# Análisis de Problema Intergaláctico

Nahuel Prieto

## 1 Descripción del Problema

**Descripción del Problema**

Problem - B - Codeforces

El malvado comandante Zargon capturó el equipo del capitán Martínez. El capitán Martínez logró escapar, pero para ese momento la nave de Zargon ya había saltado al hiperespacio. Sin embargo, Martínez sabe en qué planeta aterrizará Zargon. Para salvar a sus amigos, Martínez debe viajar repetidamente a través de portales para llegar a ese planeta. En total, la galaxia tiene $n$ planetas, indexados con números del 1 al $n$. Martínez está en el planeta con índice 1, y Zargon aterrizará en el planeta con índice $n$. Martínez puede moverse entre algunos pares de planetas a través de portales (puede moverse en ambas direcciones), la transferencia toma una cantidad positiva de segundos. Martínez comienza su viaje en el tiempo 0.
Puede suceder que otros viajeros estén llegando al planeta donde Martínez se encuentra. En este caso, Martínez tiene que esperar exactamente 1 segundo antes de poder usar el portal. Es decir, si en el tiempo $t$ otro viajero llega al planeta, Martínez solo puede atravesar el portal en el tiempo $t+1$, a menos que haya más viajeros llegando en el tiempo $t+1$ al mismo planeta. Conociendo la información sobre los tiempos de viaje entre los planetas, y los tiempos en los que Martínez no podría usar el portal en planetas particulares, determine el tiempo mínimo en el cual puede llegar al planeta con índice $n$.

### 1.1 Input

**Input**

La primera línea contiene dos enteros separados por espacio: $n$ ($2 \leq n \leq 10^5$), el número de planetas en la galaxia, y $m$ ($0 \leq m \leq 10^5$), el número de pares de planetas entre los cuales Martínez puede viajar usando portales. Luego siguen $m$ líneas, cada una conteniendo tres enteros: la $i$-ésima línea contiene los números de los planetas $a_i$ y $b_i$ ($1 \leq a_i, b_i \leq n, a_i \neq b_i$), que están conectados a través de portales, y el entero tiempo de transferencia (en segundos) $c_i$ ($1 \leq c_i \leq 10^9$) entre estos planetas.
Luego siguen $n$ líneas: la $i$-ésima línea contiene un entero $k_i$ ($0 \leq k_i \leq 10^5$) que denota el número de momentos de tiempo cuando otros viajeros llegan al planeta con índice $i$. Luego siguen $k_i$ enteros distintos separados por espacio $t_{ij}$ ($0 \leq t_{ij} < 10^9$), ordenados en orden ascendente.

### 1.2 Output

**Output**

Imprima un solo número: la menor cantidad de tiempo que Martínez necesita para ir del planeta 1 al planeta $n$. Si Martínez no puede llegar al planeta $n$ en ninguna cantidad de tiempo, imprima el número -1.

## 2 Análisis de Complejidad: Dijkstra Modificado

Definimos las siguientes variables para nuestro análisis:

- $N$: El número total de planetas (nodos).
- $M$: El número total de portales (las aristas bidireccionales, $E = 2M$).
- $K_{\text{total}}$: El número total de tiempos de llegada de viajeros en todos los planetas.

La complejidad total de la solución es la suma de la lectura de la entrada y la ejecución del algoritmo de Dijkstra modificado.

### 2.1 Complejidad de la Lectura de Entrada (Función `solve`)

1. **Lectura de $N$ y $M$**: $\mathcal{O}(1)$.
2. **Creación de estructuras**: $\mathcal{O}(N)$ para inicializar los vectores `G`, `tiempo_viajeros`, y `tiempo`.
3. **Lectura de $M$ portales**: Se leen $M$ líneas y se realizan $E = 2M$ inserciones (`push_back`) en el grafo. Costo: $\mathcal{O}(M)$.

4. **Lectura de $K_{\text{total}}$ viajeros**: Se realizan $K_{\text{total}}$ inserciones (`insert`) en total. Usando `std::unordered_set`, el costo promedio de cada inserción es $\mathcal{O}(1)$.

El costo total promedio de la lectura es: $\mathcal{O}(N + M + K_{\text{total}})$.

## 2.2 Complejidad del Algoritmo (`dijkstra`)

El algoritmo es un Dijkstra estándar con una `priority_queue`, modificado con una lógica de espera.

1. **Costo base de Dijkstra**: Cada uno de los $N$ planetas se extrae de la cola $\mathcal{O}(\log N)$. Cada una de las $E$ (o $2M$) aristas se relaja, lo que puede llevar a una inserción de $\mathcal{O}(\log N)$. **Costo base de Dijkstra:** $\mathcal{O}((N + M) \log N)$

2. **Costo de la Lógica de Espera (Optimización Clave)**: El bucle `while` para calcular `tiempo_salida` se ejecuta **una sola vez por planeta** (cuando este es extraído de la cola), no una vez por arista.

