Comparative Implementation of Shortest Path Algorithm in Python and Neo4j

Kusum Sai Chowdary[1*], Shanun Randev[1*], Tushar Sharma[1*]
[1]Columbian College of Arts and Sciences, George Washington University, Washington D.C., USA
[*]all authors contributed equally

***Abstract*— This study investigates the implementation and comparative performance of Dijkstra's algorithm using Python-based methods and Neo4j database integrated with the APOC library. The custom priority queue-based Python approach and Python's NetworkX library exhibit linear runtime dependencies, with NetworkX showcasing superior efficiency. In contrast, Neo4j's APOC library consistently outperforms in computational efficiency, especially in larger graph settings. Despite its robust performance, APOC displays algorithmic runtime variability, potentially linked to caching or other database features. These results emphasize the nuanced decision-making process when choosing between Python-based solutions and database-specific tools for optimal graph processing in diverse applications. The study sheds light on the intricate trade-offs inherent in selecting appropriate tools for effective graph algorithm implementation and optimization.**

Keywords—Dijkstra's algorithm, NetworkX, Neo4j, APOC

I. Introduction

In the intricate landscape of network analysis, the exploration of efficient algorithms for finding the optimal paths play a crucial role in optimizing various real-world scenarios. Among the plethora of algorithms available, this study zeroes in on the Dijkstra algorithm [1], a renowned method for determining the shortest path in a graph. The application of this algorithm is particularly compelling when addressing complex networks where optimizing traversal routes hold paramount importance. Its widespread use extends beyond traditional computer science applications and has found practically in fields such as transportation logistics, network routing and even in modeling social relationships [2, 3].

Graph Theory, with its roots in operations research provides a powerful framework for modeling complex relationships among entities. Networks represented as graphs offer a visual abstraction of these relationships, paving the way for algorithmic solutions to fundamental problems. Of special interest is the application of the Dijkstra algorithm to Directed Acyclic Graphs (DAGs) where the absence of cycles introduces unique challenges and characteristics, making it an intriguing area of exploration. Previous studies have compared Relational Database Management Systems (RDBMS) with Graph Database Management Systems (GDBMS) in context of shortest path problems [4, 5]. Some studies assess different graph query languages with novel processing techniques [5].

The significance of this study lies not only in the exploration of the Dijkstra algorithm but also in its real-world application. Dijkstra's ability to find the shortest paths between locations is leveraged in GPS navigation systems, ensuring efficient travel routes for commuters and delivery services. Furthermore, in the realm of telecommunication networks, Dijkstra plays a pivotal role in optimizing data routing paths, enhancing speed and reliability of data transmission. Its application is not confined to technological landscapes alone.

This study employs a Python-based implementation of the Dijkstra algorithm tapping into the capabilities of the Neo4j graph database, specifically utilizing the APOC (Awesome Procedures on Cypher) library. The choice of Neo4j stems from its aptitude for handling graph data, making it a fitting platform for evaluating the performance of pathfinding algorithms [6]. The comparative analysis with this study seeks to unravel the strengths and limitations of different environments (a database and a non-database). An integral aspect of evaluating the effectiveness of the Dijkstra lies in understanding the computational characteristics. Through this exploration we aim to provide insights into the algorithm's efficiency and applicability.

II. Methodology

A. *Dijkstra's algorithm for shortest path*

Dijkstra's algorithm is a fundamental graph theory algorithm employed for finding the shortest paths between nodes in a weighted graph. The algorithm systematically explores the graph, updating the shortest path to each node from a designated source. It operates based on the principle of greediness, selecting the node with the smallest tentative distance at each step. This efficiency makes it a popular choice for solving pathfinding problems in various applications. However, Dijkstra's algorithm cannot handle negative weights and could be applied only to graphs with positive weights. The time complexity of Dijkstra's algorithm is $O((V + E)\log V)$, where V is the number of vertices and E is the number of edges in the graph. Fig 1. Shows the time complexity variation of Dijkstra's algorithm. In the algorithm, the path is unknown at a given point. The nodes are categorized into two groups: temporary (t) and permanent (p). To begin, the distance of the source node is set to zero [distance(a) = 0], and the distance for all other nodes is initially set to infinity [distance(x) = ∞]. In Step 2, we look for the node x with the smallest $d(x)$ value. If there are no temporary nodes remaining or if the $d(x)$ value is equal to infinity, the node x is marked as permanent. This indicates that both the $d(x)$ and the parent of $d(x)$ will remain unchanged from this point onward. In Step 3 of Dijkstra's algorithm, after marking the node x as permanent, the algorithm updates the tentative distances to the neighboring

