq, estadoActual, estados)
            distribucion_der ← estrategiaUno(matrices, c1_der, c2_der, estadoActual, estados)
            p1 ← distribucion_izq[1][1:]
            p2 ← distribucion_der[1][1:]
            prodTensor ← producto_tensor(p1, p2)
            diferencia ← calcularEMD(distribucionProbabilidadOriginal[1][1:], prodTensor)
            aux ← []
            if c2_der ==[] y c1_der == [] then
                continuar
            endif

            if diferencia < menor_diferencia then
                menor_diferencia ← diferencia
                mejor_particion ← [(c2_izq, c1_izq), (c2_der, c1_der)]
            endif

            aux ← [(c2_izq, c1_izq), (c2_der, c1_der), diferencia]
            listaParticionesEvaluadas.add(aux)
        endfor
    endfor

    return mejor_particion, menor_diferencia, listaParticionesEvaluadas
}
```

Manejo de Casos Especiales y Límites

- **Casos Especiales:**

    - Si c2_der y c1_der están vacíos, la partición no se evalúa.
    - Se asegura que todas las particiones posibles se consideran para encontrar la mejor solución.
- **Límites:**
    - El algoritmo maneja eficientemente diferentes tamaños de c1 y c2.
    - En condiciones extremas (listas muy grandes), el uso de memoization y la búsqueda voraz pueden requerir optimizaciones adicionales para mantener la eficiencia.

Optimizaciones Implementadas

- **Memoization en estrategiaUno**:
    - Reduce cálculos redundantes, almacenando resultados intermedios en una tabla hash.
- **Estrategia Greedy en busqueda_voraz**:
    - Simplifica la búsqueda de la mejor partición, evaluando particiones de manera secuencial y seleccionando la mejor encontrada.

Comparaciones con Otros Enfoques

- **Programación Dinámica vs. Fuerza Bruta**:
    - La programación dinámica reduce significativamente el número de cálculos redundantes comparado con la fuerza bruta.
- **Búsqueda Voraz vs. Algoritmos Exhaustivos**:
    - La búsqueda voraz es más rápida y sencilla de implementar, aunque no garantiza la solución óptima.
    - Los algoritmos exhaustivos asegurarían la solución óptima, pero a un costo computacional mucho mayor.

Manejo de Recursos

- **Memoria**:
    - Uso eficiente mediante memoization para almacenar resultados intermedios.
- **CPU**:
    - Minimiza el tiempo de cómputo al evitar cálculos redundantes y limitar las evaluaciones a las particiones más prometedoras.

Escalabilidad

- **Escalabilidad del Algoritmo**:
    - La combinación de memoization y búsqueda voraz permite que el algoritmo maneje conjuntos de datos de tamaño creciente de manera eficiente.
    - A medida que aumenta el tamaño de los conjuntos de datos, la memoization asegura que los cálculos no se repitan, y la búsqueda

voraz limita el número de particiones evaluadas a un número manejable.

Pruebas y validación del algoritmo: Se realizaron pruebas con diferentes entradas de datos proporcionadas para este propósito y así validar la eficiencia y precisión de las particiones generadas.
https://docs.google.com/spreadsheets/d/1kb_DFqA6qlZKgJB8AAdt9-4duK76kURg/edit?usp=sharing&ouid=103283733589170182725&rtpof=true&sd=true

.

Segunda estrategia.
El código implementa una estrategia para generar particiones de conjuntos de nodos en un grafo, minimizando la diferencia entre una distribución de probabilidad original y una generada. La estrategia se basa en la eliminación secuencial de aristas del grafo, evaluando el impacto de cada eliminación en la distribución de probabilidades.

Estrategia Algorítmica

- **Inicialización**: Se comienza obteniendo matrices de probabilidad de transición y generando un estado inicial de transición.
- **Generación de Distribución Original:** Se calcula la distribución de probabilidades original entre dos conjuntos de nodos (c1 y c2) dados un estado actual.
- **Cálculo de Pérdida de Información:** Para cada arista del grafo, se calcula la pérdida de información al eliminarla. La pérdida se calcula utilizando la distancia de Earth Mover's (EMD), que mide la diferencia entre distribuciones de probabilidad.
- **Selección de Aristas:** Las aristas se ordenan según la pérdida de información calculada, de menor a mayor.
- **Eliminación de Aristas:** Se eliminan secuencialmente las aristas comenzando por aquellas que generan menor pérdida de información.
- **Generación de Particiones:** Para cada eliminación de arista, se evalúan dos nuevas particiones de los conjuntos c1 y c2 y se calcula la diferencia con la distribución original.
- **Selección de la Mejor Partición:** Se selecciona la partición que minimiza la diferencia respecto a la distribución original.

- **Visualización del Resultado:** Se muestra el grafo resultante con colores que indican las aristas eliminadas y las particiones obtenidas.

Análisis de Complejidad
Espacial:

- Se utilizan estructuras de datos para almacenar matrices de probabilidad, estados, y distribuciones de probabilidad, lo cual depende de la cantidad de nodos y estados posibles.
- El espacio adicional utilizado para almacenar resultados y estructuras de datos temporales es también lineal con respecto a la cantidad de aristas y nodos.

Justificación de la estrategia: La estrategia se enfoca en buscar eficiencia para minimizar la diferencia entre las distribuciones de probabilidad utilizando la eliminación de aristas como método.
Entre las ventajas se encuentra la eficiencia al reducir la pérdida de información de manera iterativa y la adaptabilidad a diferentes conjuntos de nodos y aristas.
Como desventajas, una de ellas es que puede ser sensible a la elección inicial de las aristas y el orden de eliminación.

Pseudocodigo:

Función calcular_perdida

