Teoría de Algoritmos
(75.29) Curso Echevarría

Trabajo Práctico 1

1 de junio de 2025

Nombre	Padron
Martín Osan	109179
Santiago Alabes	110043
Felipe Gazcon	110933
Ivan Pfaab	103862

Índice

1. Ejercicio 1 - Programación Dinámica

1.1. Presentación del problema

PROBLEMA 1 - Programación Dinámica 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

1) La cadena s, de entrada, tiene longitud mayor o igual a 1.

2) Se permite que una subcadena tenga longitud 1 (los caracteres individuales son palíndromos).

3) El objetivo es minimizar el número de substrings palindrómicos, no la longitud total.

1.3. Diseño

a) **Ecuación de recurrencia:**

Definimos $dp[i]$ como el mínimo número de substrings palindrómicos en los que se puede dividir el prefijo $s[0..i]$. Si $s[0..i]$ es un palíndromo, entonces:

$$dp[i] = 1$$

En caso contrario:

$$dp[i] = \min_{1 \leq j \leq i} \{dp[j-1] + 1 \mid s[j..i] \text{ es un palíndromo}\}$$

b) **El algoritmo cumple con ambos requisitos:**

- **Subestructura óptima:** porque la solución para cada posición depende de las soluciones óptimas de subproblemas más pequeños. Es decir, para calcular el valor óptimo en la posición i, se consideran todas las posibles particiones anteriores j tales que $s[j..i]$ es un palíndromo, y se toma el mínimo entre las soluciones $dp[j-1]+1$.
- **Subproblemas superpuestos:** porque se reutilizan cálculos memorizados para evitar resolver lo mismo muchas veces. La matriz `is_pal[i][j]` guarda si una subcadena es un palíndromo, y el arreglo `dp[i]` guarda el mínimo número de particiones hasta esa posición.

c) **Memoization:**

Se usa memoization con la matriz `is_pal[i][j]`, que guarda si $s[i..j]$ es un palíndromo o no. También se usa `dp[i]`, que almacena las soluciones a los subproblemas y evita repetir cálculos.

d) **Pseudocodigo**

```
Funcion MinPalindromo(s: cadena) -> entero
    n = longitud(s)
    Crear matriz is_pal[n][n] = FALSO

    Para i = 0 hasta n-1 hacer
        is_pal[i][i] = VERDADERO

    Para longitud = 2 hasta n hacer
        Para i = 0 hasta n - longitud hacer
            j = i + longitud - 1
```

```
            Si s[i] = s[j] entonces
                Si longitud = 2 o is_pal[i+1][j-1] = VERDADERO entonces
                    is_pal[i][j] = VERDADERO

    Crear arreglo dp[0..n-1] =

    Para i = 0 hasta n-1 hacer
        Si is_pal[0][i] = VERDADERO entonces
            dp[i] = 1
        Sino
            Para j = 1 hasta i hacer
                Si is_pal[j][i] = VERDADERO entonces
                    dp[i] = min(dp[i], dp[j-1] + 1)

    Retornar dp[n-1]
```

e) **Estructuras de Datos Utilizadas:**

- `is_pal[n][n]`: matriz booleana para memoizar si $s[i..j]$ es un palíndromo.
- `dp[n]`: arreglo de enteros donde `dp[i]` guarda la mínima cantidad de palíndromos para $s[0..i]$.

1.4. Seguimiento del Algoritmo

Utilizamos la cadena "AAB" como caso de ejemplo.

Paso 1: Inicialización

Se crea una matriz booleana `is_pal` de 3×3 (ya que la longitud de la cadena es 3), donde `is_pal[i][j]` indicará si la subcadena $s[i..j]$ es un palíndromo.

También se crea un arreglo `dp` de tamaño 3, inicializado con infinito (∞), donde `dp[i]` almacenará la mínima cantidad de palíndromos en los que se puede dividir el prefijo $s[0..i]$.

