MapReduce for Temporal Transitive Closure

Ashish Goyal
Computer Science
Case Western Reserve University
Cleveland, Ohio
axg1503@case.edu

Shraddheya Vinod Tarekar
Computer Science
Case Western Reserve University
Cleveland, Ohio
sxt887@case.edu

Yash Malhotra
Computer Science
Case Western Reserve University
Cleveland, Ohio
yxm595@case.edu

ABSTRACT

A thorough evaluation of the performance of parallel (MapReduce) and sequential (Depth first Search & A* Heuristics Search) algorithms for calculating the temporal transitive closure in sequential & distributed computing settings. The goal of the work is to reconcile the differences between previous studies on the best transitive closure technique, with a particular emphasis on the Map Reduce framework implementations. There is a wealth of information about airports, aircraft, flights, schedules, and routes available in the Open Flight Data repository. This data can be shown as a set of tables or as a directed graph with routes marked with flight information as edges and airports as nodes. For both techniques, the R&D project offers comprehensive experimental findings and benchmarks on a directed graph. According to the results, MapReduce generally performs better than Sequential, especially for deep graphs. According to the results, MapReduce generally performs better than Sequential, especially for deep graphs. Additionally, the suggested algorithm aims to identify, between two airport nodes, the top five paths in terms of overall travel time, the top five paths disregarding layovers, and the top five paths in terms of number of paths.

KEYWORDS

Temporal Graph, Transitive Closure (TC), Indirect Paths, Directed Graphs, Graph Traversal, MapReduce, Scalability, Parallel processing, Resource Utilization

I. Introduction

The R&D Project addresses the need for a distributed computation framework for temporal transitive closure algorithms, particularly in the context of large-scale graph processing. It highlights the challenges faced in distributed, recursive evaluation of the transitive.

The aim of this project is to compute the temporal transitive closure (TC) on this data within a user-specified time range T. A temporal TC encompasses all node pairs (x,y) for which there exists a path from x to y, with start and end times falling within the defined time range *T*.

II. Overview and Methodology

A. Sequential Search

In a temporal network, a sequential search for temporal transitive closure entails methodically going through each pair of vertices to see if there is a time path connecting them. This method starts by exploring every path that could possibly come from a given vertex and then repeatedly extending these paths to nearby vertices within temporal bounds. During the process, the connection of each vertex that is encountered to other vertices within the given time period is assessed. The technique finds temporal transitive links by sequentially and exhaustively traversing all of the graph's vertices and edges. This ensures that temporal restrictions are upheld during the traversal. Although its exhaustive nature may result in higher computer complexity, sequential search offers a thorough approach to computing temporal transitive closure.

1) Uninformed search

Uninformed search algorithms, often known as blind search algorithms, are used for navigating the search area with no extra information regarding the goal's position other than identifying it when it happened. These algorithms search the

entire space systematically and exhaustively, making them useful for a variety of problems, yet they are frequently less efficient than informed search strategies due to a lack of heuristics. Common uninformed search algorithms are Breadth First Search(BFS) and Depth First Search(DFS). For this project, we attempted to implement the project statement using the Depth First Search algorithm.

a) Depth First Search(DFS):

Depth First Search(DFS) is a fundamental approach for traversing and searching tree or graph data structures. It thoroughly looks for all the possible paths along a branch of the graph before backtracking. Most commonly DFS is implemented with recursion or a stack data structure. DFS is an exploration strategy that involves exploring as deep as feasible along each branch before retracing. This means that it begins at a specific node(known as "root"). After this it explores each of its neighbors before going on to the next neighbor. To avoid returning nodes and becoming locked in infinite loops, DFS keeps track of visited nodes. Once a node has been visited, it is marked as "visited" to prevent further exploration. As discussed, DFS can be implemented with either a stack or recursion, in the stack-based technique, nodes to visit are added to a stack, but in the recursive approach, the function calls itself recursively to examine nearby nodes.

