Distance-Vector Routing With SQL

Using recursive CTEs to solve the Distance-Vector Problem

Alexej Onken
alexej.onken@student.uni-tuebingen.de
Eberhard Karls Universität
Tübingen, Baden-Württemberg, Germany

ABSTRACT

"Recursive queries are useful for working with data that is organized in a hierarchical or tree-like manner", that's how the official PostgreSQL documentation similarly promotes the use of recursive queries [2]. In this article, we explore the solution of Distance-Vector Routing with SQL and compare it with the established method, which employs the Bellman-Ford algorithm. The Distance-Vector algorithm aims to calculate the shortest path from each node to every other node by having each router use its routing table to communicate exclusively with its immediate neighbours. We study the solution of the Distance-Vector problem using a recursive CTE (Common Table Expression) that filters exploding path discoveries through appropriate upper bounds and cycle detections to avoid exponential run times in certain cases. This approach is scrutinized in comparison with the established method that performs periodic updates in routing tables. Here, the implementation of the recursive CTE and the accompanying SQL script are explained in detail. We then investigate whether using recursive CTE with SQL can be practically useful in solving the Distance-Vector Routing problem.

KEYWORDS

SQL, Distance-Vector Routing, recursive CTEs, Bellman-Ford algorithm, routing tables

1 INTRODUCTION

In network research, the Distance-Vector Routing protocol is used to determine shortest paths from any node to another node [4]. It is also called a dynamic protocol as it can compute new routes independently in the event of changes in the edge weights between nodes [4]. The conventional Bellman-Ford algorithm is used to solve this problem. Since the data of each router is already available in tabular form, a solution approach using recursive CTEs is found in this article. Recursive CTEs are a special case of CTEs (temporary tables) that allow for self-referencing and iterative querying. [2]
A recursive CTE allows iterative execution of a custom SQL query, involving four tables in the process: Union Table (UT), Intermediate Table (IT), Work Table (WT), and Hash Table (HT), with the keywords `UNION` and `UNION ALL` governing the combination of results. The `EXPLAIN ANALYZE` command reveals insights into the query plan and the roles of WT, IT (not explicitly), and UT. The UT holds all records from the initial query (non-recursive part) and the recursive query. The IT temporarily stores results during iterations, and WT is utilized to execute the recursive query. The HT is used to prevent duplicates when using `UNION` and is omitted with `UNION ALL` (our case). The process unfolds as follows: First, the initial query results are stored in WT and UT. The recursive query is executed using WT, and the results are stored in IT and UT. WT is reset, and IT's contents are copied into WT before resetting IT. This process iterates until no new records are added. The recursive CTE implements an iteration by working incrementally and referencing itself to access values from previous iterations. This streamlines the process of traversing a graph, consolidating it into a single, easily readable query. Upon recursive CTE convergence, desired solutions can be obtained through a separate query.
The SQL script is made using PostgreSQL, a free open-source Relational Database-Management-System (RDBMS). The utilization of PostgreSQL in conjunction with the capabilities of Turing completeness enables smooth data integration, filtering and organization in tables. PostgreSQL adheres to the latest SQL standards and also provides advanced features like the capability to execute recursive queries as well as window functions, which are crucial components of the SQL script in this implementation.
In this article, we outline the specific PostgreSQL script employed in this approach and evaluate its time complexity in comparison to the established Bellman-Ford method. Additionally, we will delve deeper into the Branch-and-Bound method which helps to minimize the search space being examined. This technique minimizes the search space by trimming unnecessary branches and constantly establishing new milestones of the explored paths during each recursion step. Our analysis aims to assess the feasibility of using recursive CTEs to solve Distance-Vector Routing problems, and to compare its effectiveness to the traditional Bellman-Ford algorithm.

