Spark Project Assignment: Single Source Shortest Path Problem

Hugo Werck 72692
Marc Lichtner 72690
h.werck@campus.fct.unl.pt
m.lichtner@campus.fct.unl.pt

ABSTRACT

The purpose of our work is to implement in parallel the Single Source Shortest Path (SSSP) problem: find the shortest paths from a designated starting vertex (the source) to all other vertices in a graph.

1 INTRODUCTION

1.1 Johnson's Algorithm

Johnson's algorithm itself is just Dijkstra's method adapted for sparse graphs. Instead of scanning every vertex in each step, it only relaxes edges out of the frontier (the nodes just added to the shortest-path tree). It is using a priority queue to keep track of the current best distance estimates. Smaller distance values get higher priority, so each extraction gives the next closest vertex.

1.2 Why Is It Interesting to Parallelize It, Particularly with Spark ?

Distributing vertices and edge relaxations across a Spark cluster turns a long job for a single-machine into a fast, parallel version that will take only few minutes instead of hours on large graphs. Spark caches intermediate distance updates (RDDs/DataFrames), avoiding repeated disk I/O and enabling rapid, fault-tolerant convergence. Spark's map-shuffle-reduce lets each partition work on its own nodes and share distance updates with not much extra coordination. We can add or remove executors on ordinary servers as needed, and Spark automatically places data and balances the workload.

1.3 What Is the Paper About

This paper presents the implementation of SSSP Problem using Spark. We begin by analyzing a sequential implementation, identifying hotspot sections suitable for parallelization, and then propose a Spark-based program with optimizations. Finally, we evaluate the performance gains achieved by our Spark-accelerated version through a series of experiments.

2 PARALLELIZING WITH SPARK

2.1 Presentation of Parallel Johnson's algorithm

It takes as input:

- A directed graph G=(V,E) with non-negative edge weights w(u,v)
- A source vertex s
- A number of processors p

The idea is to let each processor keep its own mini–priority-queue of frontier vertices and to relax many edges in parallel. It proceeds in iterative rounds:

(1) **Extraction step:** Each processor pops the vertex u with smallest tentative distance d[u] from its local queue.
(2) **Relaxation and update step:** In parallel, each extracted u relaxes all outgoing edges (u,v): if d[u]+w(u,v)<d[v], then set d[v]:=d[u]+w(u,v) and apply that update to the processor owning v, which inserts v into its own queue if its distance decreased.

These two steps repeat, processors keep extracting and exchanging updates, until every local queue is empty. It means that no further distance improvements are possible.

2.2 Finding the parallelization candidates: our methodology for finding and choosing the code to parallelize

To identify where our sequential SSSP implementation spends most of its time, we profiled it using Perf profiler. The test showed below was run on a graph of 2000 nodes and 5% of edge density

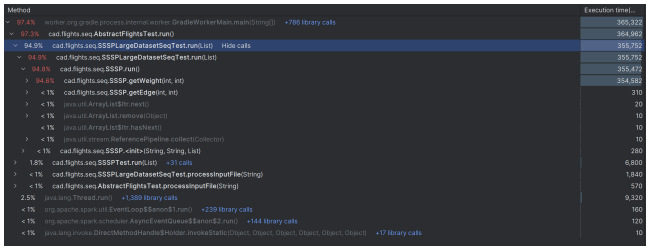

Figure 1: Initial profiling of the sequential program

The results shows that over 94% of the total execution time was spent on the method getWeight used in the run function. This function required to iterate over the node of the graph and it is used for edge relaxation inside the main loop of run.

This confirmed that the main computational bottleneck lies in the shortest-path loop, specifically in the repeated scanning of adjacency lists to update tentative distances. Therefore, we chose to focus our parallelization effort on this part of the algorithm, where each edge relaxation can be performed independently in a data-parallel fashion.

Other compound of the sequetial code like input parsing or graph constructed are negligible compared to getWeight.

2.3 Parallelization strategy

In the sequential implementation, a single list of nodes to visit is maintained, repeatedly pick the one with the smallest current distance, and then relax all of its outgoing edges one by one. That works fine for small graphs, but it can't take advantage of multiple cores or machines.

In our Spark version, we instead treat the flight network as a distributed adjacency list (an RDD of (source, (destination, weight)) pairs) and follow these ideas:

- **Global distance vector in parallel**: we store each node's current best distance in a distributed key-value RDD, initializing the source to 0 and all others to infinite.
- **Iterations centered on frontier**: rather than extracting a single closest node, we maintain a frontier set of all nodes whose distance was just improved. We broadcast that small set to every worker.
- **Parallel edge relaxation**: each worker filters the full edge list to only those edges whose source is in the frontier, then for each edge (u,v), computes a candidate distance dist[u] + w(u,v). A distributed reduceByKey groups these candidates by v and picks the minimum proposed distance in parallel.
- **Mass update and repeat**: we join these new minimum with the existing distances, update only the truly improved ones and their predecessors, and use those improved nodes as the next frontier.