Paso 2: Cálculo de Palíndromos (`is_pal`)

Se marcan como verdaderos todos los caracteres individuales, ya que toda letra por sí sola es un palíndromo:

- `is_pal[0][0]` = `True` (subcadena "A")
- `is_pal[1][1]` = `True` (subcadena "A")
- `is_pal[2][2]` = `True` (subcadena "B")

Luego se evalúan las subcadenas de longitud 2:

- `is_pal[0][1]` = `True` ("AA" es palíndromo)
- `is_pal[1][2]` = `False` ("AB" no es palíndromo)

Finalmente, la subcadena de longitud 3:

- `is_pal[0][2]` = `False` ("AAB" no es palíndromo)

Paso 3: Cálculo de `dp`

Ahora se calculan los valores de `dp` de forma secuencial:

- Para $i = 0$ (subcadena "A"):
  Como "A" es un palíndromo completo (`is_pal[0][0]` = `True`), se asigna:

$$dp[0] = 1$$

- Para $i = 1$ (subcadena "AA"):
  "AA" es un palíndromo completo (`is_pal[0][1]` = `True`), por lo tanto:

$$dp[1] = 1$$

- Para $i = 2$ (subcadena "AAB"):
  Como "AAB" no es un palíndromo completo (`is_pal[0][2]` = `False`), se buscan particiones válidas:
  - Para $j = 1$: subcadena $s[1.,2]$ = "AB" no es palíndromo.
  - Para $j = 2$: subcadena $s[2.,2]$ = "B" sí es palíndromo y $dp[1] = 1$.

  Entonces:

$$dp[2] = dp[1] + 1 = 1 + 1 = 2$$

1.5. Complejidad Temporal

La complejidad del algoritmo utilizado es $\mathcal{O}(n^2)$, debido a dos componentes principales:

- La creación de la matriz `is_pal` es de complejidad $\mathcal{O}(n^2)$, ya que se evalúa si cada subcadena $s[i..j]$ es un palíndromo.
- El cálculo del arreglo `dp` también tiene complejidad $\mathcal{O}(n^2)$, porque para cada posición i se consideran todas las posibles particiones anteriores j, y se verifica si $s[j..i]$ es un palíndromo usando `is_pal`.

Por lo tanto, la complejidad total del algoritmo es:

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

1.6. Grafico

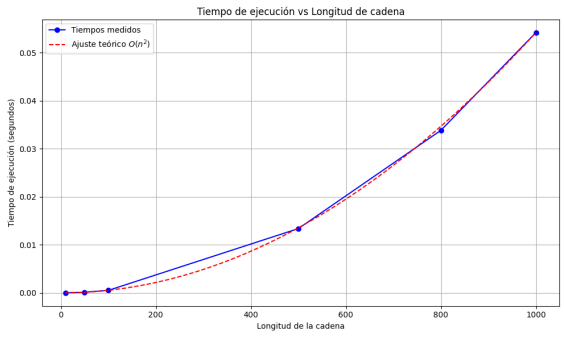

Figura 1: Descripción de la imagen.

1.7. Informe Experimental

Para verificar la complejidad del algoritmo, se ejecutó el programa con conjuntos de entradas de distintas longitudes:

- Longitudes utilizadas: `longitudes = [1, 5, 10, 50, 80, 100]`
- Estas longitudes representan tamaños reales de entrada como:
  $1 \times 10 = 10$, $5 \times 10 = 50$, ..., $100 \times 10 = 1000$
- Los resultados esperados son las longitudes de las palabras, ya que son palabras sin palíndromos.

Tiempos de Ejecución (en segundos)

Longitud real (n)	Tiempo (s)
10	1.71×10^{-5}
50	1.40×10^{-4}
100	5.06×10^{-4}
500	1.34×10^{-2}
800	3.33×10^{-2}
1000	5.33×10^{-2}

Análisis de la Complejidad

Al graficar los tiempos contra las longitudes, se observa una curva que se aproxima a una función cuadrática. Para reforzar esta observación, se calcularon los valores de $\texttt{tiempo}/n^2$, obteniendo valores muy similares:

$$\begin{bmatrix} 1{,}71 \times 10^{-7} & (n = 10) \\ 5{,}60 \times 10^{-8} & (n = 50) \\ 5{,}06 \times 10^{-8} & (n = 100) \\ 5{,}35 \times 10^{-8} & (n = 500) \\ 5{,}21 \times 10^{-8} & (n = 800) \\ 5{,}33 \times 10^{-8} & (n = 1000) \end{bmatrix}$$

En cuanto a los sets de datos particulares, obtuvimos los valores esperados:

Palabra	**resultado**
ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQPONMLKJIHGFEDCBA	1
AAB	2
ARACALACANA	3
LINILAUTOTUA	2
ABC	3
ABBA	1
ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQPONMLKJIHGFEDCBAH	2
ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQPONMLKJIHGFEDCBAHA	3
ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQPONMLKJIHGFEDCBAHAH	2

2. Ejercicio 2 - Programación Lineal

2.1. Presentación del problema

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 Premisas

- Se asume que la cantidad total de paradas publicitarias concesionadas no puede superar las 200 y puede ser menor a 200, sin la necesidad de tener publicidades en cada parada.
- Los beneficios asociados a cada cliente son fijos y determinados en base a las ofertas disponibles.
- No se consideran otros costos o beneficios adicionales fuera de los planteados en las ofertas.
- La decisión de concesionar a un cliente (variables binarias) es definitiva y no se pueden concesionar fracciones o porcentajes.

