MapReduce for Temporal Transitive Closure

Shuyuan Yang
Case Western Reserve University
Cleveland, USA
sxy841@case.edu

Zekun Feng
Case Western Reserve University
Cleveland, USA
zxf205@case.edu

Xiaoyi Leng
Case Western Reserve University
Cleveland, USA
xxl1332@case.edu

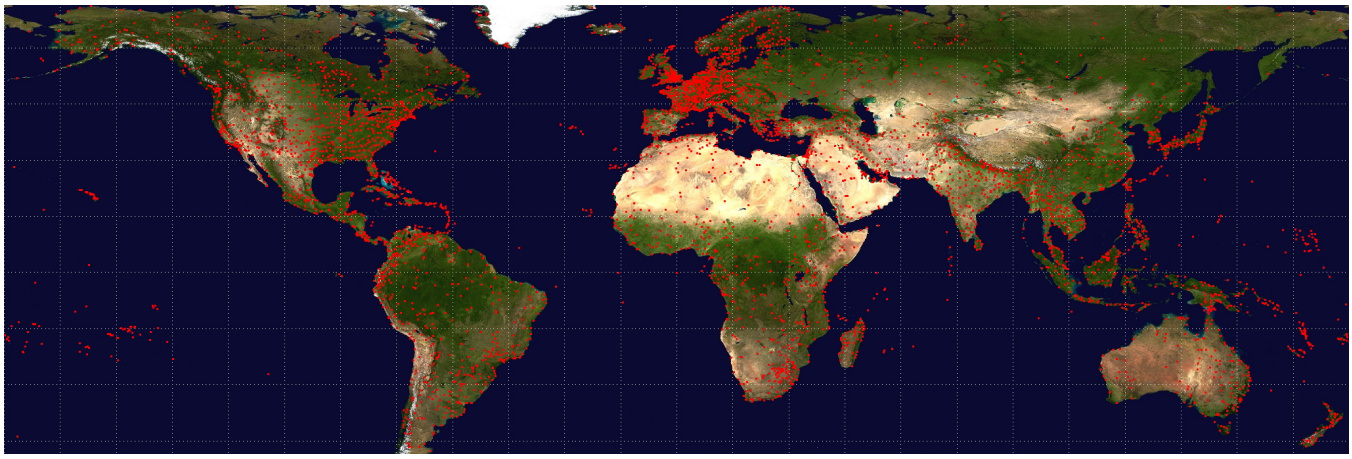

Figure 1. OpenFlights Airports Database contains **over 10,000** airports.

Abstract

Transitive closure is a classic concept in graph theory, that given relations $A \rightarrow B$ and $B \rightarrow C$, you can infer a new relation $A \rightarrow C$. Temporal transitive closure is the process of extending the transitive closure principle to temporal relations. It can be used to efficiently query and analyze data when handling time-based data.

In this paper, we convert a connection flight query problem into a directed graph with temporal transitive closure. By taking advantage of OpenFlights data, We generated a large number of flight routes as our dataset. In order to search flights, we implemented two algorithms: Floyd–Warshall and depth-first search (DFS). And running both algorithms for sequential and paralleled MapReduce editions on a Hadoop cluster. Eventually, we compared the performance of the two algorithms with two editions on datasets of multiple scales.

Keywords: Temporal Transitive Closure, Floyd–Warshall, MapReduce, DFS, Hadoop

1 Introduction

1.1 Introduction

In today's world, air travel has become a commonplace mode of transportation, and individuals are accustomed to checking flight routes before embarking on their journey. As part of the flight booking process, we usually input the departure and destination points on the airline's website, and choose one or two airports for layovers among the numerous airports worldwide, or select a direct flight. From a computer science perspective, the task of querying flight routes involves a Temporal Transitive problem that must be addressed with each search. In this paper, we explore the Temporal Transitive Closure in the context of a connection flight query problem. We convert the problem into a directed graph, using OpenFlights data to generate a large number of flight routes as our dataset. We implement two algorithms, Floyd–Warshall and depth-first search (DFS), and run both algorithms for sequential and parallel MapReduce editions on a Hadoop cluster. We compare the performance of the two algorithms with two editions on datasets of multiple scales.

