Proyecto Final de DAA

Movilidad Urbana Multimodal

Marian S. Álvarez Suri - C412
Carlos A. Bresó Sotto - C412

1. Planteamiento del problema

Se quiere extender el uso de una aplicación de transporte urbano. Esta aplicación no solo muestra las rutas de autobuses, metros y otros medios de transporte, sino que también sugiere al usuario la mejor combinación de transportes (a pie, metro, autobús, taxi callejero, ...) que en el menor tiempo posible lo lleven a su destino según el presupuesto y preferencias del viajante. Para lograr esto, se deben considerar las siguientes variables adicionales:

- **Distancias y Velocidades:** Velocidades promedio para cada medio de transporte, que pueden variar según la hora del día, las condiciones de tráfico y del clima.
- **Horarios y Frecuencias:** Horarios de salida y llegada de autobuses, metros y otros transportes.
- **Preferencias en términos de costos vs. tiempo:** algunos usuarios pueden preferir opciones más económicas, aunque tomen más tiempo.
- **Preferencias sobre la comodidad:** preferencia por menos transbordos, preferencias por transporte privado, etc...
- **Cantidad de Personas Viajando Juntas:** Disponibilidad de espacio en vehículos para grupos grandes. Preferencias sobre mantenerse juntos durante todo el viaje.
- **Puntos de Interés y Paradas:** Paradas intermedias que el usuario quiera incluir en su ruta.
- **Costo monetario:** Posibilidad de que exista un presupuesto no superable por la ruta recomendada. Tener en cuenta Tarifas de cada medio de transporte y tarifas dinámicas en taxis callejeros o vehículos privados (uber).

2. Problema de distancias y velocidades

Vamos a resolver el problema de llegar de un punto de la ciudad a otro en el menor **tiempo** posible. Para ello, vamos a establecer las velocidades a las que puede circular cada medio de transporte:

- **A pie**: 5 km/h independientemente del recorrido.
- **Autobús**: 50 km/h
- **Taxi**:
  - Carreteras principales: 100 km/h
  - Avenidas: 80 km/h

    - Ciudad: 50 km/h
    - Calles secundarias: 20 km/h
- **Metro**: 90 km/h

2.1. Modelación del problema

Vamos a modelar el grafo de la ciudad. Para cada tipo de transporte vamos a tener un grafo diferente, donde el conjunto de vértices V corresponde a los puntos donde se puede comenzar un recorrido. En el caso de los metros y autobuses, los recorridos empiezan en las estaciones y paradas establecidas. Crearemos un nodo en la esquina más cercana a cada estación o parada. Para el resto de medios de transporte, consideraremos las esquinas de las calles como vértices.
Para insertar aristas en el grafo, diremos que existe una arista que va desde el vértice u hacia el v si y solo si existe una calle que conecta directamente a u con v en esa dirección. El grafo de los taxis lo tendrá en cuenta para determinar si una calle tiene el sentido prohibido en esa dirección, pero en el grafo de los recorridos a pie, todas las aristas existirán en ambos sentidos. En el caso de los buses y metros, la arista aparecerá si existe un recorrido preestablecido entre una estación o parada y la siguiente. Llamaremos E a este conjunto de aristas.

2.2. Transbordos

Para representar los cambios de medio de transporte, vamos a conectar todos los grafos de la ciudad de la siguiente forma:
Si existe el nodo u en el grafo del medio de transporte x y también en el del medio y, creamos una arista entre u_x y u_y. Mediante esta arista, si una persona se encontraba haciendo un recorrido, por ejemplo, a pie, y llega a un punto donde existe una estación de metro, puede decidir si tomar el metro o seguir su camino a pie.
El grafo resultante de añadir estas aristas lo llamaremos grafo G_t.

2.3. Costo de las aristas

El costo de cada arista en G_t representa el tiempo que se demora en llegar de un nodo a otro tomando dicha arista.
Todas las aristas que pertenezcan al subgrafo de un mismo medio de transporte, calculan su costo c como:

$$c = \frac{distancia}{velocidad}$$

En el caso de los taxis, que pueden tener diferentes velocidades, la velocidad se ajusta según el tipo de calle que represente la arista.

Las aristas de transbordo, que conectan un medio de transporte con otro, tendrán un costo especial.

