DijkstraSpark Assignment Report

Name: Krithika Reddy Cherukupally
Panther ID: #002831440

Explanation of Your Implementation

I implemented Dijkstra's algorithm using PySpark to compute the shortest paths from node 0 to all other nodes in a weighted graph. The implementation is contained in the dijkstra_spark.py script, which leverages PySpark's RDD transformations for distributed processing. Here's a breakdown of the approach:

- **Graph Input**: The script reads the graph from /home/azureuser/weighted_graph.txt. The first line contains the number of nodes and edges, followed by lines specifying edges in the format src dst weight.
- **Graph Representation**: Edges are parsed into an RDD of the form (src, (dst, weight)) and grouped by source node to create a graph structure. This RDD is cached for performance.
- **Vertex Initialization**: Vertices are initialized as an RDD of (node, (distance, path)) pairs. The starting node (node 0) has a distance of 0 and a path [0], while all other nodes have a distance of float('inf') and an empty path.
- **Iterative Computation**: Dijkstra's algorithm is implemented iteratively using RDD transformations:
    - In each iteration, the current distances and paths are broadcast to all nodes.
    - For each edge (src, (dst, weight)), if the source node's distance is not infinite, a potential new distance and path to the destination are computed as (dst, (src_distance + weight, src_path + [dst])).
    - The script combines these updates with the current distances using reduceByKey to keep the minimum distance and corresponding path for each node.
    - The loop continues until convergence (when distances no longer change) or a maximum number of iterations (equal to the number of nodes) is reached.
- **Output**: The final distances and paths are written to /home/azureuser/dijkstra-spark/shortest_paths.txt in the format Node <id>: Distance <dist>, Path <path>. Unreachable nodes are marked with Distance INF, Path None.

The script successfully generates the shortest paths from node 0, but I also have a reference file shortest_distances_from_8168.txt showing distances from node 8168, which I can use to validate distances for specific nodes if needed.

Performance Analysis

The script's performance is influenced by the graph size (10,000 nodes) and the VM's resources. Here are the key observations:

- **Execution Time**: The previous Scala implementation took 14,379 seconds (~240 minutes) to complete, which is quite long. The PySpark version (dijkstra_spark.py) was interrupted during execution, so I don't have an exact runtime. However, based on the logs, it was on iteration 5 when stopped, and given the graph size, it might take a similar amount of time (several hours) to complete all iterations. The iterative nature of the algorithm, combined with broadcasting and collecting RDDs in each iteration, contributes to the long runtime.
- **Resource Constraints**: The VM used is an Azure B2s instance with 2 vCPUs and 4 GiB of memory, running in local mode (local[*]). The logs show a memory store capacity of 434.4 MiB, which is relatively small for a graph of this size. The limited resources likely cause frequent disk spilling and slow down RDD operations, especially during broadcasts and collects.
- **Improvement Suggestions**: Performance could be improved by:
    - Using a larger VM, such as a D4s_v5 instance with 4 vCPUs and 16 GiB of memory, to handle the memory-intensive operations better.
    - Deploying a distributed Spark cluster with multiple nodes to parallelize the computation across more workers.
    - Optimizing the script by reducing the number of collect operations (e.g., checking convergence more efficiently) or using a more scalable graph processing framework like GraphX.

Challenges and Lessons Learned

Challenges

- **SSH Access Issues**: Encountered "Permission denied (publickey)" errors when connecting to the VM, requiring me to generate and update SSH key pairs.
- **File Upload Errors**: Incorrect scp commands (e.g., using Windows paths as filenames) caused file upload issues.

- **File Removal Issues**: Special characters in filenames made it difficult to remove files using rm.
- **Scala Version Issues**: The initial Scala script (DijkstraSpark.scala) had syntax errors (e.g., EdgeDirection.Outgoi typo) and produced negative distances due to incorrect comparison logic.
- **Long Execution Time**: Both Scala and PySpark versions took hours to run due to the small VM size and large graph.
- **PySpark Script Issues**:
    - File not found errors due to filename mismatches (e.g., weighted_graph.txt vs. weighted_graphx.txt).
    - Indentation errors in the script causing syntax issues.
    - Syntax errors when running the script with incorrect command-line arguments (e.g., treating a Python function as a shell command).
    - Incorrect Scala imports in a Python script causing syntax errors.
- **Output File Confusion**: Initial runs produced output.txt (from the Scala script) instead of shortest_paths.txt, causing confusion.
- **Script Interruption**: Interrupted the PySpark script with Ctrl + Z, which stopped it before it could create shortest_paths.txt. Killing the job led to SparkContext errors (Cannot call methods on a stopped SparkContext).

Lessons Learned

- **SSH and Security**: Learned the importance of matching SSH key pairs and securing private keys.
- **Command Differences**: Understood differences between Windows (dir, type) and Linux (ls, cat) commands.
- **File Management**: Mastered correct syntax for scp and rm to handle special characters in filenames.
- **Debugging**: Gained experience debugging Scala compilation errors and PySpark scripts using error handling and debug prints.
- **Python Syntax**: Learned the importance of consistent indentation in Python scripts.

- **Running PySpark Scripts**: Discovered the correct way to run PySpark scripts using spark-submit and the difference between running a script and calling a function directly in the terminal.
- **Graph Algorithms with PySpark**: Learned how to use PySpark RDD transformations for graph algorithms like Dijkstra's.
- **Input Validation**: Understood the importance of ensuring the correct input file is used by the script.
- **Scala vs. Python**: Recognized the syntax differences between Scala and Python, especially for imports.
- **Output Verification**: Learned to verify output files to confirm script execution.
- **Dijkstra's Algorithm**: Understood the importance of correct comparison logic to avoid negative distances.
- **Script Execution**: Realized the importance of letting a script complete its execution to produce the output file.
- **Performance Considerations**: Learned that Spark performance heavily depends on resource allocation, and small VMs can significantly slow down computations.

This assignment provided valuable insights into distributed computing with PySpark, graph algorithms, and managing cloud-based VMs.