Universidad de Buenos Aires
Facultad de Ingeniería

Teoría de Algoritmos 75.29/95.06/TB024
Trabajo Práctico 2 - Curso Echevarría
1er Cuatrimestre 2025

Grupo 15
Integrantes: Bautista Corti, Tomas Amundarain, Andres Moyano, Leandro Brizuela

Índice

PROBLEMA 1

Supuestos

Para este problema se tomaron los siguientes supuestos:

- La cadena que se ingresa no contiene espacios o caracteres especiales.
- Cada cadena representa una única palabra.

Diseño

Algunas aclaraciones en cuanto a las estructuras utilizadas en la solución:

- *es_palindromo:* es una matriz nxn que representa con true si desde la posición i hasta la j se forma un palíndromo o no.
- *opt:* es una lista que va a contener las soluciones óptimas que se van armando a medida que se avanza en la ejecución.

ECUACIÓN DE RECURRENCIA:

$$opt[i] = min_{1 \leq j < i}\{opt[i-1], 1 + opt[j]\} \; con \; j \; tal \; que \; se \; forme \; un \; palíndromo \; entre \; j \; e \; i$$

Pseudocódigo

```
funcion reconstruir_solucion(cadena, es_palindromo, opt):
    n = cadena.len()
    solucion = []
    partes = opt[-1]
    j = n
    mientras partes > 0:
        para i en un rango(j - 1, -1, -1):
            si es_palindromo[i][j - 1] y opt[i] == partes - 1:
                solucion.agregar(cadena[i:j+1])
                partes -= 1
                j = i
                break
    return solucion.reverse()
```

```
funcion palindromos(cadena):
    n = cadena.len()
    es_palindromo = [[False] * n para _ en rango(n)]

    para i en rango(n):
        para j en rango(i + 1):
            si cadena[i:j+1] == cadena[j:i-1]:
                es_palindromo[i][j] = True

    opt = [0] * (n + 1)

    para i en rango(1, n + 1):
        min_cortes = i
        para j en rango(i - 1):
            si es_palindromo[j][i - 1]:
                min_cortes = min(min_cortes, opt[j] + 1)
        opt[i] = min(min_cortes, opt[i-1] + 1)

    return reconstruir_solucion(cadena, es_palindromo, opt)
```

Seguimiento

palabra = “aracalacana”

Luego del llamado a la función *palindromos:*

Paso 1:

Comienza con el armado de la matriz *es_palindromo.* Prueba para cada letra de la cadena ingresada si puede formar palíndromos que fueron evaluadas anteriormente.
En este caso la matriz queda de esta forma:

A	R	A	C	A	L	A	C	A	N	A
T	F	T	F	F	F	F	F	F	F	F
F	T	F	F	F	F	F	F	F	F	F
F	F	T	F	T	F	F	F	T	F	F
F	F	F	T	F	F	F	T	F	F	F
F	F	F	F	T	F	T	F	F	F	F
F	F	F	F	F	T	F	F	F	F	F
F	F	F	F	F	F	T	F	T	F	F
F	F	F	F	F	F	F	T	F	F	F
F	F	F	F	F	F	F	F	T	F	T
F	F	F	F	F	F	F	F	F	T	F
F	F	F	F	F	F	F	F	F	F	T

Paso 2: Cálculo de optimalidad

Se calcula la cantidad mínima en la que se puede dividir desde el principio hasta la letra i, en este caso el opt es:

$[0\ a\ r\ a\ c\ a\ l\ a\ c\ a\ n\ a]$

$[0\ 1\ 2\ 1\ 2\ 3\ 4\ 3\ 2\ 3\ 4\ 3]$

Ej: posición 8 letra “a”

- Inicializa la *min_corte* en 9.
- Itera el for hasta encontrar un posible palíndromo.
    - Encuentra el primero en la posición 6 (letra “a”) se actualiza el *min_corte* en 5.
    - Encuentra el segundo palindromo en la posición 2 (letra ”a”) se actualiza el *min_corte* en 3.

- Compara el entre *min_corte* y el *opt[i - 1] + 1* que representa el armar un palíndromo nuevo y en este caso el opt para la posición 8 es 3.

Paso 3: Armado de la solución

Se arman los palíndromos que forman la solución óptima:

- Se itera hasta la próxima letra que forme un palíndromo desde esa posición hasta el último corte.
    - Empieza por la letra “a” de la última posición hasta la letra “a” de la posición 8 para formar “ana”.
    - Continua con “c” hasta llegar a la próxima “c” que se encuentra en la posición 3 y forma “carac”.
    - Por último, termina formando el último palíndromo con las letras que faltan y se forma “ara”.

- Se devuelve la solución [“ara”, “carac”, “ana”].

Complejidad

Se analizará la complejidad de cada algoritmo diseñado y se determinará la mínima cota posible.