```
FUNCION calcular_perdida(matrices, estados, distribucionProbabilidadOriginal, c1, c2, estadoActual, arista, p)
  c1_copy <- c1.copia()
  c2_copy <- c2.copia()

  SI arista.origen EN c1_copy Y arista.destino EN c2_copy ENTONCES
    c1_copy.remover(arista.origen)
    c2_copy.remover(arista.destino)
  SINO SI arista.origen EN c2_copy Y arista.destino EN c1_copy ENTONCES
    c2_copy.remover(arista.origen)
    c1_copy.remover(arista.destino)
  SINO
    RETORNAR INFINITO
  FIN SI

  distribucion_izq <- p.generarDistribucionProbabilidades(matrices, c1_copy, c2_copy, estadoActual, estados)
```

```
    prodTensor <- p.producto_tensor(distribucion_izq[1][1:], distribucion_izq[1][1:])
    diferencia <- p.calcularEMD(distribucionProbabilidadOriginal[1][1:], prodTensor)
    RETORNAR diferencia
FIN FUNCION
```

Función generarParticiones

```
FUNCION generarParticiones(c1, c2, estadoActual, edges)
    particiones <- []
    a, b, c, lista, l <- estrategia2(c1, c2, estadoActual, edges)
    df <- DataFrame(lista, columnas=['Conjunto 1', 'Conjunto 2', 'Diferencia', 'Tiempo de ejecución'])
    RETORNAR df, particiones
FIN FUNCION
```

Función pintarGrafoGenerado

```
FUNCION pintarGrafoGenerado(c1, c2, estadoActual, nodes, edges, Node, Edge)
    p <- Nuevo ProbabilidadEP()
    mp, menorD, tiempo, lpEvaluadas, eliminadas <- estrategia2(c1, c2, estadoActual, edges)
    m <- p.datosMatrices()
    s, e <- p.generarEstadoTransicion(m)
    dpo <- p.generarDistribucionProbabilidades(m, c1, c2, estadoActual, e)

    aristas_eliminadas_perdida_cero <- Conjunto()
    arista_minima_perdida <- NULO
    minima_perdida <- INFINITO

