UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

TB024 - Teoría de Algoritmos | 2do Cuatrimestre 2025

Trabajo Práctico 1

Alumno	Padrón	Email
Maximiliano Nicolas Otero Silvera	108634	motero@fi.uba.ar
Emanuel Gaston Choque Ramirez	110895	egchoque@fi.uba.ar
Daniel Mamani	109932	dmamani@fi.uba.ar
Iñaki Pujato	109131	ipujato@fi.uba.ar

Índice

1. Problema 1

1.1. Análisis

- El programa considera que las unidades del grafo están en MB.
- Considera el envío de 10MB.
- La red es fija y sus enlaces y capacidades no se modifican.
- TCP/IP reensambla correctamente los fragmentos, no debemos ocuparnos del orden de llegada, solo de que lleguen.
- No hay pérdidas de paquetes
- Todos los enlaces son dirigidos y su flujo va en dirección del 1 al 10.

1.2. Diseño

Variables:

- $X_{i,j}$ como una variable binaria que indica si el enlace (i, j) se utiliza.
- $F_{i,j}$ como una variable continua que indica el Flujo del enlace (i, j).

La variables $F_{i,j}$ está en MB al representar el tráfico posible en ese enlace de red.

Constantes:

- W = 10, constante del tamaño total del archivo.
- Los enlaces estan limitados superiormente por su capacidad, dada por el grafo.

1.3. Modelo de Programación Lineal

Estas variables funcionan en el modelo bajo las siguientes restricciones:

- $F_{i,j} \geq 0$, no puede haber flujo negativo en un enlace
- $F_{i,j} \cdot X_{i,j} \leq Capacidad_{i,j}$, al usar un enlace, el mismo no puede tener mayor tráfico que su capacidad
- Para cada nodo:
  - $\sum_j X_{1,j} = W$
  - $\sum_j X_{j,10} = W$
  - $\sum_j X_{i,j} - \sum_k X_{k,i} = 0, \forall i \in \{2, \ ..., 9\}$

Por último, nuestro objetivo es el mínimo de enlaces utilizados, por lo que nuestra función objetivo Z será:

$$Z_{min} = \sum_i \sum_j X_{i,j}, \forall i < j$$

1.4. Solución

A continuación se detalla un pseudocódigo de la solución.

```
FUNCIÓN ProgLineal(Grafo, nodo_inicio, nodo_fin, capacidad_maxima):

  // 1. Crear modelo de optimización
  CREAR problema COMO ModeloLineal(minimizar)

  // 2. Obtener todas las aristas del grafo
  aristas = Grafo.obtener_aristas()

  // 3. Crear variables de decisión
```

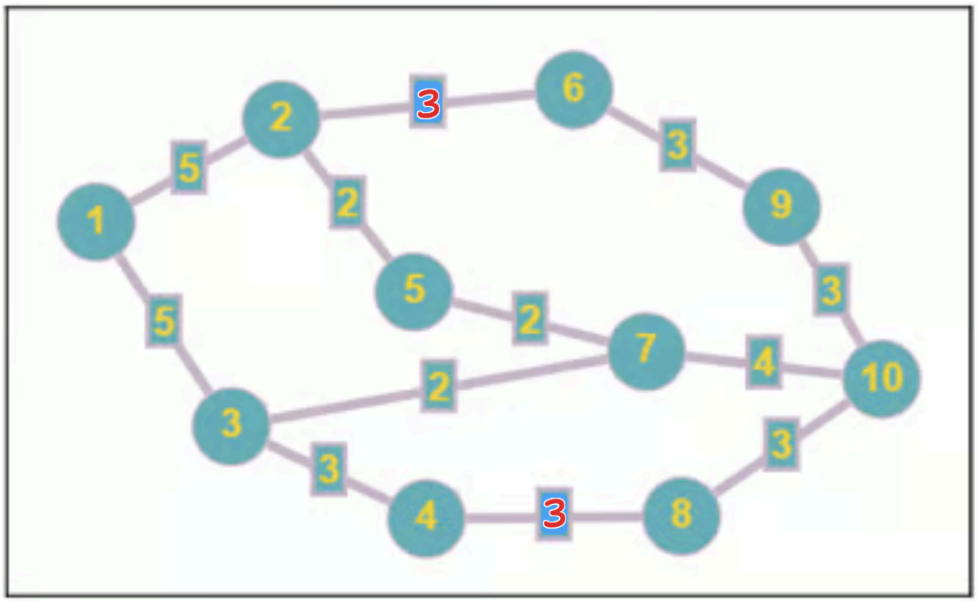
```
    PARA CADA arista (i, j) EN aristas HACER:
        X[i,j] = VariableBinaria()        // Indica si se usa el enlace (i,j)
        F[i,j] = VariableContinua(>=0)    // Flujo que pasa por el enlace (i,j)
    FIN PARA

    // 4. Función objetivo: minimizar cantidad de aristas usadas
    problema.objetivo = SUMA( X[i,j] PARA TODAS las aristas )

    // 5. Restricciones de capacidad
    PARA CADA arista (i, j) EN aristas HACER:
        capacidad = Grafo.adyacencia[i][j]
        AGREGAR restricción: F[i,j] <= capacidad * X[i,j]
    FIN PARA

    // 6. Restricciones de conservación de flujo
    nodos = todos los nodos presentes en aristas

    PARA CADA nodo n EN nodos HACER:
        inflow  = SUMA( F[i,j] PARA aristas donde j == n )
        outflow = SUMA( F[i,j] PARA aristas donde i == n )
        SI n == nodo_inicio:
            AGREGAR restricción: outflow - inflow == capacidad_maxima // debe salir todo desde inicio
        SINO SI n == nodo_fin:
            AGREGAR restricción: inflow - outflow == capacidad_maxima // debe llegar todo al fin
        SINO:
            AGREGAR restricción: inflow - outflow == 0   // nodo intermedio
        FIN SI

    FIN PARA

    // 7. Resolver el modelo
    RESOLVER problema

    // 8. retornar solución

    RETORNAR solución

FIN FUNCIÓN
```

Vemos acá como con claridad cómo se incluyen todas las restricciones y variables mencionadas en el diseño. Se comienza definiendo las variables, ambas binarias y constantes, y luego las restringimos. Todas las restricciones que hacen a nuestro modelo representar el problema real están incluidas y son tenidas en cuenta a la hora de resolver el problema.
El grafo modela la red, contemplando los enlaces con sus capacidades máximas y luego se restringe el flujo que la solución puede utilizar según este máximo.

Algo muy importante a notar es que en la definición del problema se define como uno de minimización. Esto es fundamental ya que condiciona cómo vamos a optimizar la solución. En otro caso, si buscamos por ejemplo el flujo máximo que permite la red, sería una problema de maximización. Esto es importante porque determina cómo las variables serán optimizadas en búsqueda de la solución final.

1.5. Informe de Resultados

La solución óptima encontrada por el modelo está dada por el siguiente gráfico e incluye 12 aristas:



Los caminos formados son:

- 1 -> 2 -> 6 -> 9 -> 10, con flujo 3 MB
- 1 -> 2 -> 5 -> 7 -> 10, con flujo 2 MB
- 1 -> 3 -> 7 -> 10, con flujo 2 MB
- 1 -> 3 -> 4 -> 8 -> 10, con flujo 3 MB

Está es la solución con menos enlaces utilizados.