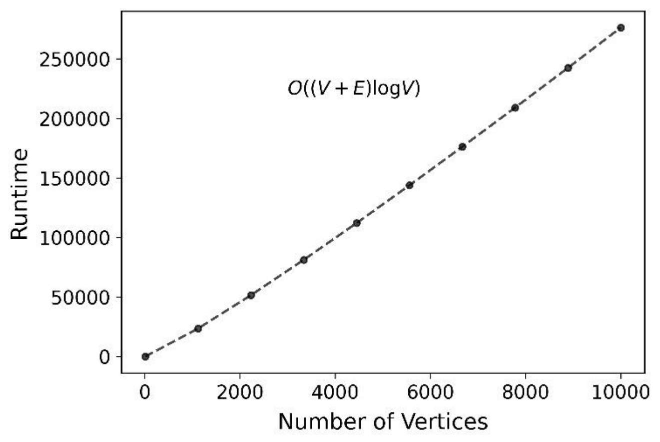

Fig. 1. Time complexity of Dijkstra's algorithm with increasing number of vertices (nodes) in graph.

nodes of *x*. For each neighbor *y* of *x*, it checks whether the path through *x* to *y* is shorter than the currently known distance to *y*. The criteria for updating the tentative distance to a neighboring node *y* is given as:

$$d(y) = \min(d(y), d(x) + w(x, y)) \qquad (1)$$

where, $w(x, y)$ is the weight associated with edge from node *x* to node *y*.

The workflow of the algorithm is summarized in Table 1.

To illustrate the working of Dijkstra's algorithm, an example problem has been shown in Fig. 2. and Table II.

B. Random graph generation

To evaluate the performance of the implemented algorithms, random connected graphs are generated with two different numbers of nodes: 1000 and 10,000. The connections in the graph are probability-based meaning there is a probability associated with having a connection between any two nodes in

TABLE I. DIJKSTRA'S ALGORITHM PSEUDO-CODE

Dijkstra's algorithm

```
function Dijkstra(Graph, source):
    create empty priority queue Q
    create empty set S
    create array distance[] and set all distances to infinity
    set distance[source] to 0
    enqueue source with distance[source] into Q

    while Q is not empty:
        current = dequeue the node having minimum distance from Q
        add current to S

        for each neighbor, weight of current:
            if neighbor is not in S:
                calculate tentative distance from source to neighbor

                if tentative distance < distance[neighbor]:
                    update distance[neighbor] to tentative distance
                    enqueue neighbor with distance[neighbor] into Q

    return distance
```

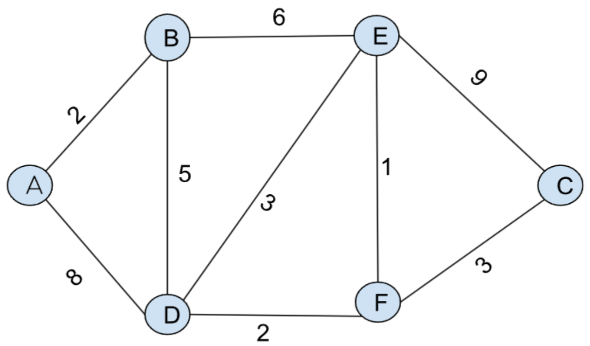

Fig. 2. Example problem graph for Dijkstra's algorithm. A to F are nodes connected with positive weighted edges.