- Hacia caminar: 0
- Hacia taxi: 5 minutos
- Hacia metro: hora de llegada del metro más cercano - hora actual
- Hacia bus: hora de llegada del bus más cercano - hora actual

Las aristas de transbordo desde un medio de transporte cualquiera hacia el metro o el bus tendrán el costo de la hora de llegada - la hora actual, donde representaremos las horas en minutos del día transcurridos. Es decir, si un metro llega a las 3:40 am y son las 3:20 am, el costo de tomar ese metro se calculará como:

$$3:40 - 3:20 = (3 * 60 + 40) - (3 * 60 + 20) = 220 - 200 = 20$$

2.4. Solución

Para solucionar el problema de hallar el camino que tome el menor tiempo en llevar de un punto de la ciudad a su destino, aplicaremos el algoritmo de Dijkstra sobre el grafo G_t.

El grafo que recibirá el algoritmo de Dijkstra es G_t, el nodo de partida será s, que es el punto donde se encuentra la persona que quiere llegar a su destino, y existirá un parámetro adicional t, que representa el momento del día en que inicia el viaje.

El valor de las aristas de transbordo hacia un metro o un bus solo se conoce durante el cómputo del algoritmo de Dijkstra, el cual depende del tiempo que haya tomado llegar hasta la estación y del horario de los transportes.
Dado que tenemos un total de 2400 minutos en el día, cuando calculemos las aristas de transbordo estaremos considerando la hora actual módulo 2400:

$$T_{actual}[v] = t + dist(s, v)\ mod\ 2400$$

El costo de las aristas de transbordo desde un nodo v hacia una estación de bus o metro u será el tiempo de espera mínimo en la parada antes de la llegada del próximo bus o metro. O sea:

$$C[v][u] = P_u(T_{actual}[v]) - T_{actual}[v]$$

Donde P_u es la hora de la primera llegada del siguiente metro o bus.
Esto será necesario para poder calcular el peso de las aristas que representan el tiempo de espera en la estación o parada antes de la llegada del próximo transporte, pues tienen horarios de llegada fijos.

2.5. Análisis de la correctitud

Por el algoritmo de Dijkstra, se garantiza que las llegadas hasta paradas de bus o estaciones de tren se realicen en el menor tiempo posible. Además, no existen valores negativos en el grafo dado que el tiempo siempre es positivo. El costo de las aristas de transbordo se calcula de esa manera porque es imposible coger un bus que ya haya pasado y no espera de más porque hace el viaje en el primer bus que llegue.

3. Problema de comodidad y número de personas viajando juntas

Estaremos resolviendo dos problemas cuya solución veremos que consiste en realizar la misma transformación en el grafo. Dichos problemas son restringir el tipo de vehículos que se desea y lograr que un número de personas determinado viajen juntos.
Para ello, vamos a definir la cantidad máxima de pasajeros que se pueden transportar:

- A pie: n
- En taxi: 4
- En bus: 70
- En metro: 500

3.1. Solución

Para eliminar los medios de transporte que el usuario no quiera tomar, durante la creación del grafo G_t no incluiremos los nodos de los subgrafos que corresponden a los medios de transporte que el usuario no desea usar. Por ejemplo, si una persona desea realizar su recorrido solo empleando taxis y el metro, al construir G_t no incluiremos ningún nodo que represente paradas de bus ni recorridos a pie.
Dada esta construcción, es posible que, al buscar el camino que el ususario debe elegir, no encontremos ninguno pues, por ejemplo, no desea usar taxis ni ir a pie y no existen paradas de bus ni de metro que conecten los dos puntos sin caminar.

Para resolver el problema de lograr que un número de personas determinado viajen juntos, la solución es equivalente al problema anterior: en este caso, dejaremos de incluir los vértices de aquellos medios de transporte cuya cantidad máxima de personas que pueden viajar juntas es menor que el número de personas viajando.

4. Problema de menor costo vs. tiempo

Veamos que resolver el problema de llegar de un punto a otro con el menor costo posible, se resume a ir caminando en todos los tramos de trayecto, puesto que el costo de caminar siempre será cero. Sin embargo, también es el de mayor tiempo, así que lo que se quiere es llegar a los lugares en un tiempo razonable y con bajo costo.

Vamos a suponer que se dispone de un presupuesto B para gastar en el recorrido. Tenemos que encontrar el camino que le permita llegar a su destino en el menor tiempo posible, pero sin exceder dicho presupuesto. Vamos a demostrar que este problema pertenece a la clase de problemas NP-Hard.

4.1. Reducción del problema de la mochila al BCSPP

Llamaremos a nuestro problema **Budget-Constrained Shortest Path Problem** (BCSPP).
Probaremos que el BCSPP es **NP-Hard** mediante una reducción polinomial desde el problema de la mochila, que sabemos de antemano que es NP-Hard. El problema de la mochila se define de la siguiente manera:

Dado un conjunto de n ítems, cada uno con un **valor** v_i y un **peso** w_i y una **capacidad máxima** W, el objetivo es seleccionar un subconjunto de ítems que maximice el valor total sin exceder la capacidad W.

