Tecnológico de Monterrey

E2. Actividad Integradora 2

PRESENTA:

Manuel Antonio Morales González A01664652
Rodrigo Fernando Rivera Olea A01664716
Alexis Chávez Juárez A01657486

Fecha de entrega: 15 de noviembre del 2024

PROFESOR:

Leonardo Cañete

Análisis y diseño de algoritmos avanzados (Gpo 652)

En esta evidencia, se realizó la simulación de la incursión de una empresa en el mercado de servicios de Internet en ciudades que inician su proceso de digitalización. Tienen problemas se relacionan tanto con la infraestructura física necesaria para la conexión entre diferentes puntos de la ciudad, como con la optimización de los recursos humanos y la capacidad de la red para transmitir información de manera eficiente. En este documento, se abordará un problema de optimización de redes que surge en este contexto. El objetivo principal es diseñar un programa en Python que, a partir de datos sobre la distancia entre colonias y la capacidad de transmisión entre ellas, determine:

Para resolver este problema, se utilizarán algoritmos clásicos de teoría de grafos y optimización combinatoria. La elección de estos algoritmos se justifica por su eficiencia y aplicabilidad a problemas de redes.

Parte 1

Python

```python
# FUNCION UNO PARA LA PARTE UNO - Algoritmo de Prim
from typing import List, Tuple

def prim(n: int, distancias: List[List[int]], capacidad: List[List[int]]) ->
str:
    """Algoritmo de Prim para encontrar la mejor manera de cablear con fibra
óptica las colonias,
    construyendo un Minimum Spanning Tree (MST) basado en distancias
mínimas.

    Args:
        n: número de nodos (colonias)
        distancias: matriz de adyacencia con las distancias (pesos)
        capacidad: matriz de capacidades de flujo máximo entre colonias

    Returns:
        str: Formateado como las conexiones del MST
    """
    seleccionado: List[bool] = [False] * n
    min: List[float] = [float('inf')] * n
    padre: List[int] = [-1] * n
    min[0] = 0

    for _ in range(n):
        u: int = -1
        for i in range(n):
            if not seleccionado[i] and (u == -1 or min[i] < min[u]):
                u = i
        seleccionado[u] = True
        for v in range(n):
```

```
            if distancias[u][v] != 0 and not seleccionado[v] and
distancias[u][v] < min[v]:
                min[v] = distancias[u][v]
                padre[v] = u

    conexiones: List[Tuple[int, int]] = []
    for i in range(1, n):
        conexiones.append((padre[i], i))

    # Formatear la salida
    conexiones_str = ", ".join([f"({con[0]}, {con[1]})" for con in
conexiones])
    return conexiones_str
```

En la primera parte, el algoritmo de Prim se emplea para construir un árbol de expansión mínima (MST) a partir de un grafo. Este proceso selecciona las conexiones entre nodos que minimizan el costo total, utilizando una matriz de adyacencia como representación del grafo. Durante su ejecución, el algoritmo identifica el nodo con el costo más bajo que aún no ha sido seleccionado y actualiza las conexiones del árbol de forma iterativa. Finalmente, genera un conjunto de pares de nodos que forman las conexiones del MST, y despliega las conexiones más óptimas entre ciudades. **Este método tiene una complejidad de $O(n^2)$.**

