.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(TB024) Curso Echeverria

Trabajo Práctico 1
Problema 1, 2 y 3

2 de junio de 2025

Grupo Nº 1

Integrantes:

Céspedes, Brian - 108219

Ledesma, Cristian - 111426

Juarez Lezama, Juan Ernesto - 110418

Janampa Salazar, Mario Rafael - 108344

Índice

1. Problema 1 - Programación Dinamíca

1.1. Consigna

Cualquier cadena puede ser descompuesta en secuencias de palíndromos. Por ejemplo, la cadena ARACALACANA se puede descomponer de las siguientes formas:

```
ARA CALAC ANA
ARA C ALA C ANA
A R A CALAC A N A
etc.
```

Desarrollar un algoritmo de programación dinámica que encuentre el menor número de palíndromos que forman una cadena dada. Por ejemplo, para ARACALACANA debería devolver 3.

1.2. Supuestos y Restricciones del algoritmo

El algoritmo del Problema 1 recibe los siguientes supuestos:

- Puede contener espacios (serán ignorados por el algoritmo).
- El algoritmo no es *case-sensitive* (transforma todo al formato de MAYÚSCULAS).
- La cadena no debe estar vacía.

1.3. Diseño y Pseudocodigo

1.3.1. Ecuación de recurrencia

$$\text{opt}(i) = \begin{cases} 1 & \text{si cadena}[0 \dots i] \text{ es un palíndromo} \\ \min\limits_{1 \leq j \leq i} (\text{opt}[i], \text{opt}[j-1] + 1) & \text{si cadena}[j \dots i] \text{ es un palíndromo} \end{cases}$$

Cumplimiento de Subestructura Óptima

- Se cumple el requisito de la subestructura óptima: se puede ver claramente cómo el problema general se va construyendo en base a las soluciones óptimas de los subproblemas anteriores.
- Si se quiere encontrar la solución óptima para toda la cadena, se aprovechan las soluciones óptimas de los prefijos de la misma.
- Por ejemplo, al calcular la solución óptima hasta la posición i, el algoritmo evalúa todos los posibles cortes anteriores j (subcadenas). Si dicha subcadena es un palíndromo, entonces actualiza la solución óptima hasta dicho momento.

Cumplimiento de Subproblemas Superpuestos

- Se cumple el requisito de los subproblemas superpuestos, ya que para construir el óptimo global se reutilizan muchas veces las soluciones a subproblemas ya resueltos previamente.
- Por ejemplo, si queremos saber si una cadena (o subcadena) es un palíndromo, directamente podemos verificarlo en una estructura de datos creada previamente, lo que nos ahorra tener que realizar dicha verificación en todas las iteraciones.

- Por otro lado, la lista de óptimos también se aprovecha para evitar tener que calcular los óptimos previos nuevamente.

Uso de Memoization

- En nuestro algoritmo, la memoization es sumamente importante ya que se usa en dos puntos clave:
  - La matriz `es_palindromo[i][j]` almacena información sobre si las subcadenas (o la cadena entera) forman un palíndromo. Más adelante en el informe, se explicará en más detalle cómo funciona dicha estructura.
  - La lista `opt[i]` también se apoya sobre los principios de la memoization, ya que almacena la información de los óptimos anteriores calculados. La lista se va construyendo incrementalmente y permite que podamos aprovechar el enfoque *bottom-up* característico de la programación dinámica.

1.3.2. Pseudocodigo

Funcion calcular_palindromos(cadena)

```
calcular_palindromos(cadena):

    longitud_cadena = LONGITUD(cadena)
    es_palindromo = MATRIZ de tamanio longitud_cadena
   longitud_cadena con todos los valores en FALSO

    DESDE longitud_subcadena = 1 HASTA longitud_cadena:
        DESDE i = 0 HASTA longitud_cadena - longitud_subcadena:
            j = i + longitud_subcadena - 1
            SI cadena[i] == cadena[j] ENTONCES:
                SI longitud_subcadena <= 2 O es_palindromo[i + 1][j
    - 1] ENTONCES:
                    es_palindromo[i][j] = VERDADERO

    RETORNAR es_palindromo
```

Funcion reconstruir_palindromos(opt, cadena)

```
reconstruir_palindromos(opt, cadena):

    diccionario = DICCIONARIO vac o
    longitud = LONGITUD(opt)

    DESDE i = 0 HASTA longitud - 1:
        SI opt[i] NO EST  EN diccionario ENTONCES:
            diccionario[opt[i]] = [i, i + 1]
        SINO:
            diccionario[opt[i]][1] = i + 1
            SI opt[i] + 1 EST  EN diccionario ENTONCES:
                ELIMINAR diccionario[opt[i] + 1]

    resultado = LISTA vac a

    PARA cada par_de_indices EN VALORES(diccionario):
        AGREGAR SUBCADENA de cadena desde par_de_indices[0] hasta
   par_de_indices[1] A resultado
```

```
18
19      RETORNAR resultado
```

Funcion encontrar_minimos_palindromos(cadena)

```
encontrar_minimos_palindromos(cadena):

    cadena = REMOVER_ESPACIOS(cadena)
    cadena = PASAR_A_MAYUSCULAS(cadena)
    longitud = LONGITUD(cadena)

    es_palindromo = calcular_palindromos(cadena)
    opt = LISTA de tamanio longitud con todos los valores en
   INFINITO

    DESDE i = 0 HASTA longitud - 1:
        SI es_palindromo[0][i] ENTONCES:
            opt[i] = 1
        SINO:
            DESDE j = 1 HASTA i:
                SI es_palindromo[j][i] ENTONCES:
                    opt[i] = MIN(opt[i], opt[j - 1] + 1)

    minima_cantidad = opt[longitud - 1]
    resultado = reconstruir_palindromos(opt, cadena)

    RETORNAR (minima_cantidad, resultado)
```

1.3.3. Implementación

```python
def calcular_palindromos(cadena: str) -> list:

    longitud_cadena: int = len(cadena)
    es_palindromo = [[False] * longitud_cadena for _ in range(longitud_cadena)]

    for longitud_subcadena in range(1, longitud_cadena + 1):
        for i in range(longitud_cadena - longitud_subcadena + 1):
            j = i + longitud_subcadena - 1
            if cadena[i] == cadena[j]:
                if longitud_subcadena <= 2 or es_palindromo[i + 1][j - 1]:
                    es_palindromo[i][j] = True

    return es_palindromo


def reconstruir_palindromos(opt: list, cadena: str) -> list:

    diccionario_reconstruccion: dict = {}
    longitud_cadena: int = len(opt)

    for i in range(longitud_cadena):
        if opt[i] not in diccionario_reconstruccion:
            diccionario_reconstruccion[opt[i]] = [i, i + 1]
        else:
            diccionario_reconstruccion[opt[i]][1] = i + 1
            if opt[i] + 1 in diccionario_reconstruccion:
                del diccionario_reconstruccion[opt[i] + 1]

    resultado: list = []
    for par_de_indices in diccionario_reconstruccion.values():
        resultado.append(cadena[par_de_indices[0]:par_de_indices[1]])

```

```
33      return resultado
34
35
36  def encontrar_minimos_palindromos(cadena: str) -> tuple:
37
38      cadena = cadena.replace(" ", "")
39      cadena = cadena.upper()
40      longitud_cadena: int = len(cadena)
41      es_palindromo: list = calcular_palindromos(cadena)
42      opt: list = [inf] * longitud_cadena
43
44      for i in range(longitud_cadena):
45          if es_palindromo[0][i]:
46              opt[i] = 1
47          else:
48              for j in range(1, i + 1):
49                  if es_palindromo[j][i]:
50                      opt[i] = min(opt[i], opt[j - 1] + 1)
51
52      minima_cantidad_palindromos: int = opt[-1]
53      resultado: list = reconstruir_palindromos(opt, cadena)
54
55      return minima_cantidad_palindromos, resultado
```

Estructuras de datos utilizadas:

- `es_palindromo[i][j]`
  Es una matriz que indica si la subcadena que va desde el índice `i` hasta el índice `j` (inclusive) es un palíndromo. Cada celda almacena un valor booleano. La diagonal principal siempre está en `True`, ya que representa subcadenas de un solo carácter, que por definición son palíndromos.
- `opt[i]`
  Es una lista donde cada posición `i` representa el mínimo número de palíndromos necesarios para cubrir la subcadena desde el inicio hasta el índice `i`. Esta estructura permite aplicar programación dinámica para construir la solución óptima paso a paso.
- `diccionario_reconstruccion`
  Es un diccionario auxiliar usado para reconstruir los palíndromos detectados. Las claves representan la cantidad de cortes (niveles de partición), y los valores son listas con dos elementos: el índice de inicio y el índice de fin + 1 de cada palíndromo, lo que permite luego extraer las subcadenas con `slicing`.