TABLE II. EXAMPLE PROBLEM SOLUTION

Node	Shortest distance	Previous node
A	0	
B	∞, 2	A
C	∞, (8 + 9 = 17), (9 + 3 = 12)	E, F
D	∞, 8, (2 + 5 = 7)	A, B
E	∞, (2 + 6 = 8)	B
F	∞, 9	D

the graph. The probability of connection for the 1000-node graph ranges from 0.005 to 0.1, while for the 10000-node graph, the graph sizes vary from 0.0005 to 0.01. Thus, for the graph with 1000 nodes, when the probability is 0.1 (i.e., 10% possibility of a connection), it implies there are 1000 × 1000 × 0.1 = 100,000 total connections in the graph. Similarly, for the graph with 10,000 nodes, when the probability is 0.01, the total number of connections in the graph would be 10,00,000. This variation in graph sizes and characteristics aims to provide a comprehensive assessment of the algorithms under each scenario. For each edge, a random positive weight between 1 and 10 is assigned. It is also ensured that there is no disconnectivity in the graph meaning there are no disjoint subgraphs or orphan nodes in the entire graph. Every node has at least one edge, either an incoming or an outgoing edge with at least one other node. A sample graph based on the above implementation for 50 nodes is displayed in Fig. 3. The graph configurations generated are summarized in Table III.

Dijkstra's algorithm is implemented using two approaches: A Python-based implementation and Neo4j's APOC library-based implementation. The Python implementation is further carried out in two ways: A custom priority queue-based approach and a NetworkX Python library-based one. The custom implementation allowed for a fine-tuned control of the algorithmic details, while the NetworkX library provided a high-level, convenient interface for graph-related operations. The latter approach makes use of a database, whereas the first one does not. The Python implementation relies solely on in-memory data structures and algorithms to compute the shortest path in a graph. On the other hand, Neo4j being a database relies on loading the data from the disk if the size of the data is large. However, loading the data from disk is slower compared to loading from RAM which is what Python libraries do.

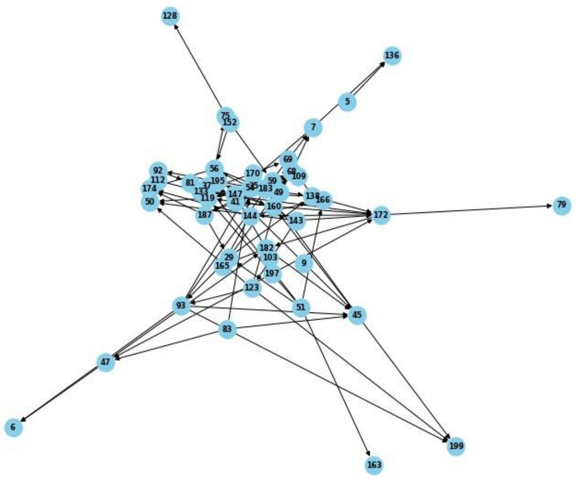

Fig. 3. An example of random connected graph.

TABLE III. SUMMARY OF CONFIGURATIONS OF BOTH 1000 NODE AND 10,000 NODE GRAPHS.

1000 nodes		10,000 nodes	
Connection probability	Number of edges	Connection probability	Number of edges
0.005	5,000	0.0005	50,000
0.0075	7,500	0.00075	75,000
0.01	10,000	0.001	100,000
0.025	25,000	0.0025	250,000
0.05	50,000	0.005	500,000
0.075	75,000	0.0075	750,000
0.1	100,000	0.01	1,00,000

Nonetheless, assessment of algorithmic performance given this difference in the working of the two environments is an interesting aspect to explore.

For the approaches discussed above, we run the algorithm for 100 random start and end nodes taken from the graph. The time taken for each complete traversal is noted and finally, the average time for all the 100 traversals is calculated. This is crucial because the traversal time could be significantly different depending on the start and end nodes the algorithm chooses at random and the algorithm's internal decisions. Although there could still be variability in the average runtime of 100 traversals on multiple executions, averaging over a larger number of traversals helps smooth out the impact of these variations, providing a more stable and representative measure of the algorithm's typical performance.

III. RESULTS

A. *Priority queue vs NetworkX*

The results for the preliminary comparison of two Python-based implementations- Custom priority queue and NetworkX library are shown in Fig. 3. The first striking observation about both implementations is the linear runtime dependence with increasing graph size (number of edges). The runtime for both