Parte 2

Python

```
# FUNCION DOS PARA LA PARTE DOS - Algoritmo del Viajero (TSP)
def tsp(n: int, distancias: List[List[int]]) -> str:
    """Algoritmo de Fuerza Bruta para resolver el Problema del Viajero
(TSP), encontrando
    la ruta más corta posible que visita cada colonia exactamente una vez y
regresa al origen.

    Args:
        n: número de nodos (colonias)
        distancias: matriz de adyacencia con las distancias (pesos)

    Returns:
        str: Formateado como la ruta más corta
    """

    def calcular_costo(ruta: List[int]) -> int:
        """Calcula el costo total de una ruta
```

```
        Args:
            ruta: Secuencia de nodos en la ruta

        Returns:
            Costo total de la ruta
        """
        costo: int = 0
        for i in range(n - 1):
            costo += distancias[ruta[i]][ruta[i + 1]]
        costo += distancias[ruta[-1]][ruta[0]]  # Volvemos al punto de
inicio
        return costo

    def permutaciones(ruta_actual: List[int], nodos_restantes: List[int]) ->
List[List[int]]:
        """Genera todas las permutaciones posibles de rutas

        Args:
            ruta_actual: Ruta actual en construcción
            nodos_restantes: Nodos restantes por visitar

        Returns:
            Lista de todas las rutas posibles
        """
        if not nodos_restantes:
            return [ruta_actual]

        rutas: List[List[int]] = []
        for i in range(len(nodos_restantes)):
            siguiente_nodo: int = nodos_restantes[i]
            nuevas_rutas: List[List[int]] = permutaciones(ruta_actual +
[siguiente_nodo], nodos_restantes[:i] + nodos_restantes[i + 1:])
            rutas.extend(nuevas_rutas)
        return rutas

    # Generar todas las posibles rutas comenzando desde el nodo 0
    nodos: List[int] = list(range(1, n))
    rutas: List[List[int]] = permutaciones([0], nodos)
    mejor_ruta: List[int] = []
    menor_costo: int = float('inf')

    # Encontrar la ruta óptima
    for ruta in rutas:
        costo: int = calcular_costo(ruta)
        if costo < menor_costo:
            menor_costo = costo
            mejor_ruta = ruta
```

```
    conexiones: List[Tuple[int, int]] = []
    for i in range(len(mejor_ruta)):
        u: int = mejor_ruta[i]
        v: int = mejor_ruta[(i + 1) % n]
        conexiones.append((u, v))

    # Formatear la salida
    conexiones_str = ", ".join([f"({u}, {v})" for u, v in conexiones])
    return conexiones_str
```

La segunda parte aborda el problema del viajero (TSP) utilizando un enfoque de fuerza bruta. Se usó TSP dado que tienen que visitar una sola vez cada ciudad para poder dejar información de la red. Este método genera todas las rutas posibles que recorren los nodos del grafo exactamente una vez y calcula el costo total de cada una, asegurando siempre el regreso al punto de partida. La ruta con el costo más bajo es seleccionada como resultado e impresa en formato de número de ciudad en parejas. **La complejidad de este enfoque es O(n!).** Además, el enfoque de fuerza bruta permite garantizar resultados precisos, lo que lo hace útil en escenarios de validación, experimentación o aprendizaje, donde la exactitud es prioritaria sobre la eficiencia computacional.

Parte 3

Python

```
# FUNCION TRES PARA LA PARTE TRES - Algoritmo de Flujo Máximo (Edmonds-Karp)
def bfs(capacidad: List[List[int]], flujo: List[List[int]], fuente: int,
lleno: int, padre: List[int]) -> bool:
    """Busca un camino aumentante en el grafo de flujo

    Args:
        capacidad: Matriz de capacidades de flujo
        flujo: Matriz de flujo actual
        fuente: Nodo origen
        lleno: Nodo destino
        padre: Lista de padres para reconstruir el camino

    Returns:
        Verdadero si se encontró un camino aumentante
    """
    visitado: List[bool] = [False] * len(capacidad)
    queue: List[int] = [fuente]
    visitado[fuente] = True

    while queue:
        u: int = queue.pop(0)

        for v in range(len(capacidad)):
```

```python
            if not visitado[v] and capacidad[u][v] - flujo[u][v] > 0:
                queue.append(v)
                visitado[v] = True
                padre[v] = u
                if v == lleno:
                    return True
    return False

def max_flujo(n: int, capacidad: List[List[int]], fuente: int, lleno: int)
-> str:
    """Calcula el flujo máximo entre dos nodos usando el algoritmo
Edmonds-Karp

    Args:
        n: Número de nodos
        capacidad: Matriz de capacidades de flujo
        fuente: Nodo origen
        lleno: Nodo destino

    Returns:
        str: Formateado como el flujo máximo
    """
    flujo: List[List[int]] = [[0] * n for _ in range(n)]
    padre: List[int] = [-1] * n
    max_valor_flujo: int = 0

    while bfs(capacidad, flujo, fuente, lleno, padre):
        flujo_camino: int = float('Inf')
        v: int = lleno

        while v != fuente:
            u: int = padre[v]
            flujo_camino = min(flujo_camino, capacidad[u][v] - flujo[u][v])
            v = padre[v]

        v = lleno
        while v != fuente:
            u = padre[v]
            flujo[u][v] += flujo_camino
            flujo[v][u] -= flujo_camino
            v = padre[v]

        max_valor_flujo += flujo_camino

    return str(max_valor_flujo)
```