    PARA cada arista EN edges HACER
        perdida <- calcular_perdida(m, e, dpo, c1.copia(), c2.copia(), estadoActual, arista, p)
        SI perdida == 0 ENTONCES
            aristas_eliminadas_perdida_cero.añadir((arista.origen, arista.destino))
        SINO SI perdida < minima_perdida ENTONCES
            arista_minima_perdida <- (arista.origen, arista.destino)
            minima_perdida <- perdida
        FIN SI
    FIN PARA
```

```
  PARA cada arista EN edges HACER
    SI (arista.origen, arista.destino) EN aristas_eliminadas_perdida_cero ENTONCES
      arista.color <- 'yellow'
      arista.dashes <- True
    SINO SI (arista.origen, arista.destino) == arista_minima_perdida ENTONCES
      arista.color <- 'violet'
      arista.dashes <- True
    FIN SI
  FIN PARA

  p1, p2 <- mp
  PARA cada i EN p1[1] HACER
    SI i NO ESTA EN p2[1] ENTONCES
      PARA cada arista EN edges HACER
        SI arista.origen == i Y arista.destino EN p2[0] ENTONCES
          arista.dashes <- True
          arista.color <- 'rgba(254, 20, 56, 0.5)'
        FIN SI
      FIN PARA
    FIN SI
  FIN PARA

  PARA cada i EN p2[1] HACER
    SI i NO ESTA EN p1[1] ENTONCES
      PARA cada arista EN edges HACER
        SI arista.origen == i Y arista.destino EN p1[0] ENTONCES
          arista.dashes <- True
          arista.color <- 'rgba(254, 20, 56, 0.5)'
        FIN SI
      FIN PARA
    FIN SI
  FIN PARA
graph <- stag.agraph(nodos=nodes, aristas=edges, config=Gui(True))
```

FIN FUNCION

Optimizaciones implementadas: Se tiene un uso de estructuras de datos eficientes para llevar a cabo los cálculos repetitivos.
Un ordenamiento inicial de aristas por pérdida de información para minimizar iteraciones innecesarias.

Manejo de recursos y escalabilidad: La estrategia gestiona de manera eficiente los recursos, utilizando estructuras de datos compactas y minimizando la memoria adicional.
Esta estrategia es escalable debido al enfoque que se maneja, lo cual es manejable para grafos que puedan ser grandes.

Pruebas y validación del algoritmo: Se realizaron pruebas con diferentes entradas de datos proporcionadas para este propósito y así validar la eficiencia y precisión de las particiones generadas.
Se realizaron validaciones comparando con las diferentes estrategias presentes en el proyecto.

Tercera estrategia.

Descripción General del Algoritmo

El algoritmo propuesto busca encontrar la mejor partición de dos conjuntos c1 y c2 que minimicen la diferencia entre la distribución de probabilidad original y la distribución generada por la partición. Para lograrlo, se utiliza la técnica de recocido simulado, que es un método de optimización metaheurístico inspirado en el proceso de enfriamiento de metales.

Pasos principales del algoritmo:

1. **Datos Iniciales:** Se obtienen las matrices de probabilidad necesarias.
2. **Generación de Estados y Distribuciones:** Se generan los estados de transición y la distribución de probabilidad original.
3. **Recocido Simulado:** Se utiliza el recocido simulado para explorar diferentes particiones de los conjuntos c1 y c2, evaluando cada una según una métrica de diferencia.
4. **Selección de la Mejor Partición:** Se elige la partición con la menor diferencia observada

Complejidad Temporal:

- **generarEstadoTransicion:** Asumimos O(E) donde E es el número de estados de transición generados.
- **generarDistribucionProbabilidades:** Esta función es llamada varias veces dentro del recocido simulado, y su complejidad depende del tamaño de las matrices y los conjuntos.
- **recocidoSimulado:** Se ejecuta mientras la temperatura es mayor que 1. Cada iteración principal reduce la temperatura en un factor constante, lo que lleva a O(log(T)) iteraciones principales, donde T es la temperatura inicial.
- **Iteraciones por temperatura:** En cada iteración principal, se realizan un número fijo k de iteraciones, cada una de las cuales incluye llamadas a generar_vecino y obtener_diferencia.
- **obtener_diferencia:** Involucra operaciones de generación de distribuciones y cálculo de producto tensor, que dependen del tamaño de los conjuntos y las matrices.

Complejidad Espacial:

- El algoritmo almacena las matrices de datos, las distribuciones de probabilidad, y las particiones evaluadas. Esto implica un uso de memoria proporcional al tamaño de los datos de entrada y el número de particiones evaluadas.

Justificación de la Estrategia

La elección del recocido simulado se justifica por:

- **Capacidad de Escapar de Óptimos Locales:** A diferencia de métodos como la búsqueda local, el recocido simulado permite aceptar soluciones peores temporalmente, lo que ayuda a escapar de los óptimos locales.
- **Flexibilidad y Simplicidad:** Es relativamente simple de implementar y ajustar, y puede ser aplicado a una amplia variedad de problemas de optimización.

Ventajas:

- Permite encontrar soluciones cercanas al óptimo global en problemas complejos.
- Es adaptable y puede ajustarse mediante parámetros como la temperatura inicial y el factor de enfriamiento.

Desventajas:

- No garantiza encontrar el óptimo global.
- Puede ser computacionalmente costoso dependiendo de la cantidad de iteraciones y el tamaño del problema.

Pseudocódigo:

Función recocido_simulado:

```
recocidoSimulado(E dict:matrices, list:estados, dict:disProbdOriginal, list: c1, list:c2, tuple: estadoActual)
{
    mejor_particion <- NULL
    menor_diferencia <- inf
    listaParticionesEvaluadas <- []
    temperatura <- 1000
    factor_enfriamiento <- 0.99
    iteraciones_por_temperatura <- 2
    tiempo <- 0

    while (temperatura > 1) do
        for i <- 1 to iteraciones_por_temperatura do
            c1_izq, c2_izq, c1_der, c2_der <- generar_vecino(c1, c2)
            diferencia <- obtener_diferencia(c1_izq, c2_izq, c1_der, c2_der, matrices, estadoActual, disProbdOriginal, estados)
            aux <- [(c2_izq, c1_izq), (c2_der, c1_der), diferencia, tiempo_actual()]
            listaParticionesEvaluadas.agregar(aux)

            if (diferencia < menor_diferencia) then
                menor_diferencia <- diferencia
                mejor_particion <- [(c2_izq, c1_izq), (c2_der, c1_der)]
            else
                probabilidad_aceptacion <- exp((menor_diferencia - diferencia) / temperatura)
                if (aleatorio() < probabilidad_aceptacion) then
                    menor_diferencia <- diferencia
                    mejor_particion <- [(c2_izq, c1_izq), (c2_der, c1_der)]
                endif
            endif
        endfor
        temperatura <- temperatura * factor_enfriamiento
    endwhile

    return mejor_particion, menor_diferencia, tiempo, listaParticionesEvaluadas
}
```

Función obtener_diferencia:

```
Función obtener_diferencia(E list:c1_izq, list:c2_izq, list:c1_der, list:c2_der, dict:matrices, tuple:estadoActual, dict:disOriginal, list:estados)
{
    distribucion_izq <- generarDistribucionProbabilidades(matrices, c1_izq, c2_izq, estadoActual, estados)
    distribucion_der <- generarDistribucionProbabilidades(matrices, c1_der, c2_der, estadoActual, estados)
    p1 <- distribucion_izq[1][1:]
    p2 <- distribucion_der[1][1:]
    prodTensor <- producto_tensor(p1, p2)
    diferencia <- calcularEMD(disOriginal[1][1:], prodTensor)

    return diferencia
}
```

Funcion generar_vecino

```
generar_vecino(E list:c1, list:c2)
{
    mitad_c1 <- longitud(c1) / 2
    mitad_c2 <- longitud(c2) / 2
    c1_izq <- random(c1, mitad_c1)
    c1_der <- c1 - c1_izq
    c2_izq <- random(c2, mitad_c2)
    c2_der <- c2 - c2_izq