2. Problema 2

2.1. Análisis

- Para la resolución de este ejercicio se utilizó la variante de Edmonds Karp del algoritmo de Ford Fulkerson. Esta decisión fue tomada ya que se pide encontrar la forma de enviar el paquete utilizando la menor cantidad de aristas posibles. Edmonds Karp utiliza BFS para encontrar un camino desde la fuente hacia el sumidero en cada iteración por lo cual, encuentra el camino con menor cantidad de enlaces.
- El algoritmo requiere que el grafo ingrese ya transformado como analizaremos en el punto **Transformación del grafo**

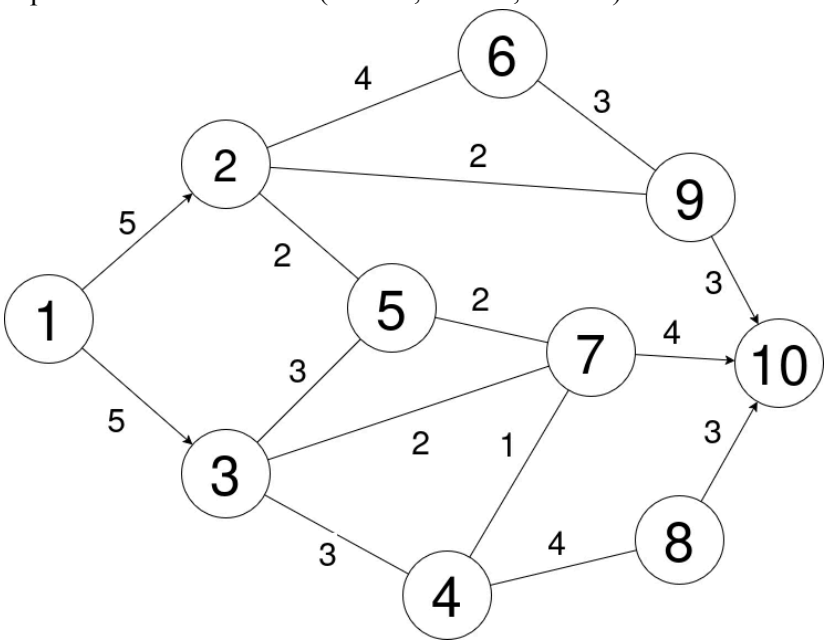
2.2. Diseño

2.2.1. Transformación del grafo

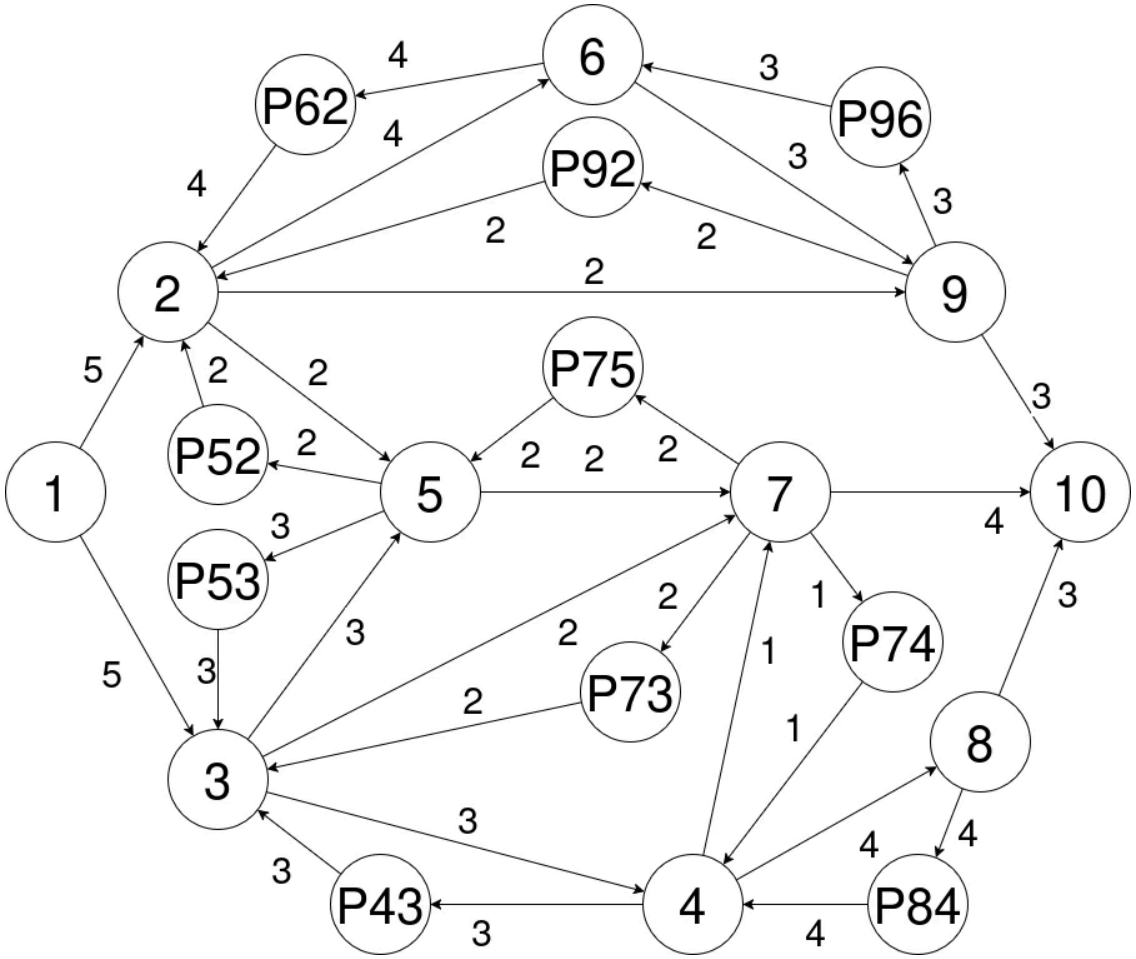
Para poder considerar un grafo como una red de flujo el mismo debe cumplir ciertas restricciones

- Debe ser un grafo dirigido
- No puede haber ciclos (A -> A)
- No puede haber aristas antiparalelas (A -> B, B -> A)
- Debe haber una única fuente
- Debe haber un único sumidero

De todas estas restricciones el grafo como se encuentra cumple únicamente con la segunda, ya que carece de ciclos. Para cumplir con sumidero y fuente únicos, dado que queremos llevar el paquete desde 1 hasta 10, podemos hacer que las aristas que conectan a 1 con 2 y 3 tengan sentido saliente (1 -> 2, 1 -> 3) y lo mismo para 10 pero en sentido entrante (7 -> 10, 8 -> 10, 9 -> 10).



Todavía nos quedan las demás aristas con ambos sentidos, formando aristas antiparalelas. Para esto, debemos generar un vértice pivote en uno de los sentidos entre cada par de vértices.
Sea PXY el vértice pivote en el sentido $X \text{ -> } PXY \text{ -> } Y$, las aristas $X \text{ -> } PXY$, $PXY \text{ -> } Y$ deben tener capacidad CXY. Por lo cual, haciendo este cambio para todas las aristas que no tenían dirección resulta



Ahora ya nos encontramos en condiciones de poder correr Ford Fulkerson para resolver la red de flujo.

Una vez que se terminó de ejecutar el algoritmo, nos retornará el flujo máximo.
Si el flujo máximo es menor a 10MB (largo del paquete a enviar) entonces el paquete no puede enviarse por la red de la facultad.
Además, utilizaremos una versión modificada del algoritmo original en la cual vamos guardando los caminos utilizados para después informar como se debe fragmentar el paquete y la cantidad de MB por camino.