2 Background

2.1 Temporal Transitive Closure algorithms

Transitive Closure is a concept used in discrete mathematics and graph theory to describe the transitivity of relations. In traditional transitive closure algorithms, relationships are typically assumed to be static, representing the transitivity between elements. A transitive closure matrix represents the transitive relationship between points. For example, even if there is no direct connection from point A to point C, but there exists a path from A to B to D and then to C, the transitive closure matrix would indicate a value of 1 for the relationship between point A and point C, indicating the transitive connection.

However, in specific situations, relationships may contain temporal characteristics, which means they change over time. Take aviation as an example, the relationships between

flights can evolve as schedules vary over time, affecting the connectivity between flights. In our project, we query the path of the different airlines from A to B in the range time T. The Temporal Transitive Closure[5] algorithm addresses this by considering temporal constraints to compute transitive closures that incorporate time-dependent properties. By incorporating time factors into the computation of transitive closures, the Temporal Transitive Closure algorithm ensures that the resulting closure adheres to temporal constraints. It is proved effective in solving problems in the fields of time-based path planning, temporal analysis, correlation analysis, etc. Furthermore, the Temporal Transitive Closure algorithm offers a more accurate and practical approach to computing transitive closures, enhancing the precision and efficiency of relationship analysis within time-constrained environments.

2.2 MapReduce

MapReduce[1] is a programming model used to process various large-scale datasets. In this model, data processing is divided into two stages: the map stage and the reduce stage. In the map stage, the input data is divided into smaller chunks and processed independently using a mapping function to generate key-value pairs. In the reduce stage, the intermediate pairs are grouped by key, and a reducing function is applied to each group to produce the final output. During processing, the underlying system automatically performs large parallel computations, effectively utilizing disk and network resources and reducing execution time.

MapReduce is a powerful model for distributed data processing, enabling efficient analysis of big data in various domains like web search, social network analysis, and genomics research.

2.3 Apache Hadoop(HDFS, MapReduce, YARN)

Apache Hadoop[6] is one of the best open-source tools for addressing data processing challenges using a distributed architecture. It consists of three main components: Hadoop Distributed File System (HDFS), MapReduce, and Yet Another Resource Negotiator (YARN).

HDFS provides reliable and scalable storage for big data. YARN manages computing resources across the cluster. In the Hadoop, MapReduce breaks down the big data into smaller data and performs them. It enables efficient analysis of big data through distributed storage, parallel processing, and resource management.

3 Methodology

3.1 Preliminary Conceptualization of the Algorithm

There exist numerous solutions for addressing the Transitive Closure problem. Considering our data structure, We opine that the most convenient form is to construct a directed graph using the data available, where each node corresponds to an airport, and each edge symbolizes a flight route with its duration as its weight. This approach provides the flexibility to experiment with various algorithms. Since we choose as Python our programming language it can very efficiently manage large data sets. When We came across the problem of transitive closure, We immediately thought of the Floyd-Warshall algorithm due to its reputation for efficiency.

3.2 Floyd–Warshall Algorithm

Algorithm 1 FLOYD-WARSHALL (sequential)

Require: A graph $G = (V, E)$ with weighted edges
Ensure: All-pairs shortest paths in G

$n \leftarrow$ number of vertices in G
$D^{(0)} \leftarrow$ the $n \times n$ matrix of edge weights
for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      if $D_{i,j}^{(k-1)} > D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}$ **then**
        $D_{i,j}^{(k)} \leftarrow D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}$
      else
        $D_{i,j}^{(k)} \leftarrow D_{i,j}^{(k-1)}$
      end if
    end for
  end for
end for
return $D^{(n)}$
procedure COUNTPATH($G, start, end, T$)
  $predecessors, distances \leftarrow$ FLOYD-WARSHALL(G)
  if $path \leftarrow [end]$**and** $path[-1] \neq start$ **then**
    $path$.append($predecessors[start][path[-1]]$)
  end if
  $total\_flight\_time \leftarrow distances[start][end]$
  if $total\_flight\_time \leq T.total\_time$ **then**
    join(path)
  end if
  return path
end procedure