En cuanto a la función *construir_solucion* se puede observar que en el peor de los casos se va a iterar n veces en el ciclo, ya que toma en cuenta el largo de la lista opt que está relacionada con el largo de la cadena. Por lo que esa función tiene una complejidad de O(n)

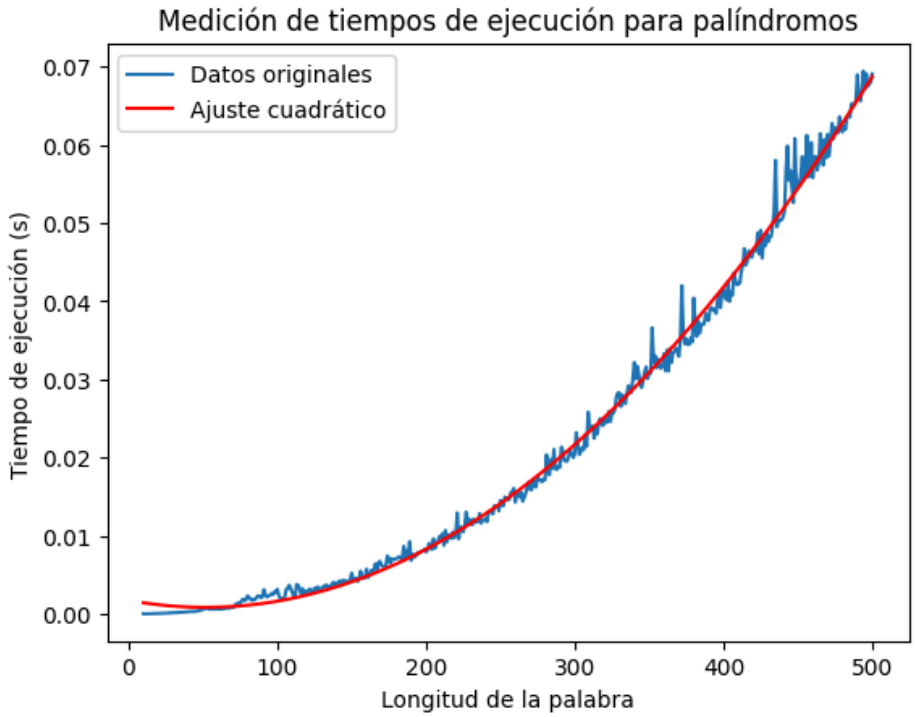
Para la función *palíndromos* se puede observar que hay dos secciones en donde hay un ciclo for anidados. En el primero, el que se encarga de armar la matriz *es_palindromo,* en el peor de los casos por cada letra se podría iterar n veces, por lo que su complejidad *es O(n^2). El* segundo ciclo, el que se encarga de armar las soluciones óptimas, al igual que el ciclo anterior por cada iteración en el peor de los casos el segundo ciclo anidado se iteran n veces más por lo que también su complejidad es $O(n^2)$. Por lo que, en el peor de los casos la complejidad es $O(n^2)$.

Informe

Para analizar la complejidad real se realizó la medición de tiempos para palabras aleatorias de largo variable entre 10 y 500 letras para cada largo se ejecutó 3 veces el algoritmo para una misma palabra, así sacar el promedio de ejecución para un largo determinado y no tener datos donde el tiempo fue afectado por causas ajenas al algoritmo.

Algunos resultados que se obtuvieron para observar cómo aumentan los tiempo:

LARGO	TIEMPO [S]
10	0.000051
11	0.000038
12	0.000036
13	0.000041
14	0.000045
28	0.000067
56	0.000129
112	0.000325



Como se puede observar en el gráfico la complejidad real del algoritmo cumple con la complejidad teórica analizada por lo que efectivamente la complejidad algorítmica es $O(n^2)$.

PROBLEMA 2

Supuestos

Para la resolución del problema se tomaron los siguientes supuestos:

- Cada cliente ocupa exactamente la cantidad de paradas que solicita en su oferta, o no se le asigna ninguna.
- El Cliente B tiene dos ofertas excluyentes: si se elige una, no puede elegirse la otra.
- Los clientes A y D son excluyentes: no se pueden aceptar las ofertas de estos clientes al mismo tiempo.
- Hay un máximo de 200 paradas disponibles.
- No se pueden asignar fracciones de paradas.

Variables

$C_A, C_{B1}, C_{B2}, C_C, C_D, C_E, C_F, C_G \in \{1, 0\}$ → Variables Binarias

- $C_i = 1$: Si se acepta la oferta del cliente i.
- $C_i = 0$: Si no se acepta.

B_i: Beneficio que ofrece la oferta del cliente i (dato conocido y constante).

P_i: Cantidad de paradas que ocuparía la oferta del cliente i (dato conocido y constante).

Modelo de Programación Lineal Entera

Como buscamos maximizar el beneficio total, la Función Objetivo a maximizar sería:

$$Z(máx) = \sum_{i=A}^{G} C_i B_i \quad \forall\, i: A, B1, B2, C, D, E, F, G$$

Además, se deberá cumplir con las siguientes restricciones:

1) Disponibilidad de paradas:

$$200 \geq \sum_{i=A}^{G} C_i P_i$$

- Solo tenemos 200 paradas disponibles.

2) Incompatibilidad entre las ofertas del cliente B:

$$1 \geq C_{B1} + C_{B2}$$

- Solo se puede aceptar una sola de las oferta del cliente B

3) Incompatibilidad entre las ofertas del cliente A y D:

$$1 \geq C_A + C_D$$

- Solo se puede aceptar la oferta del cliente A o la del cliente D, pero no ambas a la vez.

Informe

Tras resolver el modelo utilizando la librería PuLP de Python, se obtuvo una solución óptima con un beneficio total de USD 255.000. Los clientes seleccionados en esta solución son:

- Cliente A: se aceptó su oferta de USD 50.000 por 30 paradas.
- Cliente B (primera oferta): se aceptó su propuesta de USD 100.000 por 80 paradas.
- Cliente C: se aceptó su oferta de USD 100.000 por 75 paradas.
- Cliente E: se aceptó su oferta de USD 5.000 por 2 paradas.

Los restantes clientes (D, F y G) no fueron seleccionados, ya sea porque:

No maximizan el beneficio en relación al uso de paradas,

O bien por conflictos con otros clientes ya elegidos (como A y D).