This project uses the Depth-First Search (DFS) algorithm to navigate a directed network of airports and aircraft connections. The method begins at a given source airport and iteratively investigates all feasible routes to a desired destination. Each path is assessed based on airline departure times and time constraints.

The dataset contains a detailed list of airports and the flights that connect them. Each flight link includes information about the airline, the source and destination airports, and the particular departure and arrival times. This data forms a directed network, with airports representing nodes and flights representing directed edges connecting them.

The search algorithm starts at the selected source airport and creates an initial path that includes only the source, with the current time as the starting time. The Depth-First Search (DFS) algorithm begins at the source airport and iteratively investigates all feasible routes to the desired destination.

While the stack is not empty, the algorithm selects the top element, which indicates the current airport, the path traveled thus far, the current time, and the depth of investigation. If the depth exceeds the maximum allowable depth, the algorithm skips this iteration to avoid too-deep pathways. If the present airport corresponds to the target airport, the algorithm adds it to the list of found paths. Otherwise, the program investigates all neighbors of the current airport by analyzing each outbound aircraft.Flights are evaluated if they depart later than the current time and do not return to an airport on the current routing. For each valid neighbor, a new tuple is added to the stack, containing the next airport, the modified path that includes the neighbor, the new arrival time, and the depth increment. This procedure assures that the algorithm systematically explores all feasible paths while avoiding cycles and sticking to time limits, making it an appropriate solution for navigating the directed network of airports.

If a flight exists and its departure time is within the requested time range, the path is modified to include the nearby airport. The algorithm then adds a new tuple to the stack, which includes the neighbor, the revised path, the next departure time, and the increased depth. This enables the stack to keep track of the progress achieved when investigating the flight connections. By iteratively analyzing each neighbor in this manner, the DFS algorithm assures that all valid paths are investigated while remaining under time limitations and avoiding revisits.

2) Informed Search Algorithm:

Informed search, also known as heuristic search, is a type of search algorithm that uses additional information, known as heuristics, to direct the search process more efficiently than uninformed or blind search methods. This added information enables smart searches to select viable paths, lowering the time and resources required to locate a solution. The heuristic function (h(n)) assesses the cost of traveling from node n to the goal, so providing the direction for the search by indicating which paths are most likely to lead there. The

evaluation function (f(n)) guides the search process by combining the heuristic estimate (h(n)) and the actual cost of reaching the current node (g(n)). A popular expression is as follow:

$$f(n) = g(n) + h(n). \quad \text{……..(eq 1)}$$

Here,
f(n) = estimate of the total cost from start node to the targeted node.
g(n) = actual cost from start node to node n.
h(n) = estimated cost from node n to target node.

A heuristic is considered admissible if it never overestimates the cost of achieving the goal, resulting in optimal solutions when utilized in particular algorithms such as A*. A heuristic is consistent if, for each node n and its successor n', $h(n) \leq c(n, n') + h(n')$, ensuring that the evaluation function (f(n)) never lowers along the path.

Informed search algorithms are effective for large-scale search issues because they use heuristics to favor more promising paths, lowering search times by discarding less promising paths first. The efficacy of these algorithms is strongly reliant on the quality of the heuristic function employed. When the heuristic is admissible and consistent, algorithms such as A* guarantee the discovery of optimal pathways.

a) A Algorithm:*

A* Algorithm is used to find the single-pair shortest path between a weighted graphs's start node(source) and destination node is found using the A* search algorithm. The algorithm uses heuristics (h) to estimate the cost that will be incurred from the current node to the target node in addition to taking into account the actual cost from the start node to the current node (g). The next node to move until it reaches the target node is then chosen to be the one with the lowest f-value. The A* algorithm, in which the heuristic is 0 for every node, is a specific case of Dijkstra's algorithm which can be written as f(n) = g(n).(From eq 4.1)

The A* algorithm formula takes into account the functions g(n) and h(n). Suppose we are at node n at this point. Then, g(n) gives us the real cost of getting from the start node to node n. A heuristic function called h(n) calculates how much it will cost to travel from the current node to the target node. As a result, h(n) is only a rough estimate, and it is vital to ascertain how well the A* algorithm performs. The algorithm is able to estimate the total cost from the start node to the end node based on the sum of the g(n) and h(n), f(n).