1.4. Seguimiento del Algoritmo

Antes de comenzar a hacer un seguimiento, debemos tener una consideración especial con ciertos casos de cadenas que se pueden recibir.

Para ejemplificar, supongamos que tenemos la cadena `"PEPEPOPSTAR"`.

Primera posible descomposición:

- `"PEP" 'E' "POP" 'S' 'T' 'A' 'R'`
- Pero también podría ser: `'P' ."EP"E" "POP" 'S' 'T' 'A' 'R'`
- Y a su vez, también podría ser: `"PEPEP" 'O' 'P' 'S' 'T' 'A' 'R'` ← Esta es la reconstrucción que devuelve nuestro algoritmo.

Notemos que si bien podemos descomponer los palíndromos de distinta forma, la mínima cantidad de ellos siempre es la misma. En este caso, el mínimo es de siete.

Haremos un seguimiento simplificado para la cadena del ejemplo del enunciado: `'ARACALACANA'`.

a) Inicialmente, la cadena se introduce en la función `calcular_palíndromos()`. Recordemos que esta función devuelve una matriz.

Como un único carácter siempre es palíndromo, en la diagonal de la matriz tendremos siempre el valor `True` asignado.

Para subcadenas de longitud 2, si hubiese dos caracteres consecutivos iguales, también tendríamos 2 `True` consecutivos en la matriz. En el caso de esta cadena no ocurre.

Para subcadenas de longitud 3 o más, se utiliza un enfoque "desde el interior hacia el exterior", para ejemplificar esto podemos tomar de ejemplo la subcadena `CALAC`.

Primero se analiza la subcadena `'L'` la cual es un palíndromo ya que es únicamente una letra.

La subcadena `ALA` efectivamente se comprueba que es un palíndromo ya que las letras de los extremos coinciden y lo que está en medio (`'L'`) también es un palíndromo.

Por último, la subcadena `CALAC` se comprueba que también es un palíndromo siguiendo la lógica de manera análoga con el caso anterior.

A continuación, se esquematiza la matriz `es_palindromo` resultante de la cadena del ejemplo, también, se resaltan algunos puntos de interés dentro de la misma:

```
True  False True  False False False False False False False False
False True  False False False False False False False False False
False False True  False True  False False False True  False False
False False False True  False False False True  False False False
False False False False True  False True  False False False False
False False False False False True  False False False False False
False False False False False False True  False True  False False
False False False False False False False True  False False False
False False False False False False False False True  False True
False False False False False False False False False True  False
False False False False False False False False False False True
```

b) Luego de creada la matriz `es_palindromo`, podemos pasar al seguimiento en la función `encontrar_minimos_palindromos()`, la cual crea la lista de óptimos de la misma longitud que la cadena y con todos los valores en infinito.

Después de esto, se itera desde $i = 0$ hasta la longitud de la cadena:

- Cuando $i = 0$, se cumple el primer bloque `if`, ya que `es_palindromo[0][0]` es verdadero, por lo tanto $\text{opt}[0] = 1$ (esto ocurrirá siempre, ya que un solo carácter es un palíndromo, y la partición mínima en consecuencia es de 1).
- Cuando $i = 1$, ya no se cumple `es_palindromo[0][1]`, por lo tanto, se ingresa en el segundo ciclo que itera desde $j = 1$ hasta $i + 1$ (en este caso, hasta 2).
  - Cuando $j = 1$, podemos notar que `es_palindromo[1][1]` = `True`, por lo tanto $\text{opt}[1] = \text{mín}(\text{opt}[1], \text{opt}[0]+1)$, reemplazando con los valores específicos: $\text{opt}[1] = \text{mín}(\infty, 2) = 2$.
  - Cuando $j = 2$, `es_palindromo[1][2]` = `False`.
- Cuando $i = 2$, se vuelve a cumplir la condición del primer bloque `if`, ya que `es_palindromo[0][2]` = `True`, por lo tanto $\text{opt}[2] = 1$ nuevamente.
- Cuando $i = 3$, `es_palindromo[0][3]` = `False`, entonces accedemos al segundo ciclo que, en este caso iterará desde $j = 1$ hasta $j = 4$.
  - Cuando $j = 1$, `es_palindromo[1][3]` = `False`.

- Cuando $j = 2$, `es_palindromo[2][3]` = `False`.
- Cuando $j = 3$, `es_palindromo[3][3]` = `True`, entonces calculamos el óptimo como: opt[3] = mín(opt[3], opt[2] + 1), o sea, opt[3] = mín(∞, 2) = 2.

- Cuando $i = 4$, `es_palindromo[0][4]` = `False`, entonces accedemos al segundo ciclo que, en este caso iterará desde $j = 1$ hasta $j = 5$.
  - Cuando $j = 1$, `es_palindromo[1][4]` = `False`.
  - Cuando $j = 2$, `es_palindromo[2][4]` = `True`, entonces calculamos el óptimo como: opt[4] = mín(opt[4], opt[3] + 1), o sea, opt[4] = mín(∞, 3) = 3.
  - Cuando $j = 3$, `es_palindromo[3][4]` = `False`.
  - Cuando $j = 4$, `es_palindromo[4][4]` = `True`, entonces volvemos a calcular el óptimo como: opt[4] = mín(opt[4], opt[3] + 1), o sea opt[4] = mín(3, 3) = 3.

Podemos continuar iterando hasta $i = 10$, el resultado en este ejemplo es la siguiente lista, también, se resaltan algunos puntos de interés dentro de la misma:

$$\text{opt} = [\,1,\ 2,\ 1,\ 2,\ 3,\ 4,\ 3,\ 2,\ 3,\ 4,\ 3\,]$$

Es interesante notar que la lista nos da un panorama de cómo se fue creando la solución óptima progresivamente en cada subcadena. Por ejemplo, si tuviésemos "ARACALACAN" en vez de la cadena propuesta, ya no íbamos a obtener un mínimo de tres palíndromos, sino que el mínimo hubiese sido cuatro ("ARA", "CALAC", "A", "N").

c) Para finalizar el seguimiento del algoritmo, el último paso es el de reconstrucción de los palíndromos detectados. Para realizar esto, inicialmente se posee un diccionario y una lista, ambos vacíos.

Se itera desde 0 hasta la longitud de la cadena (misma longitud que la lista de óptimos).

- Cuando $i = 0$ se verifica que opt[0] = 1 no está como clave en el diccionario, por lo tanto se la agrega y como valores tendrá una lista cuyos valores iniciales serán $[0, 1]$. Resultado: $\{1 : [0, 1]\}$.
- Cuando $i = 1$ se verifica que opt[1] = 2 no está como clave en el diccionario, por lo tanto se la agrega y como valores tendrá una lista cuyos valores iniciales serán $[1, 2]$. Resultado: $\{1 : [0, 1], 2 : [1, 2]\}$.
- Cuando $i = 2$ se verifica que opt[2] = 1 sí está como clave en el diccionario, por lo tanto se actualiza el valor de dicha clave, quedando de la siguiente forma: $1 : [0, 3]$, pero también se debe eliminar la clave '2' del diccionario. Resultado: $\{1 : [0, 3]\}$.
- Cuando $i = 3$ se verifica que opt[3] = 2 no está como clave en el diccionario (lo hemos borrado en el paso inmediatamente anterior). Por lo tanto se agrega la nueva clave y se le asignan los valores iniciales $[3, 4]$. Resultado: $\{1 : [0, 3], 2 : [3, 4]\}$.

Finalmente, luego de todas las iteraciones, el diccionario resultante es el siguiente:

$$\{1 : [0, 3], 2 : [3, 8], 3 : [8, 11]\}.$$

Este diccionario nos da información muy valiosa para reconstruir las cadenas de palíndromos encontrados, en una lista adicional llamada `palindromos` se almacenan los valores de `cadena[0:3]`, `cadena[3:8]` y `cadena[8:11]`. Observemos cómo todas las subcadenas se pueden encontrar aprovechando el slicing de Python con el diccionario creado.

Para "ARACALACANA" se obtuvieron tres palíndromos: "ARA", "CALAC", "ANA".

1.5. Análisis de Complejidad

Analizaremos la complejidad de las tres funciones previamente vistas por separado:

1.5.1. Función `calcular_palindromos()`

La función `calcular_palindromos()` inicialmente calcula la longitud de una cadena, lo cual se puede realizar en tiempo constante $\mathcal{O}(1)$. Posteriormente, se debe crear una matriz de tamaño `longitud_cadena` $\times$ `longitud_cadena`, lo que implica una complejidad de $\mathcal{O}(n^2)$, siendo n la longitud de la cadena.

