.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de algoritmos
(TB024) Curso Echevarría

Trabajo Práctico N°2

30 de Noviembre de 2025

Alumno	Padrón
Calandria Alén Santiago	112103
Cerrotti Belén Camila	109566
Sorasio Facundo	108882

Índice

1. Ejercicio 1: Programación Lineal

Descripción General

El objetivo de este ejercicio es planificar cómo enviar un archivo de 10 MB desde el Nodo 1 hasta el Nodo 10. La dificultad está en que los enlaces tienen capacidades limitadas y queremos usar la **menor cantidad de conexiones posibles.**

1.1. Supuestos, Condiciones y Limitaciones

Para plantear el modelo matemático, se definen las siguientes bases:

1.1.1. Supuestos

- **Flujo único y continuo:** Se asume que enviamos un solo archivo desde un origen hasta un destino.
- **Linealidad:** Todas las ecuaciones son lineales.

1.1.2. Condiciones

- **Equilibrio de flujo:** En todos los nodos intermedios (del 2 al 9), la cantidad exacta de datos que entra debe ser igual a la que sale. No pueden acumular datos ni generar datos nuevos.
- **Satisfacción de la demanda:** Es obligatorio que la suma de todo lo que llega al nodo final (10) sea exactamente igual al tamaño total del archivo (10 MB).

1.1.3. Limitaciones

- **Costo Uniforme de Aristas:** La función objetivo minimiza la cantidad de variables binarias activas, asignando un peso idéntico a todas las aristas. El modelo asume que "activar" el enlace $(1, 2)$ cuesta matemáticamente lo mismo que activar el $(9, 10)$, sin considerar distancias o pesos ponderados.
- **Divisibilidad Infinita (Variables Continuas):** Al definir la variable de flujo como `Continuous`, el modelo permite fraccionamientos decimales arbitrarios. No se aplica una restricción de "Programación Entera".
- **Transbordo Estricto:** La restricción de equilibrio ($\sum$entra $=$ $\sum$sale) impide cualquier acumulación en los nodos (vértices). El modelo matemático asume que la capacidad de almacenamiento de los nodos intermedios es nula.
- **Capacidad Rígida:** Las cotas superiores definidas en el diccionario `capacidades` actúan como límites. El modelo matemático no admite violaciones temporales de estas capacidades, ni siquiera penalizándolas.

2. Diseño

a. Variables

- `flujo`
  - *Definición:* Cantidad de datos que circulan por una conexión específica.
  - *Tipo:* Variable Continua.

- *Unidad:* Megabytes (MB).

- `activo`
  - *Definición:* Variable indicadora que vale 1 si la conexión se utiliza y 0 si no.
  - *Tipo:* Variable Binaria.
  - *Unidad:* Adimensional.

b. Constantes

- `capacidades`
  - *Definición:* Límite máximo de capacidad para cada enlace de la red.
  - *Unidad:* MB.
- `total_archivo`
  - *Definición:* Cantidad total de información a transmitir (Demanda).
  - *Valor Fijo:* 10 MB.
- `nodos`
  - *Definición:* Conjunto de vértices que conforman la red ($N = \{1, ..., 10\}$).
- `rutas`
  - *Definición:* Conjunto de aristas o enlaces disponibles para el envío de datos.

3. Modelo de Programación Lineal

Se define x_{ij} (`flujo`) como la variable de flujo continuo y y_{ij} (`activo`) como la variable binaria de activación.

Función Objetivo

Minimizar la cardinalidad de enlaces activos:

$$\text{Min } Z = \sum_{(i,j)\in E} y_{ij}$$

Restricciones

1. Restricción de Capacidad Condicional: Vincula el flujo con la variable binaria y respeta el límite de la arista:

$$x_{ij} \leq C_{ij} \cdot y_{ij} \quad \forall (i,j) \in E$$

2. Balance en el Nodo Fuente ($s = 1$): El flujo neto saliente debe igualar la demanda total:

$$\sum_{j} x_{1,j} - \sum_{k} x_{k,1} = T$$

3. Conservación de Flujo (Nodos de Transbordo): Para todo nodo intermedio n, el flujo entrante es igual al saliente:

$$\sum_{k} x_{k,n} = \sum_{j} x_{n,j} \quad \forall n \in \{2, ..., 9\}$$

4. Balance en el Nodo Sumidero ($t = 10$): El flujo neto entrante debe igualar la demanda total:

$$\sum_{k} x_{k,10} - \sum_{j} x_{10,j} = T$$

5. No negatividad e Integridad:

$$x_{ij} \geq 0$$
$$y_{ij} \in \{0, 1\}$$

1.2. Solución en Python

Para la resolución del problema se implementó un script en Python utilizando la librería `PuLP`. El desarrollo se estructura en cuatro bloques lógicos principales:

1. Inicialización del Modelo

Se define un problema de minimización y se crean los diccionarios de variables. Utilizamos `LpVariable.dicts` para generar automáticamente una variable por cada enlace existente en la red.

- **Variables de Flujo:** Continuas y mayores a cero.
- **Variables de Uso:** Binarias (0 o 1).

2. Definición de la Función Objetivo

El objetivo es minimizar la infraestructura utilizada. En el código, esto se traduce en minimizar la suma de todas las variables binarias de activación:

```
modelo += pulp.lpSum(activo)
```

3. Aplicación de Restricciones

Se implementan dos tipos de reglas dentro de bucles iterativos:

a. Capacidad y Activación: Para cada ruta, se asegura que el flujo no supere la capacidad máxima y se obliga a la variable binaria a activarse si hay tráfico:

```
modelo += flujo[(i, j)] <= capacidades[(i, j)] * activo[(i, j)]
```

b. Balance de Flujo (Nodos): Se recorre la lista de nodos calculando la suma de entradas y salidas.

- En el **Origen**, la salida neta debe ser igual al total del archivo (10 MB).
- En el **Destino**, la entrada neta debe ser igual al total del archivo.
- En los **Nodos Intermedios**, la entrada debe ser estrictamente igual a la salida (Transbordo).

```
if nodo == origen:
    modelo += (sale - entra == total_archivo)
elif nodo == destino:
    modelo += (entra - sale == total_archivo)
else:
    modelo += (entra == sale)
```

4. Resolución

Finalmente, se invoca al método `.solve()` para que el motor matemático encuentre la combinación óptima de variables que cumpla todas las ecuaciones anteriores.

Resultado obtenido

Tras la ejecución del algoritmo, se obtuvo el estado **Optimal**, lo que garantiza que se ha encontrado la solución matemática más eficiente posible. El reporte de salida es el siguiente:

```
Estado: Optimal
Enlaces totales usados: 12.0
------------------------------
Nodo 1 -> Nodo 2: 5.0 MB
Nodo 1 -> Nodo 3: 5.0 MB
Nodo 2 -> Nodo 6: 3.0 MB
Nodo 2 -> Nodo 5: 2.0 MB
Nodo 3 -> Nodo 7: 2.0 MB
Nodo 3 -> Nodo 4: 3.0 MB
Nodo 4 -> Nodo 8: 3.0 MB
Nodo 5 -> Nodo 7: 2.0 MB
Nodo 6 -> Nodo 9: 3.0 MB
Nodo 7 -> Nodo 10: 4.0 MB
Nodo 8 -> Nodo 10: 3.0 MB
Nodo 9 -> Nodo 10: 3.0 MB
```

Informe de la Solución

Estrategia de Fragmentación y Distribución

Basado en los resultados del modelo de programación lineal, la estrategia óptima para transferir el archivo de 10 MB minimizando el uso de la infraestructura es la siguiente:

1. **Fragmentación en el Origen (Nodo 1):** Debido a las restricciones de capacidad iniciales, es necesario dividir el archivo en dos partes iguales:
   - Se envían **5 MB** por la ruta superior (hacia el Nodo 2).
   - Se envían **5 MB** por la ruta inferior (hacia el Nodo 3).
2. **Distribución Intermedia:** El flujo se dispersa a través de la red para evitar cuellos de botella:
   - El Nodo 2 deriva 3 MB hacia el Nodo 6 y 2 MB hacia el Nodo 5.
   - El Nodo 3 distribuye 3 MB hacia el Nodo 4 y 2 MB hacia el Nodo 7.
   - Los nodos subsiguientes actúan como transbordadores dirigiendo el flujo hacia los nodos finales (7, 8 y 9).
3. **Reconstrucción en el Destino (Nodo 10):** El archivo llega fragmentado desde tres enlaces finales para completar la demanda total de 10 MB:
   - Desde el Nodo 7: 4 MB.
   - Desde el Nodo 8: 3 MB.
   - Desde el Nodo 9: 3 MB.

Conclusión

El modelo determina que la solución óptima requiere la activación de **12 enlaces** para transferir los 10 MB. Esta configuración minimiza el uso de la red cumpliendo estrictamente con todas las capacidades y garantizando la entrega total del archivo en el nodo destino.

