Unraveling the Key Differences in Graph Databases Neo4j and GraphDB using RDF Dataset

Aravinda Vijayaram Kumar
Masters in Data Science,
George Washington University
aravinda.avk@gmail.com

Aaron Yang
Masters in Data Science,
George Washington University
aaron.young@gwu.edu

Luhuan Wang
Masters in Data Science,
George Washington University
luhuan.wang@gwu.edu

Abstract— As the demand for fast and scalable graph databases grows, a thorough understanding of their performance characteristics becomes critical. This paper provides a detailed comparison of the performance of two well-known graph databases, Neo4j and Ontotext GraphDB, with a focus on RDF data processing. We investigate and contrast the diverse capacities and difficulties inherent in both databases using a systematic Time evaluation.
The study delves into crucial features such as traversal efficiency and query processing times for data of differing sizes and complexities to provide a more nuanced view of how databases behave in various settings. Cypher for Neo4J and SPARQL for GraphDB are used for querying.

Keywords—RDF(Resource Description Framework), TTL(Turtle File) , DFT(Depth First Search), BFT (Breadth First search), Property graph, Semantic Web

I. INTRODUCTION

As the field of data management in data science evolves, the explosion in the amount, variety, and velocity of information has led to an ongoing pursuit of innovative storage and retrieval systems. Rapid growth in data volume, complexity and interconnectivity highlights the critical role of innovative data management systems. At the same time, Resource Description Framework (RDF) serves as a powerful standard for representing information on the web in a graph-like format, paving the way for semantic web applications [1]. In the field of data management, the emergence of graph databases has revolutionized the storage and retrieval of highly interconnected data structures. Graph databases are different from traditional relational databases. Graph databases focus more on the relationships and connectivity between data. It stores data in the form of graphs. This structure makes graph databases advantageous in processing highly interconnected and complex relationship data, such as social network analysis, recommendation systems, network security, bioinformatics and other fields.

Ontotext GraphDB is a graph database system focusing on semantic graph data management. It has highly optimized RDF storage and reasoning functions.[2]

In the field of graph database systems, Ontotext GraphDB and Neo4j are both backbones, each providing unique methods and capabilities for managing and querying graph-structured data. Known for its RDF storage and reasoning capabilities, GraphDB specializes in semantic graph data management. The concept of the Semantic Web envisions expanding the current World Wide Web by incorporating machine-interpretable metadata into published information and data. This involves enhancing existing content and data on the web with additional descriptors. The objective is to enable software programs to derive meaningful interpretations, similar to how humans process information to accomplish their objectives. [3]

On the contrary, Neo4j is known for its versatile query language and powerful performance, occupying an important position in various industries and applications. As one of the major representatives in the database field, Neo4j has extensive applications and community support. Its flexible data model and powerful query language make it ideal for working with graph data.[4]

We chose to compare Ontotext GraphDB with Neo4j because of its unique advantages in processing RDF data. GraphDB's focus is on the storage and reasoning of RDF data, which is in sharp contrast to Neo4j's common application in graph data management. Comparing the two can help us gain a deeper understanding of their respective characteristics, advantages, and limitations, and provide clearer guidance for graph database selection in different application scenarios.

Our team embarked on a comprehensive comparative analysis of Ontotext GraphDB and Neo4j to explore their effectiveness in managing RDF data and measure the time for executing various types of queries.

In this paper we will describe the basic characteristics of graph DB and conduct a systematic comparison with Neo4j in terms of versatility and complexity of processing RDF data. A comprehensive assessment of their capabilities, limitations, and their symbiotic potential to harmoniously combine to facilitate advanced data management systems. We also selected suitable database imports, aiming to highlight their unique features, query languages, Performance characteristics and application areas. By illuminating the strengths and limitations of each system, this study aims to provide valuable insights for data scientists when choosing the best graph database solution for a specific use case.

II. DATASETS

Suitable data significantly affects data management and analysis in various fields. In this study, we used 2 datasets from Data World website- ddw/ontology-v0 [5] and Medical Subject Headings – MeSH [6] specifically, we aim to compare the functionality and performance of Ontotext GraphDB and Neo4j when processing the same dataset.

