

ACTIVIDAD INTEGRADORA 2

Frida Bailleres González	AO17O8633
Marco Randu Retana Vargas	AO17O9521
Sebastian Armando Flores Lemus	AO17O9229



AGENDA

i. problemática

ii. soluciones

iii. conclusiones



i. problemática

Objetivo de la Actividad:

- Mejorar la infraestructura de las empresas proveedoras de servicios de Internet (ISP) en el contexto de la pandemia de COVID-19.
- El objetivo es optimizar la red de ISPs para manejar eficientemente el aumento en la transmisión de datos y asegurar una conectividad robusta y confiable para actividades esenciales.



i. problemática

Recursos proporcionados:

- Contexto de la actividad.
- Descripción de los problemas a solucionar (4 partes).
- Archivos de entrada y salida.



ii. soluciones

parte 1

Algoritmo: Floyd

Que es?

- Utilizado para encontrar las distancias más cortas entre todos los pares de nodos en un grafo. Es efectivo tanto para grafos dirigidos como no dirigidos y puede manejar pesos negativos en las aristas.



ii. soluciones

parte 1

Algoritmo: Floyd

Criterios de selección de algoritmo:

- Capacidad única para calcular las distancias más cortas entre todos los pares de nodos en un grafo.
 - Necesita conocer no solo la distancia más corta entre un par específico de nodos, sino entre cada par posible dentro del grafo.



ii. soluciones

parte 1

Complejidad: $\mathcal{O}(n^3)$

Esta complejidad viene de los tres bucles anidados que recorren todos los nodos del grafo



ii. soluciones

parte 1

```
def floyd_marshall(matriz_distancias):
    N = len(matriz_distancias)
    distancias_minimas = [[float('inf')] for _ in range(N)] for _ in range(N)]

    for i in range(N):
        for j in range(N):
            if i == j:
                distancias_minimas[i][j] = 0
            elif matriz_distancias[i][j] != 0:
                distancias_minimas[i][j] = matriz_distancias[i][j]

    for k in range(N):
        for i in range(N):
            for j in range(N):
                if distancias_minimas[i][k] + distancias_minimas[k][j] < distancias_minimas[i][j]:
                    distancias_minimas[i][j] = distancias_minimas[i][k] + distancias_minimas[k][j]

    return distancias_minimas
```



ii. soluciones

parte 2

Algoritmo: Problema del viajante (TSP)

Que es?

- Consiste en encontrar el camino más corto que recorre n ciudades o puntos de interés y vuelve al punto de partida, pasando por cada ciudad exactamente una vez. Este algoritmo no solo nos ayuda a encontrar la lista del camino más corto, si no que también nos da el peso total de la ruta más corta.



ii. soluciones

parte 2



Algoritmo: Problema del viajante (Naive)

Criterios de selección de algoritmo:

- Este algoritmo es uno de los más estudiados en la computación, es usado en múltiples industrias como astronomía, logística y planeación. Este algoritmo es la base de la optimización de entregas de productos para miles de empresas. Debido a que es muy estudiado hay diferentes interpretaciones que varían en complejidad.
- La solución 'Naive' es la mas sencilla de implementar y tiene la misma complejidad que la solución usando DP, y una complejidad muy similar a la solución usando 'BackTracking'.

ii. soluciones

parte 2

Complejidad: $O(n!)$

Este algoritmo tiene una complejidad $O(n!)$, donde "n" es la cantidad de vértices en el grafo. Llegamos a esta complejidad ya que generamos todas las permutación de las ciudades. Este problema es NP-hard y es uno de los más complejos y más estudiados en la computación. No existe una solución conocida en tiempo polinomial.



ii. soluciones

parte 2



```
def Tsp(matriz_distancias, nodo_inicial, num_nodos):
    def generar_permutaciones(nodos):
        if not nodos:
            return []
        return [[nodo] + p for i, nodo in enumerate(nodos) for p in generar_permutaciones(nodos[:i] + nodos[i+1:])]

    vertices = list(range(num_nodos))
    vertices.remove(nodo_inicial)

    peso_minimo = maxsize
    lista_ordenada = None

    for perm in generar_permutaciones(vertices):
        peso_actual = 0
        k = nodo_inicial
        orden_recorrido = [k] + perm
        for j in perm:
            peso_actual += matriz_distancias[k][j]
            k = j
        peso_actual += matriz_distancias[k][nodo_inicial]
        orden_recorrido.append(nodo_inicial)

        if peso_actual < peso_minimo:
            peso_minimo = peso_actual
            lista_ordenada = orden_recorrido

    return lista_ordenada, peso_minimo
```

ii. soluciones

parte 3

Algoritmo: Ford Fulkerson

Que es?