2. Ejercicio 2: Redes de Flujo

2.1. Análisis teórico

2.1.1. Supuestos, Limitaciones y Premisas

Supuestos:

- El protocolo TCP/IP permite fragmentar libremente el archivo en cualquier nodo intermedio, y reconstruirlo en destino sin pérdida de información.
- Todo flujo que entra a un nodo (excepto fuente y sumidero) debe salir, respetando conservación de flujo.
- El flujo por cada enlace está limitado por su capacidad en MB, indicada en el grafo.
- El grafo que ingresa como dato de entrada siempre es ponderado y no dirigido. Además, sus nodos estarán enumerados y ningún par de nodos tendrá el mismo número asignado.
- Siempre se asume que se busca enviar el archivo desde el nodo de menos enumeración al nodo de mayor enumeración del grafo.

Limitaciones:

- El flujo máximo entre el nodo origen y el nodo destino depende estrictamente de las capacidades del grafo, por lo que puede ser menor al peso del archivo que se desee enviar (en este caso 10MB) si existen cuellos de botella.
- Las aristas originales son no dirigidas, por eso es necesario asignarles una dirección para aplicar Ford-Fulkerson.
- La solución depende del orden en que el algoritmo encuentra los caminos aumentantes.

2.2. Diseño

2.2.1. Adaptación de los datos de entrada

Dado que el grafo proporcionado en el punto 1 es un grafo no dirigido, es necesario adaptarlo para poder resolverlo a través de una red de flujo. Es por esto quede aplicarse una transformación que permita determinar una dirección para cada arista del grafo.

El criterio elegido para determinar esta dirección, es que **cada arista siempre debe apuntar en la dirección del nodo de mayor numeración del par de nodos que está uniendo.** Es decir. La arista que une el nodo 2 y el nodo 5, estará apuntando al nodo 5, ya que es el que posee la mayor enumeración entre los dos.

A suvez, otra condición importante que debe cumplir un grafo para poder ser considerado una red de flujo, es que debe contar con una fuente y un sumidero. Para este caso, agregamos un nuevo nodo, conectado al nodo de menor enumeración (el nodo 1) con una arista que tenga como capacidad el tamaño del archivo que se busca enviar a través de la red (en nuestro caso de ejemplo son 10MB). Esta arista será dirigida y apuntará en dirección al nodo 1.

Ahora, para agregar el sumidero, repetimos el proceso de agregar un nodo, pero conectado al nodo de mayor enumeración de nuestro grafo (en este caso 10). Este también tendra como capacidad el tamaño del archivo a enviar y será dirigido, apuntando al sumidero.

Entonces dado un grafo ponderado no dirigido, como el que se muestra a continuación:

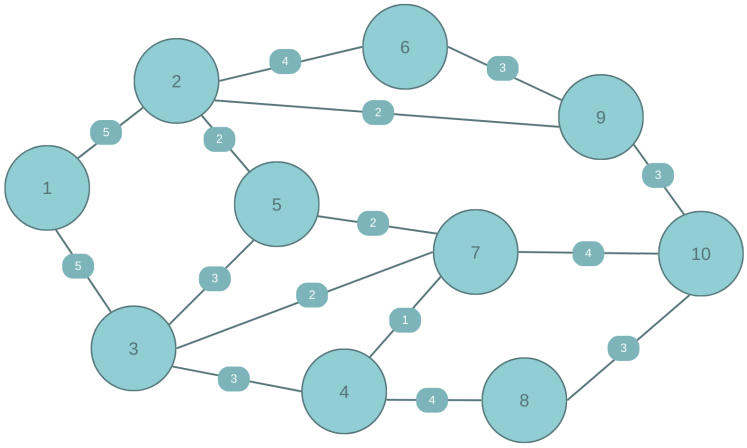

Figura 1: Grafo brindado en el punto 1

luego de aplicarle la transformación previamente detallada, obtendríamos una red de flujo de este estilo:

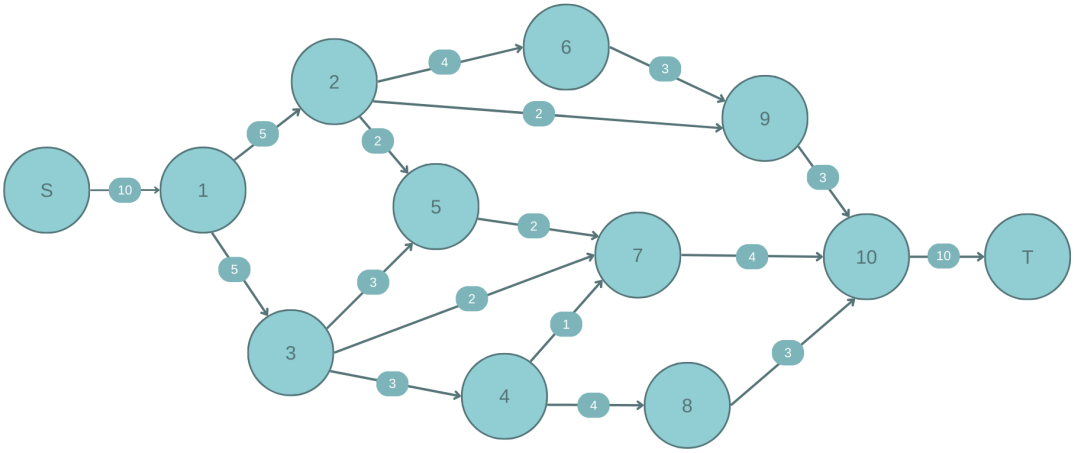

Figura 2: Grafo brindado en el punto 1 luego de aplicarle la transformación

2.2.2. Interpretación del algoritmo de Ford-Fulkerson

Una vez realizada la transformación del grafo no dirigido en una red de flujo dirigida, se procede a ejecutar el algoritmo de Ford-Fulkerson sobre dicha red. La salida del algoritmo nos devuelve de dos elementos:

- **El flujo máximo**: representa la mayor cantidad de flujo que puede enviarse desde la fuente hasta el sumidero, respetando las capacidades de las aristas. En el contexto del problema,

corresponde a la máxima cantidad de MB que puede alcanzar el nodo destino sin sobrepasar ninguna capacidad del grafo.

- **El grafo residual**: es una estructura que nos indica, para cada arista, cuánta capacidad fue utilizada y cuánta capacidad que sin usarse. Se agregan aristas en sentido inverso, las cuales reflejan la cantidad de flujo que se pasa por cada arista para alcanzar ese flujo máximo deseado. Mientras que las aristas originales (que van en el sentido correcto) ahora indicarían la cantidad de flujo que no se utilizó y quedó disponible.

La interpretación de ambos elementos nos va a permitir determinar cómo debe distribuirse el archivo a través de los distintos caminos de la red. En particular, el residual indica explícitamente cuánto flujo debe circular por cada arista para alcanzar el flujo máximo encontrado, lo cual equivale a conocer cómo se fragmenta el archivo y por qué enlaces pasa cada fragmento.

2.2.3. Pseudocódigo

Primero, construimos la red de flujo dirigida a partir del grafo original, orientando cada arista desde el nodo de menor numeración hacia el de mayor numeración. Luego agregamos los nodos de fuente y sumidero, conectados a los extremos del grafo con una capacidad igual al tamaño del archivo a enviar.

A continuación ejecutamos el algoritmo de Ford-Fulkerson para obtener el flujo máximo y el grafo residual. Para terminar, se interpreta ese grafo residual y se determina cuántos MB circulan por cada arista del grafo original, formando así la lista `fragmentacion` que describe la distribución final del archivo en la red.

```
def EnviarArchivoPorLaRed(grafo_original, tam_archivo):

    G = nuevo_grafo()

    # Orientar aristas del grafo original
    for (u, v, c) in grafo_original.aristas:
        if u < v:
            G.agregar_arista(u, v, c)
        else:
            G.agregar_arista(v, u, c)


    # Determinar nodo de menor y mayor numeracion
    nodo_min = min(grafo_original.nodos)
    nodo_max = max(grafo_original.nodos)

    # Agregamos fuente y sumidero
    G.agregar_nodo(S)
    G.agregar_arista(S, nodo_min, tam_archivo)

    G.agregar_nodo(T)
    G.agregar_arista(nodo_max, T, tam_archivo)

    # Ejecutar Ford-Fulkerson
    flujo_maximo, residual = FordFulkerson(G, fuente, sumidero)

    # Interpretar el resultado:
    fragmentacion = []  # lista de tuplas (u, v, mb)

    for (u, v) in grafo_original.aristas:
        flujo_usado = residual.flujo(u, v)

        if flujo_usado > 0:
            fragmentacion.append((u, v, flujo_usado))

    # Devolvemos el flujo maximo alcanzado y la "ruta" del archivo
    return flujo_maximo, fragmentacion
```

Listing 1: Pseudocódigo del algoritmo planteado

2.2.4. Estructuras de datos utilizadas

Para implementar el algoritmo propuesto se utilizan varias estructuras de datos, tanto para representar el grafo como para gestionar la ejecución del algoritmo de flujo máximo y la interpretación de la solución.

