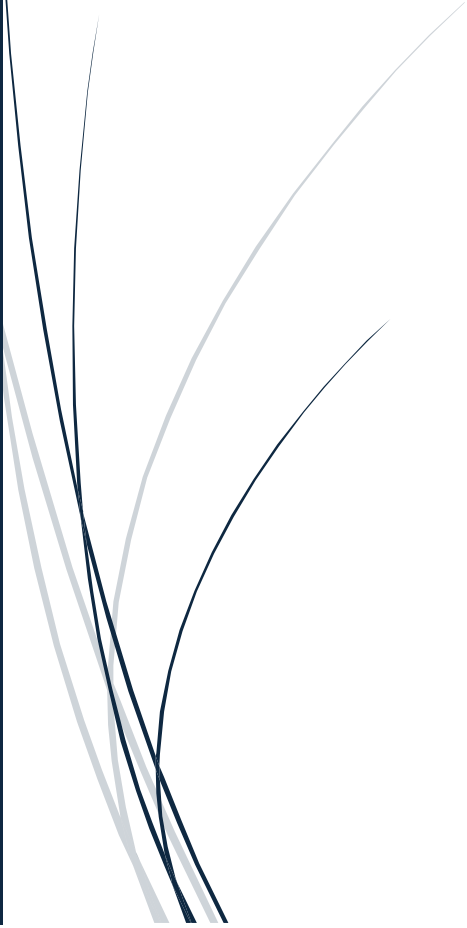




9-12-2024

PRÁCTICA 3

ALGORITMIA



LUIS NÚÑEZ FERNÁNDEZ
LUCÍA CHACÓN SANABRIA
GRUPO: 02

Índice

Tabla de contenido

1. GRAFOS: BÚSQUEDA EN PROFUNDIDAD Y COMPONENTES FUERTEMENTE CONEXAS.....	2
1.1 IMPLEMENTACIÓN DEL TAD GRAFO	2
1.2 RECORRIDO EN GRAFOS Y COMPONENTES FUERTEMENTE CONEXAS	6
1.3 COMPONENTE GIGANTE Y UMBRAL DE PERCOLACIÓN	10
2. PROGRAMACIÓN DINÁMICA	14
2.1 DISTANCIAS ENTRE CADENAS Y SUBCADENA COMÚN MÁS LARGA	14
2.2 MULTIPLICACIÓN DE MATRICES	17
<i>Pruebas para la función edit_distance:</i>	20
<i>Pruebas para la función max_subsequence_length:</i>	20
<i>Pruebas para la función max_common_subsequence:</i>	20
<i>Pruebas para la función min_mult_matrix:</i>	20

1. Grafos: Búsqueda en profundidad y componentes fuertemente conexas

1.1 Implementación del TAD Grafo

En esta sección se desarrolló un Tipo Abstracto de Datos (TAD) para representar un grafo dirigido no ponderado.

Funciones implementadas:

- ***add_node(self, vertex)***

Agrega un nodo si no existe previamente.

```
def add_node(self, vertex) -> None:
    if str(vertex) not in self._V:
        self._V[str(vertex)] = dict()
        self._E[str(vertex)] = set()
        self._init_node(vertex)
```

- ***add_edge(self, vertex_from, vertex_to)***

Crea una arista entre dos nodos, añadiéndolos si aún no estaban en el grafo.

```
def add_edge(self, vertex_from, vertex_to) -> None:
    if str(vertex_from) in self._V and str(vertex_to) in self._V:
        self._E[str(vertex_from)].add(str(vertex_to))
```

- ***nodes(self)***

Retorna las claves de los nodos en el grafo.

```
def nodes(self) -> KeysView[str]:
    return sorted(self._V.keys())
```

- ***adj(self, vertex)***

Devuelve los nodos adyacentes a un nodo dado.

```
def adj(self, vertex) -> Set[str]:
    if str(vertex) not in self._E:
        raise KeyError('Vertex ' + str(vertex) + ' not in graph')
    return sorted(self._E[vertex])
```

- *exists_edge(self, vertex_from, vertex_to)*

Determina si existe una arista entre dos nodos.

```
def exists_edge(self, vertex_from, vertex_to) -> bool:
    if vertex_to in self._E[vertex_from]:
        return True
    else:
        return False
```

- *_init_node(self, vertex)*

Inicializa los atributos de un nodo.

```
def _init_node(self, vertex) -> None:
    self._V[str(vertex)] = {'color': 'WHITE', 'parent': None, 'd_time':
None, 'f_time': None}
```