2 DISTANCE-VECTOR ROUTING WITH BELLMAN-FORD

The Distance-Vector Routing algorithm is a decentralized network protocol that collects and forwards information about the shortest paths from source to destination nodes. Each node can be considered as a router with a routing table that stores known costs to each other destination node. The network protocol operates based on the principle of exchanging routing information between neighbouring routers. This involves each router periodically transmitting its own routing table to its neighbours, while simultaneously receiving routing tables from its neighbours.
The guiding principle behind this protocol can be expressed as, *"Tell your neighbours how you see the network world"*. In Figure 1 and Figure 2, an example graph G_1 is presented [5], displaying the preliminary routing table associated with node A. The Bellman-Ford algorithm allows the router to update its own routing table based on the routing tables received from its immediate neighbours, so

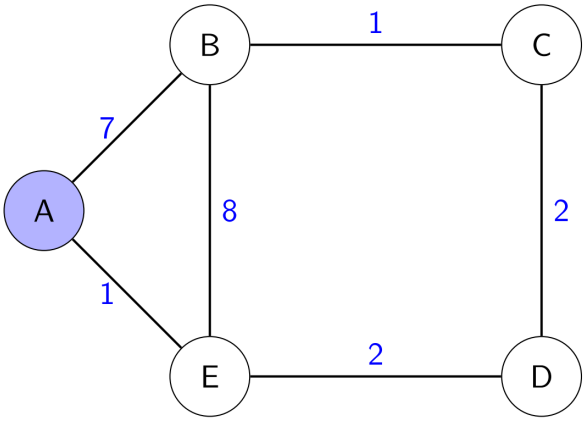

Figure 1: Illustrative graph G_1

Node A's routing table

VIA	TO B	C	D	E
B	7	–	–	–
C	–	–	–	–
D	–	–	–	–
E	–	–	–	1

Figure 2: Node A's routing table

that all edges can be relaxed in each iteration if a shorter distance to a destination router is discovered. Upon successful completion of the algorithm, the shortest paths from each source to their corresponding destinations are stored in their respective routing tables. However, the Bellman-Ford algorithm also has some drawbacks. If a connection breaks down or a new router is added, new information in the network can only spread slowly, as routers must wait for updated routing tables to be exchanged with their respective neighbours, resulting in suboptimal routes in the meantime. Routing loops can also occur when routers send packets with outdated information, leading to delays and congestion [3].

Let $G = (N, E)$ be an undirected, connected graph where N represents the set of nodes, and E represents the set of edges. Let $d_X(Y)$ represent the shortest path distance between nodes X and Y, where $X, Y \in N$ and all edge weights in E are non-negative. Each node $X \in N$ can be considered as a router, which has its own routing

Initial configuration

FROM	TO A	B	C	D	E
A	0	7	–	–	1
B	7	0	1	–	8
C	–	1	0	2	–
D	–	–	2	0	2
E	1	8	–	2	0

Figure 3: Optimal shortest paths

table. Each routing table in its simplest state only possesses knowledge about its direct neighbours $V \in N$, and the cost c associated with reaching those neighbours. In each iteration, every router sends its own routing table to its neighbours, who in turn update their routing tables based on the received information. Specifically, for each node $X \in N$, its routing table is updated according to the Bellman-Ford Equation:

$$d_X(Y) = \min_V c(X, V) + d_V(Y) \tag{1}$$

where $c(X, V)$ is the cost of the edge between nodes X and V, and $d_V(Y)$ is the shortest path distance from node V to node Y as calculated by its own routing table.

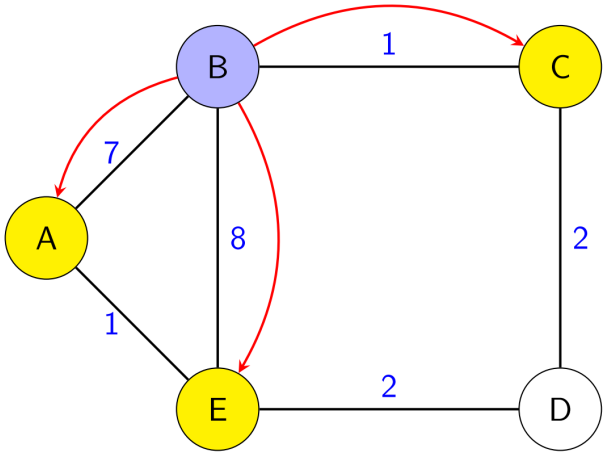

Figure 4: Optimal shortest paths

The Bellman-Ford Equation serves as a guide for updating the routing tables in each iteration. After sufficient iterations, the shortest path distances in each routing table will converge to their true values. The initial naive configuration is illustrated in Figure 3, where