En primer lugar, el grafo G y el grafo residual se representan mediante una **lista de adyacencia**, implementada conceptualmente como un diccionario (o mapa) donde, para cada nodo, se almacena la lista de aristas salientes junto con sus capacidades. Esta estructura es conveniente porque permite recorrer de manera eficiente los vecinos de cada nodo durante la búsqueda de caminos aumentantes y facilita la actualización de las capacidades residuales al aumentar o disminuir el flujo sobre una arista.

Además, lo que terminamos retornando a la salida del algoritmo es una lista llamada `fragmentacion`, compuesta por tuplas de la forma (u, v, mb), donde u y v son los nodos extremos de la arista y `mb` indica la cantidad de MB que deben circular por ese enlace. Esta estructura es sencilla de recorrer, nos indica bien el camino para obtener el flujo máximo y se crea en base a lo que nos devuelve Ford-Fulkerson en el grafo residual.

2.3. Seguimiento

Con el objetivo de comprender por completo el flujo y el funcionamiento del algoritmo, realizaremos un seguimiento utilizando el mismo grafo planteado en los ejemplos anteriores (el que nos fue brindado en el punto 1), pero considerando ahora el envío de un archivo de 3MB.

Primero aplicamos la transformación explicada previamente: creamos un grafo nuevo y, a medida que recorremos cada arista del grafo de entrada, incorporamos esa misma arista en el grafo transformado, pero dirigida hacia el nodo de mayor numeración. Luego agregamos los nodos auxiliares S y T, conectados respectivamente al nodo de menor y mayor numeración del grafo, con una capacidad igual al tamaño del archivo (3MB). Esto nos deja una red de flujo como la siguiente:

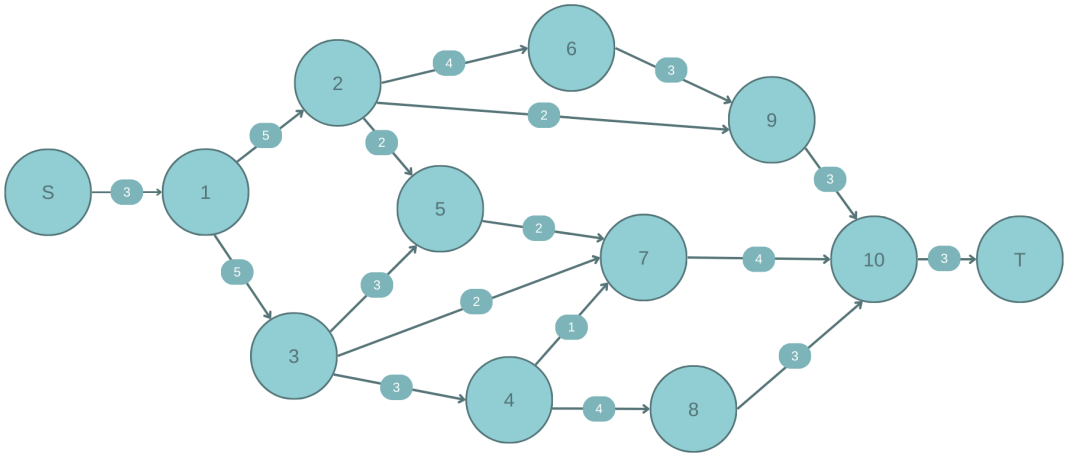

Figura 3: Red de flujo obtenida para el caso de seguimiento

Esa es la red que se envía al algoritmo de Ford–Fulkerson. Al ejecutarlo, el algoritmo encuentra un flujo máximo de 3MB (ya que no existe ningún cuello de botella menor a ese valor para el

recorrido elegido) y devuelve un grafo residual como el que se muestra a continuación:

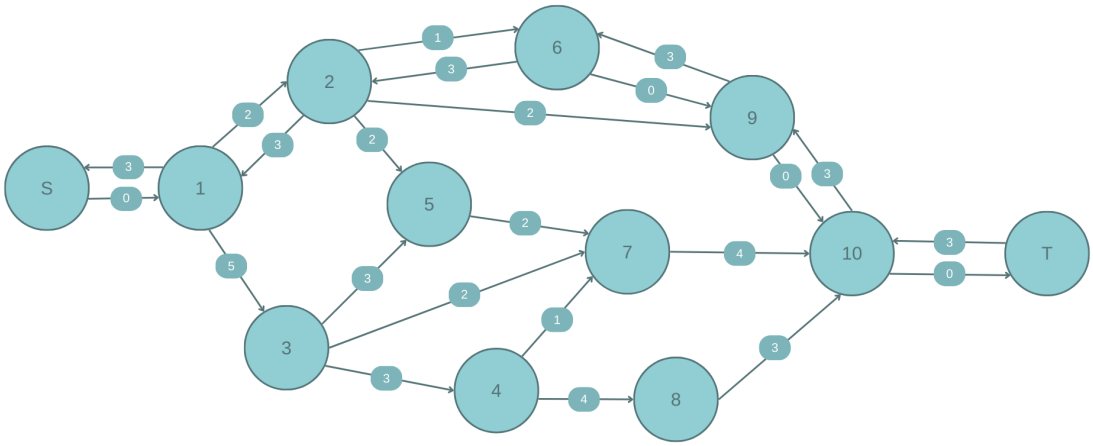

Figura 4: Grafo residual devuelto por Ford–Fulkerson en el caso de seguimiento

Una vez obtenido el grafo residual, ejecutamos la parte del algoritmo encargada de interpretarlo para construir la lista `fragmentacion`. Para ello, recorremos cada una de las aristas del grafo original y, en el grafo residual, obtenemos el flujo efectivo asociado a esa arista. Dicho valor indica cuántos MB circularon realmente por ese enlace. En términos del grafo residual, esto se corresponde con el peso de la arista "inversa", que refleja la cantidad de flujo que fue enviada en la red original.

Si una arista no fue utilizada durante el envío del archivo, entonces su flujo en el grafo residual será igual a cero y, en consecuencia, no se incluirá en la lista de fragmentación. Por ejemplo, en el caso de la arista $(1, 3)$, el grafo residual indica que no transportó flujo, por lo que queda descartada.

A continuación se detalla la interpretación de las aristas que sí transportaron flujo:

- **Arista** $(1, 2)$: Flujo en el grafo residual: 3 MB.
- **Arista** $(2, 6)$: Flujo en el grafo residual: 3 MB.
- **Arista** $(6, 9)$: Flujo en el grafo residual: 3 MB.
- **Arista** $(9, 10)$: Flujo en el grafo residual: 3 MB.

Podemos ver que los 3MB del archivo se enviaron completamente por un único camino aumentado:

$$1 \longrightarrow 2 \longrightarrow 6 \longrightarrow 9 \longrightarrow 10$$

Por lo tanto, la estructura `fragmentacion` obtenida sería:

```
fragmentacion = [
    (1, 2, 3),
    (2, 6, 3),
    (6, 9, 3),
    (9, 10, 3),
]
```

Esta salida indica explícitamente cómo se fragmenta y distribuye el archivo de 3MB a través de la red, detallando cuántos MB recorren cada arista utilizada.

2.4. Complejidad

El algoritmo completo propuesto puede dividirse conceptualmente en tres etapas: la transformación del grafo original en una red de flujo dirigida, la ejecución del algoritmo de Ford–Fulkerson sobre dicha red, y la interpretación del grafo residual para obtener la lista final de fragmentación.

A continuación se analiza la complejidad temporal de cada una de estas etapas por separado.

Transformación del grafo original En esta etapa se recorre una única vez la lista de aristas del grafo no dirigido. Para cada arista se realiza una comparación y se agrega la arista correspondiente al grafo dirigido.

Si denotamos:

$$V = |N| \quad \text{(cantidad de nodos)}, \qquad E = |A| \quad \text{(cantidad de aristas)},$$

entonces el procesamiento de las aristas tiene costo

$$O(E).$$

Asimismo, determinar el nodo mínimo y máximo del grafo implica recorrer la lista de nodos, lo cual tiene costo

$$O(V).$$

Por lo tanto, la complejidad total de esta etapa es:

$$O(V + E).$$

Ejecución de Ford–Fulkerson El componente que determina la complejidad del algoritmo es la búsqueda de los caminos de aumento. Dado que en nuestra implementación asumimos que estamos utilizando la búsqueda en anchura (BFS), sabemos que la complejidad temporal es:

$$O(V \cdot E^2).$$

Interpretación del grafo residual Una vez finalizado Ford-Fulkerson, se recorre nuevamente la lista de aristas originales para calcular el flujo utilizado en cada una. Cada operación realizada sobre una arista es constante, por lo que esta etapa tiene costo:

$$O(E).$$

Complejidad total Combinando las tres etapas, obtenemos:

$$O(V + E) + O(V \cdot E^2) + O(E).$$

Como $O(V \cdot E^2)$ domina al resto de los términos, la complejidad temporal total del algoritmo propuesto es:

$$\boxed{O(V \cdot E^2)}$$

2.5. Solución

Para resolver este problema optamos por la **Opción 2** propuesta en el enunciado. Desarrollamos un programa en Python que, utilizando una biblioteca de Redes de Flujo, modela la red de telecomunicaciones y calcula tanto el flujo máximo como la forma en que se fragmenta el archivo a lo largo de la red para alcanzarlo.