Funciones auxiliares para manejar los grafos:

- *read_adjlist(file: str)*

Crea un grafo leyendo un archivo en formato de lista de adyacencia. Agrega nodos y aristas según las líneas del archivo. También muestra los archivos en el directorio actual.

```
def read_adjlist(file: str) -> Graph:
    ''' Read graph in adjacency list format from file.
    '''

    print(f"Files in {os.getcwd()}:")
    print(os.listdir(os.getcwd()))

    G = Graph()
    with open(file, 'r') as f:
        for line in f:
            l = line.split()
            if l:
                u = l[0]
                G.add_node(u)
                for v in l[1:]:
                    G.add_edge(u, v)

    return G
```

- ***write_adjlist(G: Graph, file: str)***

Esta función guarda un grafo G en un archivo en formato de lista de adyacencia. Por cada nodo, escribe su identificador seguido de sus nodos adyacentes en una línea. Cada nodo y sus conexiones están separados por espacios.

```
def write_adjlist(G: Graph, file: str) -> None:
    '''Write graph G in single-line adjacency-list format to file.
    '''

    file_path = os.path.join(os.path.dirname(__file__), file)
    with open(file_path, 'r') as f:
        for u in G.nodes():
            f.write(f'{u}')
            f.writelines([f' {v}' for v in G.adj(u)])
            f.write('\n')
```

- ***restart(self)***

Esta función reinicia todos los nodos del grafo a su estado inicial, restableciendo sus atributos (*color*, *parent*, *d_time*, *f_time*).

```
def restart(self) -> None:
    for v in self.nodes():
        self._V[v]['color'] = 'WHITE'
        self._V[v]['parent'] = None
        self._V[v]['d_time'] = None
        self._V[v]['f_time'] = None
```

- ***__init__(self)***

Inicializa un grafo vacío con dos diccionarios: uno para los nodos (*_V*) y otro para las aristas (*_E*).

```
def __init__(self):
    self._V = dict()
    self._E = dict()
```

En el método ***__init__*** se utilizan dos estructuras privadas para almacenar el grafo:

- *self._V*: Diccionario que almacena los nodos del grafo, junto con atributos necesarios para los algoritmos (color, tiempo de descubrimiento y finalización, y nodo padre).
- *self._E*: Diccionario de listas de adyacencia, donde cada clave es un nodo y sus valores son los conjuntos de nodos conectados.

Para comprobar el correcto funcionamiento de estas funciones hemos utilizado el ejemplo que viene en el enunciado de la práctica. Este es el código utilizado:

```
import graph_24 as g
G = g.Graph ()

G.add_node(0)
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)

G.add_edge(0, 1)
G.add_edge(2, 1)
G.add_edge(1, 4)
G.add_edge(4, 3)
G.add_edge(5, 4)
G.add_edge(3, 0)
G.add_edge(5, 2)

print (G)
```

El resultado que obtenemos al ejecutar el primer apartado genera el siguiente output:

```
e506301@1-6-1-6:~/UnidadH/Practica3$ python3 prueba_graph.py
Vertices:
0: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
1: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
2: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
3: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
4: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
5: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}

Aristas:
0: {'1'}
1: {'4'}
2: {'1'}
3: {'0'}
4: {'3'}
5: {'4', '2'}
```

Como podemos comprobar, se muestra correctamente los nodos y las aristas del grafo en el formato especificado. Este output nos permite verificar que la implementación básica del TAD Grafo está funcionando según lo esperado. Podemos observar cómo los nodos han sido inicializados con sus atributos (*color*, *parent*, *d_time*, *f_time*) correctamente establecidos, y las conexiones entre ellos reflejan fielmente la estructura del grafo que se introdujo en el programa. Esto confirma que tanto la creación como la representación de grafos mediante listas de adyacencia han sido implementadas de manera adecuada.

1.2 Recorrido en grafos y componentes fuertemente conexas

En este apartado se implementan y prueban dos aspectos fundamentales de los grafos dirigidos: la búsqueda en profundidad (DFS) y la identificación de las componentes fuertemente conexas utilizando el algoritmo de Tarjan.