    return c1_izq, c2_izq, c1_der, c2_der
}
```

Manejo de Casos Especiales y Límites

1. **Tamaño Pequeño de Conjuntos:** Si c1 o c2 tienen muy pocos elementos, el algoritmo sigue funcionando pero con menos particiones posibles.

2. **Distribuciones Degeneradas:** Si las distribuciones de probabilidad son muy similares o idénticas, la diferencia será pequeña y el recocido simulado puede estabilizarse rápidamente.
3. **Altas Dimensiones:** Si las matrices y los conjuntos tienen dimensiones muy altas, puede ser necesario ajustar los parámetros del recocido simulado para mantener la eficiencia.

Optimizaciones Implementadas

1. **Reducción de Temperatura:** El uso de un factor de enfriamiento gradual ayuda a equilibrar la exploración y explotación.
2. **Aceptación Probabilística:** Permite aceptar peores soluciones temporalmente para evitar quedar atrapado en óptimos locales.

Comparaciones con Otros Enfoques

1. **Fuerza Bruta:** Evaluar todas las particiones posibles es impracticable debido a su complejidad exponencial.
2. **Búsqueda Local:** Es más rápida pero puede quedar atrapada en óptimos locales.
3. **Algoritmos Genéticos:** Pueden ser más robustos en encontrar el óptimo global, pero son más complejos de implementar y ajustar.

Manejo de Recursos

- **Memoria:** Se utilizan estructuras de datos eficientes para almacenar las matrices y las particiones evaluadas.
- **CPU:** El uso del recocido simulado equilibra el uso de CPU al enfriar gradualmente y reducir el número de evaluaciones necesarias.

Escalabilidad

El algoritmo está diseñado para ser escalable, ajustando parámetros como la temperatura inicial y el factor de enfriamiento para manejar conjuntos de datos más grandes sin sacrificar eficiencia. A medida que aumentan los datos, la eficiencia del recocido simulado asegura que el rendimiento se mantenga aceptable.

Documento de Diseño del Software (DDS)

1. Arquitectura del sistema:

La arquitectura se basa en un modelo monolítico, dividido en componentes de front-end y back-end, ambos escritos en Python donde el frontend utiliza el framework Streamlit.