2.2.2. Pseudocódigo

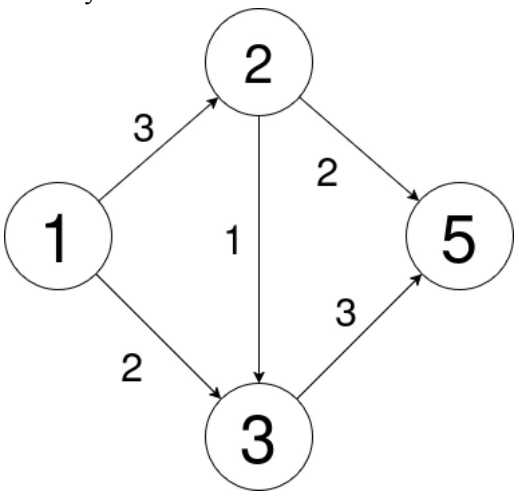
```
FUNCIÓN EdmondsKarp(Grafo, fuente, sumidero, caminos):
   INICIAR flujo_maximo = 0, camino_encontrado = []

   // Mientras exista un camino desde la fuente al sumidero
   // BFS actualiza camino_encontrado
   MIENTRAS BFS(Grafo, fuente, sumidero, camino_encontrado) sea VERDADERO hacer:

      // 1. Buscamos el cuello de botella (flujo mínimo) en el camino hallado
      flujo_camino = INFINITO
      v = sumidero

      MIENTRAS v != fuente hacer:
         u = camino_encontrado[v]  // u es el nodo anterior a v en el camino
         flujo_camino = min(flujo_camino, Grafo[u][v].capacidad)
         v = u
      FIN MIENTRAS

      // 2. Guardamos el camino utilizado junto con su flujo máximo
      caminos.agregar( (camino_encontrado, flujo_camino) )

      // 3. Actualizamos las capacidades residuales del grafo
      v = sumidero
      MIENTRAS v != fuente hacer:
         u = camino_encontrado[v]
         // Restar flujo en la arista original
         Grafo[u][v].capacidad = Grafo[u][v].capacidad - flujo_camino
         // Sumar flujo en la arista residual
         Grafo[v][u].capacidad = Grafo[v][u].capacidad + flujo_camino

         v = u
      FIN MIENTRAS

      // 4. Actualizamos el flujo total
      flujo_maximo = flujo_maximo + flujo_camino
   FIN MIENTRAS

   RETORNAR flujo_maximo
FIN FUNCIÓN

FUNCIÓN Algoritmo(Grafo, fuente, sumidero, flujo_esperado):
   INICIAR caminos = []
   flujo_maximo = EdmondsKarp(Grafo, fuente, sumidero, caminos)

   SI flujo_maximo < flujo_esperado:
      IMPRIMIR “No se puede fragmentar el paquete en la red de la facultad”
      RETORNAR

   PARA CADA c EN caminos:
      IMPRIMIR “Camino {{ c[0] }} con flujo {{ c[1] }}”

FIN FUNCIÓN
```

2.2.3. Estructuras de datos utilizadas

Además del grafo que modela la red, utilizamos dos listas, una para guardar el camino que BFS encuentra desde la fuente al sumidero en cada iteración y otra que almacena estos caminos para así mostrar los que son necesarios para fragmentar el paquete utilizando la menor cantidad de aristas.

2.3. Seguimiento

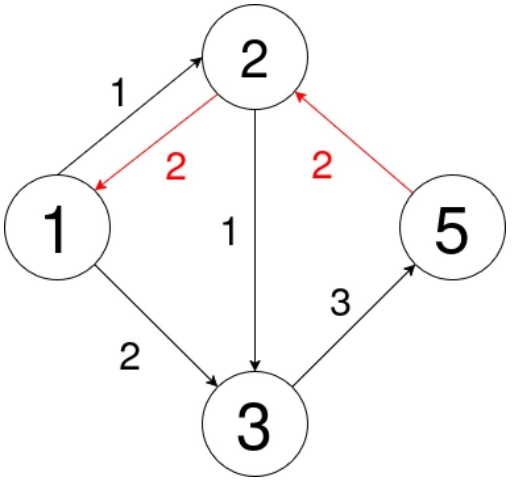
Dado el siguiente grafo, y la fuente '1' y el sumidero '5'



Edmonds Karp

Iteración 1

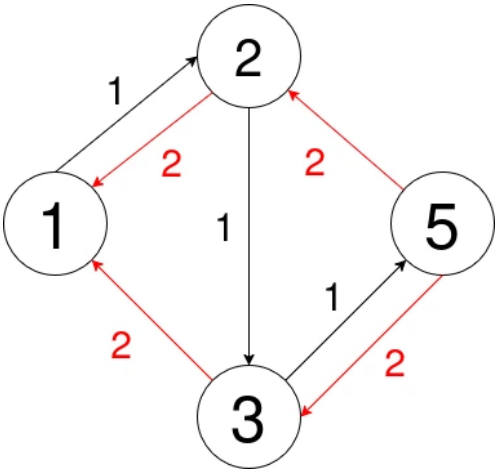
Edmonds Karp buscará el camino más corto desde la fuente hasta el sumidero. Por lo cual en esta primera iteración tomará el camino 1 -> 2 -> 5 y luego buscará la capacidad mínima del camino, en este caso 2. El flujo máximo alcanzado es 2, se agrega la tupla ([1 -> 2 -> 5], 2) a los caminos recorridos y el grafo residual resultara:



Si estuviéramos utilizando Ford Fulkerson, se hubiera seleccionado el primer camino que se encontrara realizando un DFS desde la fuente hasta el sumidero. Por esto mismo, FF podría seleccionar por ejemplo el camino 1 -> 2 -> 3 -> 5, por lo cual FF no es el algoritmo indicado si buscamos minimizar la cantidad de aristas utilizadas.

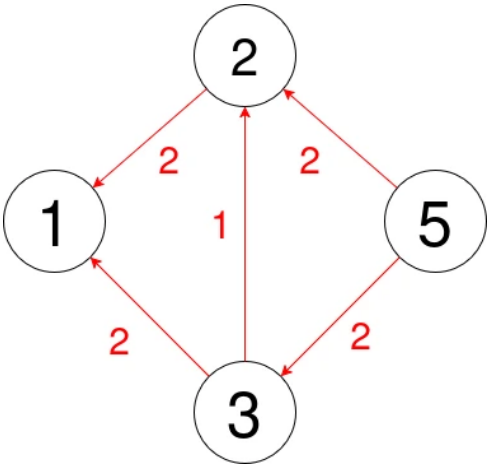
Iteración 2

El camino más corto ahora es 1 -> 3 -> 5 con capacidad ociosa 2. Por lo cual, el flujo máximo aumenta a 4, se agrega la tupla ([1 -> 3 -> 5], 2) a los caminos recorridos y el grafo residual resulta



Iteración 3

Nos queda un único camino desde 1 hasta 5, 1 -> 2 -> 3 -> 5 con capacidad ociosa 1. Por lo cual, el flujo máximo aumenta a 5, se agrega la tupla ([1 -> 2 -> 3 -> 5], 1) a los caminos recorridos y el grafo residual resulta:



Iteración 4

No existe otro camino desde 1 hasta 5, se retorna el flujo máximo encontrado (5).

Volviendo al algoritmo

Ahora que tenemos el flujo máximo alcanzado y los caminos utilizados, vemos si $flujo\_maximo < flujo\_esperado$. En ese caso, el archivo no se puede enviar en la red modelada por el grafo. En caso contrario, imprimimos y guardamos los caminos junto con el largo del fragmento para cada camino.

2.4. Complejidad

V: $Cantidad\ de\ vértices\ del\ grafo$
E: $Cantidad\ de\ aristas\ del\ grafo$
C: $Capacidad\ máxima\ del\ grafo$

Sabemos que Edmonds Karp tiene complejidad temporal $O(VE^2)$, la pequeña modificación que realizamos para poder recolectar los caminos utilizados no sobrepasa este umbral por lo cual se sigue manteniendo el mismo. Por otro lado, al imprimir los caminos recolectados el peor de los casos es $O(VE)$ ya que está demostrado que el máximo de caminos que Edmonds Karp encuentra es VE. Por lo cual la complejidad temporal del algoritmo resulta $O(VE^2)$.