En la tercera parte, el algoritmo de Edmonds-Karp se utiliza para calcular el flujo máximo en un grafo dirigido. Este método emplea búsquedas en anchura (BFS) para encontrar caminos aumentantes entre un nodo fuente y un nodo destino, ajustando los flujos iterativamente según las capacidades disponibles en las aristas. Cada iteración aumenta el flujo total hasta que ya no se pueden encontrar caminos que aumenten la capacidad. **La complejidad de este algoritmo es O(VE^2)**.

Main

Python

```python
# FUNCION PRINCIPAL
def main() -> None:
    """Función principal que lee los datos de entrada y ejecuta los
algoritmos correspondientes."""
    with open("input.txt", "r") as file:
        n: int = int(file.readline().strip())

        distancias: List[List[int]] = []
        for _ in range(n):
            row: List[int] = list(map(int, file.readline().strip().split()))
            distancias.append(row)

        capacidad: List[List[int]] = []
        for _ in range(n):
            row: List[int] = list(map(int, file.readline().strip().split()))
            capacidad.append(row)

    # Función Parte 1 - Prim
    print(prim(n, distancias, capacidad))

    # Función Parte 2 - TSP
    print(tsp(n, distancias))

    # Función Parte 3 - Flujo máximo
    print(max_flujo(n, capacidad, 0, n - 1))

if __name__ == "__main__":
    main()
```

El main integra las soluciones al leer los datos de entrada desde un archivo y ejecutar cada algoritmo con base en las matrices de distancias y capacidades de flujo proporcionadas. Los resultados son presentados de manera organizada, mostrando el árbol de expansión mínima, la ruta óptima del viajero y el flujo máximo entre los nodos fuente y destino.

Conclusión grupal

El código desarrollado aborda tres problemas fundamentales en el análisis de grafos, demostrando la versatilidad y utilidad de estos algoritmos en distintos contextos. El uso del algoritmo de Prim permite optimizar la construcción de redes minimizando costos, mientras que el enfoque de fuerza bruta para el problema del viajero destaca por su precisión al garantizar la ruta óptima, a pesar de sus limitaciones en escalabilidad. Por otro lado, el algoritmo de Edmonds-Karp muestra cómo se puede maximizar la capacidad de flujo en un grafo dirigido de manera eficiente. Estos algoritmos, muchos vistos en clase, ayudaron a la correcta implementación de la solución para las ciudades que se desean conectar. La integración de estos métodos en una estructura modular y funcional resalta la importancia de comprender y aplicar técnicas específicas según las necesidades del problema; el tipado, comentarios y el docstring formaron parte de las buenas prácticas aprendidas a lo largo del curso. Este enfoque no solo ofrece soluciones claras y precisas, sino que también sirve como base sólida para explorar optimizaciones y extensiones en el análisis de grafos en escenarios más complejos.