Formalmente, queremos:

$$\text{Maximizar } \sum_{i=1}^{n} v_i x_i \quad \text{sujeto a } \sum_{i=1}^{n} w_i x_i \leq W,$$

donde $x_i \in \{0, 1\}$ indica si el ítem i está incluido en la mochila.

Para demostrar que el BCSPP es NP-hard, construiremos una instancia del BCSPP que corresponda a una instancia del problema de la mochila. La idea es modelar la selección de ítems como la elección de caminos en un grafo, donde cada arista representa la inclusión de un ítem.

4.2. Construcción del grafo

Nuestro problema recibe una lista de n tuplas que tienen un peso w_i y un valor v_i no negativos que representan los objetos de la mochila. Además, recibe un peso máximo W que no puede excederse.

1. **Nodos**: Creamos $n+2$ nodos: $u_0, u_1, u_2, \ldots, u_{n+1}$, donde el nodo u_0 es el nodo de inicio, y u_{n+1} es el nodo de destino. Los nodos u_i, $i \in [1, n]$ representan los objetos de la mochila.
2. **Aristas**:
   - Para cada par de nodos u_i y u_j con $j > i$, creamos una arista desde u_i hasta u_j.
   - Asignamos a la arista (u_i, u_j) un **costo** igual a w_j y un **tiempo** igual a $-v_j$.
   - El costo y tiempo de todas las aristas (u_i, u_{n+1}) es de 0.
3. **Presupuesto**: Establecer el presupuesto $B = W$.

4.3. Correspondencia entre las soluciones

Cada camino del grafo con t mínimo y costo menor que B es una solución óptima al problema de la mochila porque:

- Si una arista (u_i, u_j) está en el camino, significa que se ha seleccionado el ítem u_j, $j \neq u_{n+1}$.
- El tiempo total del camino es la suma de los tiempos de las aristas, que es $-\sum_{k \in S} v_k$, donde S es el conjunto de ítems seleccionados. Minimizar este tiempo equivale a maximizar $\sum_{k \in V} v_k$, donde V es el valor de los elementos seleccionados.
- El costo total del camino es la suma de los pesos de las aristas en él, que no debe exceder $B = W$.

Por lo tanto, encontrar un camino en el grafo que cumpla con la restricción de presupuesto y minimice el tiempo es equivalente a resolver el problema de la mochila.

Hemos demostrado que cualquier instancia del problema de la mochila puede ser transformada en una instancia del BCSPP en tiempo polinomial, y su solución se convierte a una solución al problema de la mochila también en tiempo polinomial. Dado que el problema de la mochila es NP-Hard, esto implica que el BCSPP también es NP-Hard.

4.4. Solución del BCSPP

Para resolver este problema, como ya sabemos que pertenece a la clase NP-Hard, podemos calcular todos los caminos que existen desde el nodo de partida s hasta el nodo de destino d y seleccionar aquel camino que tome el mínimo tiempo y cumpla la restricción de necesitar un presupuesto menor que el presupuesto con que se cuenta B.

4.5. Propuesta de pseudocódigo

```
def encontrar_camino_minimo_tiempo(grafo, s, d, B):
    caminos = []  # Lista para almacenar todos los caminos posibles

    def dfs(nodo_actual, tiempo_acumulado, costo_acumulado, visitados):
        if nodo_actual == d:
            caminos.append((tiempo_acumulado, costo_acumulado))
            return

        for vecino, (tiempo_uv, costo_uv) in grafo[nodo_actual].items():
            if vecino not in visitados:
                nuevo_tiempo = tiempo_acumulado + tiempo_uv
                nuevo_costo = costo_acumulado + costo_uv
                if nuevo_costo <= B:  # Solo explorar si no se supera el
    presupuesto
                    visitados.add(vecino)
                    dfs(vecino, nuevo_tiempo, nuevo_costo, visitados)
                    visitados.remove(vecino)

    dfs(s, 0, 0, set([s]))

    # Filtrar caminos que cumplen con el presupuesto
    caminos_validos = [camino for camino in caminos if camino[1] <= B]

    if not caminos_validos:
        return None

    # Seleccionar el camino con el menor tiempo
    camino_optimo = min(caminos_validos, key=lambda x: x[0])
    return camino_optimo
```

4.6. Análisis de complejidad

Como el algoritmo explora todos los caminos posibles desde el nodo de partida s hasta el nodo de destino d, tiene una complejidad exponencial en el peor caso, en el cual el número de caminos posibles es del orden de $O((V-1)!)$, donde V es el número de nodos.