2.3. Planteo del problema

2.3.1. Función Objetivo

El modelo busca maximizar la ganancia total, sujeta a la disponibilidad de paradas y restricciones de competencia y oferta.

$$\text{máx}\, Z = 50000A + 100000B_1 + 120000B_2 + 100000C + 80000D + 5000E + 40000F + 90000G$$

2.3.2. Variables

- A: variable binaria que indica si se concesiona al cliente A ($A = 1$) o no ($A = 0$).
- $B1$: variable binaria que indica si se concesiona la primer opción ofrecida por el cliente B ($B1 = 1$) o no ($B1 = 0$).
- $B2$: variable binaria que indica si se concesiona la segunda opción ofrecida por el cliente B ($B2 = 1$) o no ($B2 = 0$).
- C: variable binaria que indica si se concesiona al cliente C ($C = 1$) o no ($C = 0$).
- D: variable binaria que indica si se concesiona al cliente D ($D = 1$) o no ($D = 0$).
- E: variable binaria que indica si se concesiona al cliente E ($E = 1$) o no ($E = 0$).
- F: variable binaria que indica si se concesiona al cliente F ($F = 1$) o no ($F = 0$).
- G: variable binaria que indica si se concesiona al cliente G ($G = 1$) o no ($G = 0$).

2.3.3. Restricciones

$$30A + 80B_1 + 120B_2 + 75C + 50D + 2E + 20F + 100G \leq 200$$
$$A + D \leq 1 \quad \text{(No concesionar A y D simultáneamente)}$$
$$B_1 + B_2 \leq 1 \quad \text{(Elegir como maximo solo una opción del cliente B)}$$
$$A, B_1, B_2, C, D, E, F, G \in \{0, 1\}$$

2.4. Metodología y Herramienta Utilizada

Para resolver este problema, se utilizó la biblioteca `PuLP` en Python, una herramienta de modelado para optimización lineal y entera. `PuLP` permite definir variables, restricciones y funciones objetivo de manera sencilla y luego resolver el modelo con solvers externos como CBC, Gurobi, CPLEX, entre otros.

2.4.1. Pasos principales en el código:

- **Creación del problema**: Se define el problema como de maximización.

```
    # Crear el problema
    prob = LpProblem("Maximizar_Beneficio_Publicidad", LpMaximize)
```

- **Definición de variables**: Se crean variables binarias para cada cliente y opción.

```
    # Variables binarias para cada cliente y opci n
    A = LpVariable('A', cat=LpBinary)
    B1 = LpVariable('B1', cat=LpBinary)
    B2 = LpVariable('B2', cat=LpBinary)
    C = LpVariable('C', cat=LpBinary)
    D = LpVariable('D', cat=LpBinary)
    E = LpVariable('E', cat=LpBinary)
    F = LpVariable('F', cat=LpBinary)
    G = LpVariable('G', cat=LpBinary)
```

- **Restricciones**: Se establecen las restricciones de paradas, exclusividad de opciones y competencia.

```
    # Restricci n total de paradas
    prob += (
        30 * A + 80 * B1 + 120 * B2 + 75 * C + 50 * D + 2 * E + 20 * F + 100 *
     G
    ) <= 200

    # Restricci n de competencia entre A y D
    prob += A + D <= 1

    # Restricci n para las opciones del cliente B
    prob += B1 + B2 <= 1
```

- **Función objetivo**: Se especifica la función objetivo a maximizar.

```
    # Funci n objetivo: maximizar beneficios
    prob += (
        50000 * A +
        100000 * B1 +
        120000 * B2 +
        100000 * C +
        80000 * D +
        5000 * E +
        40000 * F +
        90000 * G
    )
```

- **Resolución**: Se llama al solver para encontrar la solución óptima.

```
    # Resolver el problema
    prob.solve()
```

2.5. Conclusión

Los resultados de la optimización indican que la concesión de publicidad debe realizarse a los clientes **A**, **B (con la opción de 80 paradas)**, **C** y **E** para maximizar el beneficio total. Específicamente, se seleccionaron las opciones que suman un beneficio de 255.000 USD, utilizando 187 paradas. Esto muestra que se respetaron las restricciones de disponibilidad de paradas y competencia.
La decisión de concesionar a estos clientes optimiza el uso del espacio limitado y evita conflictos, garantizando así la mayor rentabilidad posible para la concesionaria.