Luego tenemos un ciclo que itera sobre la longitud total de la cadena desde 1 hasta n; dentro de este ciclo existe además otro ciclo que itera por todas las longitudes posibles de subcadenas desde el punto en el que estemos actualmente. Esto se puede traducir en una complejidad $\mathcal{O}\left(\sum_{i=1}^{n} i\right)$, que es equivalente a:

$$\mathcal{O}\left(\frac{n(n+1)}{2}\right) = \mathcal{O}(n^2)$$

El resto de operaciones presentes en cada ciclo se pueden realizar en tiempo constante $\mathcal{O}(1)$, ya que implican solamente acceder a índices en una lista, comparaciones o asignaciones.

Complejidad resultante: $\mathcal{O}(n^2) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$

1.5.2. Función `reconstruir_palindromos()`

La función `reconstruir_palindromos()` inicialmente también calcula la longitud de una cadena, lo cual se puede realizar en tiempo constante $\mathcal{O}(1)$. Luego se itera desde 0 hasta la longitud de la cadena. Dentro de este ciclo, todas las operaciones se pueden realizar en tiempo constante $\mathcal{O}(1)$ ya que solo implican bloques condicionales y manipulación de diccionarios. La complejidad temporal entonces es de $\mathcal{O}(n)$.

Luego, se itera sobre todos los valores del diccionario antes creado, para almacenar la información en una lista. En cada iteración se realizan operaciones de slicing sobre subcadenas, las cuales tienen una complejidad de $\mathcal{O}(k)$, siendo k la longitud de la subcadena.

Sin embargo, esta complejidad es inversamente proporcional al tamaño del diccionario: si tuviésemos un slicing del estilo `cadena[0:n]` tendría una complejidad de $\mathcal{O}(n)$ pero el diccionario tendría solo una clave, por lo que se requeriría una única iteración. Por el contrario, si tuviésemos n claves en el diccionario, significa que todas las subcadenas son de un solo carácter de longitud. Recorreremos el diccionario en un tiempo de $\mathcal{O}(n)$, y cada slicing se puede realizar en $\mathcal{O}(1)$. Por lo tanto, la complejidad temporal nuevamente es de $\mathcal{O}(n)$.

Complejidad resultante: $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$

1.5.3. Función `encontrar_minimos_palindromos()`

La función `encontrar_minimos_palindromos()` primero reemplaza todos los espacios en la cadena, y luego convierte la misma a mayúsculas. Ambas operaciones se realizan en $\mathcal{O}(n)$. Calcular la longitud de la cadena se hace en tiempo constante $\mathcal{O}(1)$.

Luego se genera la matriz `es_palindromo`, llamando a la función `calcular_palindromos()`, que ya analizamos previamente y tiene una complejidad temporal de $\mathcal{O}(n^2)$. Por último, se genera la lista de óptimos de la misma longitud que la cadena, cuya confección se realiza también en $\mathcal{O}(n)$.

Después, el primer ciclo itera desde 0 hasta la longitud de la cadena. Dentro del mismo existe otro ciclo que itera desde 1 hasta $i + 1$. Es un caso parecido al ciclo que analizamos en la función `calcular_palindromos()`, el cual tenía una complejidad:

$$\mathcal{O}\left(\frac{n(n+1)}{2}\right) = \mathcal{O}(n^2)$$

El resto de operaciones en ambos ciclos se pueden realizar en tiempo constante $\mathcal{O}(1)$.

Por último, se accede a la última posición en una lista, que nuevamente es $\mathcal{O}(1)$. Luego se llama a la función `reconstruir_palindromos()` que ya analizamos que tiene una complejidad de $\mathcal{O}(n)$.

Complejidad resultante `encontrar_minimos_palindromos()`:

$$\boxed{\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)}$$

Complejidad resultante (temporal):

$$\boxed{\mathcal{O}(n^2)}$$

La complejidad espacial resultante está dada por la matriz `es_palindromo`, la cual tiene un tamaño de longitud_cadena × longitud_cadena. Esta es la estructura de datos más "robusta" que posee el algoritmo. El resto de estructuras es lineal y como mucho pueden llegar a tener una complejidad espacial de $\mathcal{O}(n)$.

Complejidad resultante (espacial):

$$\boxed{\mathcal{O}(n^2)}$$

1.6. Set de Datos

En el archivo `test_aleatorio.txt` se encuentran las pruebas realizadas sobre los sets de datos generados aleatoriamente con el programa `ejercicio1_test_aleatorio.py`.

Para medir los tiempos de ejecución, se optó por utilizar otro programa: `ejercicio1_test_tiempo_de_ejecucion.py`. Los datos obtenidos se pueden visualizar en el archivo `test_tiempos_de_ejecucion_entrega.txt`. Sobre este archivo se realizó el gráfico que se verá en la siguiente sección. Si se corre nuevamente el script, se escribirá un archivo `test_tiempos_de_ejecucion.txt`, lo cual puede ser útil para comparar los resultados obtenidos en otra máquina con los de la entrega.

Como tercera opción, se encuentra el programa `ejercicio1_test_manual.py`, el cual permite al usuario introducir una cadena de caracteres a demanda y luego probar el algoritmo.

1.7. Graficos

1.7.1. Tiempos de Ejecucion

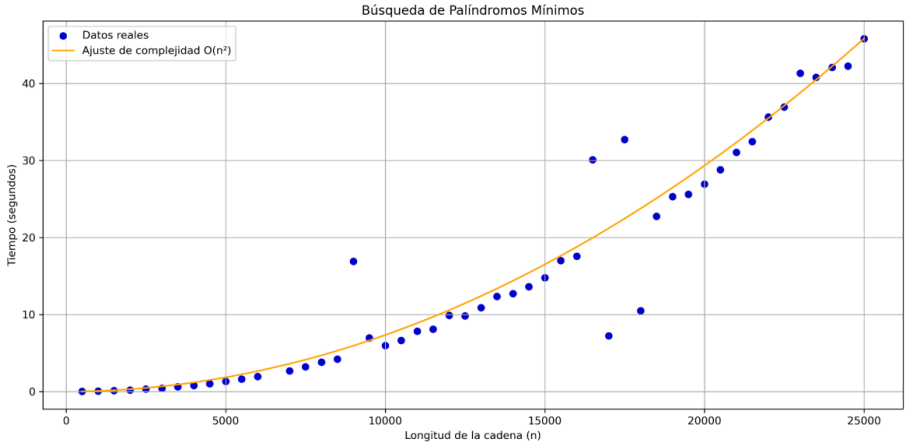


Figura 1: Gráfico que relaciona la longitud n de una cadena con el tiempo de ejecución

Las mediciones fueron realizadas en una computadora con las siguientes características:

- **Procesador:** [Intel i5 1235U, 1.3GHz].
- **Memoria RAM:** [16GB de RAM]
- **Sistema Operativo:** [WINDOWS 11]
- **Intérprete de Python:** [Python 3.12.3]

1.8. Conclusión

Podemos concluir que el algoritmo propuesto se corresponde con la metodología de programación dinámica. Se ha indicado que se cumplen con los requisitos de subestructura óptima y además con los de los problemas superpuestos. Se ha mostrado en qué se beneficia el algoritmo de utilizar memoization, reduciendo drásticamente la complejidad temporal. Además, es notorio el enfoque bottom-up para resolver el problema. La complejidad temporal del algoritmo se condice con el gráfico expuesto, ya que se observa la tendencia cuadrática del mismo.

2. Problema 2 - Programacion Lineal

2.1. Consigna

Concesiones Argentina 2000 SRL tiene la concesión de los espacios publicitarios en las paradas de colectivos de un municipio. Son en total 200 paradas y tiene ofertas de distintos productos para el próximo mes:

- Cliente A ofrece USD 50000 por ocupar 30 paradas
- Cliente B ofrece USD 100000 por ocupar 80 paradas o USD 120000 por 120 paradas (sólo una de ambas opciones)
- Cliente C ofrece USD 100000 por ocupar 75 paradas
- Cliente D ofrece USD 80000 por ocupar 50 paradas
- Cliente E ofrece USD 5000 por ocupar 2 paradas
- Cliente F ofrece USD 40000 por ocupar 20 paradas
- Cliente G ofrece USD 90000 por ocupar 100 paradas

Por ser competidores directos, no se puede hacer publicidad simultáneamente de los clientes A y D. Se desea determinar a qué clientes se concesionará la publicidad de las paradas para maximizar el beneficio total.

