# Implementing Dijkstra’s Algorithm Using Apache Spark

## 1. Implementation Explanation

This project aimed to implement Dijkstra’s shortest path algorithm using Apache Spark to compute the shortest distances from a single source node to all other nodes in a graph. The graph data was provided in an edge list format, consisting of 10,000 nodes and 100,000 weighted edges.

- **Spark Environment Setup:**I installed Apache Spark locally on my MacBook Air and configured it through Visual Studio Code using a Python virtual environment (spark_env). The necessary environment variables (SPARK_HOME, PATH) were updated in my .zshrc file, and Spark was linked to Python3.

- **Data Reading and Preprocessing**: I read the weighted_graph.txt file into Spark RDD using Spark’s textFile method. The first line of the file, containing the number of nodes and edges, was skipped after parsing.

- **Adjacency List Construction:** Each line was mapped into a (source_node, (destination_node, weight)) pair. The resulting RDD was grouped by key to create an adjacency list. To optimize Spark's parallelism, the RDD was repartitioned into 16 partitions.

- **Dijkstra's Algorithm with RDDs:**
  The algorithm maintained:
  - A dictionary of distances (node → shortest distance)
  - A set of visited nodes
  - At each iteration, the unvisited node with the smallest tentative distance was selected, and its neighbors’ distances were updated if shorter paths were found. Spark transformations like reduce, mapValues, filter, and lookup were leveraged to perform distributed computations efficiently.

- **Output**
  - The program outputs the shortest distance from the source node (node 0) to every other node. Unreachable nodes are assigned "INF" (infinity).

## 2. Performance Analysis

### System Configuration:

- MacBook Air (M2, 8-core CPU, 8 GB RAM)

- Local Apache Spark installation
- Python 3.10.12 environment

**Performance Observations:** Processing large graphs on a local machine, without a true Spark cluster, led to extended computation times. Warnings related to memory management and shuffle operations indicated resource limitations.

- The program took several minutes (up to 20–40+ minutes depending on settings) to complete, primarily due to limited CPU parallelism and the absence of high-memory nodes.
- Repartitioning the RDD into 16 partitions, compared to the default single partition setup provided slight improvements.
- Running on Azure VMs with distributed Spark would significantly enhance speed through actual parallelization.

## 3. Challenges and Lessons Learned

**Challenges:**

- **Azure Login Issues:** Initial difficulties accessing my Azure account delayed cloud VM setup.
- **Local Resource Limitations:** My local machine lacked the parallel computing power necessary for fast Spark operations.
- **Spark Environment Configuration:** Setting up Spark correctly on a Mac, ensuring the correct Java version compatibility, and managing Python environments required careful attention to environment variables and PATH settings.
- **Data Management:** Parsing the input data correctly and handling errors like "too many values to unpack" during early debugging phases needed careful handling of file structure.

**Lessons Learned:**

- **Big Data Tools Need Big Infrastructure:** Running Spark jobs efficiently needs substantial distributed resources, not personal laptops.
- **Importance of Testing on Small Data:** Testing the algorithm on a smaller graph before full execution would have saved troubleshooting time.
- **Stronger Understanding of Distributed Algorithms:** Implementing Dijkstra’s algorithm using Spark RDDs deepened my understanding of how graph algorithms must be adapted for distributed systems where operations are not inherently sequential.