Analysis and Implementation of Mutual Friends Algorithm Using Apache Spark

Technical Report

Author: Yehdih Mohamed Yehdih
Student ID: C20854

Abstract

This report presents a comprehensive analysis of a distributed mutual friends detection algorithm implemented using Apache Spark's PySpark framework. The implementation demonstrates the application of big data processing techniques to solve social network analysis problems through parallel computing and distributed data processing.

July 4, 2025

Contents

1 Introduction **2**

2 Problem Statement **2**

3 Technical Approach **2**
3.1 Framework Selection 2
3.2 Algorithm Design 2

4 Implementation Analysis **3**
4.1 Code Structure 3
4.2 Key Components Analysis 3
4.2.1 Data Parsing Function 3
4.2.2 Pair Generation 4
4.2.3 Set Operations 4

5 Performance Considerations **4**
5.1 Distributed Processing 4
5.2 Memory Management 4

6 Scalability Analysis **4**
6.1 Time Complexity 4
6.2 Space Complexity 4

7 Results and Validation **5**
7.1 Test Data Analysis 5
7.2 Expected Output 5

8 Optimization Opportunities **5**
8.1 Performance Enhancements 5
8.2 Algorithmic Improvements 5

9 Conclusion **5**

10 Future Work **5**

11 References **6**

1 Introduction

Social network analysis has become increasingly important in understanding relationships and connections within large-scale networks. One fundamental problem in this domain is the identification of mutual friends between users, which has applications in friend recommendation systems, community detection, and social graph analysis.

This report analyzes an implementation of a mutual friends detection algorithm using Apache Spark's PySpark framework. The solution leverages distributed computing principles to efficiently process large datasets and identify common connections between users in a social network.

2 Problem Statement

Given a social network represented as a list of users and their direct friends, the objective is to identify mutual friends between any two users. Specifically, for users A and B who are directly connected, we need to find all users who are friends with both A and B.

The challenge lies in efficiently processing this information when dealing with large-scale social networks that may contain millions of users and billions of connections.

3 Technical Approach

3.1 Framework Selection

Apache Spark was chosen for this implementation due to its:

- Distributed computing capabilities
- In-memory processing optimization
- Rich API for data transformations
- Fault tolerance mechanisms
- Scalability for large datasets

3.2 Algorithm Design

The algorithm follows a multi-stage approach:

1. **Data Parsing**: Extract user information and friend lists from input data
2. **Pair Generation**: Create user-friend pairs with associated friend sets
3. **Grouping and Intersection**: Group pairs and find common friends through set intersection
4. **Name Resolution**: Map user IDs to names for readable output
5. **Result Filtering**: Extract specific mutual friend relationships

4 Implementation Analysis

4.1 Code Structure

The implementation consists of several key components:

```python
from pyspark import SparkContext
sc = SparkContext("local", "MutualFriendsApp")

# Simulate data (in real case, use sc.textFile("file_path"))
data = [
    "1 Sidi 2,3,4",
    "2 Mohamed 1,3,5,4",
    "3 Ahmed 1,2,4,5",
    "4 Mariam 1,3",
    "5 Zainab 2,3"
]
rdd = sc.parallelize(data)

# Step 1: Extract data as (user_id, name, [friend_ids])
def parse_line(line):
    parts = line.strip().split()
    user_id = int(parts[0])
    name = parts[1]
    friends = list(map(int, parts[2].split(',')))
    return (user_id, name, friends)

users_rdd = rdd.map(parse_line)

# Step 2: Create RDD of (user, friend) -> list of user's friends
def generate_pairs(user_id, friends):
    return [((min(user_id, friend), max(user_id, friend)), set(friends))
            for friend in friends]

pairs_rdd = users_rdd.flatMap(lambda x: generate_pairs(x[0], x[2]))

# Step 3: Group by pair and intersect lists to get mutual friends
mutual_friends = pairs_rdd.reduceByKey(lambda x, y: x & y)

# Step 4: Add names (assuming names are accessible)
names_dict = users_rdd.map(lambda x: (x[0], x[1])).collectAsMap()

# Step 5: Find pair (1, 2)
result = mutual_friends.filter(lambda x: x[0] == (1, 2)).collect()

# Display results
for pair, mutual in result:
    id1, id2 = pair
    nom1, nom2 = names_dict[id1], names_dict[id2]
    mutual_names = [names_dict[friend_id] for friend_id in mutual]
    print(f"{id1}<{nom1}>{id2}<{nom2}> => Mutual friends: {mutual_names}")
```

Listing 1: PySpark Mutual Friends Implementation

4.2 Key Components Analysis

4.2.1 Data Parsing Function

The `parse_line` function efficiently extracts structured information from raw text data, converting string representations into appropriate data types for processing.

4.2.2 Pair Generation

The `generate_pairs` function creates normalized pairs using `min` and `max` functions to ensure consistent ordering, preventing duplicate pairs (e.g., (1,2) and (2,1)).

4.2.3 Set Operations

The use of Python sets and the intersection operator (`&`) provides efficient computation of mutual friends, leveraging set theory principles for optimal performance.

5 Performance Considerations

5.1 Distributed Processing

The implementation leverages Spark's distributed processing capabilities through:

- RDD partitioning for parallel execution
- Lazy evaluation for optimized computation
- In-memory caching for repeated operations

5.2 Memory Management

The algorithm efficiently manages memory by:

- Using sets for friend lists to reduce memory footprint
- Implementing lazy evaluation to avoid unnecessary computations
- Utilizing Spark's automatic memory management

6 Scalability Analysis

6.1 Time Complexity

The algorithm exhibits the following complexity characteristics:

- Parsing: O(n) where n is the number of users
- Pair generation: $O(n \times f)$ where f is the average number of friends
- Intersection: O(f) for each pair
- Overall: $O(n \times f^2)$ in the worst case

6.2 Space Complexity

Space requirements scale as:

- User data: O(n)
- Friend pairs: $O(n \times f)$
- Mutual friends results: O(p) where p is the number of pairs

7 Results and Validation

7.1 Test Data Analysis

The implementation was tested with a sample dataset containing 5 users with various friendship connections. The algorithm successfully identified mutual friends between users Sidi and Mohamed.

7.2 Expected Output

For the given test data, the algorithm identifies the mutual friends between users 1 (Sidi) and 2 (Mohamed), demonstrating correct functionality.

8 Optimization Opportunities

8.1 Performance Enhancements

Several optimization strategies could improve performance:

- Implementing broadcast variables for name dictionaries
- Using custom partitioning strategies
- Applying caching for frequently accessed RDDs
- Implementing graph-specific optimizations

8.2 Algorithmic Improvements

Alternative approaches could include:

- Graph-based algorithms using GraphX
- Bloom filters for large-scale approximate matching
- Incremental processing for dynamic graphs

9 Conclusion

This implementation demonstrates an effective approach to solving the mutual friends problem using Apache Spark's distributed computing framework. The solution successfully leverages PySpark's capabilities to process social network data efficiently while maintaining code clarity and maintainability.

The algorithm's design principles of data locality, parallel processing, and set-based operations make it suitable for scaling to larger datasets. Future enhancements could focus on performance optimization and extending functionality to support more complex social network analysis tasks.

10 Future Work

Potential extensions to this work include:

- Integration with real-time data streams
- Support for weighted friendships

- Implementation of friend recommendation algorithms
- Performance benchmarking with larger datasets
- Extension to multi-degree mutual friend detection

11 References