The ddw/ontology dataset is contains the core set of entities that data.world *matches* against and everything in this dataset

is managed by data.world and is constantly being expanded and improved. The Medical Subject Headings (MeSH) RDF is a linked data representation of the MeSH biomedical vocabulary produced by the National Library of Medicine. It has a total of 15,439,946**.**

This study strives to reveal the unique properties and potential advantages of these two database software solutions in processing data.

Both the datasets were loaded into Ontotext GraphDB and Neo4j and comparative analysis was performed in Python. The comparison focuses on several key aspects, including but not limited to data retrieval efficiency, query performance, scalability, and ease of use. Standardized query and performance metrics are used to evaluate and compare the capabilities and limitations of each database software.

III. METHODOLOGY

In this project, we executed a comprehensive comparative analysis between Ontotext GraphDB and Neo4j, with a primary focus on assessing their versatility and time performance in processing RDF data. To achieve this, both Ontotext GraphDB and Neo4j were configured with standardized settings to maintain fairness in our comparison, and a set of queries was designed to cover a spectrum of operations, from loading to traversals to deletion.

Performance metrics, including query execution times, were systematically collected and analyzed under varying conditions. The side-by-side comparison enabled us to derive insights into the databases' strengths and limitations, offering a nuanced understanding of their performance in different scenarios.

Both the databases were installed and a repository was created in GraphDB and its URL was to connect via Python. Similarly, we created a local DBMS of the 5.11 version and installed the neosemantics (n10s) plugin which enables the use and processing of RDF data in Neo4j [7]. The connection URL of the database was obtained to be connected via a Python driver.

To execute queries in GraphDB we used SPARQL which serves as the standard query language and protocol for both Linked Open Data and RDF databases. Crafted to interrogate a diverse range of data, it adeptly retrieves information concealed within non-uniform data structures and stored across multiple formats and origins[8]. Likewise, we used Cypher language which is a declarative graph query language and is efficient in querying property graphs, to execute queries in the Neo4j database.

The types of scenarios considered were

a. Loading the Data into the Databases

b. Basic querying the data using Select and Match

c. Traversals such as Depth First Search and Breadth First search, Best Path, Full Text Search, and Pattern Matching.

d. Deletion of the Data from the Databases.

The overall execution was done in Python by using different libraries for each database such as SparqlWrapper [9] for GraphDB and neo4j[10] driver for Neo4J database. The plots were plotted using the matplotlib library.

IV. EXECUTION

A. Loading the Data to the DB

The queries for loading the RDF data are given in the following table-

GraphDB	Neo4J
LOAD <file:///{file_path1}>	a. CALL n10s.graphconfig.init b. CREATE CONSTRAINT n10s_unique_uri FOR (r:Resource) REQUIRE r.uri IS UNIQUE c. CALL n10s.rdf.import.fetch("file://--path--","Turtle");

Table 1. Query for loading data in DB

The loading of data in GraphDb was simple and with use of one command, but in Neo4j, the graph has to be initialized, then a unique URI constraint has to be created and only after that can the data be loaded.

In both cases, many Turtle (TTL) files which stands for "Turtle Terse RDF Triple Language." with different triple counts were uploaded. It is a widely used syntax for representing RDF (Resource Description Framework) data in a human-readable and compact form.

An NT file that stands for "N-Triples." Was also uploaded to both databases. N-Triples is a simple syntax for representing RDF. The execution times were measured, plotted, and analyzed.

B. Querying the DB

Basic Queries such as SELECT and MATCH were executed with different limit sizes to compare the time performances of the databases. The general query format is given in the table below-

GraphDB	Neo4J
select * where { ?s ?p ?o . } limit 1000	MATCH (n) RETURN n LIMIT 1000

Table 2. Basic Select/Match Queries format

Basic search queries were executed for different limits such as 100,1000, 10000, and so on upto 1000000, to measure their querying time.

C. Traversals/Searches

In this study, we executed 5 types of traversals/searches in both databases- Depth First Traversal, Breadth First traversal, Best/shortest path, Pattern Matching search, and Full-text Search.