El diseño del sistema consta de componentes de front-end donde se tienen en cuenta archivos para gestionar la configuración completa de la interfaz que se le muestra al usuario final. El back-end consta de la utilización del main, que es donde se integra finalmente la lógica utilizada en el proyecto, a su vez, consta de algunos módulos para manejar la lectura de datos y otro para la lógica, algoritmos y procesamiento de grafos.

Diagrama de secuencia

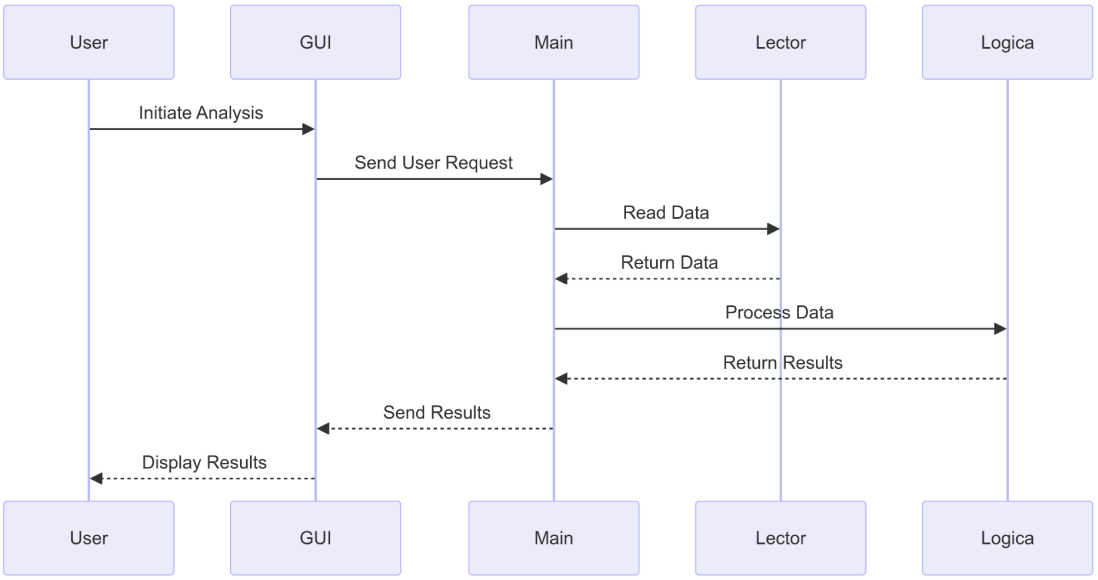
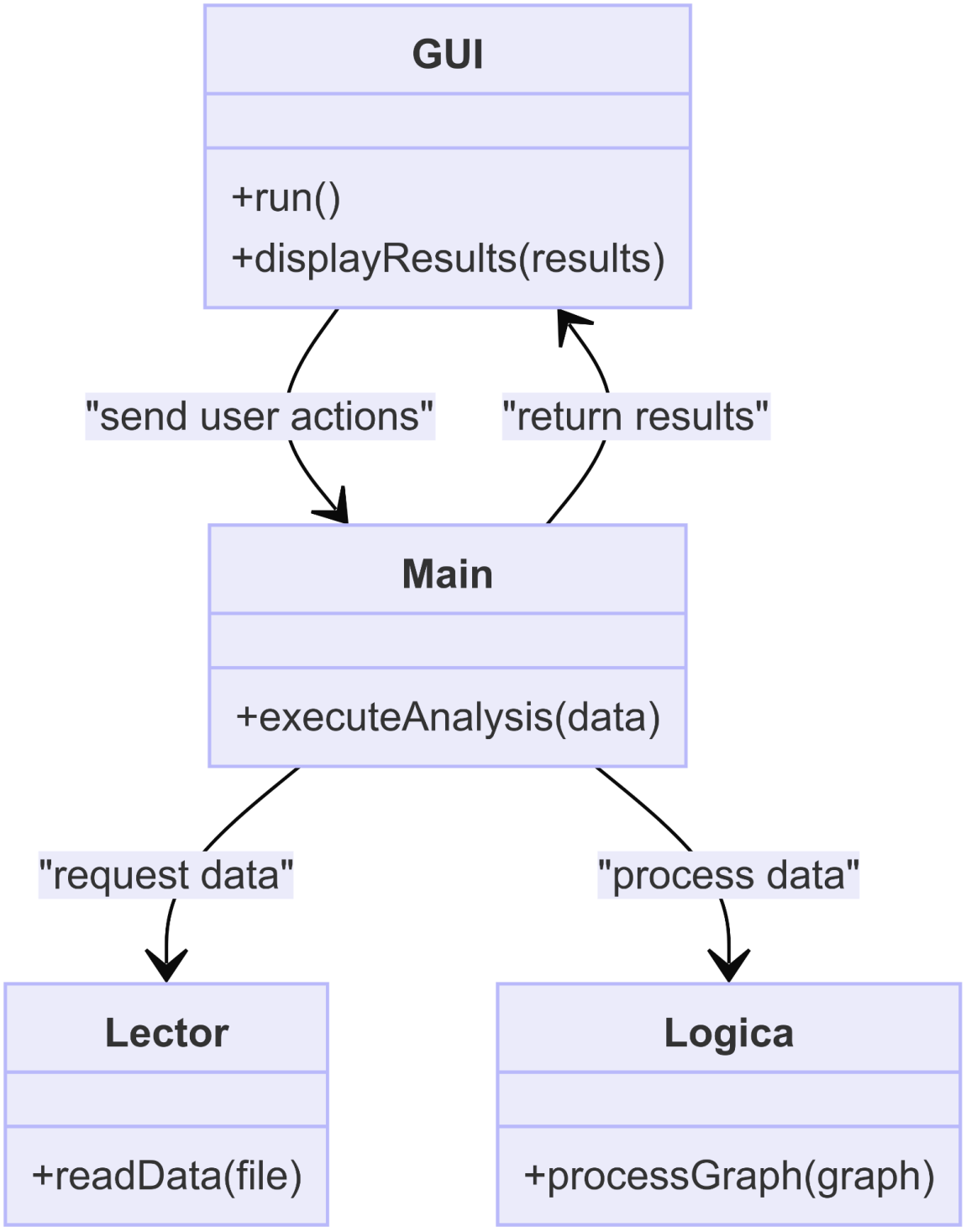


Diagrama de clases

En este diagrama podemos observar un comportamiento muy general de la aplicación, donde la main conecta con cada uno de los componentes presentes en el sistema, en la búsqueda de integrar tanto información del frontend como del backend.



2. Capa de presentación Front - End:

Tecnologías Utilizadas: El front-end del proyecto se implementó utilizando Streamlit, un framework en Python que facilita la creación de aplicaciones web interactivas.

Componentes de la Interfaz:

- **Elementos de Entrada:** Formularios para cargar grafos en formato JSON.
- **Visualización:** Gráficos interactivos para visualizar los grafos y los resultados del análisis de las estrategias implementadas.
- **Interacción del Usuario:** Múltiples opciones en el menú para que el usuario elija qué procedimiento hacer durante su estadía en la aplicación.

3. Capa de Lógica de Negocio (Back-End):

Estructura del Back-End:

- **Módulo lector:** Encargado de la lectura de archivos.
- **Módulo lógica:** Contiene las funciones para realizar el análisis de grafos. Implementa algoritmos que permiten explorar propiedades y características de los grafos cargados. Todas estas funciones se encuentran organizadas en archivos los cuáles contienen los siguientes nombres:
    - Data
    - Estrategias
    - Lógica_Arista
    - Lógica_Grafo
    - Lógica_Nodo
    - Probabilidad

Diseño de la Base de Datos:

- **Formato de Datos:** No se utiliza una base de datos tradicional. Los datos se manejan mediante archivos JSON que contienen las estructuras de los grafos.
- **Ejemplo de Estructura de Datos:** Nodos y sus conexiones se presentan en archivos JSON.

Servicios Utilizados:
Pandas: Para la manipulación y análisis de datos en archivos Excel.
Streamlit: Para la interfaz de usuario que permite cargar archivos y visualizar resultados.

Lógica de Procesamiento en el Servidor:

- **Procesamiento de Archivos:** Leer y convertir datos de Excel en estructuras de datos manejables dentro de Python.