		TO			
FROM	A	B	C	D	E
A	0	7	8	-	1
B	7	0	1	-	8
C	8	1	0	2	9
D	-	-	2	0	2
E	1	8	9	2	0

Figure 5: Optimal shortest paths

each node is aware only of the distances to its adjacent neighbours. Figures 4 and 5 demonstrate the update process during an iteration. Specifically, Figure 4 illustrates how node B communicates its routing table to nodes A, C, and E. It should be noted that during each iteration, each node (in Figure 4, we only consider Node B) updates its routing table based on the routing tables it receives from its direct neighbors. The iteration is considered complete when all nodes have updated their routing tables. Typically, multiple iterations are required for convergence, as mentioned earlier. On the other hand, Figure 5 highlights the changes made during the transition, as indicated by the red numbers, in comparison to the initial configuration in Figure 3. Figure 6 shows the optimal shortest paths for the same graph in Figure 1.

		TO			
FROM	A	B	C	D	E
A	0	6	5	3	1
B	6	0	1	3	5
C	5	1	0	2	4
D	3	3	2	0	2
E	1	5	4	2	0

Figure 6: Optimal shortest paths

The cost tables are symmetric due to the symmetry property of distance, which states that $d_X(Y) = d_Y(X)$. It is important to note that the final optimal cost table does not correspond to the optimal routing table since each node has its own routing table. Rather, the column maxima over all neighbours V_i in the optimal routing tables are the final optimal paths.

The runtime depends on the network structure and the number of iterations required for updates. It is assumed that the number of iterations is proportional to the number of nodes in the network. If two nodes in a network are far away from each other, it may take several iterations for the information to propagate from one node to the other. For instance, if the maximum number of hops between two nodes in a network is 3, the algorithm may take at most 3 iterations to propagate information from one router to another. In the worst case, if the longest hop count between two nodes is $n-1$ (where n is the total number of nodes in the network), it may take up to $n-1$ iterations for the information to reach all nodes in the network. Therefore, the number of iterations required for convergence depends on the length of the maximum shortest path (in terms of hop count and cost) between any two nodes in the network, which is often referred to as the diameter D of the network [7].

D is particularly useful in estimating the average runtime of the Bellman-Ford algorithm in Distance-Vector Routing because the number of iterations required for convergence is directly proportional to the diameter of the network. Each iteration of the Distance-Vector Routing algorithm involves checking every edge in each node's routing table, resulting in $\mathcal{O}(n \cdot |E|)$ operations per iteration. The hop count D of the network, representing the longest number of hops along the shortest paths between any two nodes, is proportional to the number of iterations required for convergence in Distance-Vector Routing. It is worth mentioning that our assumption entails parallel computation, whereby each router executes updates autonomously, due to the decentralized character of Distance-Vector Routing. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(D \cdot n \cdot |E|)$. Since D is proportional to the number of iterations, we can write the time complexity as $\mathcal{O}(D \cdot n \cdot |E|)$ or $\mathcal{O}(n^2 \cdot |E|)$ in the average case, where n is the number of nodes in the network.

During each iteration of the algorithm, every node forwards its routing table to its neighbours and updates its own routing table based on the received information. In a network with n nodes, each node has at most $n-1$ neighbours, and thus, sends and receives $n-1$ messages in each iteration. In total, $\mathcal{O}(n \cdot |E|)$ operations are performed in each iteration, as each edge in every routing table is checked. Given that each node potentially updates its routing table at least once, there are n iterations. The worst case scenario for the Bellman-Ford algorithm in Distance-Vector Routing results in a runtime of $\mathcal{O}(n^2 \cdot |E|)$. In the best case for Bellman-Ford in Distance-Vector Routing, initial optimal routing tables require only a single network iteration, yielding a time complexity of $\mathcal{O}(|E|)$.

3 PLAN OF ATTACK WITH SQL

To determine the shortest paths from each node to every other node in a graph using SQL, we first define the schema of the input graph as $graph(from, to, via, cost)$. We then perform a recursive query on the given graph. Here, we explore all possible paths from a node while cutting off unprofitable paths using window functions. It is also of great importance to ensure that no cycles occur during this process. We also record the sum of the costs of the edges recorded by our explorers. Explorers are sent on a journey to explore all possible travel routes for the prescribed and legal triples $(from, to, via)$.