2.5. Informe de Resultados

2.5.1. Fragmentación

Existe la posibilidad de enviar el archivo utilizando la red de la facultad, el mismo se debe fragmentar de la siguiente manera:

1. 5MB desde 1 a 2 y 5MB desde 1 a 3.
2. 2MB desde 2 a 9, 1MB desde 2 a 6 y 2MB desde 2 a 5. 2 MB desde 3 a 7 y 3MB desde 3 a 4.
3. 2MB desde 9 a 10. 2MB desde 7 a 10. 1MB desde 6 a 9. 2MB desde 5 a 7. 3MB desde 4 a 8.
4. 1MB desde 9 a 10. 2MB desde 7 a 10. 3MB desde 8 a 10.

O lo que es lo mismo que

Camino 1 -> 2 -> 9 -> 10 con flujo 2MB
Camino 1 -> 3 -> 7 -> 10 con flujo 2MB
Camino 1 -> 2 -> 6 -> 9 -> 10 con flujo 1MB
Camino 1 -> 2 -> 5 -> 7 -> 10 con flujo 2MB
Camino 1 -> 3 -> 4 -> 8 -> 10 con flujo 3MB

Utilizando en total 13 aristas únicas.

2.5.2. PL vs RF

Pese a que programación lineal otorga un mejor resultado que redes de flujo el factor decisorio en este caso es la complejidad. Al resolverlo con PL tenemos a las variables X (entera bivalente) y F (continua) por lo cual es un problema de PL Mixto el cual tiene una complejidad exponencial de $O(2^E)$ mientras que al resolverlo con redes de flujo analizamos que la complejidad es de $O(VE^2)$. En conclusión, PL puede otorgar un mejor resultado que RF pero su tiempo de ejecución escala exponencialmente a medida que aumentan las aristas del grafo que modela la red de la facultad.

3. Problema 3

3.1. Enunciado

Tenemos un conjunto de n objetos, donde el tamaño S_i del i-ésimo objeto cumple que $0 < S_i < 1$. El objetivo es empaquetar todos los objetos en el mínimo número posible de recipientes de tamaño unitario: cada recipiente puede contener cualquier subconjunto de los objetos cuyo tamaño total no exceda 1. El problema así planteado es NP-Hard (Karp, 1972).

Se desea encontrar un algoritmo eficiente que encuentre una solución aproximada con una garantía de a lo sumo 2 veces el valor de la solución óptima.

3.2. Análisis

3.2.1. Supuestos, condiciones y limitaciones

- El algoritmo procesa los objetos en el orden dado sin re-ordenarlos (Next Fit; Johnson, 1973).
- Los tamaños S_i son constantes, $0 < S_i < 1$, y no pueden dividirse ni deformarse.
- Todos los recipientes tienen capacidad $C = 1$.

3.2.2. Espacio de soluciones factibles

El problema planteado es una variante del *Bin Packing Problem*. Si se intentara resolverlo mediante fuerza bruta para encontrar el óptimo global, se tendría que evaluar todas las posibles asignaciones de n elementos en k recipientes.

El espacio de particiones de n elementos crece según los Números de Bell $B(n)$, con comportamiento asintótico $B(n) \sim (\frac{n}{\ln(n)})^n$ (Rosen, 2019). Este crecimiento super exponencial refuerza la inviabilidad de fuerza bruta en instancias grandes para un problema NP-Hard (Karp, 1972), justificando una heurística con garantía.

Por ejemplo, para valores pequeños de n, la cantidad de combinaciones explota rápidamente, haciendo que un enfoque exacto sea computacionalmente inviable (NP-Hard) para conjuntos de datos grandes. Esto justifica la necesidad de un algoritmo de aproximación eficiente como alternativa práctica.

3.3. Diseño

3.3.1. Algoritmo propuesto

Se implementa Next Fit por su simplicidad y garantía 2-aproximada (Johnson, 1973; Coffman et al., 1997).

La estrategia consiste en mantener abierto únicamente el recipiente actual. Si el siguiente objeto cabe en él, se coloca allí; de lo contrario, se cierra el recipiente actual (no se vuelve a usar) y se abre uno nuevo para el objeto.

3.3.2. Pseudocódigo

```
FUNCION next_fit(items S):
  // inicializar lista de recipientes R = [r_1]
  j = 1

  PARA CADA item S_i EN S:
    SI S_i entra en el recipiente r_j:
      agregar S_i a r_j
    SINO:
      j += 1
      crear r_j y agregar S_i

  RETORNAR j // cantidad de recipientes usados
```

3.3.3. Garantía de aproximación

El algoritmo de *Next Fit* garantiza que la cantidad de recipientes utilizados nunca exceda el doble de la cantidad óptima de recipientes.

Teorema: Para cualquier lista de ítems (L) se cumple $(NF(L) \leq 2 \cdot OPT(L)$ (Johnson, 1973; Coffman et al., 1997; Kleinberg & Tardos, 2006).

Demostración: Sea $k = NF(L)$. Denotemos como $C(R_i)$ la suma de tamaños de los ítems en el recipiente R_i. El algoritmo abre el recipiente R_{i+1} sólo cuando el primer ítem que irá allí (llamémoslo x_{i+1}) no entra en R_i. Por lo tanto para cada i con $1 \leq i \leq k - 1$:

$$C(R_i) + x_{i+1} > 1$$

y como $C(R_{i+1}) \geq x_{i+1}$, se obtiene $C(R_i) + C(R_{i+1}) > 1, \quad (1 \leq i \leq k - 1)$.

Tomando pares alternados disjuntos hay al menos $[\frac{k-1}{2}]$ de ellos, así:

$$S(L) = \sum_{j=1}^{k} C(R_j) > \frac{k-1}{2}$$

Dado que cada recipiente aporta a lo sumo 1 unidad de capacidad:

$$OPT(L) \geq S(L) > \frac{k-1}{2} \Rightarrow k < 2 \cdot OPT(L) + 1 \Rightarrow k \leq 2 \cdot OPT(L)$$

Concluye que *Next Fit* es un algoritmo 2-aproximado.

3.3.4. Estructuras de datos utilizadas

La ausencia de búsqueda entre recipientes cerrados elimina la necesidad de estructuras más complejas (Coffman et al., 1997).

Para la implementación del algoritmo *Next Fit*, se han seleccionado estructuras de datos lineales

simples.

Dado que la lógica del algoritmo es estrictamente secuencial y *online* (no revisa decisiones pasadas ni mira hacia el futuro), no se requieren estructuras de acceso aleatorio complejo ni de ordenamiento.

1. **Arreglo Dinámico / Lista (para la entrada)**

- **Descripción:** Se utiliza una lista (implementada como list en Python) para almacenar la secuencia de objetos de entrada S.
- **Justificación:** El algoritmo requiere recorrer los elementos una única vez en el orden dado. La lista permite iteración secuencial en tiempo $O(n)$. Al no requerir reordenamiento (como en *Decreasing First Fit*), no se incurre en costos adicionales de procesamiento previo.

2. **Lista de listas (para la salida)**

- **Descripción:** La estructura principal de retorno es una lista donde cada elemento es, a su vez, una lista que representa un recipiente conteniendo los tamaños de los objetos asignados.
- **Justificación:**

Eficiencia Espacial: Esta estructura crece dinámicamente según la cantidad de recipientes necesarios (k), evitando reservar memoria para el peor caso (n recipientes).

Acceso Local: El algoritmo *Next Fit* solo necesita interactuar con el último recipiente creado (el "recipiente actual"). El acceso al último elemento de una lista y la inserción de un nuevo elemento al final (append) son operaciones de tiempo constante amortizado $O(1)$.