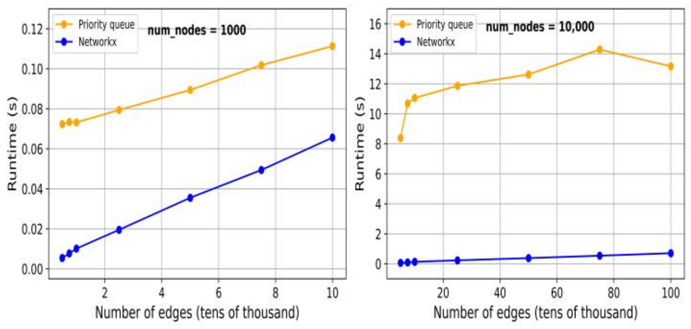

Fig. 4. Runtime performance comparison between custom priority queue implementation and NetworkX library for 1000 and 10,000 nodes.

graph settings- 1000 and 10,000 nodes, suggests that the NetworkX library is more efficient than the custom priority queue implementation. Though, the runtime using both implementations for the 1000 node graph is of the order of 10^{-2} s. Based on these observations, it is evident that NetworkX is a better choice to compare algorithm performance with Neo4j's APOC library which we do now.

B. *NetworkX vs APOC*

Fig. 4. shows the runtime performance comparison between NetworkX library and Neo4j's APOC library. The experimental comparison between NetworkX and APOC for Dijkstra's algorithm reveals distinct performance trends. NetworkX exhibits a linear increase in runtime as graph density rises, while APOC consistently outperforms in terms of computational efficiency, showcasing lower runtimes for both 1000 node and 10,000 node graph settings. Notably, the relative difference between them becomes more pronounced with an increase in the number of nodes, emphasizing APOC's robust efficiency. This is because the runtime in APOC for both graph settings with varying density is of the order of 10^{-3}s. However, its algorithmic runtime performance exhibits notable variability, potentially attributed to caching or other database functionalities. This is a plausible explanation as to why a decreasing trend is observed in runtime. These findings underscore the nuanced trade-offs between algorithmic choices and database-specific optimizations, providing valuable insights for practitioners navigating the selection of graph processing tools for diverse applications.

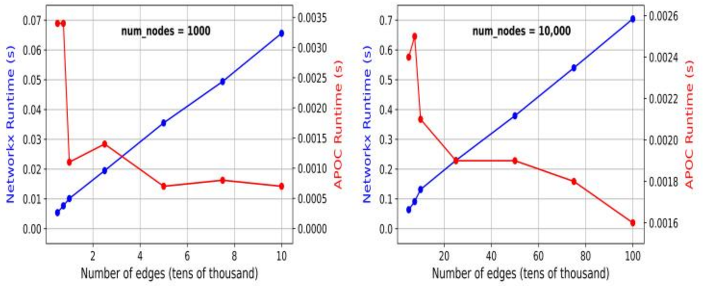

Fig. 1. Runtime performance comparison between custom priority queue implementation and NetworkX library for 1000 and 10,000 nodes.

IV. Conclusions

In conclusion, this study delved into the implementation and comparative performance analysis of Dijkstra's algorithm using Python-based approaches and Neo4j database’s APOC library. Notably, the custom priority queue-based Python implementation and the NetworkX library displayed linear runtime dependencies with increasing graph size, with NetworkX demonstrating superior efficiency. When contrasted with Neo4j's APOC library, APOC consistently outperformed in terms of computational efficiency, particularly evident in larger graph settings. Despite APOC's robust efficiency, its algorithmic runtime exhibited notable variability, potentially linked to caching or other database functionalities. These findings highlight the nuanced considerations in choosing between Python-based solutions and database-specific tools for optimal graph processing in diverse applications.

References

[1] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[2] E. Hall, "Time-dependent, shortest-path algorithm for real-time intelligent vehicle highway system applications," Transportation Research Record, vol. 1408, pp. 94-100, 1993.

[3] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in SODA, vol. 5, pp. 156-165, January 2005.

[4] V. Patras, P. Laskas, K. Koritsoglou, I. Fudos, and E. Karvounis, "A comparative evaluation of RDBMS and GDBMS for shortest path operations on pedestrian navigation data," in Proc. 6th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), 2021, pp. 0-4.

[5] M. Miler, D. Medak, and D. Odobasić, "The shortest path algorithm performance comparison in graph and relational database on a transportation network," Promet - Traffic - Traffico, vol. 26, no. 1, pp. 75-82, 2014.

[6] J. J. Miller, "Graph database applications and concepts with Neo4j," in Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, vol. 2324, no. 36, pp. 141-147, March 2013.