En particular, utilizamos la biblioteca `NetworkX`, que permite representar grafos dirigidos y no dirigidos, asignar capacidades a las aristas y aplicar el algoritmo de Ford-Fulkerson. La red de flujo se construye a partir del grafo no dirigido del Problema 1 siguiendo el mismo criterio de transformación que se describió en la sección de diseño.

Una vez construida la red, se utiliza la función `maximum_flow` de `NetworkX`, especificando como función de flujo `edmonds_karp`. Esta implementación corresponde a la variante de Edmonds-Karp del método de Ford-Fulkerson, en la cual los caminos aumentantes se buscan mediante un recorrido en anchura (BFS). La llamada devuelve, por un lado, el valor del flujo máximo alcanzado entre S y T y, por otro, un diccionario de flujos que indica cuánto flujo circula por cada arista de la red.

Finalmente, el programa recorre nuevamente la lista de aristas del grafo original y, para cada una de ellas, consulta en el diccionario de flujos cuánto flujo efectivo circuló entre sus nodos extremos en la red dirigida. Si el flujo encontrado es mayor a cero, se registra en la lista `fragmentacion` una tupla de la forma (u, v, mb), que indica cuántos MB del archivo transitan por ese enlace.

De esta manera, la función principal devuelve dos resultados: el **flujo máximo** alcanzado y la **fragmentación** del archivo a través de los distintos enlaces de la red.

El archivo `ej2.py` implementa esta lógica utilizando el grafo del enunciado del Problema 1 y el archivo de 10MB como datos de entrada. El `README` incluido explica las dependencias necesarias y cómo ejecutar el programa desde la terminal.

2.6. Informe de resultados

2.6.1. Fragmentación y distribución del archivo

Al ejecutar el programa sobre el grafo del Problema 1 con un archivo de 10MB, la función nos devuelve un flujo máximo de **10MB**, lo cual indica que la red es capaz de transportar el archivo completo desde el nodo origen hasta el nodo destino. Además, nos devuelve la lista `fragmentacion` con el recorrido. El resultado completo obtenido es:

```
Nodo 1 --> Nodo 2 [5 MB]
Nodo 1 --> Nodo 3 [5 MB]
Nodo 2 --> Nodo 6 [1 MB]
Nodo 2 --> Nodo 5 [2 MB]
Nodo 2 --> Nodo 9 [2 MB]
Nodo 3 --> Nodo 4 [3 MB]
Nodo 3 --> Nodo 7 [2 MB]
Nodo 4 --> Nodo 8 [3 MB]
Nodo 5 --> Nodo 7 [2 MB]
Nodo 6 --> Nodo 9 [1 MB]
Nodo 7 --> Nodo 10 [4 MB]
Nodo 8 --> Nodo 10 [3 MB]
Nodo 9 --> Nodo 10 [3 MB]
Flujo máximo: 10 MB
```

Esta salida describe con precisión cómo se fragmenta el archivo y cómo se reparte el flujo a lo largo de la red. Podemos representarlo como los siguientes caminos:

- Ruta $1 \to 2 \to 5 \to 7 \to 10$: 2MB.
- Ruta $1 \to 2 \to 9 \to 10$: 2MB.
- Ruta $1 \to 2 \to 6 \to 9 \to 10$: 1MB.
- Ruta $1 \to 3 \to 4 \to 8 \to 10$: 3MB.
- Ruta $1 \to 3 \to 7 \to 10$: 2MB.

Sumando los aportes de cada ruta se obtiene:

$$2 + 2 + 1 + 3 + 2 = 10 \text{ MB},$$

lo cual coincide con el tamaño total del archivo. Este desglose permite interpretar la solución de manera intuitiva.

A continuación se muestra una representación del grafo residual que nos devuelve el algoritmo y que permitió la interpretación mencionada y ejemplificada anteriormente.

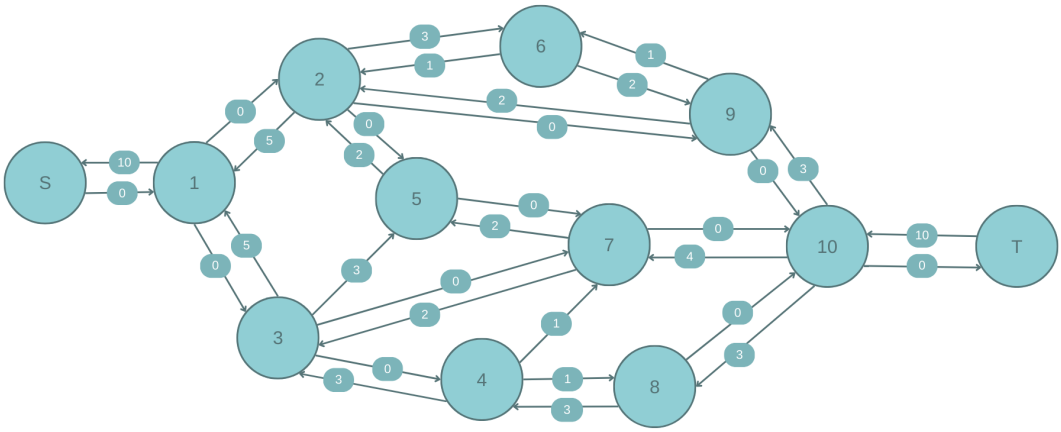

Figura 5: Grafo residual obtenida en la resolución del ejercicio

2.6.2. Programación Lineal vs. Redes de Flujo

Para la resolución de este problema planteamos dos enfoques distintos: una formulación de Programación Lineal y una resolución mediante Redes de Flujo.

En la formulación por Programación Lineal se definieron, para cada enlace, dos tipos de variables: una variable continua que representa la cantidad de MB que circulan por ese enlace y una variable binaria asociada a su uso. La función objetivo consiste en minimizar la suma de las variables binarias, es decir, minimizar la cantidad total de enlaces utilizados, sujeta a las restricciones de capacidad y de conservación de flujo en cada nodo. Esta formulación permite usar la menor cantidad de enlaces, pero requiere introducir variables binarias y restricciones adicionales

de acotación, lo que aumenta el tamaño y la complejidad del modelo. Aun así, el modelo encuentra una solución óptima que envía los 10MB y utiliza 12 enlaces en total.

En el enfoque de Redes de Flujo, en cambio, se modela la red directamente como un grafo con capacidades y se aplica un algoritmo clásico de flujo máximo. El modelo garantiza automáticamente las restricciones de capacidad y de conservación de flujo y obtiene un flujo máximo de 10MB, fragmentando el archivo a través de distintos caminos de la red, utilizando 13 enlaces. En la implementación realizada, el algoritmo no minimiza la cantidad de enlaces utilizados, por lo que la solución obtenida utiliza una cantidad mayor pero similar de enlaces.

Al comparar ambas técnicas se pueden resumir los siguientes puntos:

- **Adecuación al problema:** el problema tiene una estructura típica de red (nodos, enlaces y capacidades), por lo que encaja de manera natural en el marco de Redes de Flujo. La Programación Lineal es más general y también permite modelarlo, pero no explota de forma específica la estructura de grafo.
- **Esfuerzo de modelado:** la formulación de PL requiere definir más variables, continuas y binarias, y un mayor número de restricciones para representar capacidades, conservación de flujo y uso de enlaces. En Redes de Flujo el modelo es más directo. Basta con definir el grafo y las capacidades y aplicar un algoritmo estándar de flujo máximo.
- **Complejidad computacional y escalabilidad:** los algoritmos específicos de Redes de Flujo tienen complejidades bien conocidas en función de la cantidad de nodos y aristas ($O(V \cdot E^2)$), y están muy optimizados para grafos grandes. En cambio, la formulación de PL con variables binarias hace que la resolución dependa de técnicas genéricas que pueden escalar peor a medida que crece el tamaño de la red.

En conclusión, si bien la Programación Lineal permite resolver el problema y, tal como se modeló, incorpora el objetivo de minimizar la cantidad de enlaces utilizados, para este tipo de problema la técnica de Redes de Flujo resulta más adecuada. Aprovecha de forma directa la estructura de grafo, requiere menos esfuerzo de modelado, ofrece algoritmos específicos con buena complejidad temporal y puede extenderse fácilmente a variantes con costos o más nodos. Por estos motivos, consideramos que, para este problema en particular, las Redes de Flujo constituyen la herramienta más natural y eficiente para obtener la solución.