5. Problema de paradas y puntos intermedios

El problema que resolveremos a continuación consiste en encontrar el camino más corto desde un punto de salida s a un punto de destino d, pasando por varios puntos de parada.

5.1. Modelación del problema

Emplearemos el grafo G_t construido anteriormente, en el cual tenemos en cada arista el tiempo que toma llegar de un nodo a otro o, en el caso de las aristas de transbordo, cambiar de medio de transporte.

5.2. Solución

Vamos a realizar una reducción del grafo G_t a un subgrafo, en el cual intentaremos resolver el problema directamente.
Supongamos que se tienen k puntos de parada $p_1, p_2, \ldots, p_k$, además de s y d. Se utiliza el algoritmo de Dijkstra para calcular los caminos más cortos desde s a cada p_i, entre cada par de puntos de parada p_i y p_j, y desde cada p_i a d.

Una vez tengamos las distancias entre esos nodos, crearemos un nuevo grafo G_s donde los nodos son $s, p_1, p_2, \ldots, p_k, d$, y las aristas tienen el coste del camino más corto entre ellos. En este nuevo grafo, debemos buscar el camino más corto desde s

a d que pase por todos los puntos de parada. Este problema se reduce al **Problema del Viajante (TSP)**, que es NP-Hard, el cual resolveremos usando el algoritmo de Held-Karp.

5.3. Propuesta de pseudocódigo

```
def encontrar_camino_mas_corto(G_t, s, d, puntos_parada):
    G_s, cost_matrix = reducir_grafo(G_t, s, d, puntos_parada)

    return tsp_dp(cost_matrix)

# Funcion para reducir el grafo G_t a un subgrafo G_s
def reducir_grafo(G_t, s, d, puntos_parada):
    # Crear un nuevo grafo G_s con nodos s, d y los puntos de parada
    G_s = Graph()
    G_s.V = [s] + puntos_parada + [d]
    E = []
    cost_matrix = [][]

    for i in nodos:
        distancias = dijkstra(G_t, i)
        for j in nodos:
            if i != j:
                E.add((i, j))
                cost_matrix[i][j] = distancias[j]

    G_s.E = E
    return G_s, cost_matrix
```

```
def tsp_dp(cost_matrix):
    n = longitud_de_cost_matrix
    # Diccionarios para almacenar costos y caminos
    dp = {}
    path = {}

    # Casos base
    dp[{0}] = {0: 0}  # Costo para llegar a la ciudad 0 desde la ciudad 0 es 0
    path[{0}] = {0: [0]}  # Camino para llegar a la ciudad 0 es [0]

    for i from 1 to n - 1:
        dp[{i}] = {i: infinito}
        path[{i}] = {i: None}

    for size from 1 to n - 1:
        foreach subset in combinations(range(1, n), size):
            subset = subset.add(0)
            dp[subset] = {}
            path[subset] = {}
            foreach city i in subset:
                if i == 0: continue

                prev_subset = subset - {i}
                dp[subset][i] = infinito
                path[subset][i] = None

                foreach j in prev_subset:
                    if dp[prev_subset][j] + cost_matrix[j][i] < dp[subset][i]:
                        dp[subset][i] = dp[prev_subset][j] + cost_matrix[j][i]
                        path[subset][i] = path[prev_subset][j] + [i]

    # Devolvemos el camino y su costo que recorre todos los vertices empezando en
    0 y terminando en n
    return dp[all_cities][n], path[all_cities][n]
```

5.4. Análisis de la complejidad temporal

El algoritmo primero reduce el grafo original G_t a un subgrafo G_s que contiene únicamente los nodos clave: el origen s, el destino d y los puntos de parada intermedios. Esta reducción implica ejecutar el algoritmo de Dijkstra desde cada uno de estos nodos, lo que resulta en una complejidad temporal de:

$$O\left(k \times (E_t + V_t \log V_t)\right)$$

donde k es el número de nodos en G_s, E_t son las aristas y V_t los nodos del grafo original.
El algoritmo de Held-Karp que utilizamos para resolver el problema del viajante tiene complejidad:

$$O\left(k^2 \times 2^k\right)$$

En conjunto, la complejidad temporal total del algoritmo es:

$$O\left(k \times (E_t + V_t \log V_t) + k^2 \times 2^k\right)$$