Justificación de la Ausencia de Estructuras Complejas:

A diferencia de algoritmos como *First Fit* o *Best Fit*, que deben buscar entre todos los recipientes abiertos para encontrar espacio disponible (lo que justificaría el uso de Árboles Binario de Búsqueda o Heaps Maximales para optimizar la búsqueda a $O(\ln(n))$), *Next Fit* descarta inmediatamente cualquier recipiente cerrado. Al eliminar la necesidad de búsqueda histórica, una estructura lineal simple es suficiente y garantiza la complejidad temporal $O(n)$ y la mínima sobrecarga de memoria (overhead).

3.4. Seguimiento

El siguiente ejemplo ilustra la mecánica básica de *Next Fit* tal como se describe en Johnson (1973).

Para el conjunto $S = [0.6,\ 0.3,\ 0.7,\ 0.2,\ 0.1]$ y capacidad $C = 1$:

Paso	Objeto(S_i)	Estado recipiente actual (Espacio libre)	¿Entra?	Acción	Estado final de recipientes
Inicio	-	Recipiente 1 (Libre: 1.0)	-	Inicializar	[R1(0.0)]
1	0.6	R1 (Libre: 1.0)	Sí ($0.6 \leq 1.0$)	Agregar a R1	[R1(0.6)]
2	0.3	R1 (Libre: 0.4)	Sí ($0.3 \leq 0.4$)	Agregar a R1	[R1(0.6, 0.3)]
3	0.7	R1 (Libre: 0.1)	No ($0.7 > 0.1$)	Cerrar R1. Abrir R2. Agregar a R2	[R1(Cerrado), R2(0.7)]

4	0.2	R2 (Libre: 0.3)	Sí ($0.2 \leq 0.3$)	Agregar a R2	[R1(Cerrado), R2(0.7, 0.2)]
5	0.1	R2 (Libre: 0.1)	Sí ($0.1 \leq 0.1$)	Agregar a R2	[R1(Cerrado), R2(0.7, 0.2, 0.1)]

Resultado Final: Se utilizaron 2 recipientes.
- Recipiente 1: Contiene $\{0.6,\ 0.3\}$ (Lleno: 0.9).
- Recipiente 2: Contiene $\{0.7,\ 0.2,\ 0.1\}$ (Lleno: 1.0).

3.5. Complejidad

3.5.1. Complejidad del algoritmo Next Fit

- **Complejidad Temporal:** $O(n)$
  - El algoritmo procesa cada objeto exactamente una vez en orden secuencial.
  - Para cada objeto se realizan operaciones de tiempo constante: verificación de capacidad disponible y asignación del recipiente actual.
  - No requiere búsqueda, ordenamiento ni backtracking.

- **Complejidad Espacial:** $O(n)$
  - En el peor caso, cada objeto requiere su propio recipiente (cuando todos los objetos tienen tamaño > 0.5).
  - El espacio utilizado es proporcional al número de recipientes generados, que está acotado por n.

3.5.2. Tamaño del espacio de soluciones factibles

Para n objetos, el problema de *Bin Packing* presenta un espacio de soluciones que corresponde a todas las posibles particiones del conjunto de objetos en subconjuntos (recipientes).

Número de particiones posibles: El tamaño del espacio está dado por los Números de Bell $B(n)$, con comportamiento asintótico $(\frac{n}{\ln(n)})^n$.

Comparación de complejidades:

Enfoque	Complejidad temporal	Garantía de solución
Algoritmo exacto (Fuerza Bruta)	$O(B(n)) \approx O((\frac{n}{\ln(n)})^n)$	Óptima
Next Fit (propuesto)	$O(n)$	2-aproximación

Justificación del enfoque: La reducción de la complejidad super-exponencial a lineal hace viable el procesamiento de instancias de tamaño real, manteniendo una garantía de aproximación acotada.

3.6. Sets de datos

3.6.1. Descripción de los conjuntos de datos

Para validar el comportamiento del algoritmo *Next Fit* se diseñaron 6 conjuntos de datos que

representan diferentes escenarios del problema *Bin Packing*:

1. **Caso Básico**
   - **Datos**: [0.6, 0.3, 0.7, 0.2, 0.1]
   - **Características:** Distribución mixta que permite combinaciones eficientes.
   - **Propósito:** Evaluar el comportamiento estándar del algoritmo.
2. **Ítems Grandes**
   - **Datos:** [0.9, 0.8, 0.95, 0.85, 0.7]
   - **Características:** Todos los ítems ocupan > 70% de la capacidad del recipiente.
   - **Propósito:** Analizar el comportamiento cuando la mayoría de ítems requieren recipientes individuales.
3. **Ítems Pequeños**
   - **Datos:** [0.1, 0.2, 0.15, 0.05, 0.3]
   - **Características:** Todos los ítems son $\leq$ 30% de la capacidad.
   - **Propósito:** Verificar la eficiencia cuando múltiples ítems pueden compartir recipiente.
4. **Peor Caso Teórico**
   - **Datos:** [0.51, 0.51, 0.51, 0.51, 0.51]
   - **Características:** Todos los ítems son > 50% de la capacidad.
   - **Propósito:** Demostrar el peor escenario donde se alcanza el factor aproximación máximo (2.0).
5. **Caso Mixto**
   - **Datos:** [0.4, 0.6, 0.2, 0.8, 0.1, 0.9, 0.3, 0.7]
   - **Características:** Alternancia entre ítems grandes y pequeños.
   - **Propósito:** Evaluar la robustez del algoritmo con patrones variados.
6. **Caso Aleatorio**
   - **Datos:** 20 valores generados aleatoriamente entre 0.01 y 0.99 (seed=42, reproducible).
   - **Características:** Distribución uniforme, tamaño representativo.
   - **Propósito:** Simular condiciones reales de entrada de datos.

Los patrones (grandes, pequeños, mixtos, peor caso > 0.5) son típicos en evaluación de heurísticas de Bin Packing (Coffman et al., 1997).

3.6.2. Análisis de resultados por conjunto

3.6.2.1. Métricas de evaluación

Para cada conjunto se midieron las siguientes métricas:

- **Recipientes utilizados:** Cantidad total de recipientes requeridos.
- **Cota inferior teórica:** Número mínimo de recipientes según la suma total de ítems.
- **Factor de aproximación:** Ratio entre recipientes utilizados y la cota inferior.
- **Eficiencia:** Porcentaje de utilización promedio de los recipientes.
- **Tiempo de ejecución:** Tiempo computacional en milisegundos.

3.6.2.2. Resultados obtenidos

Conjunto	Ítems	Recipientes	Factor aprox.	Eficiencia	Tiempo trimmed (ms)
Básico	5	2	1.00	95.00%	0.0048
Ítems grandes	5	5	1.00	84.00%	0.0024

Ítems pequeños	5	1	1.00	80.00%	0.0017
Peor caso	5	5	1.67	51.00%	0.0060
Mixto	8	4	1.00	100.00%	0.0045
Aleatorio	20	10	1.25	78.20%	0.00769

3.6.3. Validación de la garantía teórica

Los resultados confirman que en todos los casos el factor de aproximación se mantiene dentro del límite teórico:

- **Factor máximo observado:** 1.67 (Peor caso)
- **Factor máximo teórico:** 2.00
- **Margen de seguridad:** 16.5%

El caso "Peor caso" con ítems de tamaño 0.51 demuestra empíricamente el comportamiento límite del algoritmo, donde cada ítem requiere su propio recipiente debido a que ningún par puede compartir espacio.