If a better path is found for nodes, the method chooses the lowest f-value node as the next current node to explore and keeps updating g(n) and h(n). Until the algorithm reaches the target node, this procedure keeps going.

In this project, we are implementing A* algorithm, and the heuristic we are using is geographic distance, as we are trying to find the distance over the globe.

3) Comparison of using A over DFS.*

There are several reasons to opt for informed search algorithms over uninformed search algorithms. Some of them can be defined as optimal solutions, heuristic guidance, time efficiency and so on. Let's talk more about the above reasons in detail.

Using A* algorithm, it ensures that the best course will be found if the heuristic is consistent. We have used Geographic Distance as an heuristic which can be an optimal heuristic. Whereas, for DFS it just delivers the first solution it finds. This solution may or may not be the quickest or the most effective route. It makes no guarantee that it will find an optimal path.

As discussed before, A* is using Geographic Distance as an heuristic which estimates the cost from the node to the goal. The algorithm uses this heuristic to give priority to the paths that seem more likely to succeed. However, with DFS, the path finding is aimless, which can lead to pointless solutions.

A* algorithm will always find a solution if there exists one. DFS is not perfect, it might not be able to solve graphs with a huge number of paths.

A* minimizes the total cost of searching a path and hence the search time is also reduced. DFS does not work with huge data and it might cause inconsistency.

B. MapReduce:

MapReduce was invented to deal with the processing of large data which is dispersed over computer clusters. Traditionally there was a centralized server to store and process data. This method was not suitable for large data causing bottlenecks while functioning with multiple files together.

MapReduce basically divides a huge block of work into smaller chunks of it and assigns them to different computers. At the end, the results from different systems get accumulated at one place and amalgamate to form the resulting solution.

The MapReduce algorithm can be done using mostly two concepts which are Map and Reduce. The Map task basically takes the block of data and it transforms this data into another set of data with each element separated into key value pairs. Then the result came through this, and is fed to the Reduce task. This Reduce task compresses the original set of the data which is a key value pair into a more manageable set. The Reduce task is always done after mapping the data together.

The end output is defined by an output formatter which converts the key value pair from the Reducer. This converted data is resulted into a file by using a record writer.

1) Map Reduce Algorithm for finding the Top 5 Worst 5 and All paths for 2 Given Airports (Nodes):

Based on user inputs, this code maps out flight paths and reduces them, showing the top 5 routes in terms of overall journey time. This is an overview:

1. Mapping Flights:Consistent departure and arrival timings for flights are generated by the `map_flights` function throughout the day. It adds flights that occurred within the given time range to the results list by iterating over the flight data.

2. Reducing Flights: - The `reduce_flights` function determines every feasible flight path, within a given time range, from a source airport to a destination airport.It explores every potential path using a recursive method (`find_paths`) and stores the results in `all_routes`. Total trip time is used to sort the routes.

3. Loading Data: CSV files are used to load route and airport data.
4. User Inputs: - The user enters the start time, duration of the time window, and IATA codes for the source and destination airports.

5. Mapping and Reducing: - Datetime objects are created from the user's inputs. Based on user inputs, flights are mapped and subsequently decreased.

6. Showing Outcomes: - If pathways are located, the software outputs each path's specifics, such as the start and end times as well as the overall journey time. The top 5 routes with the shortest overall travel time are also printed. It notifies the user if no pathways could be located

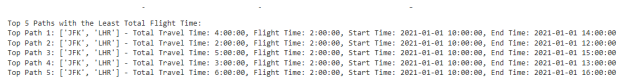


Figure: Top 5 paths according to least travel time

Based on inputs from the user, this code maps out flight paths and reduces them, highlighting the top 5 routes with the shortest overall flight time. This is an overview of the functions like reducing flights, total time calculation etc. remains the same.