El uso total de paradas en esta solución fue:
30 (A) + 80 (B1) + 75 (C) + 2 (E) = 187 paradas ($255000.0), cumpliendo la restricción de las 200 disponibles.

La selección final prioriza las ofertas que dan más beneficio en relación a la cantidad de paradas que ocupan. Por ejemplo, se descartó a Cliente G (que ofrecía mucha plata) porque sus 100 paradas complicaban el uso del espacio total disponible.

Este tipo de modelo ayuda a tomar decisiones cuando hay muchas opciones y restricciones, como en este caso. Además, es fácil de ajustar si cambian las ofertas, se suman nuevos clientes o se modifican las condiciones del problema.

PROBLEMA 3

Supuestos

Para este problema se tomaron los siguientes supuestos:

- La matriz de distancias que recibe el algoritmo es simétrica; la distancia de la antena i a la j es la misma que la de antena j a i.
- Las distancias de diagonal principal de la matriz son igual a 0.
- Se asume que los valores de distancia son confiables y representan adecuadamente si dos antenas pueden conectarse o no.
- Al momento de crear el grafo para el flujo: si una antena `i` no tiene al menos k conexiones posibles dentro del umbral `D`, el problema se declara **inválido** inmediatamente. Se asume que un grafo que pasó la instancia de creación tiene la densidad suficiente para que todos tengan al menos k vecinos validos.
- El nodo `AntenaDestinoj` tiene una capacidad b hacia el sumidero, limitando cuántas veces puede ser usada como backup. Si se intenta intentan usar una antena i como backup más de b veces, se genera saturación, lo cual se refleja en el grafo como **flujo osioso**.
- Cada conexión `inicio_i -> destino_j` tiene capacidad 1, lo que significa que **cada arista representa una única relación de respaldo**. Esto impone la lógica: una conexión = una relación `Antena i → backup Antena j`.
- Si el flujo máximo encontrado es menor que `n * k`, significa que **no fue posible asignar los k backups a cada antena respetando `D` y b**.
- Si el grafo es muy disperso (pocas antenas cercanas entre sí), es muy probable que muchas pruebas resulten no factibles.

Diseño

- Pseudocódigo:

función antenas_mediante_flujo(n_antenas, matriz_distancias, k, b, D):

 # creación del grafo:
for i in range(n_antenas):
se crea un nodo antena_inicioi y otro antena_destinoi
Al origen se lo une a cada inicio i con peso k
Por cada destino se lo une al sumidero con peso b
if distancia entre el inicio i y la antena destino j es < D:
unir inicio hacia destino con peso 1
if antena inicio i tiene menos de k conexiones a destinos:
retornar un diccionario vacío

Usar el grafo en una función que calcule el flujo máximo y un grafo residual

if flujo máximo < n_antenas*k:
retornar un diccionario vacío

for cada flujo entre antena_inicioi a antena_destinoj
if flujo == 1:
antenaj esta en el backup de antenai
retornar diccionario resultante

Con esta función resuelve el problema de saturación de antenas conectadas en un radio D, retornando los backups de todas las antenas respetando k, b y D o retornando un diccionario vacío si no se puede cumplir las condiciones en alguna de las instancias que el algoritmo puede detectar que el problema es irresoluble.

Diseño del grafo:

La existencia de antena_inicioi y antena_destinoj refleja la simetria de la matriz distancias. La decisión va más allá de este motivo: la antena_inicioi representa a la antena i y las antenas destino a las que está conectado representa las posibles antenas que tendrá en su backup.

El origen estará conectado a cada antena_inicio con un peso k. Esta decisión representa la restricción de un backup de k elementos, antena_inicioi puede estar conectada a más antenas que las necesarias (no menos que k por una razón que se dará en el apartado **Diseño de corte**) así que que el flujo que llega desde el origen limita las opciones que tendrá Edmonds-Karp al resolver el flujo y permite, si es posible, encontrar el backup con las limitantes de k y b.

Cada antena_destino estará conectada al sumidero con peso b, representando que cada antena no debe exceder b apariciones. Esto limita el flujo que llegará a cada antena_destino; si una antena cumple con b significa que si es solicitada más veces existirá un flujo ocioso entre antena_inicioi y la antena_destinoj que cumplio con b. Esto puede provocar que una antena que consiguió k antenas en su backup al momento de creación del

grafo ya no los pueda cumplir, si sucede esto se finaliza el algoritmo, se hablará más en profundidad en el apartado **Diseño de corte**. Esto puede suceder dependiendo de cuán restrictivo es b, independientemente si la antena tiene muchas antenas más cercanas a D o consiguió las justas y suficientes.

Diseño de corte:

Durante la creación del grafo si una antena i (representada por antena_inicioi) no se puede conectar a k antenas j siguiendo la restricción que dicta D el algoritmo retorna un diccionario vacío porque por lo menos la antena i no puede cumplir el objetivo del algoritmo (pueden existir más, pero el algoritmo corta al encontrar la primera que no cumple la condición).

```
    if conexiones_validas < k:
                print(f"Antena {i+1} no puede conectarse a {k}
backups válidos.")
                return dict()
```

Esta es una restricción básica del problema, las distancias representadas por la matriz no garantizan que el problema sea resoluble porque D restringe las opciones que tienen las antenas para conectarse entre sí. Esta restricción es útil pero no suficiente; aun habiendo las suficientes conexiones válidas no se garantiza que las antenas aparezcan menos de b ni que, aun cumpliendo esa condición de apariciones, se pueden seguir cumpliendo que se pueden crear los backup de k longitud.

Según el corte mínimo de determinará si el flujo es suficiente para la restricción de b y k, la cantidad de antenas * k es el flujo mínimo que debe tener el grafo para ser resoluble.