3.6.4. Generación y reproducibilidad de datasets

Para garantizar la reproducibilidad experimental y facilitar la validación de resultados:

1. **Casos deterministas:** Los datasets básicos, ítems grandes/pequeños, peor caso y mixto están predefinidos para asegurar resultados consistentes entre ejecuciones.
2. **Caso aleatorio:** Se genera dinámicamente con distribución uniforme en el rango [0.01, 0.99] con precisión de 2 decimales para simular datos del mundo real.
3. **Escalabilidad:** La estructura modular permite agregar nuevos casos de prueba sin modificar la lógica del algoritmo.

Los datasets están diseñados para cubrir el espectro completo de comportamientos del algoritmo, desde el caso óptimo hasta el peor caso teórico, proporcionando una validación integral del rendimiento.

3.7. Tiempos de Ejecución

3.7.1. Metodología de medición

Para validar empíricamente la complejidad temporal teórica $O(n)$, se realizaron experimentos sistemáticos con datasets de diferentes tamaños:

- **Tamaños evaluados:** 50, 100, 200, 500, 1000, 2000, 5000 objetos.
- **Warmups:** 10.
- **Presupuesto de tiempo:** $\geq$ 1 segundo por tamaño (ajuste dinámico de repeticiones).
- **Medición:** perf_counter_ns; GC deshabilitado por muestra; media recortada (trim=20%) y mediana.
- **Generación de datos:** Objetos aleatorios con distribución uniforme [0.1, 0.9], seed = 42.
- **Observación:** Con pocas repeticiones (30/40) se observaron outliers por ruido del SO; se mitigó con warmups, trim y presupuesto temporal.

3.7.2. Resultados experimentales

Los experimentos confirman el comportamiento lineal esperado:

Tamaño (n)	Tiempo trimmed (ms)	Mediana (ms)	Factor de aproximación
50	0.0089	0.0089	1.348
100	0.0170	0.0170	1.294
200	0.0371	0.0371	1.346
500	0.0933	0.0931	1.353
1000	0.1869	0.1866	1.381
2000	0.3732	0.3723	1.364
5000	0.9340	0.9338	1.357

Fuente: escalabilidad_1s.csv

3.7.3. Análisis gráfico

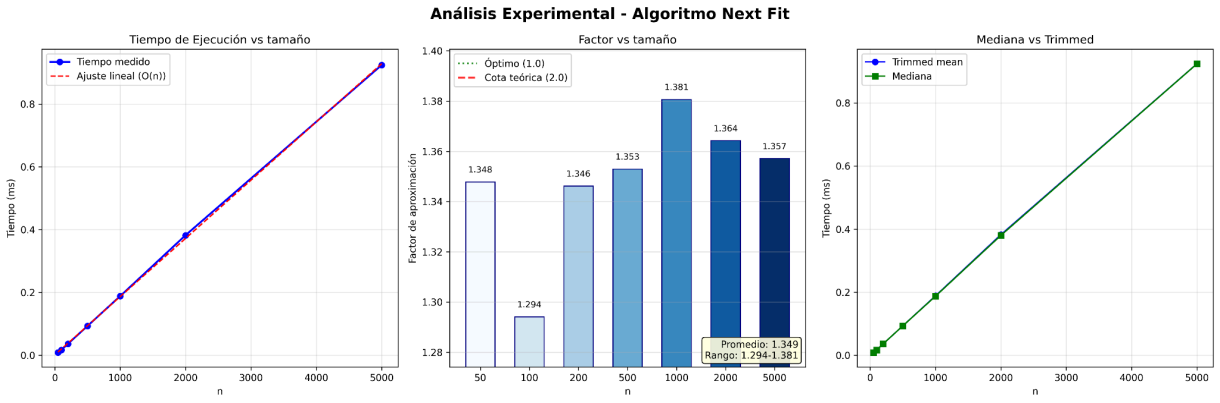


Figura 1: Análisis experimental de tiempos de ejecución del algoritmo Next Fit (corrida canónica ≥ 1s por n)

Observaciones clave:

1. **Linealidad confirmada:** pendiente normalizada ≈ 1.00.
2. **Escalabilidad:** El algoritmo mantiene eficiencia constante hasta datasets de 5000 objetos.
3. **Robustez estadística:** La mediana y la media coinciden; baja sensibilidad a outliers.
4. **Factor de aproximación estable:** ≈ 1.35, muy por debajo de la cota teórica 2.0.

3.7.4. Validación de la complejidad

- **Complejidad teórica:** $O(n)$.
- **Complejidad empírica:** Confirmada como lineal.
- **Eficiencia relativa:** Coeficiente 1.00 (pendiente normalizada ≈ 1.00)
- **Factor de aproximación promedio:** 1.35 (muy inferior a la cota 2.0).

Con 30/40 repeticiones se observaron desviaciones en $n = 2000$ atribuibles al ruido del sistema (scheduler/GC). Al usar la metodología canónica ($\geq 1s\ por\ n$, warmups = 10, trim = 20%) los residuos disminuyen y el ajuste lineal se estabiliza.

La pendiente constante en el gráfico tiempo vs tamaño confirma que el algoritmo *Next Fit* efectivamente opera en tiempo lineal, cumpliendo con las expectativas teóricas y siendo adecuado para aplicaciones en tiempo real con datasets.

La linealidad observada concuerda con la complejidad teórica $O(n)$ del recorrido único del algoritmo (Kleinberg & Tardos, 2006).

3.8. Informe de Resultados

Se evaluó *Next Fit* sobre 6 conjuntos (básico, ítems grandes, ítems pequeños, peor caso, mixto, aleatorio) y pruebas de escalabilidad (50-5000 ítems).
Se midió:

- Recipientes usados.
- Cota inferior: $LB = [\sum_{i=1}^{n} S_i]$.
- Factor: $F = \frac{bins\ usados}{LB}$.
- Eficiencia: $E = \frac{\Sigma S_i}{bins\ usados}$.
- Desperdicio: $W = bins\ usados - \sum S_i$.
- Tiempo (ms).

3.8.1. Resultados resumidos

Caso	Ítems	Usados	Cota	Factor	Eficiencia	Desperdicio
Básico	5	2	2	1.00	95.00%	0.10
Ítems grandes	5	5	5	1.00	84.00%	0.80
Ítems pequeños	5	1	1	1.00	80.00%	0.20
Peor caso	5	5	3	1.67	51.00%	2.45
Mixto	8	4	4	1.00	100.00%	0.00
Aleatorio	20	10	8	1.25	78.20%	2.18

Factor observado: entre 1.00 y 1.67 (≤ 2.00). El peor caso valida la cota teórica; los demás muestran alto aprovechamiento. El caso aleatorio mantiene buen rendimiento (1.25).

3.8.2. Escalabilidad

- Promedios ($50 \rightarrow 5000$ ítems) confirman tiempo lineal $O(n)$.
- Factor estable $\approx 1.34; 1.37$.
- Coeficiente de linealidad cercano a 1.00.
- Tiempo para $n = 5000 < 2ms, \approx 0.93ms$ (corrida canónica).

3.8.3. Interpretación

- Alta eficiencia cuando hay complementariedad (básico, mixto).
- Degradación esperable cuando todos los ítems > 0.5.
- El factor práctico (≈ 1.35) está muy por debajo del límite 2-OPT.
- Variación del factor: mínima al crecer $n \rightarrow$ robustez.

La estabilidad del factor (≈ 1.35) muy por debajo de la cota 2 está alineada con resultados clásicos de heurísticas simples (Johnson, 1973; Coffman et al., 1997).

3.8.4. Limitaciones y mejoras

- Limitaciones: orden de llegada fijo, sin reordenamiento, sensible a secuencias adversariales (> 0.5).
- Posibles mejoras: comparar con *First Fit / FFD*, fijar semilla aleatoria, registrar desviación estándar de tiempos y factores, experimentar con otras distribuciones.

