UNIVERSIDAD NACIONAL DEL ALTIPLANO – EP INGENIERÍA DE SISTEMAS

CURSO: Algoritmos y Estructuras de Datos
SEMANA: 6
NIVEL: Avanzado
DOCENTE: Mg. Aldo Hernán Zanabria Gálvez

TÍTULO DE LA PRÁCTICA

Optimización de Rutas Urbanas Usando el Algoritmo de Dijkstra

1. OBJETIVO GENERAL

Aplicar el **Algoritmo de Dijkstra** para modelar, analizar y optimizar rutas urbanas en una red vial, implementando soluciones eficientes en **C++ y Python**, con énfasis en la comparación de rendimientos, estructuras de datos utilizadas y análisis de complejidad.

2. OBJETIVOS ESPECÍFICOS

- Comprender y aplicar el algoritmo de Dijkstra como solución al problema de caminos mínimos desde un origen único.
- Representar redes urbanas como grafos dirigidos y ponderados.
- Desarrollar implementaciones eficientes en C++ y Python utilizando **colas de prioridad** y **listas de adyacencia**.
- Evaluar y comparar el desempeño en diferentes tamaños de grafo (10, 100, y 1000 nodos).
- Analizar la complejidad computacional, limitaciones y aplicaciones reales del algoritmo.

3. PROBLEMA A RESOLVER

La Municipalidad de Puno desea optimizar el tránsito urbano entre puntos estratégicos (plazas, hospitales, mercados, etc.) utilizando tecnología de rutas inteligentes. Se requiere implementar un sistema que modele las interconexiones urbanas como un **grafo dirigido y ponderado**, donde los pesos representan **tiempos estimados de viaje** en minutos. A partir de un punto de origen (por ejemplo, la Plaza de Armas), se debe calcular el **camino más corto** hacia todos los demás puntos de la ciudad.

4. FUNDAMENTO TEÓRICO

4.1. Algoritmo de Dijkstra

Es un algoritmo **greedy** para resolver el problema de caminos mínimos de una sola fuente, sobre grafos ponderados sin aristas negativas. Utiliza un mecanismo de **relajación iterativa** de aristas y selección progresiva del nodo con la menor distancia acumulada mediante una **cola de prioridad**. Su correcta implementación requiere una

estructura de datos eficiente como **listas de adyacencia** para representar el grafo y un **min-heap** para gestionar las distancias tentativas.

4.2. Complejidad Computacional

Con colas de prioridad, su eficiencia es de **O((V + E) log V)**. Para grafos dispersos (E ≈ V), resulta muy eficiente. No puede usarse en grafos con pesos negativos. El uso de listas de adyacencia permite una representación espacial óptima de grafos urbanos reales.

4.3. Aplicaciones Prácticas

El algoritmo se aplica en navegación GPS, redes de telecomunicaciones, planificación logística, análisis de tráfico y sistemas de ruteo. En esta práctica se adaptará a un entorno urbano simulado para optimizar tiempos de traslado entre puntos de interés.

5. DESARROLLO DE LA PRÁCTICA

Actividad 1: Modelado del Grafo Urbano

- Representar 3 versiones del grafo: con **10, 100 y 1000 nodos**.
- Generar conexiones ponderadas aleatorias representando distancias o tiempos.
- Utilizar listas de adyacencia para almacenar el grafo.

Actividad 2: Implementación del Algoritmo

En C++:

- Utilizar vector<vector<pair<int, int>>> para listas de adyacencia.
- Usar priority_queue (min-heap) para gestionar distancias mínimas.

En Python:

- Usar diccionarios anidados (dict[str, list[tuple[str, int]]]).
- Utilizar heapq como min-heap.

Actividad 3: Pruebas y Comparación

- Ejecutar ambos programas sobre los tres grafos generados.
- Medir el tiempo de ejecución con herramientas como time (Python) y chrono (C++).
- Evaluar el uso de memoria y velocidad relativa.

Actividad 4: Análisis y Documentación

- Comparar los resultados obtenidos entre lenguajes.
- Explicar qué estructura de datos usaste, cómo afectó la eficiencia.

- Describir cómo escaló el algoritmo al aumentar el tamaño del grafo.

6. EJEMPLO DE CÓDIGO BASE

C++ (fragmento)

```cpp
vector<int> dijkstra(int n, vector<vector<pair<int, int>>>& graph, int
start) {
    vector<int> dist(n, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>>
pq;
    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto [v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Python (fragmento)

```python
import heapq

def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    queue = [(0, start)]

    while queue:
        d, u = heapq.heappop(queue)
        if d > dist[u]: continue
        for v, w in graph[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heapq.heappush(queue, (dist[v], v))
    return dist
```

7. RESULTADOS ESPERADOS

- Tabla de tiempos de ejecución para cada tamaño de grafo y cada lenguaje.
- Gráfico comparativo (tiempo vs. número de nodos).
- Conclusión sobre cuál lenguaje fue más eficiente y por qué.
- Reflexión sobre la importancia de seleccionar estructuras adecuadas.

8. CRITERIOS DE EVALUACIÓN

Criterio	Puntaje
Modelado correcto de grafos	20%
Implementación correcta en C++ y Python	25%
Medición y análisis de resultados	25%
Complejidad computacional y estructuras analizadas	20%
Presentación y conclusiones	10%

9. ENTREGABLES

- Código fuente documentado en C++ y Python.
- Informe técnico en Word o PDF con los siguientes apartados:
  - Introducción y objetivos
  - Marco teórico completo
  - Desarrollo e implementación
  - Pruebas y resultados
  - Análisis y comparación
  - Conclusiones

Referencias

- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269-271.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Knuth, D. E. (1973). *The Art of Computer Programming*, Volume 1: Fundamental Algorithms. Addison-Wesley.
- Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms*. McGraw-Hill.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.