The depth-first traversal is an algorithm employed for navigating or searching tree and graph data structures. In this algorithm, the exploration begins at the root node and it extends as far as possible along each branch before retracing steps through backtracking.[11]

Similarly, in Breadth First Traversal, starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.[12]

Conducting a depth-first search (DFS) or breadth -first search(BFS) on a graph expressed through RDF data involves employing SPARQL to navigate the graph structure and extract the required information. It's crucial to recognize that SPARQL, unlike conventional graph query languages such as Cypher for Neo4j, doesn't inherently cater to graph traversal. Despite this, a DFS/BFS-like approach can be achieved using SPARQL by implementing recursive queries on the graph.

The query for DFS and BFS in SPARQL for GraphDB is considered as same as it's essential to clarify that the SPARQL query itself remains constant for both traversal methods. The query is given below-

```
PREFIX dwo: <https://ontology.data.world/v0#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?stateLabel ?geonameID
WHERE {
  ?state a dwo:UsState ;
        rdfs:label ?stateLabel ;
        geo:lat ?lat ;
        geo:long ?long ;
        dwo:geonameId ?geonameID .
}
```

Similarly in Neo4j, we used simple cipher queries to perform Depth First Traversal and Breadth First Traversal, the queries are as given below-

DFT-

```
MATCH path = (startNode)-[*]->(endNode) WHERE endNode.uri = 'https://entities.data.world/usstate_alabama' RETURN path;
```

This query employs a depth-first traversal strategy to explore paths in the graph. The MATCH clause identifies a variable path representing the traversal from a startNode to an endNode. The [*] syntax signifies an unrestricted relationship traversal

BFT-

```
MATCH path = (startNode)-[*1..]->(endNode) WHERE endNode.uri = 'https://entities.data.world/usstate_alabama' RETURN path;
```

This query employs a breadth-first traversal strategy. The [*1..] syntax specifies a variable-length traversal with a minimum of 1 relationship, indicating that the traversal should explore neighboring nodes at the same level before moving to subsequent levels.

Further we compared the performances of the Best/Shortest path queries in both databases. The queries are as given below-

GraphDB SPARQL-

```
PREFIX dwo: <https://ontology.data.world/v0#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?state1Label ?state2Label
WHERE {
  ?state1 a dwo:UsState ;
        rdfs:label ?state1Label .
  ?state2 a dwo:UsState ;
        rdfs:label ?state2Label .
  FILTER(?state1 != ?state2)
}
```

As stated earlier in SPARQL is not intended for traversals, but this query is written to simulate a simple form of pathfinding to find the best path between two states that is provided in the datasets.

Neo4J-

```
MATCH path = shortestPath((startNode)-[*]-(endNode))
  WHERE startNode.uri = 'https://entities.data.world/usstate_alabama'
  AND endNode.uri = 'https://en.wikipedia.org/wiki/Maine'
```

```
RETURN path;
```

The query intends to find and return the shortest path between two nodes in the graph. It starts from a node with the URI 'https://entities.data.world/usstate_alabama' and ends at a node with the URI 'https://en.wikipedia.org/wiki/Maine'. The shortest path function is employed to find the shortest path between two nodes. It explores all possible paths but returns only the shortest one based on the relationship weights, usually it assumes all relationships as equal weights unless specified.

Next, we performed two searches pattern matching and Full-text search and compared their execution performances. The queries for pattern matching are-

GraphDB-

```
PREFIX dwo: <https://ontology.data.world/v0#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

CONSTRUCT {
  ?state a dwo:UsState ;
        rdfs:label ?stateLabel ;
        geo:lat ?lat ;
        geo:long ?long ;
        dwo:geonameId ?geonameID .
}
WHERE {
  ?state a dwo:UsState ;
        rdfs:label ?stateLabel ;
        geo:lat ?lat ;
        geo:long ?long ;
        dwo:geonameId ?geonameID .
}
```

This query uses a CONSTRUCT keyword to create a new graph by matching a pattern that is given to it in the where clause.

Neo4J-

```
MATCH (subject)-[*]->(object)
WHERE subject.uri = 'https://entities.data.world/usstate_colorado'
RETURN subject, object;
```

This Cypher query is used to perform a pattern match traversal in a graph. The WHERE clause again is used to specify the required condition/pattern be matched.

