Can LLMs Study Graphs?

By,
Northwestern University Capstone Team
(Yuexin Chen, Sharika Mahadevan, Han Wen Liang, Kiran Jyothi Sheena, Hanwei Hu)

Table of Contents

Introduction

The Border Gateway Protocol (BGP) is integral to the functioning of the internet, dictating how packets are routed between autonomous systems (AS)[1]. However, the complexity and sheer volume of BGP data present significant challenges in analysis and interpretation. Traditional methods often fall short in providing clear insights into the intricate relationships and dynamics within BGP networks. This project addresses the critical need for advanced tools and methodologies to effectively interpret BGP data, focusing on the transformation of this data into a structured and analyzable format.

This project employs a two-pronged approach: First, it involves the conversion of BGP data into a graph structure, with each AS represented as a node. This transformation is crucial for structuring the data into an analyzable format. Second, the project integrates an LLM to interpret the graph. This combination was chosen for its potential to provide deeper, more intuitive insights into the graph's structure and the relationships between nodes.

Our results prove the high abilities of LLM to decode the BGP data, including the degree, routing loops, path length and so on, which means LLM is able to identify if there is any anomaly or disruption in the BGP network at specific time. The integration of LLMs demonstrated a significant improvement in the interpretation of complex graph structures, providing a user-friendly interface for network analysis.

The project's methodology and results have significant implications for network management and security, offering a novel approach to understanding and optimizing internet routing protocols.

Literature review

Internet data connectivity relies on Autonomous Systems (AS) and the interconnecting paths between them. The connectivity at any given moment is established through the Border Gateway Protocol (BGP). To provide a visual and analyzable framework, we conceptualize the system as a graph structure, wherein the ASNs serve as nodes and the data transfer paths act as edges. This representation facilitates the execution of graph operations, allowing us to extract valuable insights such as identifying the shortest path or detecting changes in the network path. One of the most important applications of this is to analyze whether a routing change has occurred between times, which could lead to the detection of a BGP hijacking event. There have been many incidents recorded where agents maliciously redirect traffic for their own advantage[2]. Hence, there have been attempts to detect them using mathematical and machine learning techniques. Path based mathematical algorithms[3] and deep learning techniques using LSTMs[4] have been used for this purpose. For this study, we explore whether a Large Language Model (LLM) could be utilized for this purpose. LLMs are intelligent machine learning models which are capable of pattern recognition and text completion. They enable developing easy interfaces for a layman to interact with a complex machine learning system.

There has already been some studies on using LLMs to analyze graph systems. Wang.et al[5] explored different prompting techniques like build-a-graph and Algorithmic prompting to teach an LLM to perform graph related tasks. Chen et.al[6] also explores using LLMs to act as feature enhancers or predictors for graph based systems. However, they showcase that the performance of LLM towards interpreting and analyzing graphs is not great. Moreover, the LLMs would be limited by the token length of the input messages and hence would not be applicable for large graphs describing the ASN networks.

Instead, LLMs can be employed to generate the queries used to interact with graph databases. This would enable the user who is unaware of the query syntax or underlying graph structure to easily interact with the network. The toolkit langchain has developed a system which interacts with the graph database Neo4j using the help of OpenAI API. This approach is similar to that of Zhang et.al [7] who experimented with a method to generate Graph APIs from user input and then generating a user output based on the results of the API. We adopt this approach to develop a function which can interact with Neo4j to generate user readable output. We make use of OpenAI API for this purpose and utilize a python environment.

Dataset description

In this project, we use Border Gateway Protocol (BGP) data. This data is available in MRT format at RIPE RIS platform [8]. These data types are fundamental to understanding the dynamics of internet routing and are pivotal in our analysis.

BGP Routing Information Base (RIB) snapshots, commonly known as BViews data, provides a snapshot of the global BGP routing table at a given point in time. This dataset includes:

1. Autonomous System Numbers (ASNs): These are unique identifiers for each participant in the BGP network, essential for understanding the network's structure.1 - 65535. Private: 64512 - 65535. Only public ASN can use internet
2. Peer Relationships: Information about how different ASNs are interconnected, which is crucial for mapping the network topology.
3. Additional Attributes: To enrich our analysis, we have incorporated supplementary data such as the country of organization, physical addresses, downstream of the AS and other relevant metadata for each ASN. This additional information provides a more comprehensive view of the network, allowing for more nuanced analysis, such as geographical routing patterns.

BGP Update messages are dynamic data that reflect real-time changes in the BGP routing table.(To comply with bViews data, we also choose 2008-08-01:16:00) They include:

1. Update Messages: These messages announce new routes or withdraw previously announced routes. According to Wang and her colleagues, there are two types of update: announcements and withdrawal. And three subcategories under announcements including New, duplicate, and implicit withdrawal.
2. KeepAlive Messages: These are periodic messages that ensure the stability of the connection between BGP peers.
3. Open Messages: Used to establish a connection between BGP peers, containing information like the BGP version, the ASN of the sender, and the hold time.
4. Notification Messages: Sent in response to errors or special conditions in the BGP network.

For this particular project, we create all our results only based on the BViews data and use that to find the difference between two snapshots. To compare the graph at different time frames, we append the timestamp attribute to the node names. Noticeably, in the real world the difference between graphs could be bigger.