Floyd-Warshall algorithm [4], is a well-known algorithm used to solve the transitive closure problem. The main advantage of the Warshall algorithm is its efficiency and simplicity. It can solve the transitive closure problem in $\theta(N^3)$ time complexity, where n is the number of nodes in the graph. This makes it particularly useful for large-scale problems with many nodes. Instead of considering how many roads to take, Warshall algorithm thinks in terms of intermediate points. Warshall divides the route from point A to point B into two types: those that pass-through point 0 and those that do not. Then, it is divided into those that pass-through point 1 and those that do not, and so on, until the final intermediate point. This concept of intermediate point is very similar to that of an airport used for transferring flights, which is highly relevant to the problem at hand. Therefore,

in this article, we decided to use this algorithm as the first solution to solve the temporal transitive closure problem. In terms of the code, We have designed a "countpath" function. This function requires a directed graph, a starting airport, a destination airport, and a time limit as inputs. The first step is to use the BFS algorithm to get all flights that can be linked to the starting airport. The second step is to use the Floyd-Warshall algorithm to get the shortest path between the starting airport and the destination airport. While calculating the shortest path from the start to the end, the predecessor node of each node on the path is also recorded and stored in a dictionary. After the shortest path is calculated, it will check whether there is a path from the start to the end. If there is, the path is constructed by searching for its predecessor nodes from the end node and storing these nodes in a "path" list in order. Finally, the path is reversed to obtain the path from the start to the end. The last step is to filter out the eligible routes by comparing the total flight time with the time limit.

3.3 DFS Algorithm

Algorithm 2 DFS (sequential)

```
1:  procedure COUNTPATH(G, start, end, b)  ▷ Initialize all vertices as not visited
2:      for all v ∈ G.vertices do
3:          visited[v] ← false
4:      end for
5:      return DFS(G, start, end, T, visited)
6:  end procedure
7:  procedure DFS(G, v, end, T, visited) ▷ Mark the current vertex as visited
8:      visited[v] ← true
9:      count ← 0
10:     if v = end then
11:         return 1
12:     end if
13:     for all w ∈ G.adjacent_vertices(v) do
14:         if not visited[w] then
15:             total_flight_time ← G[v][w]['weight']
16:             if total_flight_time ≤ T.total_time then
17:                 count ← count+ DFS(G, w, end, T, visited)
18:             end if
19:         end if
20:     end for
21:     return count
22: end procedure
```

The DFS (Depth-First Search)[7] algorithm is a commonly used technique for traversing or searching trees or graphs. Its fundamental concept involves starting from a vertex in the graph, traversing along a path as far as possible, backtracking to the previous node, and continuing down another path until all nodes have been visited.

Compared to the Warshall algorithm, the DFS algorithm boasts a lower space complexity. It only requires a stack to store nodes during the traversal process, while the Warshall algorithm needs to construct a two-dimensional matrix to store the relationships between each node. In terms of programming, we also designed a function similar to before, named Countpath. This function also requires a directed graph, a starting airport, a destination airport, and a time limit. In this function, the first step is to use the DFS algorithm to obtain all flight paths from the starting airport to the destination airport. Then, the flight time of each path is compared to the time limit to obtain the flight paths that meet the constraints. However, an issue with this approach is the possibility of closed loops formed by flight paths when search depth is not limited. To address this problem and reduce the required computation time, the default search depth was set to 3 in this experiment.

3.4 MapReduce Algorithm

Algorithm 3 MapReduce

```
1:  procedure MAPPER(V, E, T)
2:      for all e ∈ E do
3:          if e.weight ≤ T then
4:              std::out ← e
5:          end if
6:      end for
7:  end procedure
8:  procedure REDUCER(V, E, start, end, T)
9:      for all e ∈ E do
10:         G'(V', E') ← e
11:     end for
12:     DFS(G', start, end, T) | FLOYD-WARSHALL(G', start, end, T)
13: end procedure
```

To speed up the algorithm's running time, the mapper's primary task is to read data in parallel. In this case, we split the dataset into equal parts and send them to the mappers. Each mapper queries the flight time of each received flight route in sequence and sends all flight routes that meet the temporal query criteria to the MapReduce core in the format of $(source, destination, timestamps)$ key-value pairs.