The queries for the Full Text Search is given below-

GraphDB-

```
PREFIX dwo: <https://ontology.data.world/v0#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?stateLabel
WHERE {
  ?state a dwo:UsState ;
        rdfs:label ?stateLabel .
  FILTER(CONTAINS(LCASE(?stateLabel), "new"))
}
```

The SPARQL query is used to perform a case-insensitive full-text search in an RDF dataset, specifically looking for instances of the class dwo:UsState with labels that contain the term "new."

Neo4J-

```
MATCH (node)
  WHERE node.rdfs__label CONTAINS 'Colorado'
  RETURN node
```

The given Cypher query performs a case-sensitive full-text search in a Neo4j graph database, specifically looking for nodes where the property rdfs__label contains the term 'Colorado'. These type of full-text search queries are valuable for extracting information from an RDF dataset based on textual patterns.

D. Deletion of Data

Once the Traversals are done we compare the execution performance for deleting the data of different sizes from both databases. The general format of the delete queries in both graphs are given below-

GraphDB	Neo4J

DELETE {
?subject ?predicate ?object .
}
WHERE {
{
SELECT *
WHERE {
	MATCH (n)
	WITH n
	LIMIT 100
	DETACH DELETE n;

?subject ?predicate ?object . } LIMIT 100 } }	

Table3. Delete Query general format

This delete query is executed for different triple/node counts of 100, 1000 upto 100000 or 1000000 if the memory heap permits in Neo4J depending on the specification of the system.

V. RESULTS

In this section, we present the results of the comparative analysis between GraphDB and Neo4J, with a focus on handling the RDF data. The results for each type of query execution are tabulated and plotted.

The Execution time for loading the data nto the database for different triplet counts is tabulated in Table 4 and plotted in Fig 1-

Triplet Count	GraphDB	Neo4J
157	0.847975492477417 seconds	0.1910383701324463 seconds
579	0.7824344635009766 seconds	0.1515660285949707 seconds
1433	0.23038530349731445 seconds	0.2263948917388916 seconds
7481	0.7272355556488037 seconds	1.1153607368469238 seconds
15705	1.2057037353515625 seconds	0.27726054191589355 seconds
46479	1.2367780208587646 seconds	1.055119514465332 seconds
305779	6.211588621139526 seconds	5.927391052246094 seconds
477366	8.40849256515503	2.750755548477173 seconds
15439946	310.032653093338 seconds	250.40910816192627 seconds

Table 4. Query Execution time for loading data

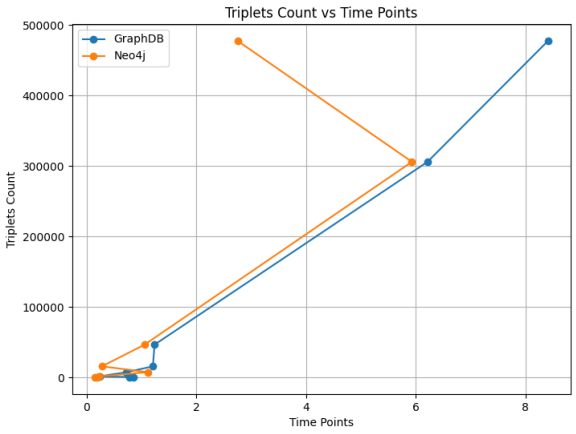

Fig 1. Plot for loading Data into Databases

From the results, we can observe a common pattern as the triplets increase the load time also increases. GraphDB demonstrates a consistent performance throughout and is a little faster when loading smaller triplet counts as shown for 7481 counts, but it is outperformed by Neo4J in loading the data, especially at larger data sizes.

Further, the results from the execution of the search queries is tabulated and plotted in Table 5 and Fig 2 respectively.