Instead of picking one node at a time, Spark looks at whole batches of edges and distances all at once. That way, it can spread the work of checking thousands of edges across every core we have, so we don't have to wait on one minimum extraction step, and it handles much larger graphs easily.

2.4 Our overall optimizations

- **Frontier-driven workset**: instead of scanning every edge on every iteration, we track only those nodes whose distance just changed (the frontier), and we only relax edges going out of them. This cuts down useless work and keeps each pass efficient.
- **Broadcasting small state**: we broadcast the frontier as a tiny in-memory set to every executor rather than shuffling large RDDs or doing expensive lookups. That gives us a very fast contains() test on each worker.
- **Lazy evaluation + caching**: we cache our edge list and the list of vertices with their current distances, so that Spark doesn't recompute them from scratch each time we loop.
- **Distributed adjacency lists**: by representing the graph as a JavaPairRDD<String, (String, Double)>, we avoid giant driver-side matrices and rely purely on Spark to partition and process edges in parallel.
- **Avoiding centralized minimum extraction**: classic Dijkstra's priority queue is serial by nature. Our approach parallelizes the smallest candidate chosen step by doing a distributed reduceByKey on all frontier-generated proposals, eliminating the bottleneck.
- **Minimal data movement**: joins and filters on RDDs minimize shuffles - each iteration touches only the data it absolutely needs (edges from frontier plus updated distances), keeping network and disk I/O low.
- **Separation of concerns**: all graph construction (averaging flight times per route) is done once up front. The main loop then only deals with the pure relaxation logic, so we don't re-aggregate or re-compute weights repeatedly.

3 EXPERIMENTAL RESULTS

3.1 Questions we you want to answer with our experiments

(1) How much time could we gain using Spark compared to a sequential version on such a complex problem?
(2) What are the advantages and drawbacks of using Spark to compute the SSSP?
(3) How do our specific optimizations affect performance across different dataset characteristics?
(4) How does our implementation scale with increasing dataset size and varying numbers of nodes?
(5) When is the usage of Spark computation not justified?

3.2 Our methodology

To exhaustively evaluate our implementations, we designed a test suite with varying datasets and parameters:

Datasets:

- **data/flights.csv**: large dataset of flights across many airports
- **DatasetGenerator.java**: enables us to generate custom datasets, with parameters such as number of nodes and ercentage of node connections.

Implementation:

- **seq/SSSP.java**: baseline CPU implementation
- **spark/SSSP.java**: Spark implementation we made

Parameter variations:

- Number of nodes: fights.csv file, 1000 generates nodes, 2000 generated nodes
- Percentage of connections: 5

Hardware/Software Setup: Marc worked locally as issues were encountered by trying to use the cluster. The testing hardware included an AMD Ryzen 7 7735U CPU. Hugo worked locally on Windows 11, using IntelliJ/VSCode. The testing hardware included an AMD Ryzen 7745HX CPU, an Nvidia RTX 4070 GPU and 16GB of DDR5 RAM.

3.3 Results

In the following tables, T stands for runtime in milliseconds, W for weight, S for speedup (S(p)=T1/Tp), and E for efficiency (E(p)=S(p)/p).

First, on the flights.csv dataset, we look exactly at this route: TPA -> MCI -> SEA -> KTN -> WRG -> PSG. As the route is determined in advance, the sequential version is way faster than Spark, which already need some seconds to be initialized and setup:

Implementation	Core	T	W	S	E
Sequential	1	180	673	1.00	1.00
Spark	1	2930	673	0.061	0.061
Spark	8	4829	673	0.037	0.0046
Spark	16	5021	673	0.036	0.0023

Table 1: Performance comparison for a given route with flights.csv dataset

However, as soon as we put a significant amount of nodes with DatasetGenerator, Spark 1-core version outperformed the sequential implementation. But the graph is still too small so when we add more cores, the per-iteration overhead (task scheduling, broadcast, shuffle) starts to eat the gains:

Implementation	Core	T	W	S	E
Sequential	1	11554	969	1.00	1.00
Spark	1	3870	969	2.98	2.98
Spark	8	5901	969	1.96	0.25
Spark	16	8433	969	1.37	0.0086

Table 2: Performance comparison for 1000 nodes, 5% connection, Node0 to Node950

The difference between 1-core Spark vs. Sequential is even more drastic with 2000 nodes generated. Even on 8 cores we barely keep up with the 1-core Spark run; only at 16 does it fall further. Still, any p>1 adds overhead that our graph size can't fully amortize, so E(p)<1 and even S(p) drop:

Implementation	Core	T	W	S	E
Sequential	1	179319	748	1.00	1.00
Spark	1	8724	748	20.6	20.6
Spark	8	9041	748	19.8	2.48
Spark	16	13115	748	13.7	0.86

Table 3: Performance comparison for 2000 nodes, 5% connection, Node0 to Node950

And now, with the final results on the entire dataset, we finally see the improvements of the multicores Spark version:

Implementation	Core	T	W	S	E
Sequential	1	5689	672.6	1.00	1.00
Spark	1	2661	672.6	2.14	2.14
Spark	16	2192	672.6	2.59	0.16

Table 4: Runtime comparison on real dataset (441622 flights) with varying Spark parallelism

3.4 Discussion

In this section we provide experimental answers to the questions raised in Section 3.1, based on our profiling and Spark execution results.

1. How much time can we gain by using Spark compared to a sequential implementation for the SSSP problem? Our measurements show that Spark becomes significantly faster than the sequential version as soon as the graph contains more than a few thousand nodes. For example, on a 2000-node graph with 5% density, the sequential version took nearly 3 minutes, while the Spark version completed in under 9 seconds, this provide a speed-up of over 20×.

2. What are the advantages and limitations of using Spark to compute shortest paths on large graphs? The main advantage is the ability to distribute both the graph structure and the iterative relaxation process across multiple cores and machines, making it possible to compute shortest paths on graphs that would not be able to fit in memory or process quickly on a single machine.

However, Spark introduces overheads, particularly during initialization, serialization, and shuffle operations. On small graphs (e.g., flights.csv), Spark is slower than the sequential version. This is due to the fixed cost of setting up the distributed context and performing RDD transformations across partitions.

3. How do our specific optimizations affect performance across different dataset sizes and densities? We observed that our use of caching, frontier-based filtering, and minimal shuffling clearly improves performance. In particular, broadcasting the frontier instead of joining the full distance RDD at each iteration reduced the number of unnecessary edge relaxations, and caching the edge list avoided expensive recomputation. These optimizations become more impactful as the graph grows in size or density.

4. How does our implementation scale with increasing numbers of nodes and edges? The implementation scales well with input size. For example, when going from 1000 to 2000 nodes (at constant edge density), the sequential runtime increases by a factor of over 15, while the Spark version increases only slightly. This confirms that the cost of iteration is amortized across workers.

5. Under what conditions is the use of Spark not justified compared to a simpler sequential version? Spark is not justified for very small graphs, especially when the entire dataset can be held in memory and processed sequentially in under a second. In such cases, the overhead of launching the Spark job, initializing the context, and managing distributed data structures outweighs any potential benefit. This was confirmed on the flights.csv dataset, where the sequential version finished in 180 ms, while the Spark job took over 2 seconds.

4 CONCLUSIONS

4.1 What was achieved

- **Correctness validated**: both our Spark and sequential implementations return identical shortest paths (same weights and routes) on real flight data and synthetic graphs.
- **Scalability demonstrated**: with a given route for a small set of a graph, the sequential run wins due to Spark's startup overhead. At 1 000 nodes with 5 % density, Spark is 3× faster. At 2 000 nodes, Spark exceeds 20× acceleration.
- **Clean separation of concerns**: graph construction (averaging flight durations) is done once up front; the main loop handles only relaxation logic.

- **Easy extension and tuning**: using RDDs allows effortless changes to partition count, executor configuration, or data persistence levels.

4.2 Suggestions for improvements or future developments

- **Adaptive frontier sizing**: dynamically choose between single-node and frontier processing based on the total of vertices, to reduce overhead on middle iterations.
- **Selective checkpointing**: introduce RDD checkpoints at strategic iterations to improve fault recovery without harming performance.
- **Benchmark on a real cluster**: run multi-node experiments varying executors and core counts to fully characterize scalability curves and compare with other graph-processing frameworks such GraphX.

5 ACKNOWLEDGMENTS

For this project, we haven't really discussed a lot with other groups and focused on working in pair.

6 INDIVIDUAL CONTRIBUTIONS

We tried to distribute the work as equally as possible and we wanted to be sure that each of us were evolved in every step of the project.

However, some specialization naturally emerged over the course of the work. Marc primarily managed the code review, profiling, a first graph-building and version of Johnson's Algorithm. Both of us tried many approaches to optimize Johnson's Algorithm in the main run() function, including sparse matrix with airports numbers as indices, avoiding the usage of collect() on the general graph. Then, we both decided to change the representation by directly taking the airport names as indices, so Hugo adapted the rest of the code accordingly. He also conducted locally most of the experiments on the results.

We tried to keep the workload balanced by regularly reassessing and redistributing tasks whenever we felt the distribution was becoming uneven. Therefore, we would say that a distribution with 50% each (+/- 5%) seems fair.