Subsequently, MapReduce sorts all the received key-value pairs and sends them to the reducer. The reducer, based on the received key-value pairs, adds edges to a new subgraph in sequence, ultimately forming a subgraph that meets the time conditions. Compared to the original graph, this subgraph greatly reduces the number of vertices and edges. Based on this subgraph, the reducer is able to call either Warshall or DFS, any path search algorithm. It then finds paths that

completely meet the time conditions based on the provided source and destination.

4 Experiment

4.1 Environment

In order to realistically demonstrate the parallel computing capabilities of MapReduce, we created 4 identical virtual machines on a Xeon® server. Each virtual machine has 2 processor cores, 8GB of memory, and 40GB of SSD storage space. The communication bandwidth between the virtual machines is determined by the disk I/O. In this case, the upper limit is about 2 Gbps, which is similar to the typical cluster. These 4 nodes form an Apache Hadoop cluster, with each node serving not only as a DataNode for HDFS and a WorkerNode for computation but also handling part of the cluster's task and resource allocation.

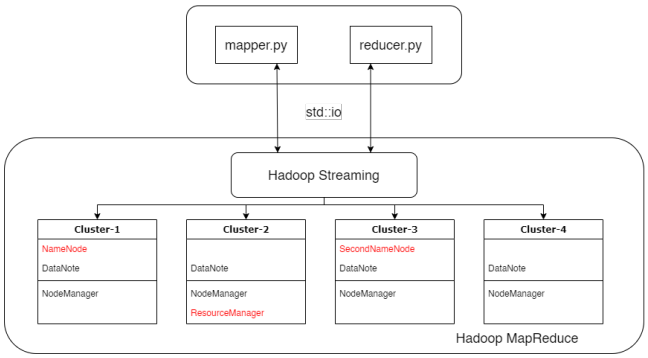

Figure 2. Hadoop Architecture.

The specific architecture implemented is shown in Figure 2. Cluster-1 and Cluster-3 are the master node and backup master node of the entire cluster, respectively, managing HDFS and controlling access to files. Cluster-2 hosts YARN, which splits the data according to demand and prepares to send it to worker containers. For this cluster, each node has 8 v-core containers and there are total 32 containers. Therefore, we set the default number of data partitions as 30, which can run all jobs at once.

Considering that our experiments involve multiple algorithms and a large amount of data processing work, we chose to write our programs in Python instead of Java, which is originally supported by Hadoop. Although this may sacrifice some runtime efficiency, it does not affect the way MapReduce works. Specifically, as shown in Figure 2, our programs communicate with the Hadoop core through the Hadoop Streaming API. Each container runs the mapper and reducer programs separately, with data passed through the standard I/O interface. Similar to the original MapReduce program, during the Mapper phase, each container receives data partitions that have been split by YARN. After the Mapper computation is completed, the intermediate data is sent back to Hadoop and sorted before being sent to the reduce worker for final output processing.

4.2 Dataset

4.2.1 Data Type Filling. The foundational data for our study was obtained from the OpenFlights website. Specifically, we downloaded three distinct data packages - "airport.dat," "airlines.dat," and "routes.dat" - each of which provided relevant information for our research. The "airport.dat" package contained the names of airports, their respective countries of origin, as well as their respective latitude and longitude coordinates and time zone information. The "airlines.dat" package provided the names and IDs of airlines as well as their locations. The "routes.dat" package provided information on flight routes, including the departure and arrival airports, as well as the number of intermediate stops made during the flight. We utilized the Pandas library to read and store the data packages in DataFrame format. After removing any irrelevant data elements, we merged the information using Pandas' merge function to create a new DataFrame. At this point, we have completed the initial processing stage.