Query Size	GraphDB	Neo4J
100	0.01995110511779785 seconds	2.0970842838287354 seconds
1000	0.03729748725891113 seconds	0.034908294677734375 seconds
10000	0.16156792640686035 seconds	0.07380294799804688 seconds
100000	1.1192455291748047 seconds	1.0791707038879395 seconds
1000000	10.309787273406982 seconds	5.585696220397949 seconds

Table 5. Query Execution time for basic select/match queries

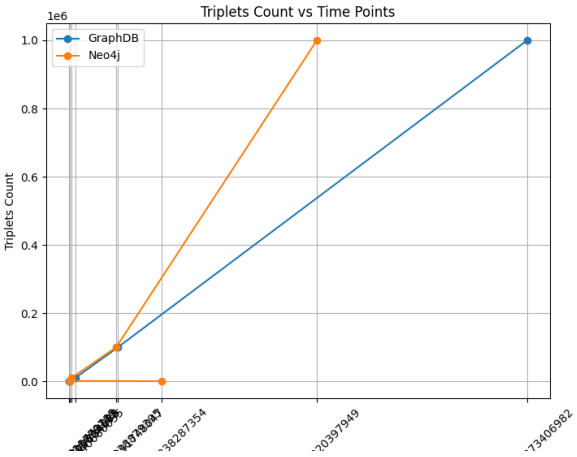

Fig 2. Plot for basic search queries in Databases

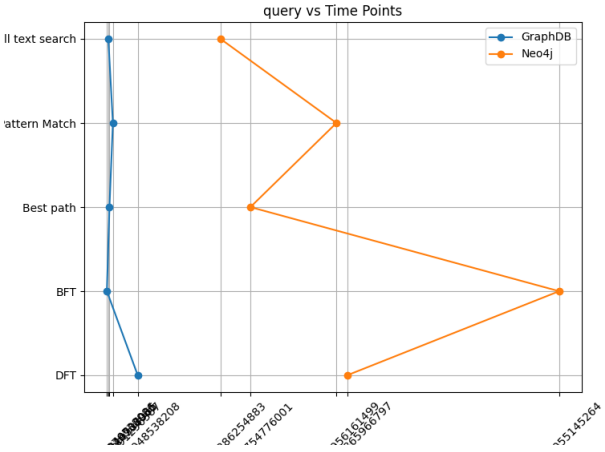

Fig 3. Plot for different traversals in Databases

analyzing the results of the supplied SELECT and MATCH queries run on Neo4j and GraphDB, certain trends surface, quite similar to the loading data. GraphDB generally exhibits faster execution times on small to medium result sets, while Neo4j performs better for higher limits and gives the results almost at half the time of GraphDB for 1000000 records.

The performance of different traversals and searches are tabulated and plotted in Table 6 and Fig 3 respectively.

Query	GraphDB	Neo4J
DFT	0.282304048538208 seconds	2.1563243865966797 seconds
BFT	0.00589 seconds	4.054851055145264 seconds
BestPath	0.02892279624938965 seconds	1.290935754776001 seconds
Pattern Matching	0.05983996391296387 seconds	2.060992956161499 seconds
Full Text Search	0.018899440765380 86 seconds	1.022809028625 4883 seconds

Table 6. Query Execution time for different traversals/searches

From the results, we can observe a clear pattern that GraphDB outperforms the Neo4J database in all types of queries and traversals, with significantly lower execution times. One of the reasons for this kind of performance might be GraphDB's RDF-centric architecture which probably makes processing RDF triple patterns and gr aph traversals more effective.

The execution times for deleting the data from the databases is tabulated and plotted in Table 7 and Fig 4 respectively.

Query Size	GraphDB	Neo4J
100	0.13862895965576172 seconds	2.1626358032226562 seconds
1000	0.16256809234619 14 seconds	0.4136426448822 0215 seconds
10000	0.45677804946899414 seconds	0.5808196067810059 seconds
100000	2.5747480392456 055 seconds	0.9787862300872 803 seconds

