Implementation Report: Dijkstra's Algorithm with Apache Spark

Your Name / Team Name

April 23, 2025

1 Introduction

This report describes the implementation of Dijkstra's shortest path algorithm using Apache Spark on Azure VMs. The algorithm computes the shortest path from a source node to all other nodes in a weighted graph.

2 Implementation Details

2.1 Algorithm Overview

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights. Our implementation adapts this algorithm to work in a distributed environment using Apache Spark's Resilient Distributed Dataset (RDD) abstraction.

2.2 Key Components

- **Graph Representation:** The graph is stored as an adjacency list using Spark RDDs.
- **Distance Tracking:** A Python dictionary (or equivalent structure within Spark) maintains the shortest known distance to each vertex.
- **Active Nodes Set:** To optimize performance, we track which nodes need processing in each iteration.
- **Broadcast Variables:** Current distances are shared efficiently across the cluster using Spark's broadcast mechanism.

2.3 Implementation Strategy

The algorithm follows these steps:

1. Parse the input graph file to build an adjacency list representation (RDD).
2. Initialize the distance of the source node to 0 and all other nodes to infinity.
3. Iteratively perform the following:
   - Broadcast current distances to all worker nodes.
   - Compute new candidate distances through active nodes using RDD transformations.
   - Update distances when shorter paths are found (e.g., using reduceByKey or aggregateByKey).
   - Identify nodes with updated distances for the next iteration.
4. Terminate when no distances improve in an iteration or a maximum iteration count is reached.

2.4 Optimization Techniques

- **Active Node Filtering:** Only process nodes (and their neighbors) whose distances were updated in the previous iteration. This avoids redundant computations.
- **Early Termination:** Stop the iterative process as soon as an iteration completes without any distance updates.
- **Broadcast Variable Management:** Use broadcast variables for read-only data (like the current distance map) that needs to be accessed by all tasks on worker nodes, reducing communication overhead compared to sending the data with each task.
- **RDD Caching/Persistence:** Cache the adjacency list RDD in memory ('.cache()' or '.persist()') if it's reused across iterations to avoid recomputing it from the source file.

3 Performance Analysis

3.1 Testing Environment

- **Azure VM:** Standard_D4s_v3 (4 vCPUs, 16 GB RAM) - Specify if this was the master, workers, or both.
- **Apache Spark:** Version 3.1.2 (Standalone cluster mode assumed, specify configuration if different).
- **Test Dataset:** Weighted graph with 10,000 nodes and 100,000 edges (Specify graph characteristics if known, e.g., density, degree distribution).

3.2 Execution Time

The following table summarizes the execution time for different graph sizes on the specified environment:

Table 1: Execution Time vs. Graph Size

Graph Size (Nodes)	Execution Time (seconds)	Iterations
1,000	3.2	12
5,000	8.7	24
10,000	15.4	31

3.3 Scaling Analysis

- **Single-node vs. Cluster:** A 43% performance improvement (reduction in execution time) was observed when scaling from a single worker node to 4 worker nodes for the 10,000-node graph. (Clarify if the single node test was local mode or a 1-worker cluster).
- **Memory Usage:** Peak memory usage reached approximately 2.4GB on the driver/worker nodes during the execution on the 10,000-node graph.
- **Communication Overhead:** Analysis of Spark UI logs indicated that broadcast operations consumed approximately 15% of the total execution time, highlighting it as a significant component of overhead.

3.4 Bottlenecks

- The most significant performance bottleneck identified was the overhead associated with broadcasting the updated distance map in each iteration, especially as the number of nodes with updated distances grew.
- For larger graphs, collecting updated distances back to the driver (if done inefficiently) or shuffling data during reduce operations could also become expensive.

4 Challenges and Lessons Learned

4.1 Challenges

- **Dataflow Management:** Designing the sequence of RDD transformations to efficiently update distances and identify active nodes for the next iteration required careful planning.
- **Memory Optimization:** Tuning Spark's memory configurations (executor memory, overhead) and choosing appropriate persistence levels was crucial to prevent OutOfMemoryErrors, especially with larger graphs or intermediate RDDs.
- **Broadcast Overhead:** Minimizing the size and frequency of broadcast variables was necessary to mitigate their impact on performance.
- **Termination Conditions:** Implementing a robust and correct termination condition that works reliably across distributed nodes (ensuring all nodes agree that no further updates are possible) needed careful consideration.

4.2 Lessons Learned

- **RDD Operations:** Gained practical understanding of the performance implications of different RDD transformations (e.g., 'map', 'flatMap', 'reduceByKey', 'join') in the context of iterative graph algorithms.
- **Broadcast vs. Join:** Learned the trade-offs between using broadcast variables for smaller lookup tables versus performing RDD joins for larger datasets.
- **Performance Tuning:** Developed insights into tuning Spark configurations (e.g., number of executors, cores per executor, memory settings) and monitoring application performance using the Spark UI.
- **Active Set Approach:** Confirmed that filtering computations based on an active set of nodes significantly reduces the workload in sparse update algorithms like Dijkstra's.

5 Conclusion

This implementation successfully adapts Dijkstra's algorithm for distributed execution using Apache Spark. The approach effectively leverages RDDs and broadcast variables to manage the computation across a cluster. While broadcast overhead presents a bottleneck, the implementation demonstrates reasonable performance and scalability for graphs of moderate size (up to tens of thousands of nodes), showing performance improvements with additional worker nodes. Further optimizations using graph-specific libraries or advanced techniques could enhance performance for larger-scale problems.