- El algoritmo de Ford-Fulkerson es un método utilizado para encontrar el flujo máximo en una red de flujo dirigido
- El algoritmo trabaja iterativamente buscando caminos de aumento en la red hasta que no se puedan encontrar más caminos. Un camino de aumento es un camino desde el origen hasta el destino a lo largo del cual es posible aumentar el flujo.



ii. soluciones

parte 3



Algoritmo: Ford Fulkerson

Criterios de selección de algoritmo:

- Aplicabilidad a determinar flujo máximo entre dos puntos: Al buscar la capacidad máxima de transmisión de datos entre dos ubicaciones específicas (como las colonias i y j), este algoritmo resulta apropiado, ya que se enfoca en encontrar el flujo máximo entre un origen y un destino en una red.
- Versatilidad en el contexto de redes de datos: Este algoritmo es versátil y aplicable a diferentes tipos de redes, incluyendo aquellas de transmisión de datos.

ii. soluciones

parte 3



Complejidad: $O(V + E)$

La complejidad de tiempo de un algoritmo BFS es $O(V + E)$, donde V es el número de nodos (vértices) y E es el número de aristas.

ii. soluciones

parte 3



```
def ford_fulkerson(matriz_flujos, nodo_inicial, nodo_final):
    N = len(matriz_flujos)
    matriz_flujos_residuales = [list(row) for row in matriz_flujos]
    flujo_maximo = 0

    while True:
        padres = bfs(matriz_flujos_residuales, nodo_inicial, nodo_final)
        if padres is None: # No hay más caminos de aumento
            break

        # Encontrar el flujo mínimo en el camino encontrado
        flujo_camino = float('inf')
        v = nodo_final
        while v != nodo_inicial:
            u = padres[v]
            flujo_camino = min(flujo_camino, matriz_flujos_residuales[u][v])
            v = u

        # Actualizar el flujo máximo y el grafo residual
        flujo_maximo += flujo_camino
        v = nodo_final
        while v != nodo_inicial:
            u = padres[v]
            matriz_flujos_residuales[u][v] -= flujo_camino
            matriz_flujos_residuales[v][u] += flujo_camino
            v = u

    return flujo_maximo
```

```
def bfs(matriz_flujos_residuales, nodo_inicial, nodo_final):
    N = len(matriz_flujos_residuales)
    visitados = [False] * N
    cola = [nodo_inicial]
    visitados[nodo_inicial] = True
    padres = [-1] * N

    while cola:
        u = cola.pop(0)
        for v, capacidad in enumerate(matriz_flujos_residuales[u]):
            if capacidad > 0 and not visitados[v]:
                cola.append(v)
                padres[v] = u
                visitados[v] = True
                if v == nodo_final:
                    return padres

    return None
```

ii. soluciones

parte 4

Algoritmo: Dijkstra

Que es?

- Encuentra el camino más corto desde un nodo de origen a todos los demás nodos en un grafo ponderado. Es especialmente útil en grafos donde los pesos de las aristas representan distancias o costos.



ii. soluciones

parte 4

Algoritmo: Dijkstra

Criterios de selección de algoritmo:

- Adecuado para encontrar el camino más corto desde un único nodo de origen (la nueva contratación) a todos los demás nodos (las centrales) en un grafo. Esta característica lo hace ideal para determinar cuál es la central más cercana geográficamente a la nueva contratación



ii. soluciones

parte 4

Complejidad: $O(V \log V + E \log V)$

- Donde V es el número de vértices o nodos E es el número de aristas en el grafo



ii. soluciones

parte 4

```
def dijkstra(matriz_distancias, nodo_inicio):
    N = len(matriz_distancias)
    distancias = [float('inf')] * N
    distancias[nodo_inicio] = 0
    cola_prioridad = [(0, nodo_inicio)]

    while cola_prioridad:
        distancia_actual, nodo_actual = heapq.heappop(cola_prioridad)
        if distancia_actual > distancias[nodo_actual]:
            continue

        for vecino, peso in enumerate(matriz_distancias[nodo_actual]):
            if peso > 0:
                distancia = distancia_actual + peso
                if distancia < distancias[vecino]:
                    distancias[vecino] = distancia
                    heapq.heappush(cola_prioridad, (distancia, vecino))

    return distancias
```

```
def encontrar_central_mas_cercana_con_dijkstra(matriz_distancias, nuevo_punto_idx):
    distancias = dijkstra(matriz_distancias, nuevo_punto_idx)
    return distancias
```

```
def ampliar_grafo_con_nuevo_punto(N, matriz_distancias, centrales, nuevo_punto):
    N_ampliado = N + 1
    for i in range(len(centrales)):
        distancia_a_nuevo_punto = distance.euclidean(centrales[i], nuevo_punto)
        matriz_distancias[i].append(distancia_a_nuevo_punto)
    fila_nuevo_punto = [distance.euclidean(central, nuevo_punto) for central in centrales] + [0]
    matriz_distancias.append(fila_nuevo_punto)
    return N_ampliado, matriz_distancias
```