During their exploration, they log the costs of their travel, including the stopovers, and compare them with each other at the end. Legal triples are determined by the requirement of a mandatory first stopover at a direct neighbour from a starting point. Once the recursion process is complete, we can extract the shortest paths for all tuples $(from, to)$ $d_{X_i}(Y_i)$ by computing the minimum of the shortest paths over all legal triples for a specific $d_X(Y)$:

$$d_{X,V}(Y) = \min\{d_{X,V_1}(Y), \ldots, d_{X,V_n}(Y)\} = d_X(Y) \tag{2}$$

An initial sample input table for the Distance-Vector Routing algorithm is depicted in Figure 7. This can be imagined as the collection of all routing tables in the network stacked on top of each other. Initially, each router in the network is assumed to only have knowledge of the costs $c(X, V_i) \neq \infty$ to its immediate neighbours, representing the most basic state. The *via* column therefore represents a direct neighbour V_i, provided that the cost entry is not infinite. Nodes

from	to	via	cost
A	B	B	3
A	B	C	Inf
A	C	B	Inf
A	C	C	23
A	D	B	Inf
A	D	C	Inf
⋮	⋮	⋮	⋮
D	C	C	5

Figure 7: Sample input table for SQL graph

that are far away from any reference router, characterized by a hop count greater than 1, or those for which the entries in the *via* column are not equal to the *to* column, are labeled with a cost value of infinity (Inf) to indicate their unknown state. The accompanying graph G_2 is depicted in Figure 8.

We choose this approach to make the Bellman-Ford Equation 1 more intuitive. The recursive part of the Bellman-Ford Equation is solved using explorers that are sent on the journey from each legal triple. The non-recursive part (first term in the Bellman-Ford Equation 1) is solved in a final transformation after the shortest paths are computed over the converged recursive CTE.

To keep the exponential nature of this approach in check, it is essential to disqualify explorers whose travel costs are too expensive. This can be achieved using a Branch-and-Bound method by cutting off and not pursuing the explorers with the highest costs. While this approach can significantly reduce the number of searched paths and improve efficiency for finding shortest paths in large networks with many nodes and edges, it is important to note that there is no guarantee that the search space will always be reduced.

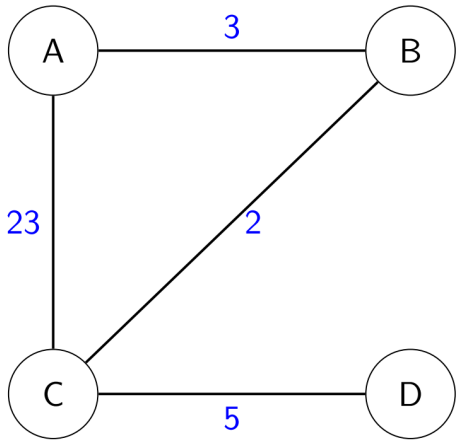

Figure 8: Sample graph G_2 [6]

4 STRUCTURE OF THE RECURSIVE CTE

The SQL code provided employs a recursive CTE to compute the shortest paths between routers in an undirected graph. The recursive CTE comprises of two main parts: the non-recursive part and the recursive part, followed by a final transformation using the converged recursive CTE. Furthermore, the repository https://github.com/jexela/Distance-Vector-Routing-With-SQL contains the code for generating two sample graphs G_1 and G_2. In addition, a User-Defined Function (UDF) named `array_smallest` is provided, which computes the minimum value from an array. This repository offers an opportunity to test the SQL implementation.

4.1 Non-Recursive Part

In the non-recursive section of the CTE (base case), we prepare an appropriate initial table to start the calculation of the shortest paths. The recursive CTE code can be found on the subsequent page.

Lines 2-8 of the code split legal triples into static and dynamic components. The static portion always informs the explorers of their origin (`from`), the first mandatory stopover (`first_stopover`), and the final destination of their journey (`to`). The dynamic part, on the other hand, offers a snapshot to aid orientation during recursion. Line 8 logs the travel costs to the current stopover. Legal triples during table initialization are ensured by input graph population, see Figure 7.