3. Ejercicio 3 - Redes de Flujo

3.1. Presentación del problema

Se está construyendo una red WAN con n antenas y se quiere 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. Se quiere 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.

3.1.1. Supuestos

Se proponen los siguientes supuestos para la resolución del problema:

- Se asume que $d[i, j] = d[j, i] \forall i, j$.
- Una antena no se puede ser parte de su propio conjunto backup.
- Para que una antena i pueda ser parte del conjunto backup de otra antena j, la distancia que las separa tiene que ser estrictamente menor a D. De ser $d[i, j] = D$, i no sera parte del conjunto backup de j.
- Se asume que $0 \leq k \leq n - 1$ y $0 \leq b \leq n - 1$.
- No se aceptaran matrices no cuadradas como d.
- Los argumentos D, k y b deben ser enteros positivos, puesto que no pueden tener -2,5 backups o estar a -2 km.
- Aclaracion: se utilizo la libreria de python NetworkX, la misma sera citadada debidamente en la seccion de Complejidad.

3.2. Diseño

3.2.1. Modelación de Red de Flujo

La solución que se propone hace uso del concepto de Redes de Flujo. Para poder modelar el problema como una red de flujo válida se siguen los siguientes pasos:

- Se crean dos vértices para el grafo: uno será la fuente (s) y otro será el sumidero (t)
- Por cada antena i en el problema se crean dos vértices: i_in y i_out
- Se agregan aristas que salgan desde s y lleguen hasta cada unas de las antenas $antena\_in$ con peso igual a b
- Se agregan aristas que salgan desde cada uno de los vértices correspondientes a la forma $antena\_out$ con capacidad igual a k
- Por cada i_in se agrega una arista con peso 1 dirigida a j_out si $d[i, j] < D$ y $i \neq j$

Ya que a cada *antena_in* le entrará un flujo igual a la cantidad máxima de conjuntos backups a la que puede pertenecer una sola antena, se garantiza que dicho límite nunca sea roto. La cantidad de flujo que puede llegar a t será igual a la cantidad de antenas multiplicada por el tamaño del conjunto backup, por lo que si dicho flujo termina siendo encontrado, se habrá encontrado una solución para el problema. Si el flujo que llega a t no es igual a la cantidad de antenas multiplicada por el tamaño del conjunto backup, entonces el problema no tiene solución. Si una arista desde i_in hacia j_out es usada para transportar flujo, eso significa que la antena i es parte del conjunto backup de j.

A continuación se presenta un grafo construido bajo el procedimiento anteriormente detallado para un problema en el cual hay 4 antenas y los pares de antenas cuya distancia es menor a D es $(1,2),(1,3),(3,4)$. Tener en cuenta que aquellas aristas sin peso especificado tienen peso igual a 1.