2.2. Supuestos y Restricciones del algoritmo

El algoritmo del Problema 2 presenta los siguiente supuestos:

- El algoritmo planteado recibe un archivo con el nombre çlientes.csv", con el formato: cliente, paradas, propuesta. Donde cliente representa el nombre del cliente, paradas la cantidad de paradas que desea ocupar, y propuesta la oferta económica por dicha ocupación.
- Las restricciones entre propuestas se consideran únicamente de a pares, se conocen de antemano y se encuentran en forma de constante en el código (e.g Clientes A y D son competidores, y por ende no pueden tomarse ofertas de ambos a la vez).
- Cada propuesta es indivisible: se acepta por completo o se descarta.
- Se busca maximizar las ganancias sin superar el máximo de 200 paradas disponibles.

2.3. Variables

En este modelo de programación lineal, se utilizan variables de decisión binarias para representar la aceptación o rechazo de propuestas de clientes.

- $Y_i = 1$ Las variables son de decisión binaria, y cada una representa si se acepta o no la propuesta de un cliente (1 para afirmativo, 0 para negativo).
- Las variables consideradas son: A, B1, B2, C, D, E, F, G

Estas variables serán utilizadas en las restricciones y en la función objetivo, de modo que reflejen las decisiones óptimas bajo los criterios planteados.

2.4. Modelacion de Programacion lineal

- **Función objetivo:** La función objetivo busca maximizar las ganancias totales a partir de aceptar propuestas de clientes con una cantidad finita de paradas.

  Max $Z = 50000 \cdot Y_A + 100000 \cdot Y_{B1} + 120000 \cdot Y_{B2} + 100000 \cdot Y_C + 80000 \cdot Y_D + 5000 \cdot Y_E + 40000 \cdot Y_F + 90000 \cdot Y_G$

- **Restricciones:**
  - Las paradas concesionadas no pueden superar la cantidad máxima de paradas disponibles:

    $$30 \cdot Y_A + 100 \cdot Y_{B1} + 120 \cdot Y_{B2} + 75 \cdot Y_C + 50 \cdot Y_D + 2 \cdot Y_E + 20 \cdot Y_F + 100 \cdot Y_G$$

  - Clientes A y D no se pueden publicitar simultáneamente:

    $$Y_A + Y_D \leq 1$$

  - Solo una propuesta por cliente:

    $$Y_{B1} + Y_{B2} \leq 1$$

2.5. Solución Obtenida

Variables iguales a 1

- Cliente A: 30 paradas, USD 50000
- Cliente B1: 80 paradas, USD 100000
- Cliente C: 75 paradas, USD 100000
- Cliente E: 2 paradas, USD 5000

Solución

- Z = 255000.0
- Paradas usadas: 187

2.6. Interpretación

- Se aceptan las propuestas de los clientes: A, B1, C, E.
- No se acepta la de D (por competencia directa con A), ni la de B2 (ya que se aceptó la propuesta B1).
- Se maximizan los beneficios a USD 255.000 dentro del límite de 200 paradas.
- Se respetan todas las restricciones del problema (capacidad máxima e incompatibilidad entre propuestas).
- Todo lo planteado garantiza que la solución es factible y se encuentra dentro del dominio del problema, es decir, dentro del conjunto de combinaciones válidas de decisiones que cumplen con las restricciones detalladas.

Además, se justifica el uso de la programación lineal para este tipo de problemas de asignación y optimización, dado que, según Mitchell et al. (2011), herramientas como PuLP permiten modelar y resolver de manera eficiente problemas lineales con restricciones múltiples, facilitando la obtención de soluciones óptimas en contextos como el presente.

Referencias

- Mitchell, S., et al. (s.f.). *PuLP: A linear programming toolkit for Python*. COIN-OR. Recuperado el 23 de mayo de 2025, de `https://coin-or.github.io/pulp/index.html`

3. Problema 3 - Redes de Flujo

3.1. Consigna

Estamos construyendo una red WAN con n antenas y queremos que tenga un buen nivel de tolerancia a fallas. Dada una antena, su conjunto de *backup* de tamaño k es el conjunto de k antenas que se encuentran a una distancia menor a D. Queremos evitar que una antena pertenezca al conjunto de *backup* de más de b antenas, precisamente para evitar que un fallo pueda afectar a una porción importante de la red.

Suponer que conocemos los valores D, b y k, y que tenemos una matriz $d[1..n, 1..n]$ con las distancias entre antenas, de forma tal que $d[i, j]$ es la distancia entre la antena i y la j.

Plantear un algoritmo de complejidad polinomial que encuentre el conjunto de *backup* de tamaño k de cada una de las n antenas, de forma tal que ninguna aparezca en más de b conjuntos de *backup*, o bien, que indique que no existe una solución posible. Debe apoyarse en el algoritmo estándar de Ford-Fulkerson, la variante escalada o la de Edmonds-Karp.

3.2. Supuestos y Restricciones del algoritmo

El algoritmo desarrollado para el Problema 3 parte de los siguientes supuestos y restricciones:

- La matriz de distancias es simétrica y no permite conexiones de una antena consigo misma.
- Se utiliza una única lista de antenas que se divide artificialmente en dos grupos: las que necesitan respaldo (A) y las que pueden brindarlo (B).
- Solo se permiten conexiones $A_i \to B_j$ si la distancia entre ambas antenas es estrictamente menor a un umbral $D_{\text{mínima}}$.
- Cada antena del grupo A puede pedir a lo sumo k respaldos (capacidad de salida), y cada antena del grupo B puede ofrecer a lo sumo b respaldos (capacidad de entrada).
- La red de flujo construida cumple con las condiciones necesarias para aplicar el algoritmo de Edmonds-Karp:
  - No se aceptan bucles (aristas de un vértice a sí mismo).
  - No se permiten ciclos de dos vértices (aristas antiparalelas entre A_i y B_i).
  - Existe una única fuente (FUENTE) y un único sumidero (SUMIDERO).
  - Todas las capacidades son enteras y positivas.
- El problema se modela como un flujo máximo, y se considera solución factible si el flujo total alcanza el valor esperado: $n \times k$, donde n es la cantidad de antenas que requieren respaldo (conjunto A), y k es la cantidad máxima de respaldos que cada antena del conjunto A puede solicitar.

3.3. Diseño y Pseudocodigo

3.3.1. Adaptación de los datos a una red de flujo

Para resolver el problema, se construye una red de flujo dirigido que representa las restricciones de respaldo entre antenas. Se parte de una única lista de antenas que se divide conceptualmente en dos subconjuntos:

- Conjunto A: antenas que requieren respaldo.
- Conjunto B: antenas que pueden ofrecer respaldo.

Se agregan dos nodos especiales:

- **FUENTE**, que conectará a cada antena del conjunto A.
- **SUMIDERO**, que recibirá el flujo proveniente de las antenas del conjunto B.

A continuación se describe la forma en que se crean las aristas y sus capacidades:

- Para cada antena A_i en el conjunto A, se añade una arista FUENTE $\rightarrow A_i$ con capacidad k. Aquí, k representa la cantidad máxima de respaldos que puede solicitar cada antena que requiere backup.
- Para cada antena B_j en el conjunto B, se añade una arista $B_j \rightarrow$ SUMIDERO con capacidad b. Aquí, b representa la cantidad máxima de respaldos que puede ofrecer cada antena que actúa como backup.
- Entre un nodo A_i y un nodo B_j se inserta una arista $A_i \rightarrow B_j$ con capacidad 1 si y solo si:
  1. $i \neq j$ (no se permite que una antena se respalde a sí misma).
  2. La distancia entre la antena i y la antena j es estrictamente menor que el umbral $D_{\text{mínima}}$.

  De este modo, cualquier camino de flujo desde FUENTE hasta SUMIDERO tiene la forma

$$\text{FUENTE} \rightarrow A_i \rightarrow B_j \rightarrow \text{SUMIDERO}$$

y transporta una unidad de respaldo desde la antena A_i hacia la antena B_j. Si en conjunto se logra enviar el flujo deseado, significa que se ha encontrado una asignación válida de respaldos para todas las antenas que lo requieren.

3.3.2. Interpretación de la salida de Ford-Fulkerson (vía Edmonds-Karp)

Una vez construida la red, se aplica el algoritmo de Ford-Fulkerson (implementado típicamente vía Edmonds-Karp) para calcular el flujo máximo desde la fuente hasta el sumidero. La interpretación es la siguiente:

- Si el flujo máximo obtenido es igual a $n \times k$, donde n es la cantidad de antenas en el conjunto A y k es el número de respaldos que cada antena puede solicitar, entonces existe una asignación factible que satisface todas las demandas de respaldo.
- Si el flujo máximo es menor que $n \times k$, significa que no se pueden ofrecer respaldos a todas las antenas de A según las restricciones de distancia y capacidad, y por lo tanto no existe solución factible bajo los parámetros dados.