- **Algoritmos de Análisis:**
    - Cálculo de rutas más cortas.

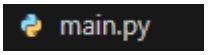
- Identificación de componentes conexos.
- Otros análisis relacionados con la teoría de grafos.

4. Componentes del sistema

Para el Front-end tenemos los siguientes:

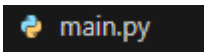
GUI.py: Este archivo se encarga de manejar la configuración establecida para el frontend de la aplicación.

main.py: Si bien este archivo contiene y recibe todo lo necesario para la ejecución del sistema, gran parte del componente frontend se encuentra alojado allí, dando forma y una alta experiencia al usuario final.

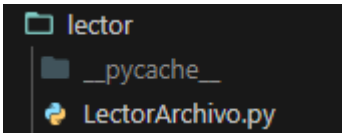


Para el Back-end tenemos los siguientes:

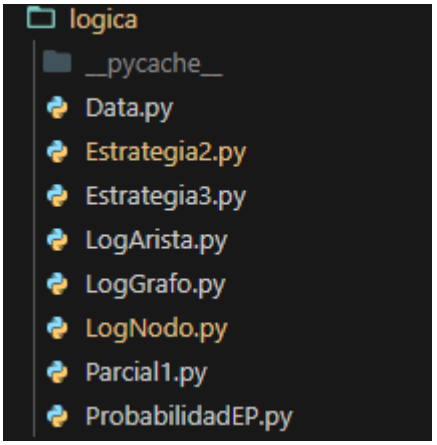
main.py: Este archivo da el punto de entrada principal a la lógica central y la coordinación entre los diferentes módulos de la aplicación.



lector/: Este directorio contiene lo necesario para la carga, lectura y guardado de los diferentes archivos.



logica/: Este directorio contiene la lógica de análisis de grafos, algoritmos y sus diferentes operaciones



Estos componentes en conjunto permiten la carga, procesamiento, análisis y visualización de datos e información necesaria de los grafos.

5. Comunicación entre componentes
El flujo de comunicación funciona con solicitudes realizadas del frontend al backend a través de llamadas directas a funciones de Python. Este programa no funciona con un API REST o similar, sino que se trata de un programa monolítico donde los componentes se comunican internamente.

El back-end procesa la solicitud y envía los resultados de vuelta para ser interpretados y visualizados por el programa.

Este enfoque simplifica la comunicación al evitar la sobrecarga de protocolos de red, manteniendo todo dentro del mismo entorno de ejecución Python.

Limitaciones y Perspectivas Futuras:

Enuncia todos los problemas que se les presentaron, como los resolvieron y específicamente que no pudieron resolver.