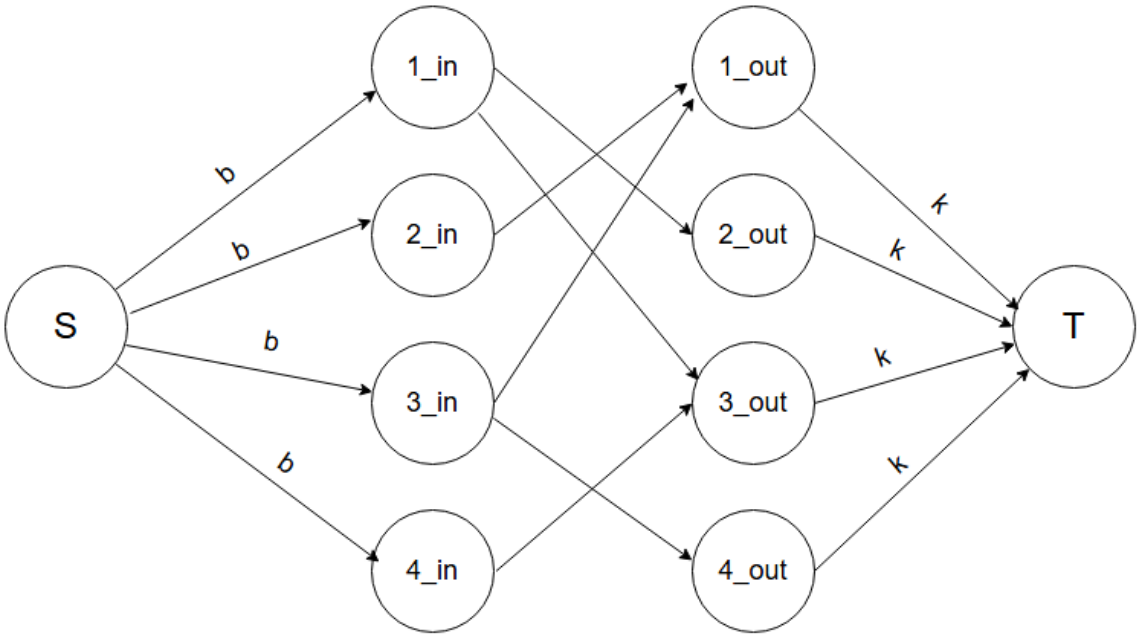

Figura 2: Grafo de ejemplo

3.2.2. Pseudocódigo

```
Entrada: D: numero positivo distancia maxima, b: numero entero positivo cantidad
    maxima de antenas que puede respaldar una antena, k: numero entero positivo
    numero minimo de antenas como respaldo de una antena, d: array de arrays de las
     distancias.
Salida: En caso de que exista una red posible, retorna el grafo residual, imprime
    las aristas con flujo 1. En caso de que no exista la red lo avisa con una
    impresion, y no retorna nada.

WAN_network(Distancia Maxima, Limite de antenas backupeadas por antena, Minimo de
    antenas backup, arrays de distancias)

    si d no es cuadrada:
        return

    si algun parametro ademas de d no es un numero entero positivo:
        return

    network = create_graph(...)
    #Llamado a la funcion create_graph(...), descrita mas abajo

    flujo_maximo_posible, backups = Algoritmo Edmonds-Karp, descrito mas abajo

```

```
    si flujo_maximo_posible < cantidad de antenas * Minimo de antenas backup
        No se puede alcanzar el numero minimo de antenas backup para cada antena.
        return
    sino (si se alcanzo el numero esperado):
        imprimir los backups
        #Llamado a print_connections, descrita mas abajo

    return backups

print_connections(diccionario de backups):
    por cada nodo u del grafo de backups:
        por cada nodo v adyacente a u:
            flow (flujo) = valor de la clave v del diccionario u (flujo de la
    arista)
            si flujo es 1:
                imprimir arista u -> v , u es backup de v

create_graph(...)
    grafo = Grafo dirigido, pesado.

    grafo agrego nodos S y T

    por cada antena dentro del array d:
        grafo agrego nodos antena In y antena Out
        grafo agrego arista s -> antena In, con peso = Limite de antenas
    backupeadas por antena
        grafo agrego arista antena Out -> T, con peso = Minimo de antenas por
    backup

    por cada antena i:
        por cada antena j:
            si j != i y la distancia de i a j es menor a Distancia Maxima:
                grafo agrego arista i In -> j Out, con peso 1

    return grafo


Algoritmo Edmonds Karp (network):
    flujo_acumulado = 0
    grafo_residual = network, con

    mientras haya un camino de s a t posible (usando BFS):

        tomar la capacidad minima del recorrido

        flujo_acumulado += capacidad minima

        por cada par u, v del recorrido:
            flujo arista(u, v) del grafo_residual -= capacidad minima
            flujo arista(v, u) del grafo_residual += capacidad minima

    return flujo_acumulado, grafo_residual
```

3.2.3. Estructuras utilizadas

- Matrices (array de arrays): Utilizamos las matrices como parametro de la funcion principal WAN_network, donde se almacena en el array i en la posicion j la distancia que hay de la antena i a la j. Es utilizada solo para la creacion de la red principal.
- Grafos: La Red principal es modelada como un grafo, donde representamos las antenas del problema como dos vertices: vertice IN y verice OUT, y tambien estaran los vertices fuente y sumidero s y t. Se usara el struct Graph de la libreria NetworkX y luego la implementacion de diccionario anidado.