3.3.3. Propiedades del grafo construido

Al construir la red, surgen ciertas propiedades estructurales que permiten verificar de forma inmediata si es posible encontrar una solución factible. En particular, se destacan las siguientes:

Flujo total requerido Para satisfacer completamente las demandas de respaldo, se necesita un flujo total de:

$$\text{Flujo requerido} = n \times k$$

donde n es el número de antenas y k la cantidad de respaldos que solicita cada una.

Capacidad total disponible El conjunto de nodos B (las mismas antenas vistas como posibles oferentes) puede proporcionar como máximo:

$$\text{Capacidad disponible} \;=\; n \times b$$

donde b representa la cantidad máxima de respaldos que cada antena puede ofrecer.

Condición necesaria de factibilidad Para que exista siquiera la posibilidad de una solución completa, debe cumplirse:

$$n \times k \;\leq\; n \times b \qquad \Longrightarrow \qquad k \leq b.$$

Si esta desigualdad no se cumple, la red nunca podrá transportar el flujo necesario, incluso si las conexiones fueran ideales.

Suficiencia estructural (conectividad) Además de la condición anterior, cada antena debe estar conectada a al menos k antenas distintas dentro del conjunto B, cumpliendo las restricciones de distancia y sin incluirse a sí misma. Esta conectividad garantiza que el flujo pueda ser efectivamente distribuido.

Conclusion de lo Analizado

- La desigualdad $k \leq b$ es condición necesaria para alcanzar un flujo total de $n \times k$.
- La factibilidad también depende de la existencia de al menos k conexiones válidas desde cada nodo A_i hacia nodos B_j.
- Si ambas condiciones se cumplen, el flujo máximo alcanzará $n \times k$, confirmando que existe una solución factible para el valor de $D_{\text{mínima}}$ considerado.

3.3.4. Pseudocodigo

A continuación se presentan las funciones más relevantes del desarrollo, expresadas en pseudocódigo.

```
Funcion crear_grafo_antenas(matriz_distancias, D_maxima, k, b)
    g := nuevo Grafo dirigido
    g.agregar_vertice("SUMIDERO")
    g.agregar_vertice("FUENTE")

    Para i desde 1 hasta longitud(matriz_distancias) + 1
        Para cada grupo en {"A", "B"}
            nodo := grupo + i
            g.agregar_vertice(nodo)
            Si grupo = "A" entonces
                g.agregar_arista("FUENTE", nodo, capacidad := k)
            Sino
                g.agregar_arista(nodo, "SUMIDERO", capacidad := b)

    Para i desde 0 hasta longitud(matriz_distancias)
        Para j desde 0 hasta longitud(matriz_distancias[0]
            Si i DISTINTO j y matriz_distancias[i][j] MENOR A
   D_maxima entonces
                origen := "A" + (i+1)
                destino := "B" + (j+1)
                g.agregar_arista(origen, destino, capacidad := 1)

```

```
    Retornar g
Fin Funcion
```

Listing 1: Construcción del grafo a partir de los datos del problema

A continuación se presenta el pseudocódigo correspondiente a la implementación del algoritmo de Edmonds-Karp, utilizado para calcular el flujo máximo en una red.

```
Funcion flujo(grafo, fuente, sumidero)
    flujo := diccionario vacio

    Para cada vertice en grafo
        Para cada adyacente en grafo.adyacentes(vertice)
            flujo[(vertice, adyacente)] := 0

    grafo_residual := inicializar_Grafo_Residual(grafo)

    Mientras obtener_camino(grafo_residual, fuente, sumidero) NO ES None hacer
        camino := obtener_camino(grafo_residual, fuente, sumidero)
        capacidad := minimo_peso_del_camino(grafo_residual, camino)

        Para i desde 1 hasta longitud(camino) - 1
            Si estan_unidos(grafo, camino[i-1], camino[i]) entonces
                flujo[(camino[i-1], camino[i])] :=
                    flujo[(camino[i-1], camino[i])] + capacidad
            Sino
                flujo[(camino[i], camino[i-1])] :=
                    flujo[(camino[i], camino[i-1])] - capacidad

            actualizar_grafo_residual(grafo_residual, camino[i-1], camino[i], capacidad)

    Retornar flujo
Fin Funcion
```

Listing 2: Implementación del algoritmo de Edmonds-Karp para encontrar el flujo máximo

La función `obtener_camino` es un componente fundamental del algoritmo de Edmonds-Karp, ya que implementa una búsqueda (BFS) sobre el grafo residual para encontrar un camino aumentante desde la fuente hasta el sumidero. Esta estrategia garantiza que se utilicen los caminos más cortos posibles en términos de cantidad de aristas, lo cual es clave para asegurar la cota temporal polinomial del algoritmo.

A continuación, se detalla su implementación en pseudocódigo.

```
Funcion obtener_camino(grafo, fuente, sumidero)
    visitados := conjunto vacio
    padre := diccionario vacio

    cola := nueva cola
    cola.encolar(fuente)
    visitados.agregar(fuente)

    Mientras cola NO ES vacia hacer
        actual := cola.desencolar()

        Si actual = sumidero entonces
            romper

        Para cada vecino en grafo.adyacentes(actual)
            Si vecino NO EN visitados Y peso_arista(grafo, actual,
   vecino) > 0 entonces
                visitados.agregar(vecino)
                padre[vecino] := actual
                cola.encolar(vecino)

    Si sumidero NO EN padre entonces
        Retornar None

    Retornar reconstruir_camino(padre, fuente, sumidero)
Fin Funcion
```

Listing 3: Búsqueda e(BFS) para encontrar un camino aumentante en el grafo residual

Estructuras de datos utilizadas:

- **Diccionario (diccionario clave-valor):** Utilizado para representar el *flujo* de cada arista en el grafo y también para registrar los *padres* de cada nodo durante la búsqueda BFS. Permite accesos rápidos y asignaciones eficientes.
- **Conjunto (set):** Utilizado para almacenar los nodos *visitados* durante la búsqueda BFS. Facilita operaciones de pertenencia rápidas, lo cual es esencial para evitar ciclos o exploraciones redundantes.
- **Cola (FIFO):** Empleada en la función `obtener_camino` para implementar la búsqueda (BFS). Garantiza que los nodos se procesen en el orden en que se descubren.
- **Lista o arreglo indexado:** Utilizado para construir y recorrer los *caminos aumentantes*, tanto en la función de flujo como en la reconstrucción de caminos.
- **Grafo dirigido (con aristas con peso y capacidad):** Es la estructura principal sobre la que se modela el problema. Se utiliza tanto para representar el grafo original como el grafo residual.

3.3.5. Implementación

```python
def crear_grafo_antenas(matriz_distancias, distancia_maxima,
                        grupo_k_backups, grupo_b_antenas):

    g = grafo.Grafo(es_dirigido=True)
    g.agregar_vertice(SUMIDERO)
    g.agregar_vertice(FUENTE)
    for i in range(1, len(matriz_distancias) + 1):
        for grupo in GRUPOS:
            grupo_antena = grupo + str(i)
            g.agregar_vertice(grupo_antena)
            if grupo == "A":
                g.agregar_arista(FUENTE, grupo_antena, grupo_k_backups)
            else:
                g.agregar_arista(grupo_antena, SUMIDERO, grupo_b_antenas)

    for i in range(len(matriz_distancias)):
        for j in range(len(matriz_distancias[0])):
            if i != j and matriz_distancias[i][j] < distancia_maxima:
                g.agregar_arista(GRUPOS[0] + str(i+1), GRUPOS[1] + str(j+1), 1)

    return g
```

```python
def flujo(grafo, fuente, sumidero):
    flujo = {}
    for vertice in grafo:
        for adyacente in grafo.adyacentes(vertice):
            flujo[(vertice, adyacente)] = 0

    grafo_residual = utilidades.inicializar_Grafo_Residual(grafo)

    while (camino := utilidades.obtener_camino(grafo_residual, fuente, sumidero))
    is not None:
        capacidad_residual_camino = utilidades.minimo_peso_del_camino(
    grafo_residual, camino)
        for i in range(1, len(camino)):
            if grafo.estan_unidos(camino[i-1], camino[i]):
                flujo[(camino[i-1], camino[i])] += capacidad_residual_camino
            else:
                flujo[(camino[i], camino[i-1])] -= capacidad_residual_camino
            utilidades.actualizar_grafo_residual(grafo_residual, camino[i-1],
    camino[i], capacidad_residual_camino)

    return flujo
```

```python
def obtener_camino(grafo, fuente, sumidero):
    visitados = set()
    padre = {}

    cola = deque()
    cola.append(fuente)
    visitados.add(fuente)

    while cola:
        actual = cola.popleft()
        if actual == sumidero:
            break
        for vecino in grafo.adyacentes(actual):
            if vecino not in visitados and grafo.peso_arista(actual, vecino) > 0:
                visitados.add(vecino)
                padre[vecino] = actual
                cola.append(vecino)

    if sumidero not in padre:
        return None

    return reconstruir_camino(padre, fuente, sumidero)
```