Si el flujo es menor, significa que no fue posible encontrar backups de tamaño k para cada antena **sin exceder** b repeticiones de ninguna antena.

```
if(flujo_maximo < (antenas * k)):
        print(f"El flujo maximo {flujo_maximo} no es suficiente para
abastecer n*k {antenas*k}\nSignificando que hay 1 o mas antenas que
aparecen mas de b veces en los k backups")
        return dict()
```

Cuando se mira la capacidad ociosa del grafo final, que haya una arista desde un inicio hasta un destino sin usar significa que el algoritmo de búsqueda del flujo máximo no logró conectar un destino pretendido por una antena i, no pudiendo cumplir la restricción de k antenas en su backup.

Esta condición garantiza que:

- Cada nodo `inicio_antenai` pudo entregar exactamente k unidades de flujo.
- Ningún `destino_antenaj` recibió más de b unidades de flujo.
- Por lo tanto, **la solución cumple con los requisitos del problema**.

Para el grafo dirigido se eligió el de la librería **networkx**. Para encontrar el flujo máximo utiliza Edmonds-Karp y la función lo hace entrega además el grafo resultante con los flujos ociosos que quedan al final de la resolución. Internamente se comporta como un diccionario anidado por lo que es muy eficiente temporalmente y facilita la conversión de la salida de Edmonds-Karp a una solución del problema de las antenas.

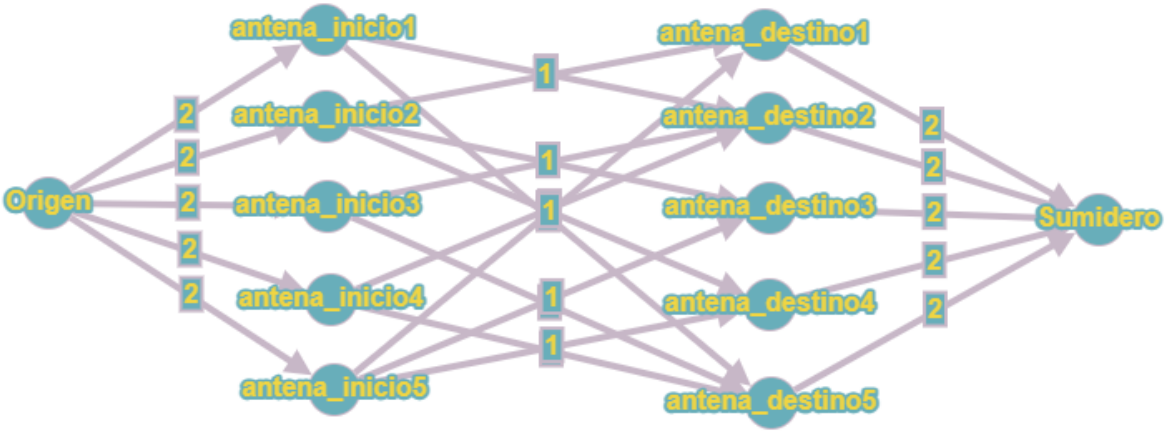
Seguimiento

A continuación se realizará un seguimiento del algoritmo diseñado para una entrada:

```
(5, [
    [0, 3, 4, 4, 3],
    [3, 0, 3, 3, 7],
    [4, 3, 0, 4, 3],
    [4, 3, 4, 0, 3],
    [3, 7, 3, 3, 0]
], 2, 2, 4)
```

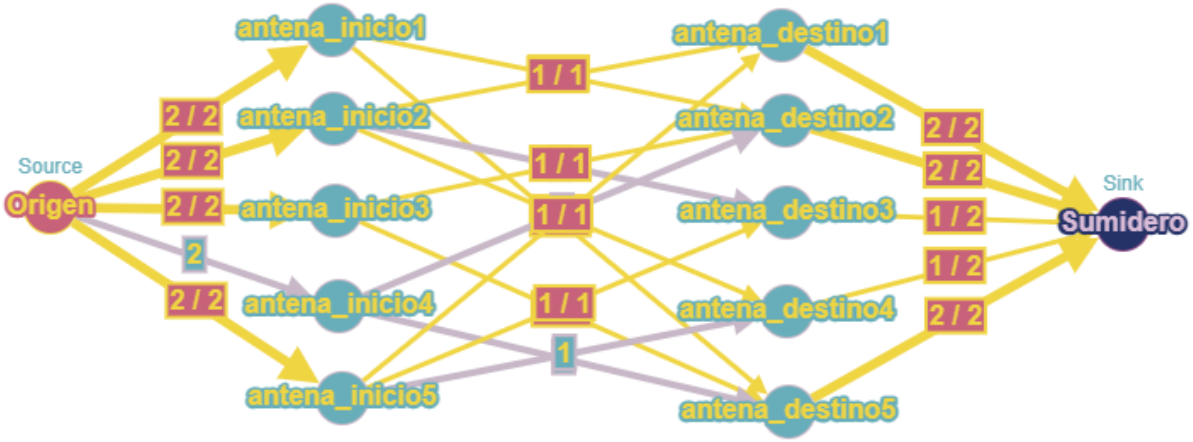
Que representa que hay 5 antenas, la matriz que representa las distancias entre antenas, un k = 2, un b = 2 y un D = 4