   Para un planeta $u$, el bucle `while` se ejecuta $c_u$ veces (bloqueos consecutivos). Cada `find()` en el `unordered_set` es $\mathcal{O}(1)$ en promedio.

   El costo total de *todos* los bucles de espera (sumando todos los planetas) es la sumatoria de $\sum_{u=1}^{N} \mathcal{O}(c_u)$, que está acotada por $K_{\text{total}}$.

   **Costo total de espera (promedio):** $\mathcal{O}(K_{\text{total}})$

## 2.3 Complejidad Total (Promedio)

Sumamos los términos dominantes:

$$\mathcal{O}((N + M) \log N + K_{\text{total}})$$

## 2.4 Nota sobre el Peor Caso (Colisiones de Hash)

El análisis promedio anterior asume que `std::unordered_set` provee una búsqueda de $\mathcal{O}(1)$. Sin embargo, esta estructura (una tabla hash) tiene un peor caso.

Si un juez utiliza un caso de prueba malicioso donde muchos tiempos de viajeros en un planeta causan "colisiones" en la tabla hash, la operación `find()` colapsa de $\mathcal{O}(1)$ a $\mathcal{O}(k_u)$, donde $k_u$ es el número de viajeros en ese planeta.

En este escenario, el costo de la lógica de espera (que calculamos en $\mathcal{O}(K_{\text{total}})$) explota. El costo de ese bucle `while` para un solo planeta se convierte en $\mathcal{O}(k_u^2)$.

$$\text{Costo Total (Peor Caso): } \mathcal{O}((N + M) \log N + \sum_{u=1}^{N} k_u^2)$$

Si el peor caso concentra todos los $K_{\text{total}}$ viajeros en un solo planeta, esto se degrada a $\mathcal{O}((N + M) \log N + K_{\text{total}}^2)$. Este escenario es el que probablemente causaba el "Time Limit Exceeded" (TLE) en la implementación no optimizada (la que calculaba `tiempo_salida` dentro del bucle `for` de las aristas).

# 3 Implementación en C++

```cpp
#include <bits/stdc++.h>
using namespace std;

void initialize_single_source(vector<long long> &tiempo, const int source)
{
    fill(tiempo.begin(), tiempo.end(), LLONG_MAX);
    tiempo[source] = 0;
}


void relax(const long long tiempo_salida, const int v, const long long w, vector<long long> &tiempo)
{
    if (tiempo[v] > tiempo_salida + w)
    {
        tiempo[v] = tiempo_salida + w;
    }
}

void dijkstra(const int source, const int n, const vector<vector<pair<int, long long>>> &G, vector<long long> &tiempo,
              const vector<unordered_set<long long>> &tiempo_viajeros)
{

    initialize_single_source(tiempo, source);
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long long, int>>> pq;
    pq.push({0, source});

    while (!pq.empty())
    {
        auto [tiempo_llegada, u] = pq.top();
        pq.pop();
        if (tiempo_llegada > tiempo[u])continue;
        if (u == n) break; // llegamos al destino

        long long tiempo_salida = tiempo[u];
        const auto &viajeros_en_planeta = tiempo_viajeros[u];
        while (viajeros_en_planeta.find(tiempo_salida) != viajeros_en_planeta.end())
        {
            tiempo_salida++;
        }

        for (const auto &[v, w] : G[u])
        {
            long long old_tiempo = tiempo[v];
            relax(tiempo_salida, v, w, tiempo);
            if (tiempo[v] < old_tiempo)
            {
                pq.push({tiempo[v], v});
            }
```

```
        }
    }
}

long long viaje_intergalactico(const int n, const vector<vector<pair<int, long long>>> &G, vector<long long> &tiempo,
                               const vector<unordered_set<long long>> &tiempo_viajeros)
{

    dijkstra(1, n, G, tiempo, tiempo_viajeros);

    return (tiempo[n] == LLONG_MAX) ? -1 : tiempo[n];
}

void solve(istream &in, ostream &out)
{
    int n, m;
    in >> n >> m;
    vector<vector<pair<int, long long>>> G(n + 1);
    vector<unordered_set<long long>> tiempo_viajeros(n + 1);
    vector<long long> tiempo(n + 1);

    for (int i = 0; i < m; ++i)
    {
        int u, v;
        long long w;
        in >> u >> v >> w;
        G[u].push_back({v, w});
        G[v].push_back({u, w});
    }

    for (int i = 1; i <= n; ++i)
    {
        int k;
        in >> k;
        for (int j = 0; j < k; ++j)
        {
            long long t;
            in >> t;
            tiempo_viajeros[i].insert(t);
        }
    }

    out << viaje_intergalactico(n, G, tiempo, tiempo_viajeros) << "\n";
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    solve(cin, cout);
    return 0;
}
```

viaje_intergalactico.cpp