3.4. Seguimiento del Algoritmo

Para hacer un seguimiento del algoritmo, vamos a tener en cuenta el archivo `antenas_pruebas.txt`. Como podemos observar, es un archivo con tres parámetros: la distancia mínima (D), la cantidad máxima de backups que puede tener cada antena (k) y la cantidad máxima de veces que una antena puede ser usada como backup de otra (b). También contiene una matriz con la distancia entre cada antena:

```
D = 4
k = 2
b = 2
Antena1|0|2|2|4
Antena2|2|0|3|2
Antena3|2|3|0|1
Antena4|4|2|1|0
```

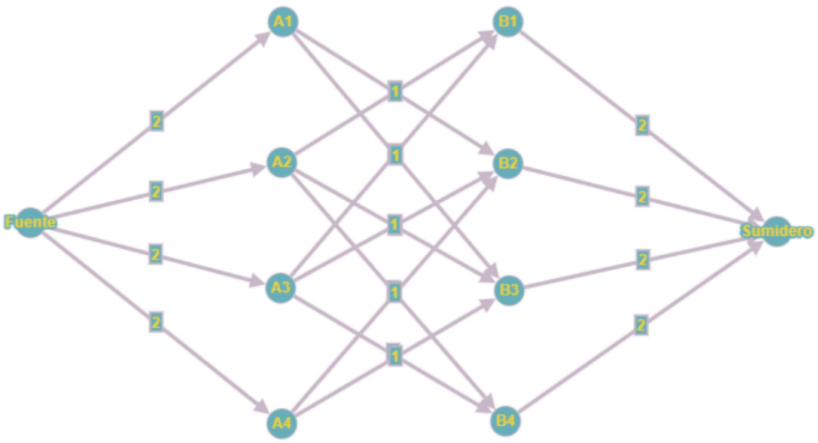
Primero vamos a tener la idea de un grafo bipartido, siendo el grupo A las antenas que apuntan a las antenas del grupo B que tienen distancia menor a D (solo apuntan si son menores a D). Tanto el grupo A como el grupo B están compuestos por las mismas antenas, es decir, $A1 = B1$, $A2 = B2$, ..., $An = Bn$. Se duplicaron las antenas para poder encontrar el flujo. El grupo A representa las antenas que necesitan backups, mientras que el grupo B representa las antenas que reciben backups.

Con esta idea, podemos tratar esto como un tipo de *Matching Bipartito Múltiple*. Cada antena del grupo A va a apuntar a antenas del grupo B con peso 1. Esto es así porque una vez que el flujo pasa por una arista $A1 \rightarrow B3$, significa que $A1$ se ha emparejado con $B3$, y no volverá a pasar flujo por esa arista (aunque $A1$ puede seguir emparejándose con otras antenas).

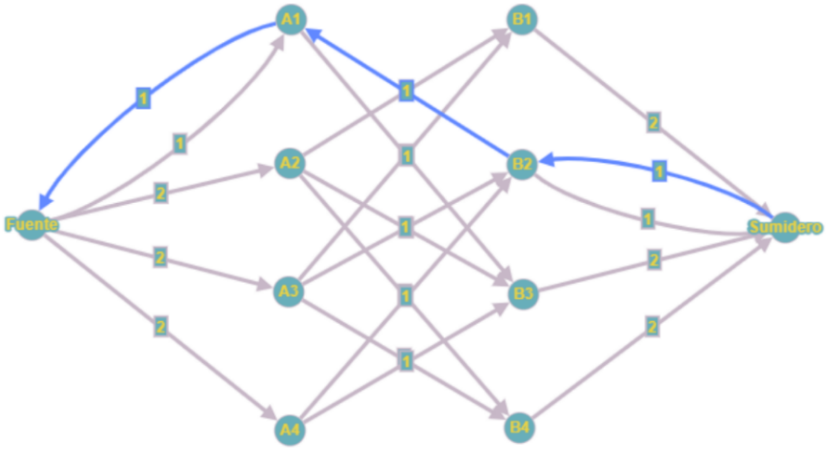
Ahora, para construir el grafo necesario para que el flujo funcione correctamente, desde la fuente se envía un flujo de valor k a cada antena del grupo A. Si pensamos que la fuente es una especie de fábrica de antenas, esto representa que desde la central se establece que cada antena puede tener como máximo k backups. Por eso, desde la fuente se apunta a cada nodo del grupo A con una arista de capacidad k.

Finalmente, cada antena del grupo B se conecta al sumidero con una arista de capacidad b, ya que queremos evitar que una antena sea utilizada como backup por más de b antenas diferentes.

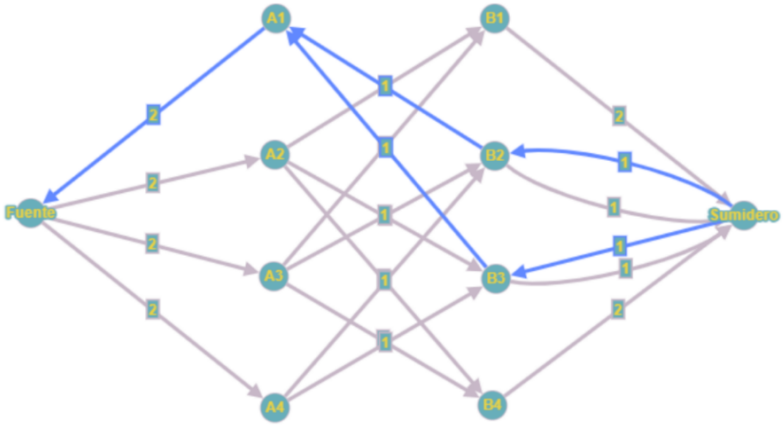
1) Grafo inicial para el flujo, también es el inicial para el grafo residual:



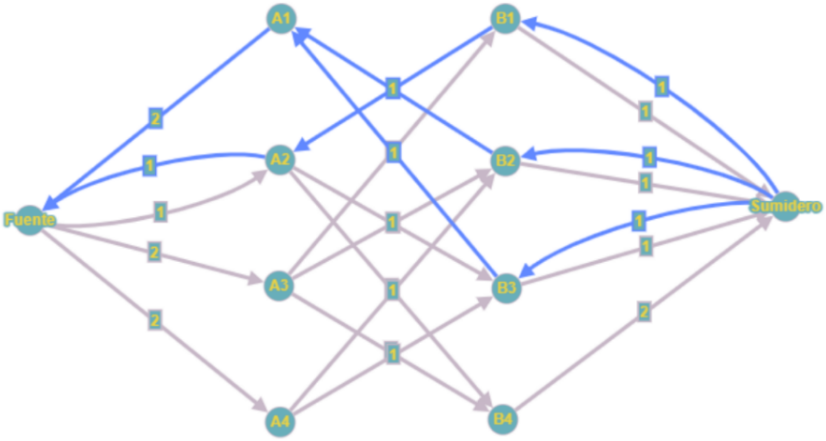
2) Vamos a buscar el camino más corto desde la Fuente hasta el Sumidero, como primera búsqueda tenemos: Fuente → A1 → B2 → Sumidero.



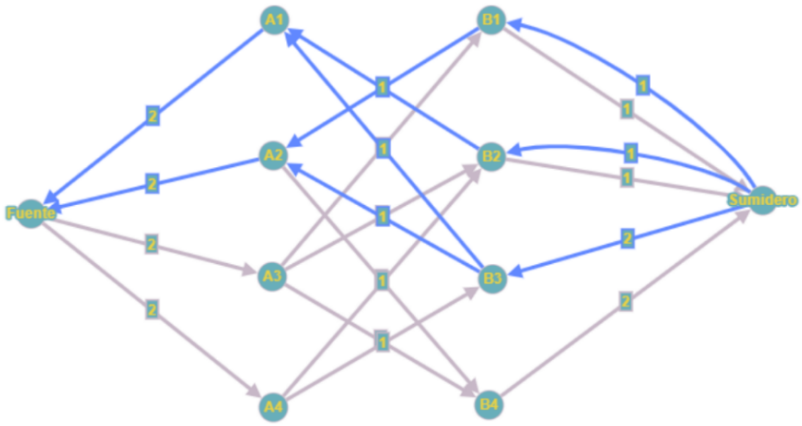
3) Siguiente camino: Fuente → A1 → B3 → Sumidero.