Funciones implementadas:

- ***dfs(self, nodes_sorted: Iterable[str] = None)***

Inicia el recorrido en profundidad sobre los nodos del grafo. Si se proporciona un orden específico de nodos, lo sigue; si no, recorre todos los nodos. Llama a la función recursiva *dfs_rec()* para explorar nodos no visitados y retorna un bosque DFS.

```
def dfs(self, nodes_sorted: Iterable[str] = None) -> List[List[Tuple]]:
    forest_forest= []
    cont = 0
    if nodes_sorted is None:
        nodes_sorted = self.nodes()
    for node in nodes_sorted:
        if (self._V[node]['color'] == 'WHITE'):
            forest = []
            cont = self.dfs_rec(node, forest, cont)
            forest_forest.append(forest)

    return forest_forest
```

- ***dfs_rec(self, vertex, path: List[str], cont: int)***

Realiza la exploración recursiva de un nodo y sus vecinos. Marca el tiempo de descubrimiento y finalización de cada nodo, y construye el árbol DFS con nodos y sus padres.

```
def dfs_rec(self, vertex, path: List[str], cont: int) -> List[Tuple]:
    self._V[vertex]['color'] = 'BLACK'
    cont += 1
    self._V[vertex]['d_time'] = cont
    path.append((vertex, self._V[vertex]['parent']))
    for a in self.adj(vertex):
        if self._V[a]['color'] == 'WHITE':
            self._V[a]['parent'] = vertex
            cont = self.dfs_rec(a, path, cont)
    cont += 1
    self._V[vertex]['f_time'] = cont
    return cont
```

- ***graph_conjugate(G: Graph)***

Genera el grafo conjugado (o transpuesto) de un grafo dirigido. Crea un nuevo grafo, añade los mismos nodos que el grafo original, y luego invierte todas las aristas del grafo original, añadiendo las aristas dirigidas en sentido opuesto. Finalmente, retorna el grafo conjugado.

```
def graph_conjugate(G: Graph) -> Graph:
    conjGraph = Graph()

    for u in G.nodes():
        conjGraph.add_node(u)

    for u in G.nodes():
        for v in G.adj(u):
            conjGraph.add_edge(v, u)

    return conjGraph
```

- ***tarjan(self)***

Implementa el algoritmo de Tarjan para encontrar las componentes fuertemente conexas de un grafo. Realiza un recorrido DFS en el grafo original, luego calcula su grafo conjugado y ordena los nodos por su tiempo de finalización. Finalmente, realiza un segundo DFS sobre el grafo conjugado y agrupa los nodos en componentes fuertemente conexas, retornándolas en una lista.

```
def tarjan(self) -> List[List[str]]:
    self.restart()
    self.dfs()

    graph_conj = graph_conjugate(self)

    nodes = self.nodes()

    # Ordenar los nodos por tiempo de finalización
    nodes = sorted(nodes, key = lambda x: self._V[x]['f_time'], reverse =
True)

    dfs_forest_conj = graph_conj.dfs(nodes)
    scc = []
    for tree in dfs_forest_conj:
        scc.append([])
        for e in tree:
            scc[len(scc)-1].append(e[0])

    return scc
```


Para comprobar el correcto funcionamiento de las funciones, hemos usado el ejemplo que viene en el enunciado de la práctica:

```
print(f' DFS forest: {G.dfs ()}')
print ()
print (G)
print ()
print(f' scc : {G.tarjan()}')
```

Este es el output que nos sale:

```
DFS forest: [[('0', None), ('1', '0'), ('4', '1'), ('3', '4')], [('2', None)],
[('5', None)]]
Vertices:
0: {'color': 'BLACK', 'parent': None, 'd_time': 1, 'f_time': 8}
1: {'color': 'BLACK', 'parent': '0', 'd_time': 2, 'f_time': 7}
2: {'color': 'BLACK', 'parent': None, 'd_time': 9, 'f_time': 10}
3: {'color': 'BLACK', 'parent': '4', 'd_time': 4, 'f_time': 5}
4: {'color': 'BLACK', 'parent': '1', 'd_time': 3, 'f_time': 6}
5: {'color': 'BLACK', 'parent': None, 'd_time': 11, 'f_time': 12}

Aristas:
0: {'1'}
1: {'4'}
2: {'1'}
3: {'0'}
4: {'3'}
5: {'2', '4'}

scc : [['5'], ['2'], ['0', '3', '4', '1']]
```

Los resultados obtenidos son correctos y coinciden con lo esperado. El algoritmo de búsqueda en profundidad (DFS) asigna correctamente los tiempos de descubrimiento y finalización de cada nodo, lo que indica que el recorrido en profundidad se ha realizado de manera adecuada. Además, el bosque DFS se genera correctamente, dividiendo los nodos en árboles según el orden en que fueron visitados, y reflejando las relaciones de padres e hijos entre los nodos. Los vértices se marcan como "BLACK", lo que confirma que han sido completamente procesados, y las aristas entre los nodos se listan correctamente, mostrando las conexiones directas.