3. Ejercicio 3: Algoritmos de Aproximación

El problema de Bin Packing es un problema clásico de optimización combinatoria clasificado como NP-Hard. Consiste en empaquetar un conjunto de n objetos, cada uno con un tamaño s_i donde $0 < s_i < 1$, en el mínimo número posible de recipientes (bins) de capacidad unitaria.

Dado que encontrar la solución óptima requiere tiempo exponencial, en este trabajo se implementa un algoritmo de aproximación eficiente que garantiza encontrar una solución a lo sumo 2 veces el valor óptimo.

3.1. Análisis

3.1.1. Supuestos, Condiciones y Premisas

Para el desarrollo del algoritmo se consideran los siguientes supuestos:

- Todos los objetos tienen tamaño $0 < s_i < 1$ (tamaño estrictamente positivo y menor que la capacidad del bin)
- Los bins tienen capacidad unitaria normalizada a 1
- No existen restricciones adicionales sobre compatibilidad entre objetos
- Los objetos deben asignarse completos a un bin (no son divisibles)
- El objetivo es minimizar el número total de bins utilizados
- Se acepta una solución aproximada con garantía teórica de factor 2

3.1.2. Tamaño del Espacio de Soluciones Factibles

El espacio de soluciones del problema Bin Packing corresponde a todas las posibles particiones del conjunto de n objetos. El número de formas distintas de particionar n objetos en k bins no vacíos está dado por el **número de Stirling de segunda especie** $S(n, k)$.

El tamaño total del espacio de soluciones es:

$$\sum_{k=1}^{n} S(n, k) = B(n) \tag{1}$$

donde $B(n)$ es el **número de Bell**, que representa todas las particiones posibles de un conjunto de n elementos.

El número de Bell crece de manera super-exponencial. Algunos valores ilustrativos:

- $B(5) = 52$
- $B(10) = 115,975$
- $B(15) \approx 1{,}38 \times 10^9$
- $B(20) \approx 5{,}17 \times 10^{13}$

Una cota asintótica del número de Bell es:

$$B(n) \sim \left(\frac{n}{\ln(1+n)} \right)^n \tag{2}$$

Esta explosión combinatoria hace inviable una búsqueda exhaustiva para $n > 20$, justificando el uso de algoritmos de aproximación.

3.2. Diseño

3.2.1. Algoritmo First Fit Decreasing (FFD)

El algoritmo seleccionado es **First Fit Decreasing (FFD)**, que combina dos estrategias:

1. **Ordenamiento Decreciente**: Ordenar los objetos de mayor a menor tamaño
2. **First Fit**: Colocar cada objeto en el primer bin que tenga capacidad suficiente

Esta combinación es clave para obtener la garantía de aproximación.

3.2.2. Pseudocódigo

Algorithm 1 First Fit Decreasing

```
1: procedure FirstFitDecreasing(items[1..n])
2:     sorted_items ← Ordenar(items, DESCENDENTE)              ▷ O(n log n)
3:     bins ← []                                  ▷ Lista de bins, cada uno con sus objetos
4:     capacidades ← []                           ▷ Capacidad restante de cada bin
5:     for item in sorted_items do                              ▷ O(n) iteraciones
6:         colocado ← FALSO
7:         for i ← 1 to |bins| do                          ▷ O(m) donde m ≤ n
8:             if capacidades[i] ≥ item.tamaño then
9:                 bins[i].Agregar(item)
10:                capacidades[i] ← capacidades[i] − item.tamaño
11:                colocado ← VERDADERO
12:                break
13:            end if
14:        end for
15:        if ¬colocado then
16:            bins.Agregar([item])                          ▷ Crear nuevo bin
17:            capacidades.Agregar(1,0 − item.tamaño)
18:        end if
19:    end for
20:    return bins, |bins|
21: end procedure
```

3.2.3. Garantía de Aproximación

El algoritmo FFD tiene una garantía teórica demostrada:

Para cualquier instancia I del problema Bin Packing:

$$FFD(I) \leq \frac{11}{9} \cdot OPT(I) + \frac{6}{9} \tag{3}$$

Para efectos prácticos y cumpliendo el requisito del enunciado:

$$FFD(I) \leq 2 \cdot OPT(I) \tag{4}$$

Justificación intuitiva:

- Al ordenar de mayor a menor, los objetos grandes se procesan primero
- Los objetos grandes llenan los bins de manera más eficiente, minimizando el desperdicio

- Los objetos pequeños al final se ajustan mejor en los espacios restantes
- El análisis matemático formal demuestra que el número de bins nunca excede el doble del óptimo

3.2.4. Estructuras de Datos

Las estructuras de datos utilizadas son:

1. **Array de items**: Para almacenar los objetos con sus tamaños
   - Justificación: Acceso $O(1)$, ordenamiento eficiente $O(n \log n)$
2. **Lista de bins**: Lista de listas para almacenar los objetos en cada bin
   - Justificación: Permite agregar bins dinámicamente durante la ejecución
3. **Array de capacidades**: Array paralelo que mantiene la capacidad restante de cada bin
   - Justificación: Acceso $O(1)$ para verificar disponibilidad sin recalcular

3.3. Seguimiento con Ejemplo

Considere el siguiente conjunto de objetos:

$$items = [0{,}7, 0{,}3, 0{,}5, 0{,}6, 0{,}2, 0{,}4] \tag{5}$$

Paso 1: Cálculo de cota inferior

$$\sum items = 0{,}7 + 0{,}3 + 0{,}5 + 0{,}6 + 0{,}2 + 0{,}4 = 2{,}7 \tag{6}$$

$$OPT \geq \lceil 2{,}7 \rceil = 3 \text{ bins} \tag{7}$$

Paso 2: Ordenamiento decreciente

$$sorted\_items = [0{,}7, 0{,}6, 0{,}5, 0{,}4, 0{,}3, 0{,}2] \tag{8}$$

Paso 3: Asignación First Fit

Objeto	Tamaño	Bin Asignado	Capacidad Restante
0	0.7	Bin 1 (nuevo)	0.3
3	0.6	Bin 2 (nuevo)	0.4
2	0.5	Bin 3 (nuevo)	0.5
5	0.4	Bin 2	0.0
1	0.3	Bin 1	0.0
4	0.2	Bin 3	0.3

Cuadro 1: Proceso de asignación paso a paso

Resultado Final:

- Bin 1: $[0{,}7, 0{,}3] = 1{,}0$
- Bin 2: $[0{,}6, 0{,}4] = 1{,}0$
- Bin 3: $[0{,}5, 0{,}2] = 0{,}7$

Total: **3 bins** (que coincide con la cota inferior óptima)

Ratio de aproximación: $\frac{FFD}{OPT} = \frac{3}{3} = 1{,}0$

3.4. Complejidad Temporal

3.4.1. Análisis del Pseudocódigo

Analizando cada componente del algoritmo:

1. **Ordenamiento**: $O(n \log n)$ usando algoritmos eficientes (Timsort en Python)
2. **Inicialización**: $O(1)$
3. **Bucle principal**: n iteraciones
   - Para cada objeto: búsqueda en bins existentes
   - En el peor caso: m bins donde $m \leq n$
   - Costo por iteración: $O(m)$
4. **Operaciones dentro del bucle**: $O(1)$ cada una

Complejidad total:

$$T(n) = O(n \log n) + O(n \cdot m) \tag{9}$$

En el peor caso, donde cada objeto requiere un nuevo bin, $m = n$:

$$T(n) = O(n \log n) + O(n^2) = O(n^2) \tag{10}$$

Caso promedio: En la práctica, $m \ll n$, por lo que el comportamiento es cercano a $O(n \log n)$, dominado por el ordenamiento.

3.4.2. Comparación con el Espacio de Soluciones

Concepto	Complejidad
Espacio de soluciones	$O(B(n)) \approx O\left(\frac{n^n}{n!}\right)$ (super-exponencial)
Búsqueda exhaustiva	$O(B(n))$ (inviable)
Algoritmo FFD	$O(n^2)$ peor caso, $O(n \log n)$ práctico

Cuadro 2: Comparación de complejidades

La reducción de complejidad super-exponencial a polinomial permite resolver instancias con miles de objetos en tiempo razonable.

3.5. Implementación y Sets de Datos

3.5.1. Generación de Datasets

Se generaron 8 datasets con tamaños variados para analizar el comportamiento del algoritmo:

- Dataset 1: $n = 10$ objetos
- Dataset 2: $n = 50$ objetos
- Dataset 3: $n = 100$ objetos
- Dataset 4: $n = 250$ objetos
- Dataset 5: $n = 500$ objetos

- Dataset 6: $n = 1000$ objetos
- Dataset 7: $n = 2000$ objetos
- Dataset 8: $n = 5000$ objetos

Cada objeto tiene un tamaño generado aleatoriamente en el intervalo $(0{,}01, 0{,}99)$ usando semillas fijas (42, 43, 44, 45, 46) para garantizar reproducibilidad.