Lines 9-16 generate the previous itinerary. An explorer who reaches their final destination should record `'FINISHED'` in their travel log. Otherwise, they should add the current stopover to their previous itinerary. Lines 17-20 establish the basis for Branch-and-Bound, which ensures that explorers who have not yet reached their destination do not have any optimal scores to display, and thus receive a value of `'infinity'`. However, triples where `destination=via` can be directly considered as the shortest paths.

Lines 21-25 employ a self-join to guarantee that the costs to the first mandatory stopover from line 8 are appropriately transferred. By using the `UNION ALL` keyword, we can ensure the transition to the recursive part of the CTE, and since no duplicates are to be assumed, we can omit `UNION` for faster execution.

```
1  WITH RECURSIVE exploration as (
2      SELECT g.origin as from,
3             g.destination as to,
4             g.via as first_stopover,
5             g.origin,
6             g.destination,
7             g.via,
8             e.cost as cost_next_hop,
9             CASE WHEN g.destination = g.via
10                 THEN array[g.origin] ||
11                      array[g.via]    ||
12                      array['FINISHED']::VARCHAR[]
13                 ELSE array[g.origin] ||
14                      array[g.via]
15            END as track,
16            e.cost as total_cost,
17            CASE WHEN g.destination = g.via
18                 THEN g.cost
19                 ELSE 'infinity'
20            END as branch_and_bound
21     FROM graph as g, graph as e
22     WHERE g.origin      = e.origin AND
23           g.via         = e.via    AND
24           e.destination = e.via
25 UNION ALL
26 ...
```

4.2 Recursive Part

```
1  ...
2  SELECT
3         .
4         .
5         .
6         CASE WHEN e.to = g.via
7              THEN e.track ||
8                   array[g.via, 'FINISHED']
9                        ::VARCHAR[]
10             ELSE e.track || array[g.via]
11        END as track,
12        e.total_cost+g.cost as total_cost,
13        array_smallest(
14        array[e.branch_and_bound]::float[] ||
15        array[min(CASE WHEN e.to = g.via
16                       THEN e.total_cost+g.cost
17                       ELSE 'infinity'
18                       END) over win]::float[]
19        ) as branch_and_bound
20 FROM exploration as e, graph as g
21 WHERE e.via         = g.origin                AND
22       g.destination = g.via                   AND
23       'FINISHED'   <> ALL(e.track)            AND
24       (SELECT cardinality(array_positions(
25               e.track[2:],g.via)
26               ) < 1)                          AND
27       e.total_cost <= e.branch_and_bound
28 WINDOW win as (
29 PARTITION BY e.from, e.to, e.first_stopover
30 ORDER BY e.total_cost
31 RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
32       FOLLOWING)
```

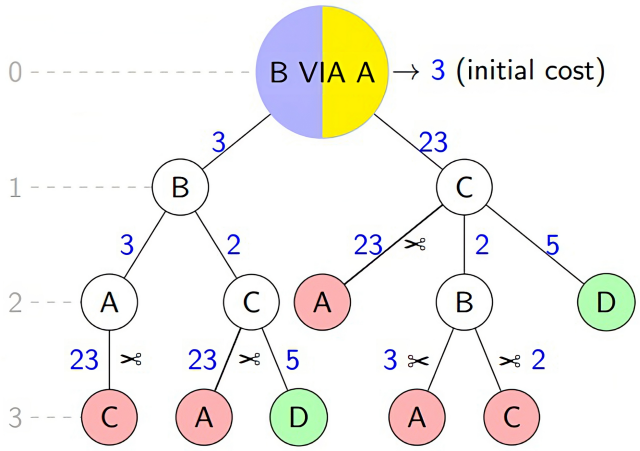

Figure 9: Optimize search tree efficiency for G_2 (Figure 8)

The recursive part of the CTE starts by selecting the origin triple (`from, to, first_stopover`) from the working table named `exploration` and fetching the snapshot (`origin,destination, via`) from the input graph. The total cost of the explored path is calculated along with the travel log in lines 6-12, which is similar to lines 9-15 in the non-recursive part.