Por otro lado, las componentes fuertemente conexas (SCC) se identifican de manera precisa, agrupando correctamente los nodos que están fuertemente conectados entre sí. El algoritmo realiza el análisis de la conectividad del grafo correctamente, identificando los

subconjuntos de nodos que forman estas componentes. En resumen, el programa funciona correctamente, produciendo una salida que refleja de forma precisa el comportamiento esperado del algoritmo y el análisis de las conexiones en el grafo.

1.3 Componente Gigante y umbral de percolación

En este apartado se estudia cómo la adición de aristas a un grafo aleatorio provoca la formación de una componente gigante, es decir, un grupo grande de nodos interconectados. El umbral de percolación es el punto crítico donde esto sucede, lo que indica un cambio en la estructura del grafo.

Funciones implementadas.

- *erdos_renyi(n: int, m: float = 1.)*

Esta función crea un grafo dirigido con n nodos, donde cada nodo tiene un número aleatorio de vecinos basado en una distribución binomial con parámetro $p = m/n$. Para cada nodo, se seleccionan aleatoriamente otros nodos como vecinos, y se agregan aristas entre ellos si no existen. El resultado es un grafo aleatorio con conexiones dirigidas entre los nodos.

```
def erdos_renyi(n: int, m: float = 1.) -> Graph:

    G = Graph()
    for i in range(n):
        G.add_node(i)

    numVecinos = binom.rvs(n, m/n, size=n)

    for u in G.nodesint():
        choices = list(G.nodesint())
        while numVecinos[int(u)] > 0 and choices:
            v = random.choice(choices)
            choices.remove(v)
            if not G.exists_edge(u, v):
                G.add_edge(u, v)
                numVecinos[int(u)] -= 1

    return G
```

- *size_max_scc(n: int, m: float)*

Genera un grafo aleatorio dirigido con n nodos y el parámetro m . Luego, calcula las componentes fuertemente conexas del grafo utilizando el algoritmo de Tarjan. La función encuentra el tamaño de la mayor componente fuertemente conexas y devuelve una tupla con el tamaño de esa componente normalizado por n y el valor de m .

```
def size_max_scc(n: int, m: float) -> Tuple[float, float]:  
    g = erdos_renyi(n, m)  
  
    componentes_conexas = g.tarjan()  
    maximo = max(len(componente) for componente in componentes_conexas)  
  
    return maximo/n, m
```

Para comprobar el correcto funcionamiento de las funciones, se utiliza la función *grafica* para representar visualmente la relación entre el tamaño normalizado de la mayor componente fuertemente conexas (SCC) y el número esperado de vecinos por nodo m en un grafo aleatorio dirigido de Erdős-Rényi. La gráfica generada por esta función muestra cómo el tamaño de la mayor SCC varía a medida que cambiamos el número promedio de vecinos por nodo.

La función *grafica* recibe una lista de puntos (pares de valores) que contienen el tamaño de la mayor componente fuertemente conexas (SCC) y el número esperado de vecinos por nodo m . Estos puntos se utilizan para crear un gráfico de dispersión utilizando matplotlib. El resultado se guarda en un archivo de imagen como percolation.png, aunque se puede activar la visualización interactiva si se descomenta la línea `plt.show()`.