- Se crean un nodo origen y por cada antena un antena_inicioi. El origen se unirá a cada inicio con peso k, en este caso 2
- Se crean por cada antena un nodo_destinoj y un sumidero. los destinos se unirán al sumidero con peso b, en este caso 2
- Las conecciones entre los inicios y destinos tendrán peso 1:
- La antena 1 esta está conectada a 2 antenas a una distancia menor a D: la antena 2 con distancia 3 y la antena 5 a 3 de distancia. A este punto se pudo formar un backup de k antena por lo que se continúa con la evaluación de las antenas.
- La antena 2 esta está conectada a 3 antenas a una distancia menor a D: la antena 1 con distancia 3, la antena 3 a 3 de distancia y la antena 4 a 3 de distancia. A este punto se pudo formar un backup de k antena por lo que se continúa con la evaluación de las antenas.
- La antena 3 esta está conectada a 2 antenas a una distancia menor a D: la antena 2 con distancia 3 y la antena 5 a 3 de distancia. A este punto se pudo formar un backup de k antena por lo que se continúa con la evaluación de las antenas.
- La antena 4 esta está conectada a 2 antenas a una distancia menor a D: la antena 2 con distancia 3 y la antena 5 a 3 de distancia. A este punto se pudo formar un backup de k antena por lo que se continúa con la evaluación de las antenas.
- La antena 5 esta está conectada a 3 antenas a una distancia menor a D: la antena 1 con distancia 3, la antena 3 a 3 de distancia y la antena 4 a 3 de distancia. A este punto se pudo formar un backup de k antena por lo que se continúa con la evaluación de las antenas.
- A continuación se mostrará el grafo resultante:



- Cómo se consiguió que las antenas puedan generar backups de k o más cumpliendo la restricción D se procede a calcular el flujo máximo mediante Edmonds-Karp:

- El flujo máximo es = 8 y el grafo residual resulta en:



- El flujo no es suficiente para cumplir k*n, eso significa que hay una o más antenas que no pudieron conectarse a otra porque esta última ya cumplio con su requisito b. En este caso se debe ver las conexiones del origen hacia los inicios: el flujo que le llega a inicio_antena4 no se usa, significando que la antena 4 no pudo conectarse a la 2 y a la 5. Esto se debe a que ambas antenas ya cumplieron su requisito b y ya no cuentan con capacidad ociosa que le permita a la antena 4 conectarse siguiendo las restricciones del modelo. En términos del problema de las antenas: “la antena 4 no pudo crear un backup de tamaño 2 porque las antenas 2 y 5 fueron solicitadas b veces”.

Complejidad

Se analizará el caso de encontrar exitosamente los backup para las n antenas, porque es el que más recursos utiliza (el “peor caso”).

Analicemos la complejidad de la construcción del grafo:

- Realiza n creaciones de nodos inicio más el origen y se los conecta. Internamente el grafo el grafo de **networkx** se comporta como un diccionario por lo que individualmente agregar los nodos tiene complejidad O(1).
- Realiza n creaciones de nodos destino más el sumidero y se los conecta. Internamente el grafo el grafo de **networkx** se comporta como un diccionario por lo que individualmente agregar los nodos tiene complejidad O(1).
- La combinación de las 2 operaciones resulta en una complejidad O(2n+ 2), que puede simplificarse a O(n) por la presencia de constantes.
- A cada Inicio y destino se los une por un arco con peso 1. La complejidad individual de cada una de estas operaciones es O(1). Cada una de las antenas se puede conectar a como mucho n-1 antenas. Por lo que la complejidad total de este paso es O(n*(n-1)), O(n^2) quitando la resta constante.

La complejidad de la función del flujo máximo de **networkx**:

- Como se adelantó en el apartado anterior el algoritmo que utiliza Edmonds-Karp para encontrar el flujo máximo y el grafo residual. Este algoritmo cuenta con una complejidad O($m^2 v$) donde $m = cantidad\ de\ aristas = n^2$ y $v = cantidad\ de\ vértices = 2n + 2$.
- Al retorno de Edmonds-Karp se lo adapta como un diccionario adecuado para el problema de las antenas. Para ello se enfoca en los nodos inicio y destino pero recorriendo la totalidad del grafo residual. Cada consulta tiene complejidad O(1) pero se itera en los v vértices y las m aristas, resultando en una complejidad total de O(vm).

- La sumatoria de todos los pasos que tiene el modelo resulta en una complejidad de:

$$O(n + n^2 + m^2 v + vm)$$

Recordando que $v = 2n + 2$ y m es a lo sumo n^2 nos da la complejidad de:

$$O(2n^5 + 2n^4 + 2n^3 + 3n^2 + n) \simeq O(n^5)$$

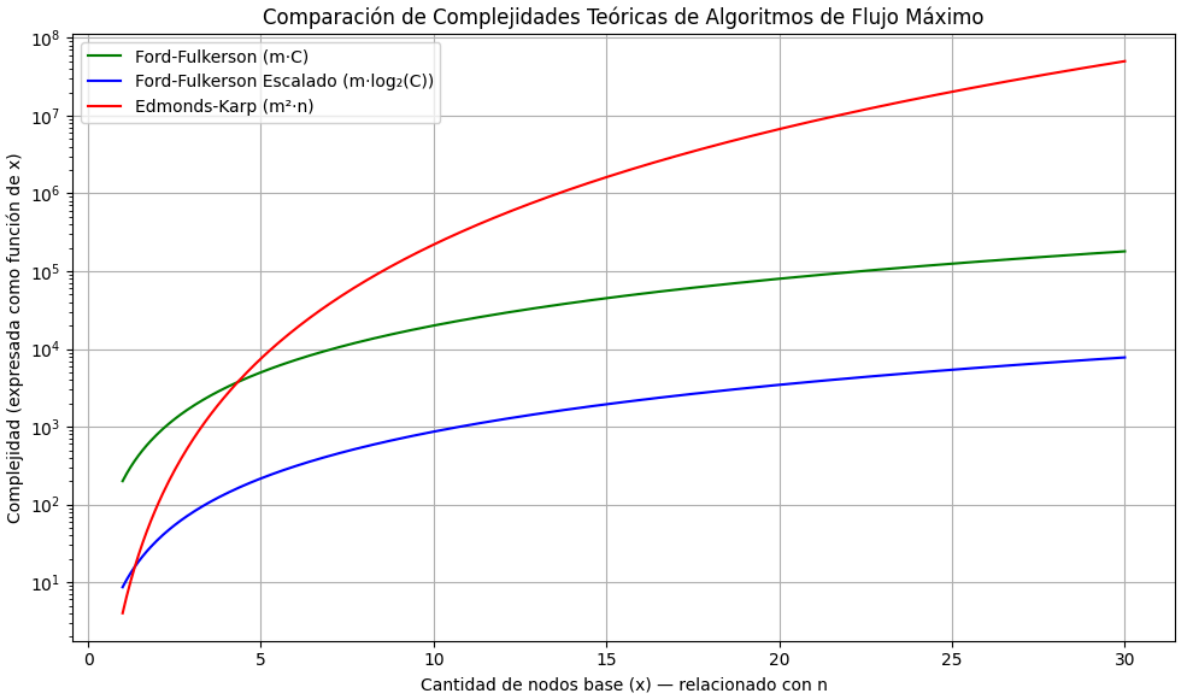
Un algoritmo polinomial de orden alto que no necesariamente es el más rápido en varias situaciones.

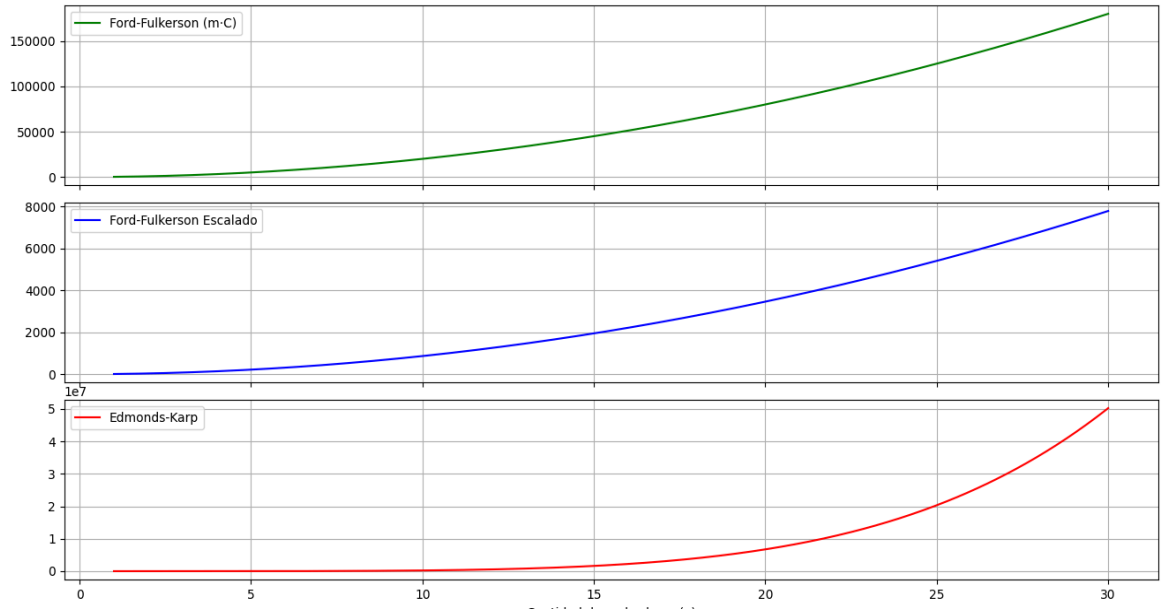
Anexo: ¿Cuál algoritmo de flujo máximo es el más rápido según la situación?

Por el diseño de algoritmo y la construcción del grafo C (flujo máximo) no puede ser superior a nk dependiendo del valor de k para el resultado del flujo. De cualquier forma eso provoca que un algoritmo que dependa de C sea pseudo polinomial pero también se ve que en determinadas instancias y dependiendo de k se lograría un tiempo menor al conseguido por Edmonds-Karp.

A continuación se presentará la evolución de los valores conseguidos por los 3 algoritmos para hallar el flujo máximo vistos en clase: Ford-Fulkerson, Ford-Fulkerson escalado y Edmonds-Karp. Los tres tendrán las mismas condiciones:

$m \leq antenas^2$, $n = 2antenas + 2$, $C = 200$ (El valor 200 es el máximo que puede tomar el flujo en las condiciones de creación de los set de datos de antenas_aleatorias.py)





Para el contexto del problema de las antenas, Edmonds-Karp es el algoritmo más lento y Ford-Fulkerson escalado es más rápido. Esto se debe a que el flujo máximo del grafo de antenas por ningún motivo va a se mayor a mn:

$C = mk$

$m \leq antenas^2$

$k < antenas$

$n = 2antenas + 2$

Tiempos de Ejecución

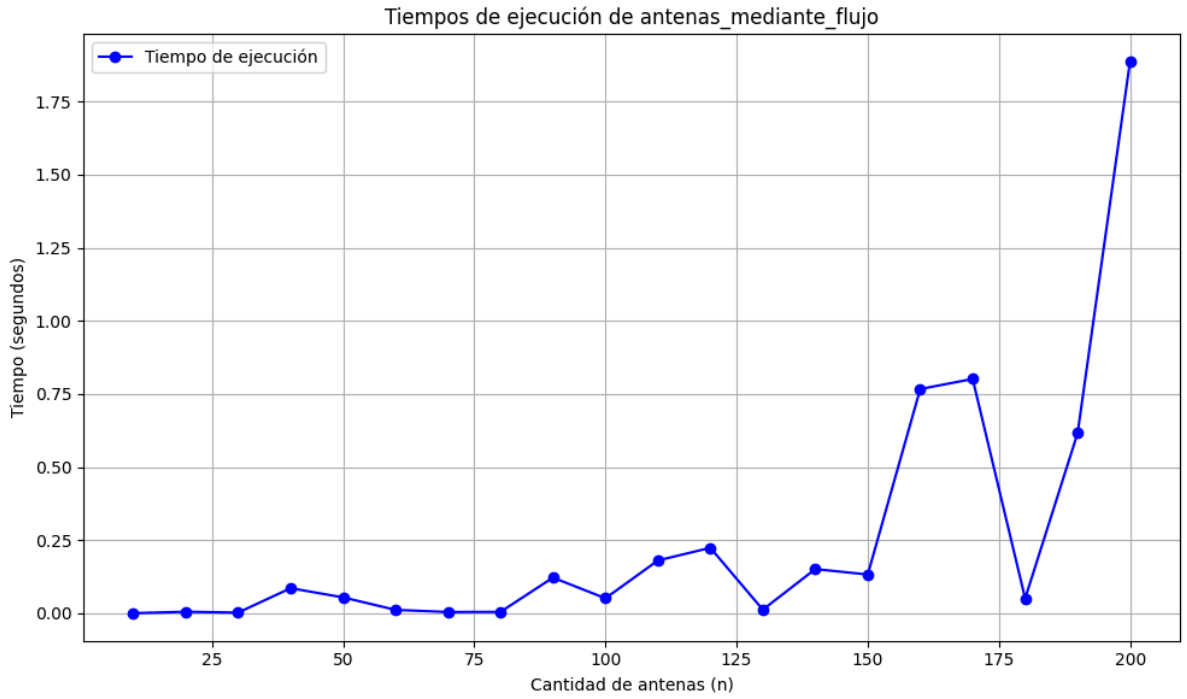
Se realizaron 30 ejecuciones del algoritmo para poder observar los tiempos que se obtienen según n, k, b y D y si se logro encontrar un backup de k longitud para las n antenas o en qué instancia de corte el algoritmo resolvió que no se puede obtener este listado.
Para generar instancias aleatorias del problema se usaron valores de n, k, b y D en rangos que permitan mostrar todas las resoluciones que dio el algoritmo.

Para las mediciones de tiempos y gráficos se utilizó un computador con un CPU Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz y 8 GB de RAM.
Se proporciona la semilla que generó esa instancia para repetir las condiciones experimentales.

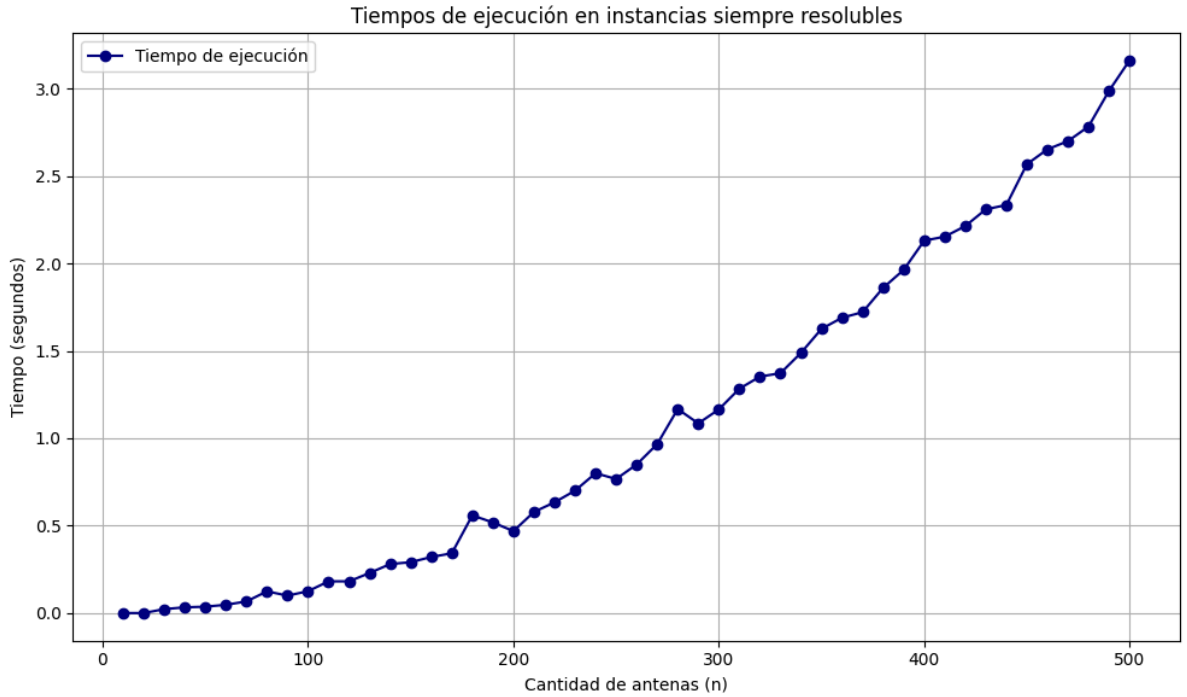
Ejecución	n	k	b	D	Existen backup	Donde finalizó	seed	Tiempo
1	12	2	18	71	Si	-	761528	0.019855
2	16	2	22	24	No	Creación G	621553	0.0030129
3	7	3	2	71	No	Aparición > b	254665	0.0056295
4	19	9	18	40	No	Creación G	585563	0.0055611
5	14	7	18	90	Si	-	571485	0.0083730
6	9	3	8	52	Si	-	159160	0.011744
7	16	4	6	91	Si	-	245561	0.011894
8	19	3	27	82	Si	-	140395	0.019342
9	13	3	13	29	No	Creación G	237252	0.0086536
10	12	6	10	47	No	Creación G	233549	0.0082457
11	15	3	3	89	Si	-	862013	0.013196
12	11	2	2	48	Si	-	17824	0.0092103
13	15	5	21	92	Si	-	726402	0.027755
14	7	3	7	95	Si	-	160136	0.0088174
15	6	3	6	33	No	Creación G	42204	0.0082443
16	11	3	9	77	Si	-	574209	0.0050964
17	13	2	19	53	Si	-	76935	0.014084
18	14	6	11	93	Si	-	474849	0.0075676