ii. soluciones

parte 4

```
def dijkstra(matriz_distancias, nodo_inicio):
    N = len(matriz_distancias)
    distancias = [float('inf')] * N
    distancias[nodo_inicio] = 0
    cola_prioridad = [(0, nodo_inicio)]

    while cola_prioridad:
        distancia_actual, nodo_actual = heapq.heappop(cola_prioridad)
        if distancia_actual > distancias[nodo_actual]:
            continue

        for vecino, peso in enumerate(matriz_distancias[nodo_actual]):
            if peso > 0:
                distancia = distancia_actual + peso
                if distancia < distancias[vecino]:
                    distancias[vecino] = distancia
                    heapq.heappush(cola_prioridad, (distancia, vecino))

    return distancias
```

```
def encontrar_central_mas_cercana_con_dijkstra(matriz_distancias, nuevo_punto_idx):
    distancias = dijkstra(matriz_distancias, nuevo_punto_idx)
    return distancias
```

```
def ampliar_grafo_con_nuevo_punto(N, matriz_distancias, centrales, nuevo_punto):
    N_ampliado = N + 1
    for i in range(len(centrales)):
        distancia_a_nuevo_punto = distance.euclidean(centrales[i], nuevo_punto)
        matriz_distancias[i].append(distancia_a_nuevo_punto)
    fila_nuevo_punto = [distance.euclidean(central, nuevo_punto) for central in centrales] + [0]
    matriz_distancias.append(fila_nuevo_punto)
    return N_ampliado, matriz_distancias
```



iii. Conclusiones

- Los algoritmos como Floyd-Warshall, Dijkstra, Ford Fulkerson y el Problema del Viajante son esenciales para optimizar las redes de telecomunicaciones.
- Cada uno soluciona aspectos específicos como la planificación de rutas de fibra óptica y la capacidad de transmisión de datos.
- Estas herramientas mejoran la infraestructura y conectividad, apoyando la toma de decisiones estratégicas en el sector de internet y telecomunicaciones.



Salida de programa

The screenshot shows a dark-themed code editor interface, likely Visual Studio Code, with a circuit board background. The main area displays a Python file named `main.py`. The code implements several graph and network algorithms:

```
# ===== FUNCIÓN PRINCIPAL =====
# Función principal que ejecuta el programa
#
def main():
    # Lista de archivos de entrada
    archivos_entrada = [
        'Equipo_08_Entrada_1.txt',
        'Equipo_08_Entrada_2.txt',
        'Equipo_08_Entrada_3.txt'
    ]

    for nombre_archivo_entrada in archivos_entrada:
        # Leer los datos de entrada
        N, matriz_distancias, matriz_flujos, centrales, nuevo_punto = leer_datos_de_archivo(nombre_archivo_entrada)

        # Ampliar el grafo con la nueva central
        N_ampliado, matriz_distancias_ampliada = ampliar_grafo_con_nuevo_punto(N, matriz_distancias, centrales, nuevo_punto)

        # Índice de la nueva central
        nuevo_punto_idx = N_ampliado - 1

        # 1. Forma óptima de cablear las colonias con fibra (lista de arcos de la forma (A,B)).
        distancias_minimas = floyd_marshall(matriz_distancias)

        # 2. Ruta a seguir por el personal que reparte correspondencia (TSP)
        # Nodo inicial para recorrido
        nodo_inicial = 0
        resultado, peso = Tsp(matriz_distancias, nodo_inicial, N)

        # 3. Valor de flujo máximo de información (Ford-Fulkerson o Edmonds-Karp)
        flujo_maximo = ford_fulkerson(matriz_flujos, 0, N-1)

        # 4. La central más cercana a la nueva contratación
        distancias = encontrar_central_mas_cercana_con_dijkstra(matriz_distancias_ampliada, nuevo_punto_idx)
        central_cercana_idx = nn.amin(distancias[:,-1])
```

The interface includes a sidebar with project files like `Equipo_XX_team.txt`, `Equipo_08_Entrada_1.txt`, etc., and a terminal window at the bottom left.