Para generar la gráfica, hemos utilizado todo el código siguiente:

```
import matplotlib.pyplot as plt
from graph_24 import *
import numpy as np

points = []
n = 1000

def grafica(points, file='percolation.png') -> None:
    '''Genera una gráfica en el fichero file'''

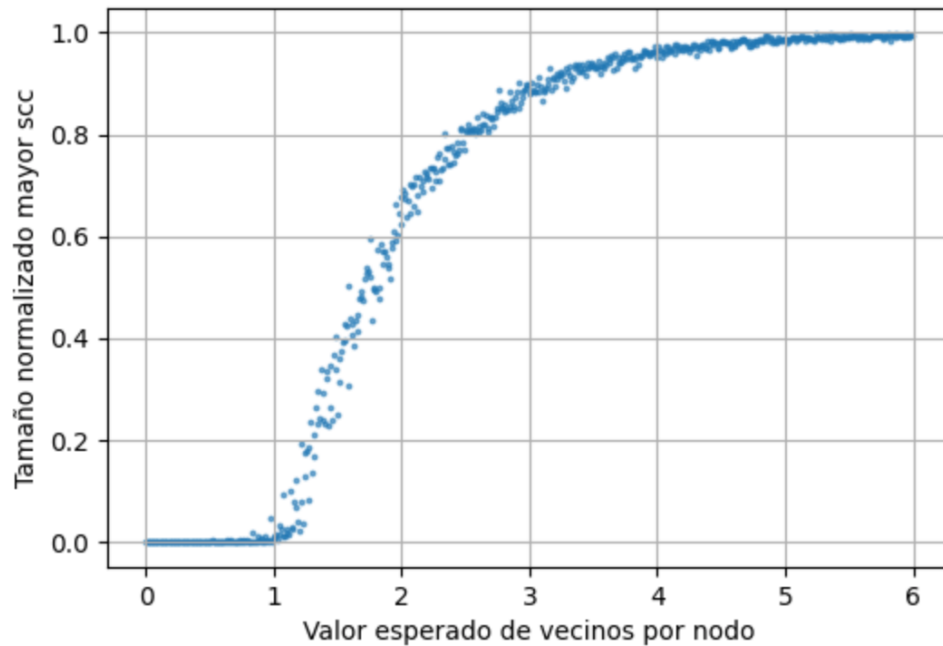
    y, x = zip(*points)

    fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    ax.scatter(x, y, alpha=0.6, s=3)
    ax.set_ylabel (f'Tamaño normalizado mayor scc')
    ax.set_xlabel (f'Valor esperado de vecinos por nodo')
    ax.grid()
    plt.savefig(file)
    #plt.show()

mmax = 6
incr = 0.01
for m in np.arange(0, mmax, incr):
    os.system('clear')
    print(f'Graficando... {((m/mmax)*100):.2f}%\n')
    points.append(size_max_scc(n, m))

grafica(points)
```

Gráfica:



Al generar la gráfica, la cual está en percolation.png, observamos un comportamiento que está en línea con lo que se espera en el análisis del umbral de percolación. Específicamente, cuando el número promedio de vecinos por nodo m es inferior a un valor crítico $m_c \approx 1$, el grafo tiende a estar muy desconectado, y la mayoría de las componentes conexas consisten en un solo nodo. Sin embargo, a medida que m supera este valor crítico, el grafo comienza a condensarse, formando una componente conexas gigante. A partir de este umbral, la mayoría de los nodos pertenecen a esta componente gigante, mientras que las componentes más pequeñas se vuelven menos frecuentes y de tamaño reducido.

Este comportamiento es un reflejo del fenómeno de percolación, donde la transición de fase es clara en el valor crítico m_c . En los grafos dirigidos, aunque el comportamiento es similar al de los grafos no dirigidos, la transición puede no ser tan abrupta debido a la naturaleza direccional de los enlaces.

La gráfica generada sigue este patrón esperado: un aumento brusco en el tamaño de la mayor SCC a medida que el valor de m supera el umbral crítico. Esto valida que la implementación está funcionando correctamente y genera el comportamiento teórico esperado para grafos aleatorios dirigidos.

2. Programación Dinámica

En este apartado se Implementan las siguientes funciones de Programación Dinámica (PD).

2.1 Distancias entre cadenas y subcadena común más larga

Funciones implementadas:

- ***edit_distance(str1: str, str2: str)***

Calcula el número mínimo de operaciones (inserciones, eliminaciones o sustituciones) necesarias para transformar una cadena en otra. Utiliza una matriz *dp* para almacenar resultados parciales, donde cada celda representa la distancia de edición entre subcadenas. Se llena la matriz comparando los caracteres de las cadenas y eligiendo la operación más barata. El resultado final es el valor en la última celda, que indica la distancia de edición mínima entre las dos cadenas completas.

```
def edit_distance(str_1: str, str_2: str) -> int:
    """
        Calcula la distancia de edición entre dos cadenas
    """

    m = len(str_1)
    n = len(str_2)

    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str_1[i-1] == str_2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])

    return dp[m][n]
```

