Análisis Técnico: Algoritmo de Dijkstra

Programación de estructuras de datos y algoritmos fundamentales (Gpo 604)

1 de diciembre de 2025

Descripción del Algoritmo

El algoritmo de Dijkstra, concebido por el científico de la computación Edsger W. Dijkstra en 1956, es un método determinista para resolver el problema del **camino más corto de origen único** (*Single-Source Shortest Path*) en grafos ponderados dirigidos o no dirigidos.

Su funcionamiento se basa en una estrategia **voraz** (*greedy*). El algoritmo mantiene un conjunto de nodos visitados cuyas distancias más cortas desde el origen ya son definitivas. En cada iteración, selecciona el nodo no visitado con la menor distancia acumulada tentativa, lo añade al conjunto de visitados y explora sus vecinos.

El núcleo del algoritmo es el proceso de **relajación** (*relaxation*). Para cada vecino v de un nodo u, si se encuentra una ruta más eficiente a través de u, se actualiza la distancia de v:

$$\text{si } d[u] + w(u, v) < d[v] \implies d[v] = d[u] + w(u, v)$$

Nota crucial: El algoritmo requiere que los pesos de las aristas sean **no negativos** ($w(e) \geq 0$). Si existieran pesos negativos, se requeriría el algoritmo de Bellman-Ford.

Análisis de Complejidad

El rendimiento del algoritmo depende drásticamente de las estructuras de datos utilizadas para representar el grafo y para gestionar la cola de prioridad de los nodos candidatos.

Complejidad Temporal

Para la implementación utilizada en el proyecto (Python con `heapq`), se asume un grafo representado mediante **listas de adyacencia** y el uso de un **Min-Heap** (Montículo Binario) como cola de prioridad.

Sea V el número de vértices (nodos) y E el número de aristas (calles).

- **Inicialización:** Se asignan distancias infinitas a todos los nodos. Esto toma tiempo lineal $O(V)$.
- **Extracción del Mínimo:** El bucle principal se ejecuta una vez por cada vértice. La operación `heappop` para extraer el nodo con menor distancia toma $O(\log V)$. Al hacerse para todos los vértices, el costo es $O(V \log V)$.
- **Relajación de Aristas:** En el peor de los casos, recorremos todas las aristas del grafo una vez. Si encontramos un camino más corto, realizamos una inserción (`heappush`) en el montículo, lo cual cuesta $O(\log V)$. Al iterar sobre todas las aristas, esto resulta en $O(E \log V)$.

Dado que en un grafo conectado $E \geq V - 1$, el término dominante es el de las aristas.

$$\boxed{\text{Complejidad Temporal Total:} \quad \mathcal{O}(E \log V)}$$

Comparativa: Si usáramos una matriz de adyacencia y un arreglo simple (sin Heap), la complejidad sería $\mathcal{O}(V^2)$, lo cual es ineficiente para mapas de calles (grafos dispersos donde $E \ll V^2$).

Complejidad Espacial

El espacio requerido se determina por el almacenamiento del grafo y las estructuras auxiliares:

- **Grafo (Lista de Adyacencia):** Se necesita almacenar cada vértice y sus conexiones. Costo: $O(V + E)$.
- **Distancias y Padres:** Se mantienen arreglos o diccionarios para `distancia`, `padre` y estado de `visitado` para cada nodo. Costo: $O(V)$.
- **Cola de Prioridad:** En el peor caso, puede llegar a almacenar vértices proporcionalmente a las aristas si no se eliminan duplicados (lazy deletion). Costo: $O(E)$.

$$\boxed{\text{Complejidad Espacial Total:} \quad \mathcal{O}(V + E)}$$

Conclusión

La implementación mediante listas de adyacencia y colas de prioridad binarias es la óptima para sistemas de navegación en mapas reales, equilibrando un uso de memoria lineal con un tiempo de ejecución casi-lineal.