3.3. Seguimiento

A continuacion una demostracion de como el programa avanza modificando sus estructuras internas.
Datos:

- D = 2, b = 2, k = 1, d = [[0, 1, 2], [1, 0, 1], [2, 1, 0]]

Paso a paso:

- Se llama a la funcion `is_square`, la que se asegurara de que que la matriz d sea cuadrada.
  - $rows = len(d) = 3$
  - $len(d[0]) = 3 == rows$
  - $len(d[1]) = 3 == rows$
  - $len(d[2]) = 3 == rows$
  - return True
- Se hacen las revisiones de que los parametros D, b y k sean aptos.
  - $D <= 0$ or not $type(D) == int \rightarrow$ Falso
  - $b <= 0$ or not $type(b) == int \rightarrow$ Falso
  - $k <= 0$ or not $type(k) == int \rightarrow$ Falso
  - return False
- $network = create\_graph(D, b, k, d)$
  - $newGraph = nx.DiGraph()$
  - newGraph es un struct DiGraph, un grafo dirigido provisto por NetworkX
  - Se añaden los vertices 'S' y 'T' al grafo
  - Se añaden los vertices correspondientes a las Antenas:
    - Para la Antena 0:
      - Se añaden los vertices '0_IN' y '0_OUT'.
      - Se añade la arista 'S'→'0_IN' con capacidad b = 2
      - Se añade la arista '0_OUT'→'T' con capacidad k = 1
    - Para la Antena 1:
      - Se añaden los vertices '1_IN' y '1_OUT'.
      - Se añade la arista 'S'→'1_IN' con capacidad b = 2
      - Se añade la arista '1_OUT'→'T' con capacidad k = 1
    - Para la Antena 2:
      - Se añaden los vertices '2_IN' y '2_OUT'.
      - Se añade la arista 'S'→'2_IN' con capacidad b = 2
      - Se añade la arista '2_OUT'→'T' con capacidad k = 1
  - Se añaden las aristas correspondientes a los posibles respaldos.
    - Para Antena 0:
      - if $0 < 2$ and $0! = 0 \rightarrow$Falso
      - if $1 < 2$ and $0! = 1 \rightarrow$Verdadero
      - Añado la arista '0_IN'→'1_OUT' con peso 1
      - if $2 < 2$ and $0! = 2 \rightarrow$Falso
    - Para Antena 1:
      - if $1 < 2$ and $1! = 0 \rightarrow$Verdadero

        - ◇ Añado la arista '1_IN'→'0_OUT' con peso 1
        - ◇ if $0 < 2$ and $1! = 1$ →Falso
        - ◇ if $1 < 2$ and $1! = 2$ →Verdadero
        - ◇ Añado la arista '1_IN'→'2_OUT' con peso 1
    - Para Antena 2:
        - ◇ if $2 < 2$ and $2! = 0$ →Falso
        - ◇ if $1 < 2$ and $2! = 1$ →Verdadero
        - ◇ Añado la arista '2_IN'→'1_OUT' con peso 1
        - ◇ if $0 < 2$ and $2! = 2$ →Falso

- max_flow, residual_ek = nx.maximum_flow(network, 'S', 'T', 'capacity', edmonds_karp)
    - flujo_acumulado = 0
    - grafo_residual = copia(network)
    - Mientras exista un camino de s a t:
        - flujo_arco = 1 = min(grafo_residual('S','0_IN'), grafo_residual('0_IN','1_OUT'), grafo_residual('1_OUT','T'))
        - grafo_residual('S', '0_IN') -= 1
        - grafo_residual('0_IN', 'S') += 1
        - grafo_residual('0_IN','1_OUT') -= 1
        - grafo_residual('1_OUT','0_IN') += 1
        - grafo_residual('1_OUT','T')) -= 1
        - grafo_residual('T','1_OUT')) += 1
        - flujo_acumulado += 1
    - Este proceso se repetira por cada camino que se encuentre mientras es actualiza el grafo residual. La libreria NetworkX provee el resultado de la funcion de manera tal que cuando lo recorremos, tengan flujo 1 las aristas que son usadas en el problema como backups. Ademas del grafo residual, retorna el flujo_acumulado
- Si el valor de max_flow es menor a la cantidad de antenas por el numero minimo de backups por antena, no hubo red posible.
- if $3 < 3 * 1$ →Falso
- Al ser falso, imprimimos y retornamos las relaciones de backup.
- 'i_IN'→'j_OUT' significa: i es un backup de j
    - Para u = 'S'
        - Para v = '0_IN' adyacente a 'S'
        - flow = 1
        - if 1 and u != 'S' and v != 'T'→Falso
        - No se imprime nada
        - Para v = '1_IN' adyacente a 'S'
        - flow = 1
        - if 1 and u != 'S' and v != 'T'→Falso
        - No se imprime nada
        - Para v = '2_IN' adyacente a 'S'
        - flow = 1
        - if 1 and u != 'S' and v != 'T'→Falso
        - No se imprime nada
    - Para u = '0_IN'