- ***max_subsequence_length(str_1: str, str_2: str)***

Calcula la longitud de la subsecuencia común más larga entre dos cadenas. Utiliza una matriz *dp* para almacenar los resultados intermedios, donde cada celda representa la longitud de la subsecuencia común más larga hasta ese punto en ambas cadenas. Si los caracteres de las cadenas coinciden, se incrementa el valor de la subsecuencia; si no coinciden, se toma el valor máximo entre las opciones anteriores. El resultado final es el valor en la última celda, que indica la longitud de la subsecuencia común más larga.

```
def max_subsequence_length(str_1: str, str_2: str) -> int:
    """
        Calcula la longitud de la subsecuencia común más larga
    """

    m = len(str_1)
    n = len(str_2)

    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif str_1[i-1] == str_2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]
```


- ***max_common_subsequence(str_1: str, str_2: str)***

Calcula la subsecuencia común más larga (LCS) entre dos cadenas. Utiliza PD para construir una tabla *dp* donde cada celda almacena la longitud de la LCS hasta ese punto. Después de llenar la tabla, se recorre hacia atrás desde la última celda para reconstruir la LCS. Si los caracteres de ambas cadenas coinciden, se agrega el carácter a la subsecuencia; si no, se sigue la dirección que ofrece la subsecuencia más larga. El resultado final es la subsecuencia común más larga.

```
def max_common_subsequence(str_1: str, str_2: str) -> str:
    """
        Calcula la subsecuencia común más larga
    """

    m = len(str_1)
    n = len(str_2)

    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif str_1[i-1] == str_2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    index = dp[m][n]

    lcs = [''] * (index+1)
    lcs[index] = ''

    i = m
    j = n
    while i > 0 and j > 0:

        if str_1[i-1] == str_2[j-1]:
            lcs[index-1] = str_1[i-1]
            i -= 1
            j -= 1
            index -= 1

        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    return ''.join(lcs)
```

2.2 Multiplicación de matrices

Función implementada:

- ***min_mult_matrix(l_dims: List[int])***

Calcula el número mínimo de multiplicaciones escalares necesarias para multiplicar una cadena de matrices utilizando programación dinámica. Toma una lista *l_dims* con las dimensiones de las matrices, y calcula los costos de multiplicar subcadenas de matrices de manera eficiente. La función utiliza una tabla *dp* para almacenar los costos mínimos y encuentra el punto de partición óptimo para cada subcadena de matrices. Finalmente, devuelve el costo mínimo de multiplicar todas las matrices en *dp[0][n-1]*.

```
def min_mult_matrix(l_dims: List[int]) -> int:
    n = len(l_dims) - 1 # Número de matrices a multiplicar

    if n <= 0:
        return 0

    dp = [[0] * n for _ in range(n)]

    # Calcular dp[i][j], que representa el costo mínimo de multiplicar
    matrices Ai...Aj
    for chain_length in range(2, n + 1):
        for i in range(n - chain_length + 1):
            j = i + chain_length - 1
            dp[i][j] = float('inf')

            for k in range(i, j):
                # Calcular el costo para dividir en (Ai...Ak) x (Ak+1...Aj)
                cost = dp[i][k] + dp[k + 1][j] + l_dims[i] * l_dims[k + 1] *
l_dims[j + 1]
                dp[i][j] = min(dp[i][j], cost)

    # El costo mínimo para multiplicar A1...An está en dp[0][n-1]
    return dp[0][n - 1]
```