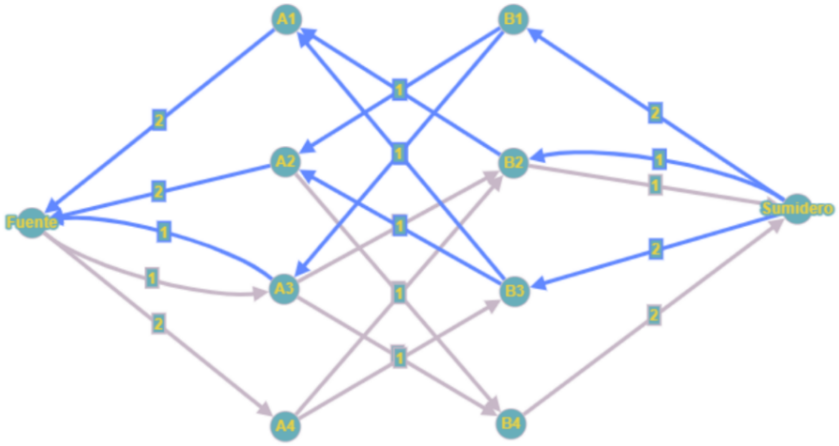
4) Siguiente camino: Fuente → A2 → B1 → Sumidero



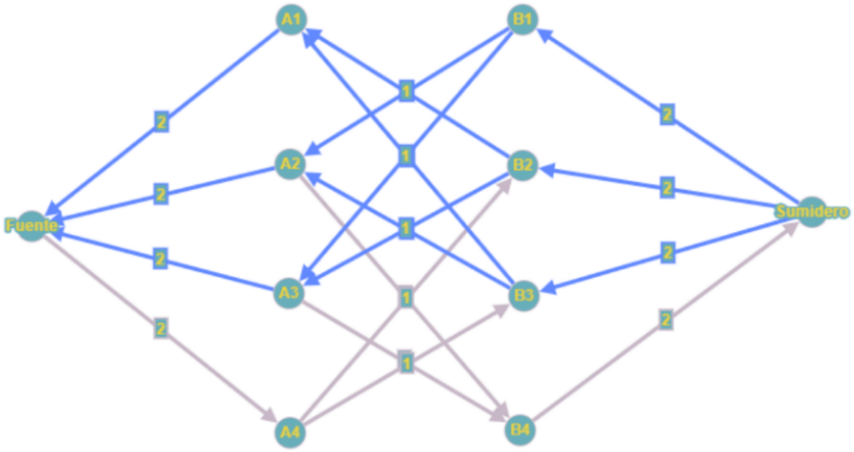
5) Siguiente camino: Fuente → A2 → B3 → Sumidero.



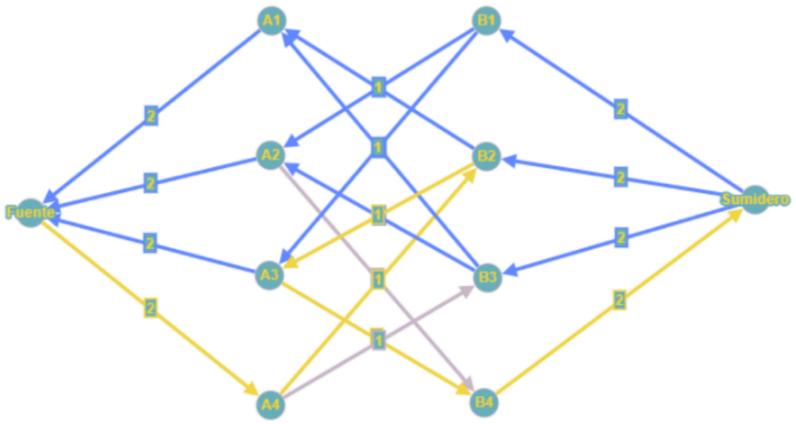
6) Siguiente camino: Fuente → A3 → B1 → Sumidero.



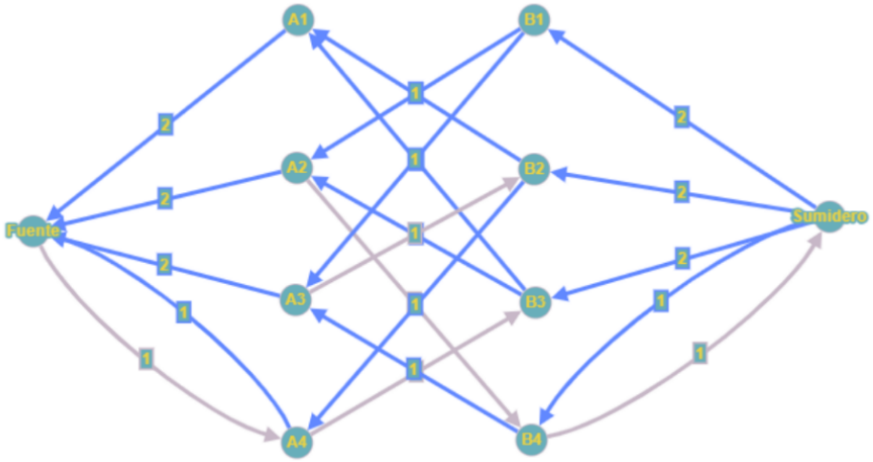
7) Siguiente camino: Fuente → A3 → B2 → Sumidero.



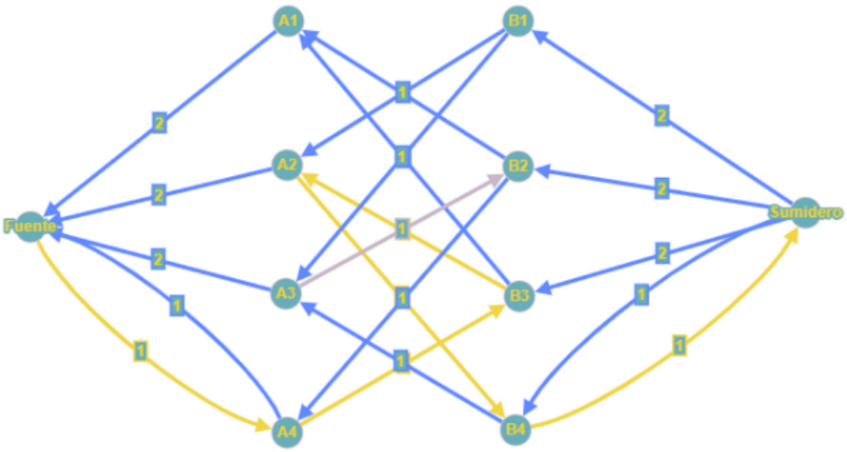
8) Siguiente camino: Fuente → A4 → B2 → A3 → B4 → Sumidero.



Dado que estamos pasando por un flujo que ya habíamos pasado anteriormente, significa que esa conexión no se va a usar. Por eso, en la red residual vemos que `B2` → `A3`, pero como ya explicamos lo anterior, queda `A3` → `B2`.

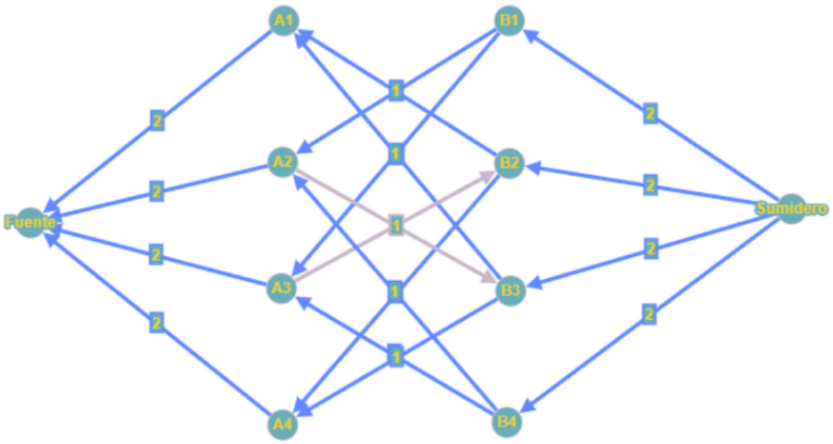


9) Siguiente camino: Fuente → A4 → B3 → A2 → B4 → Sumidero.



Como ya mencionamos en el punto anterior, se aplica la misma lógica.

10) Nos queda la red residual de esta manera:



Lo que significa que, el flujo máximo es 8, que es lo mismo que $n \times k$ (cantidad de antenas $\times$ cantidad de backups). Si se cumple esa condición, existe solución. Entonces el resultado es:

- Antena1 va a tener como backups a Antena2 y Antena3.
- Antena2 va a tener como backups a Antena1 y Antena4.
- Antena3 va a tener como backups a Antena1 y Antena4.
- Antena4 va a tener como backups a Antena2 y Antena3.