- Para v = '1_OUT' adyacente a '0_IN'
- flow = 1
- if 1 and u != 'S' and v != 'T'→Verdadero
- print('0_IN' → '1_OUT': 1, '0_IN' es backup de '1_OUT')

- Para u = '1_IN'
  - Para v = '0_OUT' adyacente a '1_IN'
  - flow = 1
  - if 1 and u != 'S' and v != 'T'→Verdadero
  - print('1_IN' → '0_OUT': 1, '1_IN' es backup de '0_OUT')
  - Para v = '2_OUT' adyacente a '1_IN'
  - flow = 1
  - if 1 and u != 'S' and v != 'T'→Verdadero
  - print('1_IN' → '2_OUT': 1, '1_IN' es backup de '2_OUT')
- Para u = '2_IN'
  - Para v = '1_OUT' adyacente a '2_IN'
  - flow = 1
  - if 0 and u != 'S' and v != 'T'→Falso
  - No se imprime nada
- Para u = '0_OUT'
  - Para v = 'T' adyacente a '0_OUT'
  - flow = 1
  - if 1 and u != 'S' and v != 'T'→Falso
  - No se imprime nada
- Para u = '1_OUT'
  - Para v = 'T' adyacente a '1_OUT'
  - flow = 1
  - if 1 and u != 'S' and v != 'T'→Falso
  - No se imprime nada
- Para u = '2_OUT'
  - Para v = 'T' adyacente a '2_OUT'
  - flow = 1
  - if 1 and u != 'S' and v != 'T'→Falso
  - No se imprime nada

- return residual_ek
- Finalizado el seguimiento.

3.4. Análisis de Complejidad

En el momento de crear el grafo se generan 2 vértices por cada antena. Luego se recorren todas las parejas de vértices comparando la distancia que las separa con D. Este paso por ende tiene complejidad $O(n^2)$ y produce un grafo con n vértices y n^2 aristas (despreciando a la fuente y al sumidero). Luego se usa la implementación de la librería *networkx* del algoritmo edmonds-karp, que resuelve redes de flujo. Según la documentación oficial: "[this] algorithm has a running time of $O(ve^2)$ for v nodes and e edges."([NetworkX Developers, , párrafo 3]) Como $v = n$ y $e = n^2$, $O(ve^2) = O(n(n^2)^2) = O(n^5)$. Por lo que la complejidad final será $O(n^2 + n^5) = O(n^5)$

3.5. Mediciones de Tiempo

Se realizaron mediciones para valores de n desde 30 hasta 300. Se eligió este intervalo debido a que se tarda demasiado en calcular los resultados para valores de n grandes. Para conseguir mayor consistencia, para cada n se ejecutó el algoritmo 5 veces y se tomó el valor promedio. Además se creó un gráfico de tipo *scatter* que indica si, para un tamaño determinado, la mayoría de las veces se encontró solución o no.