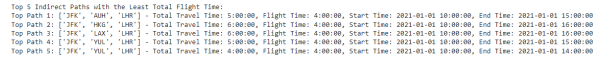


Figure : Top 5 Indirect Paths with the Least Total Flight Time

This code uses NetworkX and Matplotlib for visualization in order to map, reduce, and visualize flight pathways between airports. This is an explanation for visualizing the graph and paths by visualizers rest all functions remains as same:

1. Visualizing Paths: - Using Matplotlib and NetworkX, the `visualize_paths` function visualizes flight paths. It builds a directed graph with nodes standing in for airports, edges for flights, and edges for Visualizing all Available airports

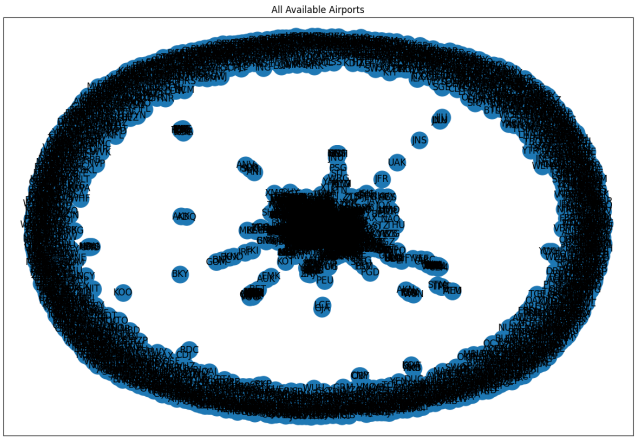
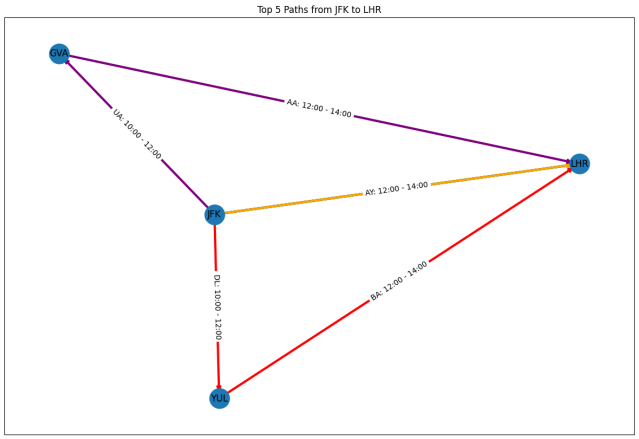


Figure: All available airports visualized as clusters based on geographical coordinates

III. Result and Discussion

The experimental results demonstrate that seminaive is the best choice for calculating the transitive closure of large graphs in the Map Reduce paradigm. The paper provides detailed performance comparisons for both algorithms on different graph types, highlighting the dominance of data volume cost over the number of rounds required, particularly for non-tree graphs.

Figure: Top 5 paths between Nodes JFK to LHR




```
Enter source airport IATA code: ORD
Enter destination airport IATA code: DEN
Enter start time (HH:MM, 24-hour format): 10:00
Enter duration of time window in hours: 1
Optimal path using A*: ['ORD', 'MCO', 'SFO', 'IAH', 'DEN']
Optimal path using MapReduce: ['ORD', 'DEN']
Time taken by A* Algorithm: 0.0042 seconds
Time taken by MapReduce Algorithm: 0.0027 seconds
```