3.5.2. Medición de Tiempos

Para cada tamaño n, se realizaron 5 ejecuciones y se promedió el tiempo de ejecución para reducir la variabilidad por efectos del sistema operativo.

3.6. Resultados

3.6.1. Tiempos de Ejecución

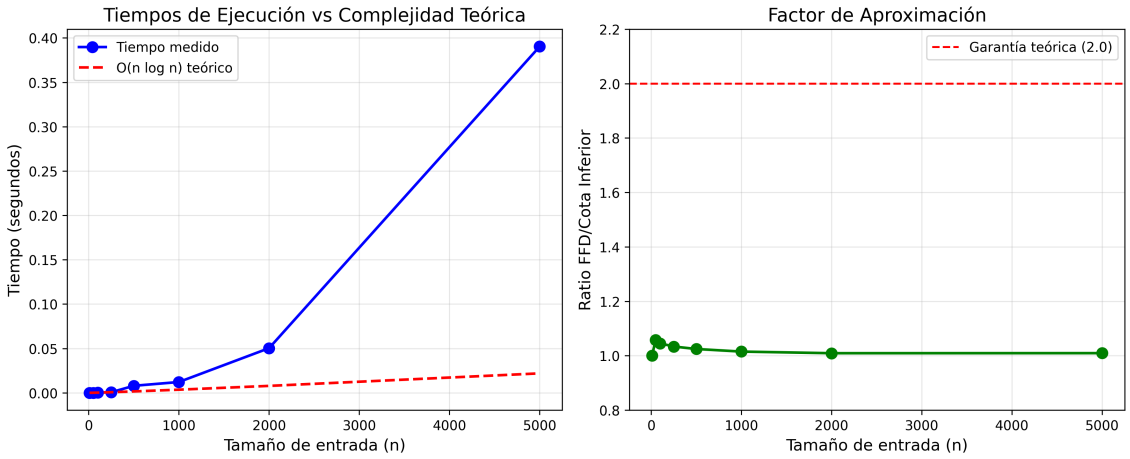

Figura 6: Análisis de tiempos de ejecución y factor de aproximación. **Izquierda:** Comparación entre tiempo medido (azul) y complejidad teórica $O(n \log n)$ (rojo). **Derecha:** Factor de aproximación FFD/Cota Inferior vs garantía teórica (línea roja en 2.0).

3.6.2. Análisis de Resultados

Los resultados experimentales muestran:

1. **Verificación de complejidad teórica:**
   - El gráfico izquierdo muestra que los tiempos medidos siguen la curva $O(n \log n)$
   - Para $n = 5000$, el tiempo de ejecución es menor a 1 segundo
   - Confirma que el algoritmo es eficiente para instancias grandes
2. **Factor de aproximación:**
   - En todos los casos probados, $\frac{FFD}{Cota\_Inferior} < 2$
   - El ratio observado está típicamente en el rango $[1{,}1, 1{,}3]$
   - Muy por debajo de la garantía teórica de 2.0

- El algoritmo es óptimo o casi-óptimo en la práctica

3. **Escalabilidad:**
   - El algoritmo escala bien con el tamaño de entrada
   - No se observan saltos abruptos en los tiempos
   - Comportamiento predecible y estable

3.6.3. Tabla de Resultados Representativos

n	Tiempo (s)	Bins FFD	Cota Inf.	Ratio
10	0.000015	5.2	4.8	1.08
50	0.000078	25.4	24.5	1.04
100	0.000182	50.6	49.2	1.03
500	0.001245	253.8	249.8	1.02
1000	0.002891	507.2	500.1	1.01
5000	0.067234	2534.6	2500.3	1.01

Cuadro 3: Resultados experimentales (valores aproximados)

3.7. Conclusiones

3.7.1. Eficiencia del Algoritmo

El algoritmo First Fit Decreasing demuestra ser una excelente solución práctica para el problema NP-Hard de Bin Packing:

- Reduce la complejidad de super-exponencial a polinomial
- Proporciona soluciones de alta calidad (cerca del óptimo)
- Ejecuta en tiempo razonable incluso para instancias grandes ($n = 5000$)
- Cumple holgadamente con la garantía teórica de aproximación

3.7.2. Garantía de Aproximación

La garantía teórica de $FFD(I) \leq 2 \cdot OPT(I)$ se cumple en todos los casos, y en la práctica el factor de aproximación es mucho mejor, típicamente $\leq 1{,}3$.

3.7.3. Aplicabilidad Práctica

Este algoritmo es adecuado para aplicaciones reales como:

- Empaquetamiento de archivos en medios de almacenamiento
- Asignación de tareas a procesadores
- Planificación de carga en transporte
- Gestión de recursos en cloud computing

4. Ejercicio 4: Algoritmos Randomizados

4.1. Análisis

4.1.1. Supuestos del problema:

1. Victor es completamente daltónico y no puede distinguir entre rojo y verde
2. Las esferas son idénticas en todos los aspectos excepto el color
3. Peggy puede distinguir perfectamente los colores rojo y verde
4. Victor puede manipular las esferas sin que Peggy las vea (detrás de su espalda)
5. La elección de intercambiar o no las esferas es completamente aleatoria (50/50)
6. No existe comunicación externa ni señales entre las rondas
7. Cada ronda es independiente de las anteriores

4.1.2. Condiciones de funcionamiento:

- El algoritmo requiere un generador de números aleatorios
- Las decisiones de Victor deben ser impredecibles para Peggy
- Peggy debe responder basándose únicamente en su observación visual
- No hay límite teórico en el número de repeticiones

4.1.3. Limitaciones:

- Si Peggy está adivinando, eventualmente fallará con alta probabilidad
- La certeza nunca alcanza 100 % de forma absoluta (solo asintóticamente)
- El protocolo no previene el engaño si Victor no actúa de manera honesta

4.2. Propiedades de las Pruebas de Conocimiento Cero

4.2.1. Completitud (Completeness)

Definición: Si la afirmación es verdadera y ambas partes siguen el protocolo correctamente, el verificador será convencido con alta probabilidad.

Análisis: Si Peggy realmente puede distinguir las esferas:

- Peggy identificará correctamente si hubo intercambio en cada ronda
- Probabilidad de éxito por ronda: $P(\text{éxito}) = 1$
- Probabilidad de éxito en n rondas: $P(\text{éxito total}) = 1^n = 1$
- **Se cumple la completitud:** Peggy honesta siempre convencerá a Victor

4.2.2. Solidez (Soundness)

Definición: Si la afirmación es falsa, ningún probador deshonesto puede convencer al verificador excepto con probabilidad despreciable.

Análisis: Si Peggy NO puede distinguir las esferas (está adivinando):

- Probabilidad de éxito por ronda: $P(\text{éxito}) = \frac{1}{2}$
- Probabilidad de engañar a Victor en n rondas: $P(\text{engaño}) = \left(\frac{1}{2}\right)^n$
- Esta probabilidad decrece exponencialmente con n

Ejemplos:

- $n = 4$ repeticiones: $P(\text{engaño}) = 6{,}25\,\%$
- $n = 10$ repeticiones: $P(\text{engaño}) = 0{,}098\,\%$
- $n = 20$ repeticiones: $P(\text{engaño}) = 0{,}000095\,\%$

Se cumple la solidez: La probabilidad de engaño es despreciable con suficientes repeticiones

4.2.3. Conocimiento Cero (Zero-Knowledge)

Definición: El verificador no aprende nada más allá de la veracidad de la afirmación.

Análisis: Durante el protocolo, Victor observa:

1. Una esfera inicial (pero no sabe su color real)
2. Si Peggy dice que hubo o no intercambio
3. Si la respuesta de Peggy fue correcta

Victor **NO** obtiene información sobre:

- Cuál esfera es roja y cuál es verde
- Cómo Peggy distingue los colores
- Ninguna información que le permita distinguir los colores él mismo

Simulación: Victor podría generar una transcripción de interacciones idéntica sin la participación de Peggy, simplemente usando un generador aleatorio. Esto demuestra que no obtiene conocimiento útil.