4.2.2 Data completion. It should be noted that "flight duration," a crucial piece of information for the research, was unavailable in the OpenFlights data. To overcome this limitation, a simulation approach was adopted. Specifically, we calculated the distance between the two airports using their respective latitude and longitude coordinates and assumed a flight speed between 800 km/h and 900 km/h, generating a random flight speed within this range. Using these parameters, flight duration, departure and arrival times were calculated, accounting for time zone differences. To simulate flight duration, we removed rows with missing data or identical departure and arrival airports, as well as non-zero intermediate stops. These critical elements were appended to the processed DataFrame and subsequently stored in a CSV file.

4.2.3 Random Generation. In order to show the performance of our algorithms in the project, we extended the database based on the available data under reasonable conditions. We randomly selected source and destination airports from the "airport.dat". To ensure the completeness of the dataset, we also randomly assigned airlines based on countries. During the random generation process, we deleted duplicate routes where the source and destination airports were the same. In the "airport.dat", each airport has two codes, the IATA code and the ICAO code. Some airports have missing IATA codes, so there might be cases where certain routes have missing values. We removed such data during the generation process. Once we generated a sufficient amount of data to demonstrate performance, we used this ".dat" file into the project.

4.3 Results and Analysis

As Figure 3 shown, we found that the sequential DFS algorithm has faster running speeds when the dataset is small. However, as the size of the dataset increases, its running time grows exponentially. In contrast, the running time of the MapReduce algorithm remains stable within a large range, with only a gentle upward trend when the dataset is very large.

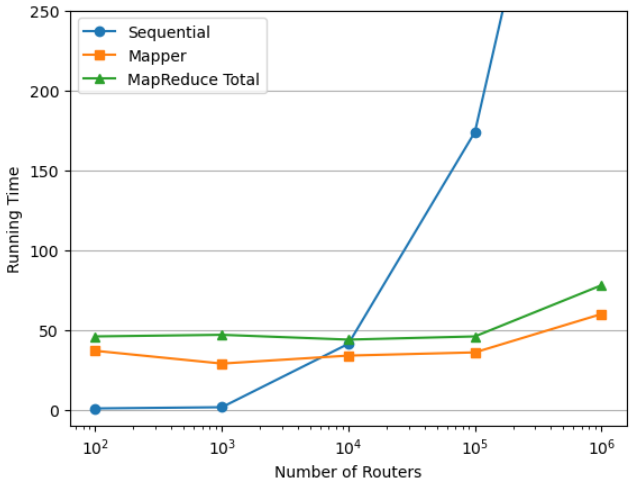

Figure 3. DFS performance comparison.

We believe this is because, during the operation of MapReduce, the startup of the cluster, task allocation, and data distribution occupy most of the time. Therefore, even though the dataset is small, MapReduce still cannot complete its execution in a short period. When the dataset is large enough, each node is allocated enough data slices, and only then does the computation time become significant.

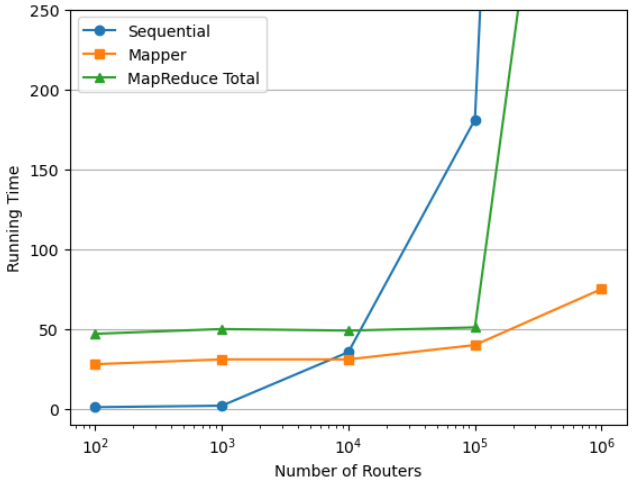

Figure 4. Warshall performance comparison.