The code between lines 13-20 represents the core of the Branch-and-Bound method and uses the window function (`win`) defined between lines 28-32. Figure 9 depicts the search tree and shows that, in each recursion stage, for each legal origin triple (`from, to, first_stopover`), it is checked within each `PARTITION` (windows do not cross partitions) whether there are any discoverers who have reached their final destination (`to`), enabled by `e.to=g.via`. If a discoverer has reached their destination, the total cost is calculated; otherwise, `'infinity'` is set. Then the minimum of all finished paths of the current recursion level is calculated, and the winner competes against the previous champion of all recursion levels, as indicated by `array_smallest`. The previous winners of all recursion stages are stored in the `branch_and_bound` column. The working table `exploration` is joined with the input graph, referred to as `graph`.

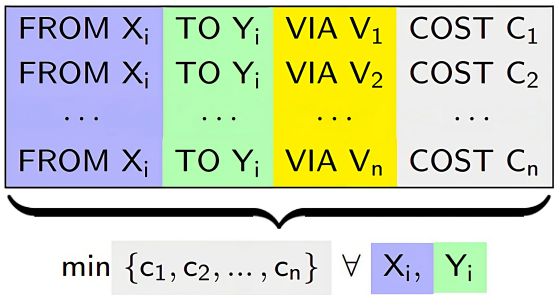

Figure 10: Last transformation after recursion

The `WHERE` statement provides a dual purpose: It not only serves as a termination condition for the recursive CTE, but also functions as a filtering mechanism for selecting relevant records. This filtering process enables the query to explore only the most promising paths, which have the potential to be lucrative.
The opening condition (line 21) secures that the next step (`e.via`) in `exploration` commences at the vertex (`g.origin`) extracted from `graph`, which in turn provides additional information about the following adjacent nodes. This ensures the logically correct chaining of nodes in the graph while the explorer traverses through the map.
The second condition in line 22 guarantees that `'infinity'` values are not incorporated, requiring `g.destination = g.via` to hold true. The third clause in line 23 validates that the record does not already have the `'FINISHED'` marker in the track array, indicating that the final destination (`e.to`) has already been reached by the explorer. The fourth clause (lines 24-26) is responsible for detecting cycles during the exploration process by checking whether the next stopover has already been visited. This verification is conducted through the travel log (`e.track[2:]`), starting from the first mandatory stopover (`e.first_stopover`) to prevent cycles. Although not necessary, we chose this approach to make the Bellman-Ford Equation 1 more tangible in our recursive CTE implementation. Alternatively, we could have started cycle detection from the beginning of the travel log (`e.track[1:]`). The fifth clause in line 27 guarantees that the present exploration path is not pricier than the best-known path discovered so far. This is done by modifying the `branch_and_bound` array, which retains the smallest total cost of each subpath for each legal origin triple (`from, to, first_stopover`, lines 13-19). In this scenario, it is important to ensure that the total costs of the exploratory paths, denoted by `e.total_cost<=e.branch_and_bound`, do not exceed the upper bound.
Finally, using the converged recursive CTE `exploration`, the shortest paths are calculated for all (X_i, Y_i) tuples as shown in Figure 10. To achieve this, a simple query can be constructed by selecting the exploration table and segmenting (`e.from, e.to`) using `GROUP BY`, and then applying the aggregate function `min` to compute the minimum value of the `e.total_cost` column over all tuples. Output tables for G_1 and G_2 can be found on pages 34-35 of the PDF slides (`Distance_Vector_Routing_With_SQL.pdf`) in the repository.