3.8.5. Conclusión

Next Fit cumple: simplicidad, tiempo O(n) confirmado, garantía $\leq 2 \cdot OPT$ y buen desempeño empírico (factor ≤ 1.67, típico ≈1.35). Adecuado como solución rápida online; puede reemplazarse por heurísticas con ordenamiento para mayor eficiencia absoluta.

3.9. Algoritmos alternativos y contexto

- *First Fit* (FF): Inserta cada ítem en el primer recipiente donde quepa, sin reordenar. Complejidad típica $O(n^2)$ sin estructuras; puede reproducirse con estructuras de búsqueda. Garantía clásica: $\leq 1.7 \cdot OPT$ (Coffman et al., 1997).

- *Best Fit* (BF): Coloca el ítem en el recipiente con menor espacio libre que aún lo admite. Similar a FF en costo y garantías: $\leq 1.7 \cdot OPT$ (Coffman et al., 1997).

- *First Fit Decreasing* (FFD): Ordena los ítems en forma descendente y aplica FF. Complejidad $O(log(n))$ por el ordenamiento. Garantía más fuerte: $\leq \frac{11}{9} \cdot OPT + 1$ ($\approx 1.222 \cdot OPT + 1$) (Johnson, 1973).

- Relajación lineal del ILP: Modela con variables binarias $x_{i,b}$ (asignación de ítem i al bins b) y y_b (bins usado). La relajación LP permite soluciones fraccionarias y se combina con técnicas de rounding/column generation (relación con *Cutting Stock*). Útil como baseline teórico y para comparar contra heurísticas (Kleinberg & Tardos, 2006).

Estos métodos ofrecen mejores factores o calidad práctica a costa de mayor complejidad u ordenamiento, y sirven como referencia para el contexto del problema.

4. Problema 4

4.1. Análisis

4.1.1. Análisis, Condiciones y Limitaciones

- Se asume que Victor es daltónico y no puede distinguir visualmente las esferas, mientras que Peggy tiene visión normal.
- El protocolo asume un canal seguro donde Victor no puede saber mediante otros medios (tacto, temperatura, marcas) cuál esfera es cuál; son idénticas salvo por el color.
- Se asume honestidad en la ejecución mecánica del protocolo: Victor elige aleatoriamente si intercambiar o no las esferas y Peggy responde honestamente basándose en lo que ve.

4.1.2. Cumplimiento de Propiedades ZKP

- **Completitud:** Si la afirmación es verdadera (esferas de distinto color), Peggy siempre detectará si hubo cambio o no. Convencerá a Victor con probabilidad 1.
- **Solidez:** Si la afirmación fuese falsa (esferas idénticas), Peggy no tendría ninguna pista visual y se vería obligada a adivinar la respuesta. Al ser una elección binaria aleatoria por parte de Victor, Peggy solo tendría un 50% de probabilidad de acertar por suerte en cada ronda, haciendo despreciable su chance de engañarlo tras varias repeticiones.
- **Conocimiento Cero:** Victor conoce de antemano si realizó o no el intercambio. La respuesta correcta de Peggy solo verifica su capacidad de distinción, sin revelar información adicional que permita a Victor identificar cuál esfera es la roja y cuál la verde.

4.1.3. Cálculo de repeticiones

Definimos la certeza deseada como $C \geq 0.90$. Sabiendo que la probabilidad de error tras k rondas es $(1/2)^k$, planteamos:

$$1 - (\frac{1}{2})^k \geq 0.9$$

Despejamos el término de probabilidad de error:

$$(\frac{1}{2})^k \leq 0.1$$

Para despejar k, invertimos la base y aplicamos logaritmo en base 2:

$$2^k \geq 10$$

$$k \geq log_2(10)$$

Resolviendo:

$$k \geq 3.3219$$

Dado que las rondas deben ser un número entero, se redondea hacia el entero superior más próximo. Por lo tanto se necesitan al menos **4 rondas** para garantizar una certeza superior al 90%.

4.2. Diseño

4.2.1. Pseudocódigo

```
FUNCIÓN ProtocoloZKP(cantidad_rondas):
    // 0 = Rojo, 1 = Verde
   esferas = [0, 1]

   PARA i DESDE 1 HASTA cantidad_rondas HACER:
      // — Turno de Victor —
      // Victor lanza una moneda (0 = no cambiar, 1 = cambiar)
      decision_victor = Random(0, 1)

      SI decision_victor == 1:
         esferas_actuales = Swap(esferas)
      SINO:
         esferas_actuales = esferas

      // — Turno de Peggy —
      // Peggy observa las esferas actuales y determina si hubo cambio
      respuesta_peggy = PeggyAnalizar(esferas, esferas_actuales)

      // — Verifico —
      SI respuesta_peggy != decision_victor:
         RETORNAR Falso // Victor descubrió el engaño

   // Si pasa todas las rondas
   RETORNAR Verdadero // Victor está convencido
FIN FUNCIÓN

FUNCIÓN PeggyAnalizar(configuracion_original, configuracion_actual):
   // Si Peggy puede distinguir los colores, compara las configuraciones
   SI configuracion_actual != configuracion_original:
      RETORNAR 1
   SINO:
      RETORNAR 0
FIN FUNCIÓN
```

4.2.2. Estructuras de datos utilizadas

- **Arreglo/Lista:** Se utiliza un arreglo simple de tamaño 2 para representar el par de esferas ya que al ser solo dos objetos cuyo estado es binario (posición original o invertida), un arreglo es la estructura más eficiente en memoria O(1) y acceso directo.
- **Variables primitivas**: La decisión de Victor (intercambiar o no) y la respuesta de Peggy se modelan como valores binarios (0 o 1).

4.3. Seguimiento

Para verificar el funcionamiento, realizamos una prueba simulando **4 rondas** (el mínimo necesario para el 90% de certeza calculado en el análisis).

- **Estado Inicial (Base):** Esfera Roja a la Izquierda (Pos[0]), Esfera Verde a la Derecha (Pos[1]).
- **Peggy:** Conoce el estado inicial y puede ver el estado actual.
- **Víctor:** Decide aleatoriamente (0 = No cambiar, 1 = Cambiar).

Ronda	Decisión de Victor (Random)	Configuración Presentada	¿Qué ve Peggy?	Respuesta de Peggy	Verificación
1	1 (Cambiar)	[Verde, Roja]	Difiere de la Base	"Hubo cambio"	**Correcto**
2	0 (No cambiar)	[Roja, Verde]	Igual a la Base	"No hubo cambio"	**Correcto**
3	1 (Cambiar)	[Verde, Roja]	Difiere de la Base	"Hubo cambio"	**Correcto**
4	1 (Cambiar)	[Verde, Roja]	Difiere de la Base	"Hubo cambio"	**Correcto**

Cómo se finalizó las 4 rondas sin fallos, victor alcanza una certeza > 90% de que peggy no está adivinando y realmente distingue las esferas.

4.4. Complejidad

El algoritmo consiste en un bucle que se repite k veces (donde k es el número de rondas necesarias para la certeza deseada). Dentro de cada iteración, las operaciones son de tiempo constante

O(1):

- Generar un número aleatorio.
- Intercambiar dos elementos en un arreglo (Swap).
- Comparar dos arreglos de tamaño fijo (2 elementos).

Por lo tanto, la complejidad temporal es **O(k)**. Dado que k es una constante configurada según la certeza deseada (en este caso 4) y no depende del tamaño de los objetos, se puede considerar **O(1)** en términos computacionales puros.