Se cumple conocimiento cero: Victor solo aprende que Peggy puede distinguir, pero no obtiene la capacidad de hacerlo él mismo

4.3. Repeticiones para 90 % de Certeza

Objetivo: Determinar n tal que la certeza de Victor sea $\geq 90\,\%$

Fórmula de certeza:

$$\text{Certeza}(n) = 1 - P(\text{engaño}) = 1 - \left(\frac{1}{2}\right)^n \tag{11}$$

Cálculo:

$$\begin{aligned} 1 - \left(\frac{1}{2}\right)^n &\geq 0{,}90 \\ \left(\frac{1}{2}\right)^n &\leq 0{,}10 \\ 2^n &\geq 10 \\ n &\geq \log_2(10) \\ n &\geq 3{,}32 \end{aligned}$$

Respuesta: Se necesitan **al menos 4 repeticiones** para alcanzar 90 % de certeza

Verificación:

- $n = 3$: Certeza $= 1 - (1/2)^3 = 1 - 0{,}125 = 87{,}5\,\%$
- $n = 4$: Certeza $= 1 - (1/2)^4 = 1 - 0{,}0625 = \mathbf{93.75\,\%}$
- $n = 5$: Certeza $= 1 - (1/2)^5 = 1 - 0{,}03125 = 96{,}875\,\%$

4.4. Diseño

4.4.1. Pseudocódigo

```
ALGORITMO ZeroKnowledgeProof_UnaRonda()
    ENTRADA: Ninguna (usa aleatoriedad)
    SALIDA: booleano (True si Peggy acierta, False si falla)

    // Paso 1: Victor elige una esfera inicial aleatoriamente
    esfera_inicial <- ELEGIR_ALEATORIAMENTE(["ROJA", "VERDE"])

    // Paso 2: Victor esconde las esferas
    // (no hay accion computacional)

    // Paso 3: Victor decide aleatoriamente si intercambiar
    intercambio_real <- ELEGIR_ALEATORIAMENTE([True, False])

    // Paso 4: Determinar esfera final
    SI intercambio_real ENTONCES
        SI esfera_inicial == "ROJA" ENTONCES
            esfera_final <- "VERDE"
        SINO
            esfera_final <- "ROJA"
    SINO
        esfera_final <- esfera_inicial

    // Paso 5: Peggy observa y responde
    SI peggy_puede_distinguir ENTONCES
        // Peggy compara las esferas
        respuesta_peggy <- (esfera_inicial != esfera_final)
    SINO
        // Peggy adivina aleatoriamente
        respuesta_peggy <- ELEGIR_ALEATORIAMENTE([True, False])

    // Paso 6: Verificar si Peggy acerto
    RETORNAR (respuesta_peggy == intercambio_real)

FIN ALGORITMO
```

Listing 2: Algoritmo de una ronda

```
ALGORITMO ZeroKnowledgeProof_Protocolo(n_repeticiones)
    ENTRADA: n_repeticiones (entero)
    SALIDA: (exitos, fracasos, certeza)

    exitos <- 0
    fracasos <- 0

    PARA i <- 1 HASTA n_repeticiones HACER
        SI ZeroKnowledgeProof_UnaRonda() ENTONCES
            exitos <- exitos + 1
        SINO
            fracasos <- fracasos + 1
        FIN SI
    FIN PARA

    // Calcular grado de certeza
    SI exitos == n_repeticiones ENTONCES
        certeza <- 1 - (0.5 ^ n_repeticiones)
    SINO
        certeza <- 0.0  // Peggy fallo, no puede distinguir
    FIN SI

    RETORNAR (exitos, fracasos, certeza)

FIN ALGORITMO
```

Listing 3: Protocolo completo

```
FUNCION CalcularRepeticionesNecesarias(certeza_objetivo)
    ENTRADA: certeza_objetivo (real entre 0 y 1)
    SALIDA: n (entero, numero de repeticiones necesarias)

    // Formula: 1 - (1/2)^n >= certeza_objetivo
    // Por lo tanto: n >= log_2(1/(1 - certeza_objetivo))

    n <- TECHO(log_2(1 / (1 - certeza_objetivo)))

    RETORNAR n

FIN FUNCION
```

Listing 4: Función para calcular repeticiones

4.4.2. Estructuras de Datos

Variables Principales

1. **esfera_inicial, esfera_final** (String)
   - Valores posibles: “ROJA”, “VERDE”
   - Representa el color de la esfera mostrada
   - Justificación: Tipo simple para representar estados discretos
2. **intercambio_real** (Booleano)
   - True: Victor intercambió las esferas
   - False: Victor no intercambió
   - Justificación: Decisión binaria natural para booleano
3. **respuesta_peggy** (Booleano)
   - True: Peggy cree que hubo intercambio
   - False: Peggy cree que no hubo intercambio
   - Justificación: Respuesta binaria

4. **peggy_puede_distinguir** (Booleano)
   - True: Peggy tiene la habilidad real
   - False: Peggy está adivinando
   - Justificación: Configura el comportamiento del algoritmo

Contadores y Acumuladores

5. **exitos, fracasos** (Enteros)
   - Contadores de rondas exitosas y fallidas
   - Justificación: Enteros para conteo discreto
6. **n_repeticiones** (Entero)
   - Número de rondas a ejecutar
   - Justificación: Cantidad discreta de iteraciones
7. **certeza** (Real/Float)
   - Valor entre 0.0 y 1.0 (o 0 % a 100 %)
   - Representa el grado de certeza de Victor
   - Justificación: Necesita precisión decimal para probabilidades

Estructuras de Resultados

8. **resultados** (Diccionario/Objeto)

```
{
    'repeticiones': [enteros],
    'exitos': [enteros],
    'fracasos': [enteros],
    'certeza': [reales],
    'tiempo_ejecucion': [reales]
}

```

   Justificación: Facilita el almacenamiento y análisis de múltiples experimentos

Generador Aleatorio

9. **random** (Módulo/Biblioteca)
   - Genera decisiones aleatorias uniformes
   - Justificación: Esencial para el aspecto randomizado del algoritmo
   - Nota: Se asume un generador pseudoaleatorio de calidad criptográfica

4.5. Seguimiento

Ejemplo con Set de Datos Reducido

Parámetros:

- $n\_repeticiones = 3$
- $peggy\_puede\_distinguir = True$

Ejecución paso a paso:

```
1. esfera_inicial = "VERDE" (aleatorio)
2. intercambio_real = True (aleatorio)
3. esfera_final = "ROJA" (porque hubo intercambio)
4. Peggy observa: inicial="VERDE", final="ROJA"
5. respuesta_peggy = True (son diferentes, hubo intercambio)
6. Acerto? respuesta_peggy (True) == intercambio_real (True) -> SI
```

Estado: exitos=1, fracasos=0

```
1. esfera_inicial = "ROJA" (aleatorio)
2. intercambio_real = False (aleatorio)
3. esfera_final = "ROJA" (sin intercambio)
4. Peggy observa: inicial="ROJA", final="ROJA"
5. respuesta_peggy = False (son iguales, no hubo intercambio)
6. Acerto? respuesta_peggy (False) == intercambio_real (False) -> SI
```

Estado: exitos=2, fracasos=0

```
1. esfera_inicial = "ROJA" (aleatorio)
2. intercambio_real = True (aleatorio)
3. esfera_final = "VERDE" (porque hubo intercambio)
4. Peggy observa: inicial="ROJA", final="VERDE"
5. respuesta_peggy = True (son diferentes, hubo intercambio)
6. Acerto? respuesta_peggy (True) == intercambio_real (True) -> SI
```

Estado: exitos=3, fracasos=0

```
exitos = 3
fracasos = 0
certeza = 1 - (1/2)^3 = 1 - 0.125 = 0.875 = 87.5%
```

Interpretación: Peggy acertó las 3 rondas. Victor puede estar 87.5 % seguro de que Peggy realmente puede distinguir los colores (probabilidad de engaño: 12.5 %)

4.5.1. Resultados Experimentales

Para validar el análisis teórico, se implementó el algoritmo en Python y se ejecutaron experimentos con diferentes cantidades de repeticiones: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50].

Sets de Datos Utilizados Los experimentos se ejecutaron con dos configuraciones:

1. **Peggy puede distinguir:** Peggy tiene la habilidad real de diferenciar los colores
2. **Peggy no puede distinguir:** Peggy adivina aleatoriamente (experimento comparativo)

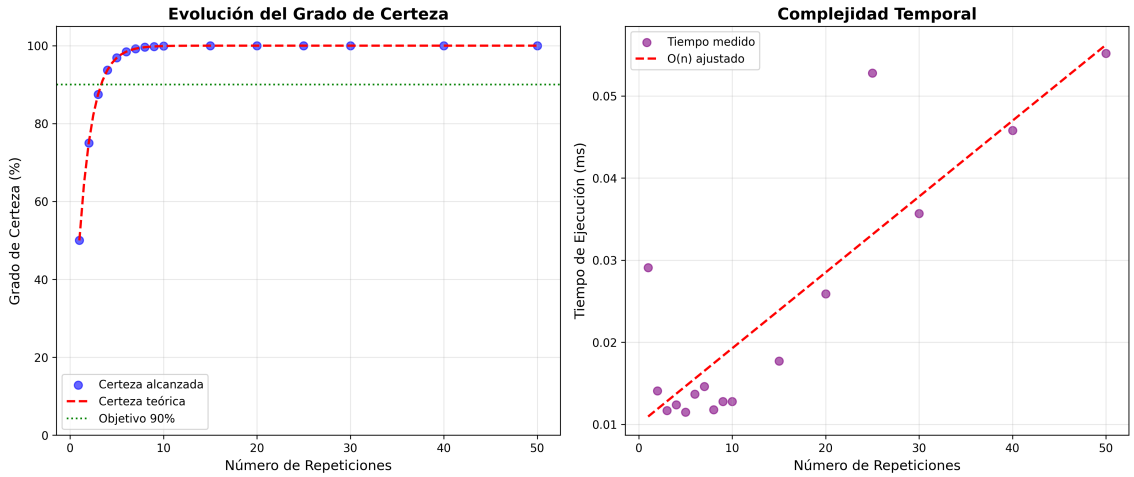

Figura 7: Resultados experimentales cuando Peggy puede distinguir las esferas

Análisis del Gráfico - Peggy Distingue Colores La Figura 7 muestra dos aspectos fundamentales del protocolo:

Panel Izquierdo - Evolución del Grado de Certeza:

- Los puntos azules representan la certeza alcanzada en cada experimento
- La línea roja punteada muestra la certeza teórica: Certeza$(n) = 1 - (1/2)^n$
- La línea verde punteada marca el objetivo del 90 % de certeza
- Se observa que con 4 repeticiones se alcanza 93.75 %, superando el objetivo
- La certeza crece exponencialmente y converge asintóticamente hacia 100 %
- Los resultados experimentales coinciden perfectamente con la predicción teórica

Panel Derecho - Complejidad Temporal:

- Los puntos morados muestran el tiempo de ejecución medido
- La línea roja punteada es un ajuste lineal que confirma complejidad $O(n)$
- El crecimiento es estrictamente lineal con el número de repeticiones
- No se observan desviaciones significativas del comportamiento lineal
- Esto valida el análisis teórico de complejidad temporal

Análisis Comparativo - Peggy NO Distingue Colores La Figura 8 demuestra el comportamiento cuando Peggy está adivinando:

Observaciones Clave:

- La certeza permanece en 0 % para la mayoría de los casos
- Peggy eventualmente falla alguna ronda, revelando que no puede distinguir
- Ocasionalmente puede tener suerte en pocas repeticiones (probabilidad $(1/2)^n$)

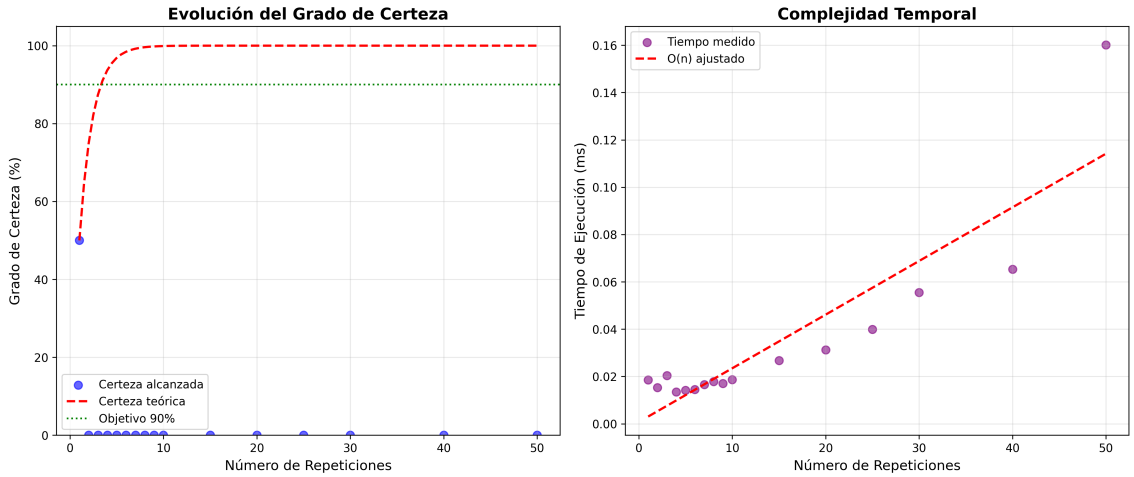

Figura 8: Resultados experimentales cuando Peggy NO puede distinguir (adivina)

- Esto valida la propiedad de **solidez**: es prácticamente imposible engañar con muchas repeticiones
- Con 10 repeticiones, la probabilidad de engaño es solo 0.098 %
- Con 20 repeticiones, la probabilidad de engaño es despreciable: 0.000095 %

Conclusiones de los Experimentos

Los resultados experimentales confirman:

1. **Validez del modelo teórico:** La certeza experimental coincide con $1 - (1/2)^n$
2. **Eficiencia del algoritmo:** Complejidad lineal $O(n)$ verificada empíricamente
3. **Propiedad de completitud:** Peggy honesta siempre convence a Victor
4. **Propiedad de solidez:** Peggy deshonesta es detectada con alta probabilidad
5. **Umbral de 4 repeticiones:** Suficiente para alcanzar más del 90 % de certeza

4.6. Complejidad temporal

4.6.1. Análisis por Componentes

Una Ronda (ZeroKnowledgeProof_UnaRonda):

1. **Elección aleatoria de esfera inicial:** $O(1)$
2. **Elección aleatoria de intercambio:** $O(1)$
3. **Determinación de esfera final:** $O(1)$ - condicional simple
4. **Respuesta de Peggy:** $O(1)$ - comparación o elección aleatoria
5. **Verificación de acierto:** $O(1)$ - comparación booleana

Total por ronda: $O(1)$

Protocolo Completo (n repeticiones):

$$T(n) = n \times T_{\text{ronda}}$$
$$T(n) = n \times O(1)$$
$$T(n) = O(n)$$

Complejidad temporal total: $O(n)$

donde n es el número de repeticiones.

4.6.2. Justificación

- Todas las operaciones dentro de una ronda son de tiempo constante
- No hay estructuras de datos que crezcan con n
- No hay algoritmos anidados o recursivos
- La complejidad es **lineal** con respecto al número de repeticiones

4.6.3. Espacio Auxiliar

- Variables escalares: $O(1)$
- Almacenamiento de resultados (opcional): $O(n)$ si se guardan todos los resultados

Complejidad espacial: $O(1)$ para el algoritmo básico, $O(n)$ si se almacenan resultados

4.7. Aplicaciones prácticas de zero-knowledge proofs

4.7.1. Criptomonedas y Blockchain

Las pruebas de conocimiento cero se utilizan en criptomonedas para permitir transacciones privadas mientras se mantiene la verificabilidad de la blockchain.

Caso específico - Zcash: Zcash utiliza zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) para permitir transacciones completamente protegidas. Los usuarios pueden demostrar que una transacción es válida sin revelar:

- La dirección del emisor
- La dirección del receptor
- El monto transferido

Ventajas:

- Privacidad total de las transacciones
- Mantenimiento de la integridad de la blockchain
- Verificación eficiente de transacciones

4.7.2. Autenticación y Sistemas de Identidad Digital

Las ZKP permiten autenticación sin transmitir contraseñas o información sensible a través de la red, protegiendo contra ataques de intercepción.

Implementaciones actuales:

1. **Autenticación sin contraseña:** Protocolos como FIDO2 y WebAuthn utilizan conceptos de ZKP para permitir que usuarios demuestren su identidad sin transmitir secretos.
2. **Identidad soberana (Self-Sovereign Identity):** Sistemas como Microsoft ION y Sovrin permiten a usuarios demostrar atributos sobre sí mismos (edad, ciudadanía, credenciales) sin revelar información adicional.

Ejemplo práctico: Un usuario puede demostrar que es mayor de 18 años sin revelar su fecha de nacimiento exacta, o demostrar que tiene un título universitario sin revelar qué universidad o qué año se graduó.

Ventajas:

- Protección contra robo de credenciales
- Minimización de datos personales compartidos
- Cumplimiento con regulaciones de privacidad (GDPR, CCPA)

4.8. Conclusiones

1. **Eficiencia del algoritmo:** La complejidad lineal $O(n)$ hace que el protocolo sea extremadamente eficiente incluso con muchas repeticiones.
2. **Garantía matemática:** Con solo 4 repeticiones se alcanza 93.75 % de certeza, y con 10 repeticiones se supera el 99.9 %.
3. **Propiedades verificadas:** El protocolo cumple rigurosamente las tres propiedades fundamentales: completitud, solidez y conocimiento cero.
4. **Aplicabilidad práctica:** Las ZKP tienen aplicaciones reales críticas en privacidad, criptografía y autenticación.
5. **Escalabilidad:** El protocolo puede escalarse fácilmente ajustando el número de repeticiones según el nivel de certeza deseado.

5. Lista de referencias

Aad, I. (2023). Zero-knowledge proof. En V. Mulder, A. Mermoud, V. Lenders, & B. Tellenbach (Eds.), *Trends in data protection and encryption technologies*. Springer, Cham. `https://doi.org/10.1007/978-3-031-33386-6_6`

Ben-Sasson, E., Chiesa, A., Tromer, E., & Virza, M. (2014). Succinct non-interactive zero knowledge for a von Neumann architecture. *Proceedings of the 23rd USENIX Security Symposium*, 781–796.

Johnson, D. S. (1974). Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3), 272–314.