5 DISCUSSION AND CONCLUSION

Distance-Vector Routing relies on nodes communicating with neighbours and propagating routing tables to determine shortest paths. E.g., the Routing Information Protocol (RIP) uses Bellman-Ford algorithm for path calculation. RIP was one of the first Distance-Vector Routing protocols used on the internet but has since been largely replaced by more advanced protocols such as Border Gateway Protocol (BGP). [1]
The time complexity of the Bellman-Ford algorithm is $\mathcal{O}(n^2 \cdot |E|)$, where n is the number of nodes and $|E|$ is the number of edges in the graph. The recursive CTE SQL implementation for Distance-Vector Routing relies on several factors, including: The number of nodes (n), the average outdegree of the router network (d_{avg}, which in turn depends on the number of edges $|E|$), and the number of legal triples ($N_{l,triples}$) from which the algorithm starts exploring. The estimated time complexity is $\mathcal{O}(n^{d_{avg}} \cdot N_{l,triples})$, with approximately d_{avg} possible turns at each intersection during exploration and a maximum path length of n or $n+1$ (as in our implementation). By incorporating techniques such as Branch-and-Bound filtering, which involves computing upper bounds for each window of legal triples, and cycle detection methods, it is possible to enhance the performance of the recursive CTE SQL algorithm for Distance-Vector Routing. However, their effectiveness will depend on the specific network structure and the costs associated with each path, which can vary greatly from network to network. As such, it is difficult to predict the exact impact on runtime. Although not accounted for in the time complexity analysis, it is important to note that the computations involved in the window functions, implicit joins and array processings can also be highly time-intensive. An alternative solution could be a decentralized approach using SQL, although it has not been obviously addressed in this article.
In contrast, the Bellman-Ford algorithm for Distance-Vector Routing is a distributed approach that enables each node in the network to calculate the shortest path to other nodes by only communicating with its adjacent nodes. This property makes it highly scalable. On the other hand, the recursive SQL implementation requires a centralized database and may not be as efficient for larger networks. Moreover, the Bellman-Ford algorithm is particularly suitable for networks with dynamic topology changes, as it can quickly adapt to such changes.
The Bellman-Ford algorithm, unlike the recursive CTE SQL approach, does not have access to a travel log, which can result in revisiting nodes multiple times and potentially creating cycles. This is because the algorithm updates the distance estimate of a node based on the distance estimates of its neighbours, which in turn can depend on the distance estimate of the current node. This process of updating distances is repeated for all nodes in the graph and can cause the algorithm to revisit nodes whose distance estimates have already been updated, potentially leading to cycles. Additionally, the travel log in the recursive CTE SQL approach allows us to see the entire route of the shortest journey from any starting router to any destination. Besides, when using recursive CTEs, it is possible to compute shortest paths in a single query, whereas the decentralized Bellman-Ford approach requires each node to communicate with its neighbors to update its routing table, leading to a considerable amount of network traffic.
The Bellman-Ford algorithm can be slow in networks with many nodes and few connections due to revisiting nodes multiple times, causing network traffic and performance issues. In contrast, the recursive CTE SQL approach considers only relevant edges, leveraging network sparsity and generating less traffic, making it faster in some cases. On the flip side, the recursive SQL approach's exponential nature can be a hindrance in large networks. Furthermore, it is important to note that the space complexity of the SQL implementation has not been considered in this analysis, and as such, the recursive CTE can become very large until it reaches convergence. Ultimately, the most suitable approach for solving Distance-Vector Routing problems depends on the specific network structure being examined.

REFERENCES

[1] Kate Brush. 2022. *Routing Information Protocol.* https://www.techtarget.com/searchnetworking/definition/Routing-Information-Protocol Last Accessed on March 13, 2023.

[2] The PostgreSQL Global Development Group. 2023. PostgreSQL 14 Documentation: Common Table Expressions. https://www.postgresql.org/docs/14/queries-with.html#QUERIES-WITH-RECURSIVE. Last Accessed on March 13, 2023.

[3] C. Hedrick. 1988. *Routing Information Protocol.* Technical Report. https://tools.ietf.org/html/rfc1058 Last Accessed on March 14, 2023.

[4] James F. Kurose and Keith W. Ross. March 5, 2012. *Computernetzwerke: Der Top-Down-Ansatz* (6 ed.). Pearson Studium. 371–379 pages.

[5] Stefan Savage. [n. d.]. *Lecture 8: Routing I - Distance-vector Algorithms.* CSE 123: Computer Networks, University of California, San Diego. https://cseweb.ucsd.edu/classes/fa11/cse123-a/123f11_Lec9.pdf Last Accessed on March 13, 2023.

[6] Wikipedia. 2023. *Distance-vector routing protocol.* https://en.wikipedia.org/wiki/Distance-vector_routing_protocol Last Accessed on March 15, 2023.

[7] Wikipedia. 2023. Network science. https://en.wikipedia.org/wiki/Network_science Last Accessed on March 19, 2023.