4.5. Sets de Datos

Para validar el comportamiento del algoritmo, se realizó un barrido experimental continuo ejecutando el protocolo desde **k=1** hasta **k=20** rondas. A continuación se describen los 4 escenarios representativos seleccionados para el análisis detallado:

1. **Caso Base:**
   - Datos: k = 1 ronda.
   - Características: Ejecución única. La probabilidad de error es del 50%.
   - Propósito: Punto de partida que demuestra la nula fiabilidad de una única iteración.
2. **Caso del Requerimiento**
   - Datos: k = 4 rondas.
   - Características: Punto de inflexión donde la certeza (93.75%) supera por primera vez el 90%.
   - Propósito: Validar empíricamente que con apenas 4 iteraciones se satisface la condición de suficiencia planteada.
3. **Caso de Alta Confianza**
   - Datos: k = 10 rondas.
   - Características: La probabilidad de engaño cae por debajo del 0.1% (aprox 0.0009).
   - Propósito: Demostrar la rápida convergencia del algoritmo hacia una seguridad robusta.
4. **Caso Asintótico**
   - Datos: k = 20 rondas.
   - Características: Probabilidad de error del orden de 10^{-6}.
   - Propósito: Verificar la estabilidad numérica y el comportamiento límite del algoritmo ante secuencias largas.

4.6. Grado de Certeza

La siguiente tabla resume los resultados de los escenarios clave definidos anteriormente:

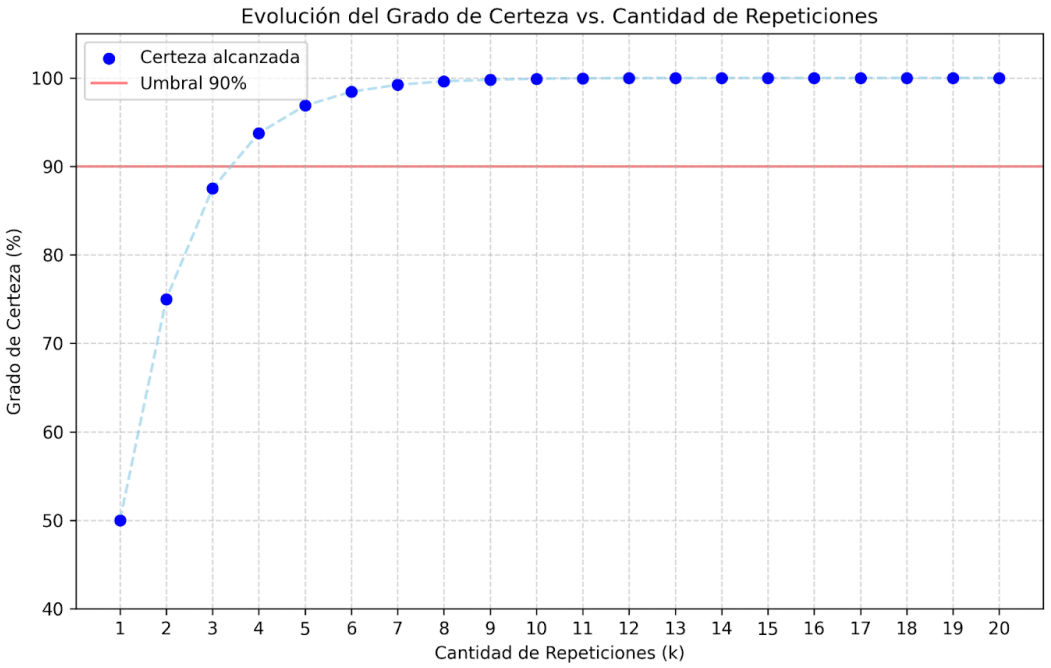
Escenario	Rondas (k)	Probabilidad de Error	Certeza Alcanzada	Resultado
Base	1	0.50000	50.000%	No Confiable
Umbral	4	0.06250	**93.750%**	**Objetivo Cumplido (>90%)**
Alta Confianza	10	0.00098	99.902%	Seguro
Asintótico	20	0.000001	99.999%	Prácticamente Infalible

Análisis: El análisis muestra que la certeza converge asintóticamente al 100%. Dado que el tiempo de ejecución para el caso máximo (k=20) se mantuvo por debajo de 1 ms, se confirma que aumentar la cantidad de rondas para mejorar la seguridad tiene un costo computacional despreciable.

4.7. Informe de Resultados

4.7.1. Análisis de la evolución de la certeza

Se generó el gráfico de dispersión que vincula la cantidad de repeticiones (k) con el porcentaje de certeza alcanzado por el verificador.



-Interpretación de la curva:

- **Fase de Crecimiento Acelerado (k=1 a k=5):** Se observa una pendiente pronunciada inicial. En las primeras iteraciones, cada ronda adicional aporta una ganancia significativa de información. Pasamos de una incertidumbre total (50% en k=1) a una certeza de alta seguridad (96.8% en k=5) en pocos pasos.
- **Cruce del Umbral:** La línea roja horizontal marca el requerimiento del 90%. El gráfico confirma visualmente que este umbral se supera en la 4ta repetición (93.75%), validando el cálculo teórico que se realizó en el Análisis.
- **Comportamiento Asintótico:** A partir de la décima iteración, la curva se aplana visualmente cerca del 100%. Aunque la ganancia porcentual parece marginal (de 99.9% a 99.99%), en términos criptográficos representa una reducción exponencial de la probabilidad de ataque.

Por lo tanto, la relación entre costo computacional (lineal, O(k)) y seguridad (exponencial, 1 - 1/2^k) es favorable. El algoritmo no requiere un número elevado de repeticiones para ser robusto; con k=20, la probabilidad de error es virtualmente nula, haciendo al protocolo viable para sistemas de tiempo real

4.7.2. Aplicaciones Prácticas de ZKP

- **Transacciones Privadas en Blockchain:** En blockchains públicas como Bitcoin, las transacciones son transparentes. Protocolos como Zcash o Monero utilizan ZKP para permitir que un usuario demuestre que tiene fondos suficientes y que la

transacción es válida, sin revelar el monto transferido ni las identidades del emisor y receptor. La red verifica la prueba matemática sin conocer los datos secretos de la transacción.

- **Sistemas de autenticación sin contraseña:** En los sistemas tradicionales, el usuario envía su contraseña al servidor para loguearse, lo cual conlleva riesgos de intercepción. Mediante ZKP (como el protocolo Fiat-Shamir o implementaciones modernas de FIDO), el usuario puede demostrar al servidor que conoce su contraseña sin enviarla nunca a través de la red. El servidor envía un "desafío" aleatorio y el dispositivo del usuario responde probando su identidad sin exponer la credencial real.

5. Anexo

5.1. Referencias

- Rosen, K. H. (2019). *Discrete Mathematics and Its Applications (8th ed.). McGraw-Hill Education.* (Ver Sección 6.5).
- Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson Addison-Wesley.
- Johnson, D. S. (1973). *Near-Optimal Bin Packing Algorithms.* PhD thesis, MIT.
- Coffman, E. G., Garey, M. R., Johnson, D. S. (1997). *Approximation Algorithms for Bin Packing.* In: Approximation Algorithms for NP-Hard Problems, PWS.
- Karp, R. M. (1972). *Reducibility Among Combinatorial Problems.* In: Complexity of Computer Computations.
- Aad, I. (2023). *Zero-Knowledge Proof*. En: Mulder, V., Mermoud, A., Lenders, V., Tellenbach, B. Trends in Data Protection and Encryption Technologies. Springer, Cham.
- Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., & Virza, M. (2014). *Zerocash: Decentralized Anonymous Payments from Bitcoin*. 2014 IEEE Symposium on Security and Privacy.
- Fiat, A., & Shamir, A. (1987). *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*. Advances in Cryptology — CRYPTO '86. Springer, Berlin, Heidelberg.