3.5. Análisis de Complejidad

En esta sección analizaremos la complejidad temporal del algoritmo propuesto para asignar backups entre antenas utilizando un enfoque basado en flujo máximo con el algoritmo de Edmonds-Karp.

1. Construcción del grafo:

Para cada una de las n antenas, se crean dos nodos: uno en el grupo A (los que necesitan backups) y otro en el grupo B (los que pueden ser backups). Adicionalmente, se agregan los nodos `FUENTE` y `SUMIDERO`, junto con las siguientes aristas:

- Desde `FUENTE` hacia cada nodo del grupo A, con capacidad k.
- Desde cada nodo del grupo B hacia `SUMIDERO`, con capacidad b.
- Desde cada nodo A_i hacia cada nodo B_j si $d[i, j] < D$, con capacidad 1.

Este procedimiento implica:

- $\mathcal{O}(n)$ nodos en cada grupo A y B.
- $\mathcal{O}(n)$ aristas desde `FUENTE` y hacia `SUMIDERO`.
- $\mathcal{O}(n^2)$ comparaciones para verificar qué conexiones cumplen la condición $d[i, j] < D$.

Por lo tanto, la construcción del grafo tiene complejidad $\mathcal{O}(n^2)$ en el peor caso.

2. Ejecución del algoritmo de Edmonds-Karp:

El algoritmo de Edmonds-Karp es una variante de Ford-Fulkerson que utiliza BFS para encontrar caminos aumentantes. Tiene una complejidad temporal de $\mathcal{O}(V \cdot E^2)$, donde V es la cantidad de vértices y E la cantidad de aristas del grafo residual.

- En este caso, $V = 2n + 2$ (las n antenas duplicadas en A y B, más `FUENTE` y `SUMIDERO`).
- En el peor caso, el número de aristas E es $\mathcal{O}(n^2)$ (todas las posibles conexiones entre A y B).

Funciones auxiliares: Dado que el grafo está implementado como un diccionario de diccionarios, las funciones auxiliares utilizadas en el algoritmo de flujo tienen complejidades de:

- `inicializar_Grafo_Residual`: Esta función realiza una copia profunda del grafo, que recorre todos los vértices y aristas. Dado que hay $\mathcal{O}(V)$ vértices y $\mathcal{O}(E)$ aristas, su complejidad es $\mathcal{O}(V + E)$.
- `obtener_camino`: Implementa una búsqueda (BFS) sobre el grafo residual. Como cada vértice tiene una lista de adyacentes (y el acceso a cada lista es en tiempo constante), el recorrido total tiene complejidad $\mathcal{O}(V + E)$.
- `minimo_peso_del_camino`: Recorre una lista de vértices (el camino aumentante) y accede al peso de cada arista. Como el camino puede tener hasta $\mathcal{O}(V)$ vértices, esta función tiene complejidad $\mathcal{O}(V)$.
- `actualizar_grafo_residual`: En cada actualización de aristas, se accede, modifica o borra una clave en el diccionario de adyacencia. Estas operaciones son de tiempo constante. Como se actualiza una arista por cada paso del camino (a lo sumo $\mathcal{O}(V)$), esta función tiene complejidad $\mathcal{O}(V)$ por iteración.

Estas funciones se ejecutan en cada iteración del ciclo principal del algoritmo de flujo. Sin embargo, todas ellas tienen complejidad lineal respecto a V o E, y no modifican la cota asintótica dominante del algoritmo de Edmonds-Karp, que se mantiene en:

$$\mathcal{O}(V \cdot E^2) = \mathcal{O}(n \cdot (n^2)^2) = \mathcal{O}(n^5)$$

3. Complejidad total:

Sumando ambas etapas, obtenemos una complejidad global de:

$$\mathcal{O}(n^2) + \mathcal{O}(n^5) = \mathcal{O}(n^5)$$

Conclusión: El algoritmo propuesto para asignar backups entre antenas mediante el enfoque de flujo máximo con Edmonds-Karp es de complejidad polinomial $\mathcal{O}(n^5)$, lo que garantiza su viabilidad para tamaños moderados de n. Las funciones auxiliares, aunque tienen costos lineales, no afectan.

3.6. Conjunto de Datos

Para realizar las pruebas experimentales del algoritmo, hemos generaron múltiples archivos de entrada simulando distintos escenarios con antenas. Estos archivos contienen la información necesaria para contruir el grafo de acuerdo a las especificaciones del problema (posiciones de las antenas, parámetros k, b y D, etc.).

Los archivos generados fueron:

- `10Antenas.txt`
- `25Antenas.txt`
- `75Antenas.txt`
- `150Antenas.txt`
- `200Antenas.txt`
- `250Antenas.txt`
- `300Antenas.txt`
- `375Antenas.txt`
- `450Antenas.txt`

Todos estos archivos fueron generados utilizando el archivo `generar_antenas.py`, el cual se encuentra en la carpeta `Herramientas/` del proyecto. Este archivo permite crear configuraciones personalizadas de antenas a partir de parámetros como la cantidad deseada y los valores de los parámetros k, b y D.

3.7. Graficos

A continuación se presenta el gráfico correspondiente a estos datos, junto con una curva teórica de referencia de complejidad $\mathcal{O}(n^5)$, para verificar la validez del análisis desarrollado anteriormente.

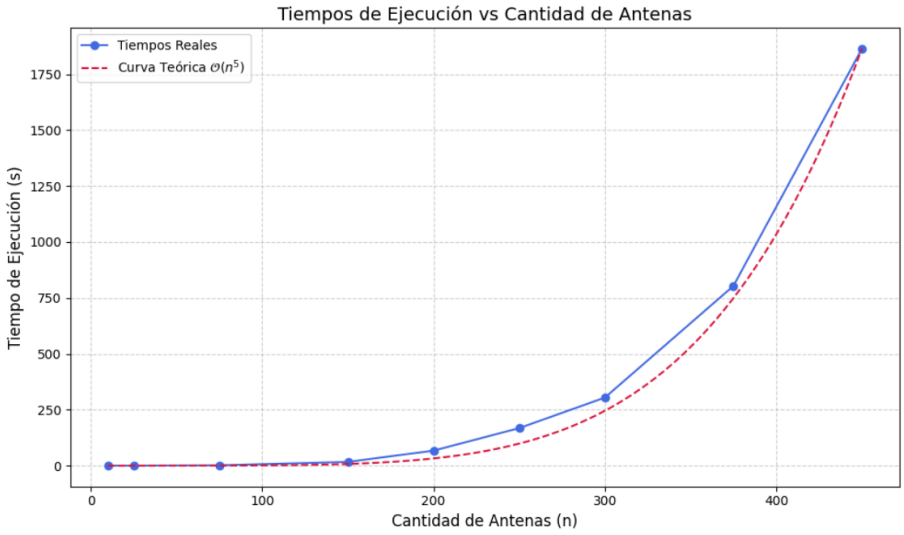


Figura 2: Comparación entre tiempos de ejecución reales y la curva teórica $\mathcal{O}(n^5)$

Como puede observarse en la Figura 2, los tiempos de ejecución obtenidos crecen de forma coherente con la complejidad teórica $\mathcal{O}(n^5)$. Esto valida empíricamente el análisis realizado para el algoritmo de flujo máximo aplicado al problema de asignación de backups entre antenas.

3.7.1. Tiempos de Ejecucion

A continuación, se expone la tabla con los datos obtenidos:

N = Cantidad De Antenas	Tiempo de Ejecución (s)
N = 10	0.0019
N = 25	0.0210
N = 75	1.5375
N = 150	16.7230
N = 200	67.1016
N = 250	167.8138
N = 300	304.3229
N = 375	803.3625
N = 450	1863.8405

Las mediciones fueron realizadas en una computadora con las siguientes características:

- **Procesador:** [Intel Core i5- 1135G7 2.40 GHz]
- **Memoria RAM:** [8GB de RAM]
- **Sistema Operativo:** [WINDOWS 11]
- **Intérprete de Python:** [Python 3.12.9]

Conclusión

En este trabajo se ha implementado y analizado un algoritmo basado en el método de flujo máximo, utilizando la variante de Edmonds-Karp, para resolver el problema de asignación de backups entre antenas. Se demostró que la construcción del grafo y la aplicación del algoritmo cumplen con la lógica necesaria para modelar correctamente las restricciones del problema.

Asimismo, el análisis teórico de complejidad $\mathcal{O}(n^5)$ fue corroborado empíricamente mediante la comparación de los tiempos de ejecución experimentales con la curva teórica, mostrando una concordancia clara y consistente.