Figure 4 displays the performance of Warshall algorithms. The performance of the sequential edition is similar to that of the DFS algorithm, with running time growing exponentially. When the dataset is too large, it cannot complete the query within tens of minutes. The performance of MapReduce also declines significantly when the dataset is large. However, we observed that the running time growth of the mapper is similar to that of DFS. On the other hand, the reducer consumes a significant amount of time. We speculate that this is due to the high time complexity of the Warshall algorithm, as our MapReduce program uses a large number of mappers and only one reducer.

In fact, the Warshall algorithm is not well-suited to the MapReduce workflow. We cannot distribute the process of traversing and finding paths to multiple reducers, as complete paths cannot be found on incomplete graphs. Although we used some strategies to limit the number of edges sent to the reducer, overly large datasets still put considerable pressure on a single reducer.

5 Related Works

In this paper, we mainly used Warshall and DFS algorithms to calculate paths for Temporal Transitive Closure in directed graphs. Both of these classic algorithms can solve the flight route query problem, but each has its drawbacks. The Warshall algorithm is designed to calculate the shortest paths in a graph, with a time complexity of $\theta(N^3)$. Although the mapper can more efficiently transform Temporal Transitive Closure into a weighted directed graph in parallel, the reducer still requires a significant amount of time for the Warshall algorithm to traverse paths. One potential solution is to use multiple MapReduce iterations to subdivide the reducer's task.

In comparison, the time complexity of the DFS algorithm appears more efficient. However, the original DFS algorithm cannot handle directed graphs with cycles, and cycles are a common occurrence in flight route graphs. We chose to use parameters to limit the maximum depth of DFS, but such a strategy may result in an inability to obtain overly long potential paths. Although in real flight route query scenarios, connection flights with more than two stops are rare, most existing flight route queries also limit the number of stops to two.

In fact, the flight route query problem based on Temporal Transitive Closure can be considered as a path search problem in graph theory. There are already many algorithms for path search, such as Greedy Best First Search (GBFS) [2] and Dijkstra's algorithm [3]. As a result, a potential research direction is to combine more path search algorithms with MapReduce to solve Temporal Transitive Closure queries.

Moreover, our research shows that MapReduce has significant advantages in computing large-scale data. This advantage is not limited to the runtime efficiency of parallel computing; this non-shared memory parallel computing greatly reduces hardware requirements. In our experiments, we found that the sequential algorithm would encounter memory overflow errors when running on extremely large datasets. In contrast, MapReduce avoids memory shortages since each container only receives a small portion of the data slices.

6 Conclusion

In our project, we generated a substantial amount of data through randomization using available data. The purpose of this approach was to validate the performance of our parallel algorithms using a large dataset. We compared the time taken by DFS and Warshall algorithms in sequential, mapper, and MapReduce total execution. The MapReduce total execution consistently demonstrated stability and speed, showcasing its high availability and Scalability.

References

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[2] Rina Dechter and Judea Pearl. 1985. Generalized best-first search strategies and the optimality of A. *Journal of the ACM (JACM)* 32, 3 (1985), 505–536.

[3] Edsger Wybe Dijkstra. 1959. Communication with an automatic computer. (1959).

[4] Robert W Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345.

[5] Vincent J Kovarik. 1994. An Efficient Method for Representing and Computing Transitive Closure Over Temporal Relations. (1994).

[6] Jyoti Nandimath, Ekata Banerjee, Ankur Patil, Pratima Kakade, Saumitra Vaidya, and Divyansh Chaturvedi. 2013. Big data analysis using Apache Hadoop. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE, 700–703.

[7] Charles Pierre Trémaux. 1865. Considérations sur les corps d'état appliquées à la guerre. *Annales des Ponts et Chaussées* 9 (1865), 5–110.

A Experiment

A.1 Source code

Codes are available on *Google Drive*.

B Responsibility

Table 1. Tasks assignment

	Shuyuan	Zekun	Xiaoyi
Data preprocessing		✓	✓
Random generation			✓
Experiment env setup	✓		
DFS algorithm		✓	
Warshall algorithm	✓	✓	
MapReduce algorithm	✓		