Table 7. Query Execution time for delete queries

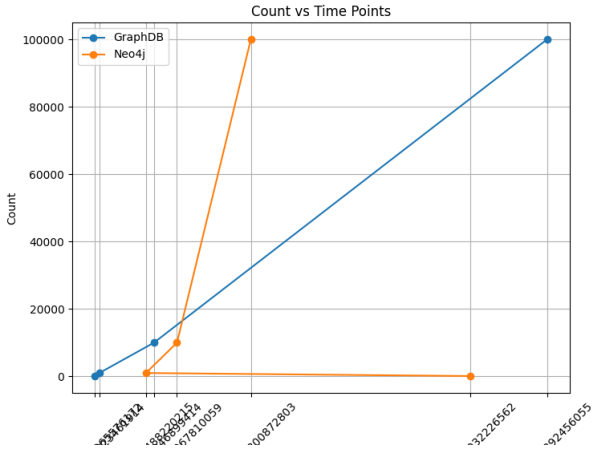

Fig 4. Plot for deleting Data from Databases

Similar to loading and query of data, in deletion, Neo4j begins with a comparatively longer execution time for deleting 100 nodes, even if its deletion timings are typically fast. While GraphDB remains consistent as the data size increases, Neo4J's performance is comparatively faster for larger data sizes as seen for 100000 triplets' deletion.

Another key observation was that in Neo4J, when a traversal is executed using the ID (node Identifier) of a node, the results are rendered much faster than compared to any other property used. One of the reasons for this observation may be that node IDs are implemented as integer values and are distinct identifiers assigned to nodes by Neo4j. Neo4j can make use of effective internal data structures, like indexes or direct lookups, to swiftly locate the node when you utilize the internal node ID for retrieval. There are certain drawbacks of using the IDs as well, such as the ID changes for a node when re-ingested, backed-up or transferred to a new DB.

VI. CONCLUSION AND FUTURE WORK

Our thorough comparison of Ontotext GraphDB and Neo4j covering data loading, search queries, traversal algorithms, and data deletion has revealed subtle differences in the functionality and performance of these two well-known graph database systems.

Even though it is a Property graph database, Neo4j showed great performance in handling RDF data, especially in loading, searching, and deletion. Neo4j performed better than GraphDB at handling higher data size. The GraphDB excelled at traversals and searches, of RDF data outperforming the Neo4j.

In this study, our executions were conducted on a Windows operating system, utilizing a desktop with 16GB RAM. It's important to note that this specific hardware and software environment could influence the results. The performance of GraphDB and Neo4j might vary significantly on different systems, such as high-capacity enterprise servers, which could yield different outcomes due to their enhanced processing capabilities.

Additionally, the choice of operating system could also impact the results. Our experiments were limited to Windows, but different operating systems like MacOS or Linux may interact differently with these databases, potentially altering their performance profiles.

For future research, it is recommended to explore these variables more thoroughly. Researchers could replicate the study across different hardware setups, varying the RAM capacity and using diverse server configurations. This would help in understanding how hardware capabilities influence the performance of these graph databases. Furthermore, conducting similar experiments on various operating systems would provide a more comprehensive view of the databases' adaptability and efficiency across different platforms.

Expanding the scope of data used in testing is another valuable avenue for future research. By varying the size and complexity of datasets, researchers can gain deeper insights into how each database scales and performs under different data loads. This would contribute to a more nuanced understanding of the strengths and limitations of GraphDB and Neo4j, offering clearer guidance for their application in real-world scenarios.

Through such explorations, the research community can better ascertain the robustness and versatility of these databases, providing more informed recommendations for their use in diverse environments.

VII. REFERENCES

[1] RDF, https://www.w3.org/RDF/

[2] Ontotext-GraphDB, https://www.ontotext.com/products/graphdb/?ref=menu

[3] What is Semantic Web, https://www.ontotext.com/knowledgehub/fundamentals/what-is-the-semantic-web/

[4] Alm Robert, Imeri Lavd, A Performance Comparison between Graph Databases Degree Project about the comparison between Neo4j, GraphDB and OrientDB on different operation (16-17)

[5] https://data.world/ddw/ontology-v0

[6] https://data.world/nlm/medical-subject-headings-mesh

[7] Neosemantics, https://neo4j.com/labs/neosemantics/

[8] What is SPARQL, https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/

[9] Sparqlwrapper library, https://pypi.org/project/SPARQLWrapper/

[10] Neo4j library, https://neo4j.com/developer/python/

[11] Depth-first Search, https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

[12] Breadth First Traversal, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?ref=lbp