Para comprobar las cuatro funciones implementadas, hemos hecho el siguiente código:

```
if __name__ == '__main__':
    # Pruebas para edit_distance
    print("Edit Distance:")
    print(edit_distance("kitten", "sitting"))
    print(edit_distance("same", "same"))
    print(edit_distance("", "hello"))
    print(edit_distance("world", ""))
    print(edit_distance("", ""))

    # Pruebas para max_subsequence_length
    print("\nMax Subsequence Length:")
    print(max_subsequence_length("ABCD", "ACDF"))
    print(max_subsequence_length("common", "common"))
    print(max_subsequence_length("abcd", "efgh"))
    print(max_subsequence_length("abcd", ""))
    print(max_subsequence_length("", "efgh"))

    # Pruebas para max_common_subsequence
    print("\nMax Common Subsequence:")
    print(max_common_subsequence("ABCD", "ACDF"))
    print(max_common_subsequence("common", "common"))
    print(f"Resultado 1: '{max_common_subsequence("abcd", "efgh")}'")
    print(f"Resultado 2: '{max_common_subsequence("abcd", "")}'")
    print(f"Resultado 3: '{max_common_subsequence("", "efgh")}'")
    print()

    # Pruebas básicas con matrices de diferentes tamaños
    test_cases = [
        [10, 20], # Solo una matriz (no requiere multiplicación)
        [10, 20, 30], # Dos matrices (10x20 y 20x30)
        [10, 20, 30, 40], # Tres matrices
        [5, 10, 3, 12, 5, 50], # Varias matrices
        [30, 35, 15, 5, 10, 20], # Más complejas
    ]

    print("Pruebas de min_mult_matrix:")
    for i, dims in enumerate(test_cases, start=1):
        print(f"Test case {i}: dimensiones {dims}")
        print(f" Mínimo número de productos: {min_mult_matrix(dims)}")
        print()

    # Caso límite: ninguna matriz
    empty_case = []
    print("Caso límite (sin matrices):", min_mult_matrix(empty_case))
    print()

    # Caso límite: una sola matriz
    single_case = [10]
```

```

print("Caso límite (una sola matriz):", min_mult_matrix(single_case))
print()

# Caso grande
large_case = [10] + [20] * 10 + [30]
print("Caso grande:", min_mult_matrix(large_case))

```

Este es el output que nos sale:

```

e506301@1-6-1-6:~/UnidadH/Practica3$ python3 graph_24.py
Edit Distance:
3
0
5
5
0

Max Subsequence Length:
3
6
0
0
0

Max Common Subsequence:
ACD
COMMON
Resultado 1: ''
Resultado 2: ''
Resultado 3: ''

Pruebas de min_mult_matrix:
Test case 1: dimensiones [10, 20]
  Mínimo número de productos: 0

Test case 2: dimensiones [10, 20, 30]
  Mínimo número de productos: 6000

Test case 3: dimensiones [10, 20, 30, 40]
  Mínimo número de productos: 18000

Test case 4: dimensiones [5, 10, 3, 12, 5, 50]
  Mínimo número de productos: 1655

Test case 5: dimensiones [30, 35, 15, 5, 10, 20]
  Mínimo número de productos: 11875

Caso límite (sin matrices): 0

Caso límite (una sola matriz): 0

Caso grande: 42000

```

En la ejecución de las pruebas, todas las funciones de programación dinámica han retornado los resultados esperados.

Pruebas para la función *edit_distance*:

- La distancia de edición entre las cadenas "kitten" y "sitting" es 3, lo que indica que tres operaciones son necesarias para transformar una cadena en la otra.
- La distancia de edición entre las cadenas "same" y "same" es 0, lo que indica que las cadenas son iguales.
- Cuando una de las cadenas está vacía, como en el caso de la cadena vacía y "hello", la distancia de edición es igual a la longitud de la cadena no vacía (5 en este caso).

Pruebas para la función *max_subsequence_length*:

- La longitud de la subsecuencia común más larga entre "ABCD" y "ACDF" es 3, correspondiendo a la subsecuencia "ACD".
- En el caso de las cadenas "common" y "common", la longitud de la subsecuencia común más larga es 6, ya que las cadenas son idénticas.
- Para las cadenas "abcd" y "efgh", no hay subsecuencia común, por lo que el resultado es 0.

Pruebas para la función *max_common_subsequence*:

- La subsecuencia común más larga entre "ABCD" y "ACDF" es "ACD".
- En el caso de las cadenas "common" y "common", la subsecuencia común es la misma palabra, "common".
- Cuando no hay subsecuencia común, como en las pruebas con "abcd" y "efgh" o con una cadena vacía, el resultado es una cadena vacía.

Pruebas para la función *min_mult_matrix*:

- Para las diferentes matrices de prueba, como [10, 20], [10, 20, 30], [5, 10, 3, 12, 5, 50] y [30, 35, 15, 5, 10, 20], la función calcula correctamente el número mínimo de multiplicaciones necesarias.
- En el caso límite, cuando no hay matrices o solo hay una matriz, el número de multiplicaciones es 0, como es esperado.

- La función también maneja correctamente casos más grandes, como $[10] + [20] * 10 + [30]$, mostrando el resultado esperado.

En resumen, todas las pruebas han pasado correctamente, y los resultados devueltos coinciden con lo esperado. Todo el código funciona según lo previsto.