19	15	6	17	73	Si	-	123270	0.010225
20	14	3	2	58	No	Aparición > b	736256	0.0068328
21	5	2	2	84	Si	-	864659	0.0032411
22	9	3	6	94	Si	-	816444	0.0085871
23	6	2	2	28	No	Creación G	168042	0.0072904
24	13	5	16	90	Si	-	209145	0.0040517
25	13	6	6	29	No	Creación G	802728	0.0061431
26	13	6	9	92	Si	-	561294	0.0075376
27	19	8	5	93	No	Aparición > b	347851	0.011597
28	10	3	11	35	No	Creación G	577977	0.011784
29	6	3	7	81	No	Creación G	666595	0.0038798
30	6	2	6	96	Si	-	358288	0.0070739
promedio	-	-	-	-	-	-	-	≃0.0092516

Gráficamente se pueden observar las diferencias de costo temporal según las diferentes respuestas que puede dar el algoritmo dependiendo de n, k, b y D y si logró o no encontrar los n backup de tamaño k:



Forzando valores donde siempre se pueda encontrar un k backup para las n antenas se puede observar una curva esperable para un algoritmo con complejidad polinomial pero de orden alto:



Informe

El objetivo de este experimento fue analizar empíricamente el comportamiento del algoritmo que resuelve “el problema de backups de antenas” mediante redes de flujo. La hipótesis inicial establece que el algoritmo tiene una cota de complejidad superior de orden $O(n^5)$, considerando que el método de flujo utilizado (Edmonds-Karp) tiene una complejidad de $O(VE^2)$), donde el grafo crece con n, la cantidad de antenas.

Se generaron instancias del problema con tamaños n (cantidad de antenas) desde 200 hasta 1000, en pasos de 10 unidades. Para cada instancia:

- Se aseguró la generación de un grafo que genere una instancia del problema de de backups de antenas siempre resoluble (para evitar tiempos nulos o triviales).
- Se midió el tiempo de ejecución con `time.perf_counter()` para mayor precisión.
- Se graficaron los tiempos medidos junto con una curva teórica de referencia basada en $O(n^5)$, escalada para ser visualmente comparable.

Aunque la curva de complejidad general sigue una tendencia creciente compatible con una cota superior de orden $O(n^5)$, se presentan **picos agudos y recurrentes** en puntos específicos $n = 460$, $n = 760$ y $n = 900$. Si bien la cota superior domina el crecimiento asintótico, en la práctica es posible que **términos de orden inferior** (como los presentes en la expresión real $O(2n^5 + 2n^4 + 2n^3 + 3n^2 + n)$ **no sean despreciables** en ciertos rangos de n.

Sin embargo, el hecho de que las variaciones se concentren en valores muy puntuales puede deberse a :

- Cuando se incrementa el número de antenas, la cantidad y densidad de conexiones también se incrementa. Sin embargo, generar aleatoriamente la matriz, n , k, b y D, no garantiza uniformidad, causando que se expanda considerablemente la cantidad de caminos posibles para el algoritmo de flujo en los picos mencionados.
- La naturaleza combinatoria de este problema puede provocar puntos de transición donde una pequeña variación en los parámetros (como n) genera una transición de difícil a fácil resolución o viceversa.
- Las interacciones con el sistema operativo y el entorno de ejecución pueden provocar estos picos. Ante la persistencia de los picos se usó otro computador con un AMD Ryzen 5 5600 para generar el gráfico pero, aunque se generaron gráficos con menor ruido, los 3 picos persistieron.
- Aunque la complejidad teórica de Edmonds-Karp es polinomial, su rendimiento en la práctica depende fuertemente del número de caminos aumentantes que deben encontrarse. En las instancias donde el número de caminos viables es alto (como ocurre al aumentar densamente las conexiones), la cantidad de iteraciones y la profundidad de cada búsqueda de caminos origen-sumidero crecen considerablemente, provocando los picos. Para los grafos generados en este problema se vio que Edmonds-Karp no era el óptimo ya que el algoritmo por cómo se genera el grafo se podían generar una gran densidad de aristas y vértices que duplican la cantidad de antenas ¿Utilizando Ford-Fulkerson escalado existirían esos picos?

Conclusión

Los resultados obtenidos muestran que el comportamiento del algoritmo concuerda con la complejidad teórica estimada $O(n^5)$, validando el análisis formal. Los picos detectados reflejan fenómenos reales y bien documentados en la literatura de algoritmos combinatorios, y no contradicen el análisis de complejidad sino que lo complementan.