Figure: Top 5 paths from JFK to LHR

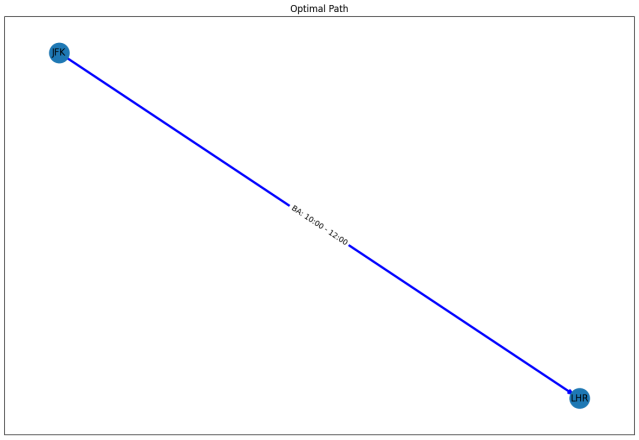


Figure: Optimal Path between 2 Nodes

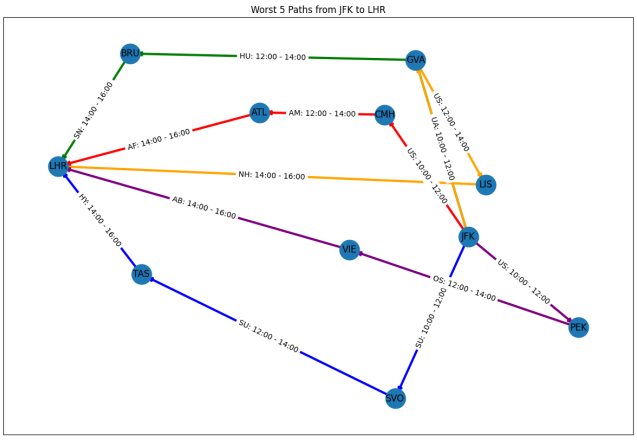


Figure: Worst 5 Paths between Nodes JFK to LHR

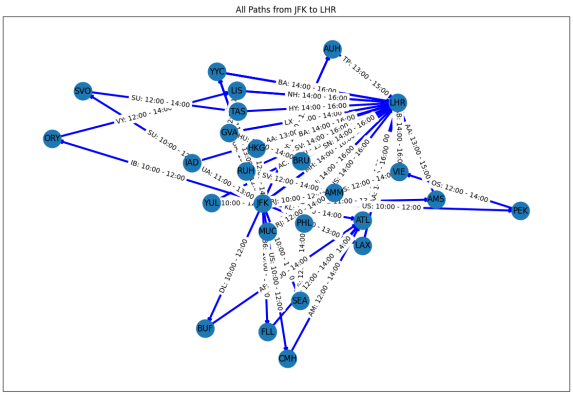


Figure: All paths from Nodes JFK to LHR

IV. CONCLUSION

Our goal is to bring more efficiency to the Map Reduce Paradigm and deliver scalable solutions for flight pattern analysis within time constraints by developing and implementing sequential and MapReduce algorithms for computing temporal TC on Open Flight Data.

For A* Heuristics search: -

- Complexity of Time: O (| V | log | V | + | E |)
- Memory Usage: High because of the upkeep of open and closed lists.
- Path Quality: The heuristic produces high-quality pathways.
- Scalability: Moderate; memory constraints may cause problems with very big graphs.

For MapReduce:

- Time complexity is expressed as O (| E | · d), where d is the longest possible path.
- Memory Usage: Because of iterative exploration, lower than A*.
- Path Quality: Moderate; this is contingent upon the caliber of paths in between. -
- Scalability: Excellent, able to manage distributed processing.

```
Enter source airport IATA code: ORD
Enter destination airport IATA code: DEN
Enter start time (HH:MM, 24-hour format): 10:00
Enter duration of time window in hours: 1
Optimal path using A*: ['ORD', 'MCO', 'SFO', 'IAH', 'DEN']
Optimal path using MapReduce: ['ORD', 'DEN']
Time taken by A* Algorithm: 0.0042 seconds
Time taken by MapReduce Algorithm: 0.0027 seconds
```

Figure: Comparative analysis of the execution time of A heuristics Search and Map Reduce algorithm.*

REFERENCES

[1] Eric Gribkoff University of California, Davis Distributed Algorithms for the Transitive Closure

[2] F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman. Map-reduce extensions and recursive queries. In Proceedings of the 14th International Conference on Extending Database Technology, pages 1–8. ACM, 2011.

[3] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In Proceedings of the 15th International Conference on Extending Database Technology, pages 132–143. ACM, 2012.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. The VLDB JournalThe International Journal on Very Large Data Bases, 21(2):169–190, 2012.