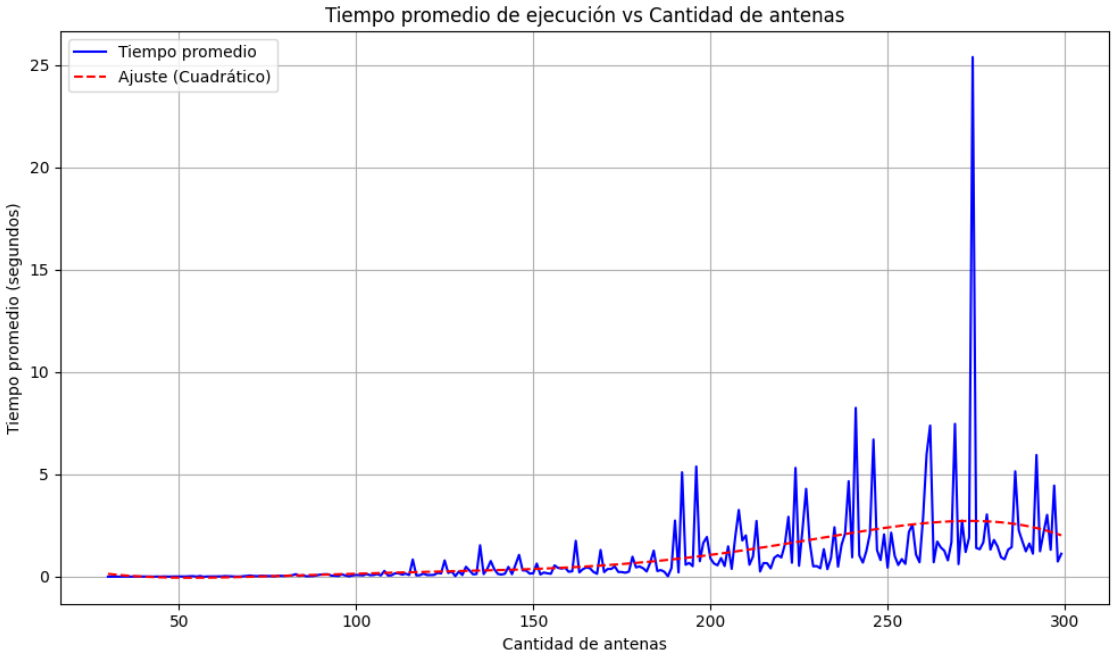

Figura 3: Grafo de ejemplo

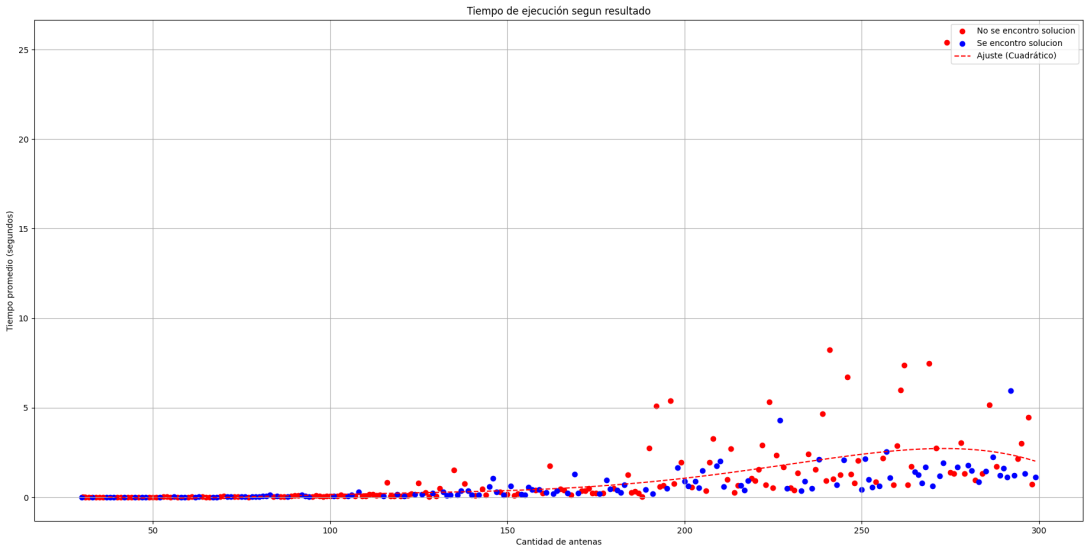

Figura 4: Grafo de ejemplo

Como se observa en los gráficos, el tiempo de ejecución del algoritmo varía mucho para los distintos valores. Se pueden ver altos picos que aparecen frecuentemente. Parecería ser por el gráfico scatter que hay una correlación con respecto a los picos y el hecho de si hay solución o no para el problema, ya que la mayoría de los picos parecerían aparecer cuando no hay solución. Se hizo un ajuste correspondiente a un polinomio de grado 5. El error cuadrático terminó siendo alto (841) lo que no sorprende viendo la inestabilidad del algoritmo.

3.6. Conclusión

Quedó en evidencia la versatilidad de las redes de flujo para poder resolver diversos problemas. La representación propuesta permitió utilizar el algoritmo de Edmonds-Karp para encontrar la solución óptima. Si bien hay picos altos de tiempo de ejecución, la complejidad es polinomial, por lo que es viable para problemas no tan grandes.

Referencias

[NetworkX Developers,] NetworkX Developers. networkx.algorithms.flow.edmonds_karp. `https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.edmonds_karp.html`. Accedido en mayo de 2025.