Inicialmente, ajustar las gráficas del grafo con Streamlit generó varios problemas. A medida que avanzábamos, adaptar las estrategias 1 y 2 resultó complicado debido al enfoque específico requerido. Durante la implementación, enfrentamos numerosos cambios y ajustes. En particular, la implementación de la estrategia 2 fue un desafío completo, ya que inicialmente no funcionaba como se esperaba. Para abordar estos problemas, recurrimos a una documentación exhaustiva y un análisis detallado de las estrategias implementadas. Esta metodología nos permitió encontrar soluciones viables y avanzar en el desarrollo del proyecto.

¿Han considerado cómo su diseño algorítmico podría evolucionar para enfrentar más eficientemente el problema?
Para la estrategia 3, que involucró el uso de una metaheurística como el recocido simulado, hemos considerado cómo nuestro diseño algorítmico podría evolucionar para abordar el problema de manera más eficiente. Al implementar el recocido simulado, nos enfocamos en ajustar los parámetros y las técnicas de enfriamiento para mejorar la convergencia hacia soluciones óptimas. Además, hemos explorado la posibilidad de integrar técnicas avanzadas de optimización y ajuste fino de parámetros para optimizar aún más el rendimiento del algoritmo.

¿Qué áreas pueden identificar en las que podrían realizar mejoras o refinamientos en términos de eficiencia?
En general, el proyecto ha demostrado una eficiencia decente en cada una de las estrategias implementadas hasta ahora. Sin embargo, reconocemos la importancia

de buscar una mayor eficiencia global para manejar grandes volúmenes de datos gráficos y probabilidades sin problemas.

Para mejorar la eficiencia, podríamos considerar varias áreas clave:

Optimización de algoritmos: Revisar y ajustar los algoritmos utilizados en cada estrategia para reducir la complejidad computacional y mejorar los tiempos de ejecución.

Uso de estructuras de datos eficientes: Implementar estructuras de datos más eficientes para el almacenamiento y manipulación de grafos y datos probabilísticos.

Paralelización y distribución: Explorar técnicas de paralelización y distribución para procesar tareas de manera simultánea y aprovechar recursos computacionales adicionales.

Mejora en la gestión de memoria: Optimizar la gestión de memoria para reducir la sobrecarga y mejorar el uso eficiente de los recursos disponibles.

Refinamiento de técnicas de optimización: Continuar refinando y ajustando las técnicas de optimización utilizadas, como el recocido simulado u otras metaheurísticas, para lograr convergencia más rápida y soluciones más óptimas.