Tools and Methods

The toolkit langchain has a module which can be used to interact with graphs stored in different graph databases, with one of them being Neo4j. Neo4j is an easy to use graph database which can be easily connected with AWS and different cloud systems. For this particular project, we make use of the free instance of Neo4j Aura DB. The free instance of Neo4j Aura can support one database instance with a maximum of 200K nodes and 400K relationships. Once we create a database, we use the database URI, username and password to connect the database with langchain.

The gpt-4 model by openai is well trained enough to produce the correct cypher queries for a given user input. Along with it, the GraphCypherQAChain function under langchain is able to produce the required output by querying the stored graph in Neo4j. However, this system has a few shortcomings,

a) User prompts need to be specific sometimes.
b) The user output cannot be finetuned. For example, if a missing path is queried, langchain outputs it does not have the required information rather than the path does not exist.
c) There is no error feedback loop. The generated cypher query can be syntactically inaccurate, producing an error in Neo4j. Langchain does not attempt to rectify the query by using the produced error.

To overcome these issues, we propose to create a customized chain, mimicking the langchain process. Individual functions can be created for each of the steps utilizing LLMs, thus enabling us to finetune the prompts for the cypher query as well as for the final user output. This also allows us to implement an error feedback loop, to improve the accuracy of the cypher queries.

The overall process would be the following.

a) Initialisation

   A python driver to interact with the Neo4j database is created using the databases's URI, username and password. Also the OpenAI API key is added as the 'OPENAPI_API_KEY' environment variable.

b) Query is refined

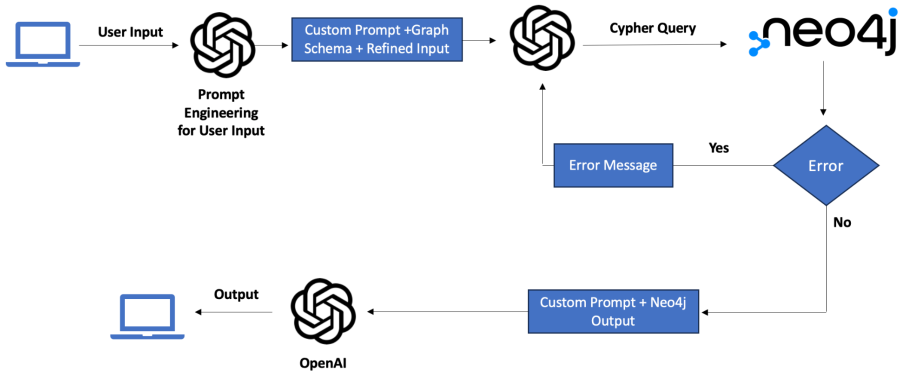
   The obtained user query is sent to the completions.create module under the OpenAI client, to be refined. This function takes raw user input and reformulates it into a more structured and precise query format. It is able to evaluate the clarity and relevance of the input, perform grammar correction and refine sentence structure to meet the user's expectations. If an input is considered unclear or irrelevant, the system prompts the user for a more explicit query. The *text-davinci-003* engine was chosen for its advanced capabilities in understanding and generating natural language. This model is particularly effective in handling a wide range of language tasks, including prompt refinement and clarification. The max_tokens parameter was set to 100, which was determined to be sufficient for generating concise yet complete refined prompts. The response from the client would contain the refined query.

c) Graph Schema is obtained
In order to obtain the correct query, a description of the node, edges and the relationship type must be passed along with each query. We get the schema of the graph by making a call using the initialized Neo4j driver. The description would describe the different node and relationship types as well as what node types are connected with each other.

d) Constructing the cypher query
In order to obtain the cypher query, we make a call to gpt-4 using the chat.completion.create function under the OpenAI client. When we make the call, we have to pass on messages to the chat function, consisting of a system prompt and the user content. We also have an option to add historical messages but we do not add it during the initial call. The response message would return the cypher query generated by the LLM.

e) Querying Neo4j
The obtained query is then run on Neo4j using the python driver and the response is obtained. In case the query returned from step c is syntactically incorrect, the driver will return a CypherSyntaxError. We then repeat step c with the error message. The chat.completion.create module is run again using the user query, the gpt response as history, and the error message along with the prompt to correct it. The obtained query is then rerun using the driver and the response is collected.

f) Obtaining the user output
To obtain the final user output, we make a call again to the OpenAI chat.completion.create module with the response from Neo4j and user query. We check whether the final output contains an apology which is returned if the LLM is not able to deduce the final output. In that case, we prompt the LLM to try again with simplified assumptions. The final output obtained is then returned to the user.



Flow diagram of Final Model

Results

The developed functions are tested against a graph database constructed using the RIP data.
For the purpose of testing, we have used subsets of BGP snapshots from collector rrc00. The first snapshot is as of 6th November 2015, UTC 00:00 and the second snapshot is of UTC 08:00. The graphs are subsetted to include only paths passing through ASNs 22804, 48479 and 28625. Different operations pertinent to that of BGP data are performed and compared with the accurate results. The custom function performs fairly well for the given operations. The following sections highlight the input provided, the Cypher query generated and the correct output.

Number of nodes

- **User input/prompt**
  How many nodes are there of type <insert node type> (ex. Node_201511060000)?
- **Cypher Query (generated by LLM)**
  MATCH (n:Node_201511060000)
  RETURN COUNT(n) AS NumberOfNodes_201511060000
- **Final Output**
  The number of nodes with type 'Node_201511060000' in the graph database is 78.
- **Findings**
  The LLM is easily able to find the number of nodes in a graph. The LLM refines the user prompt well and can generate the correct Cypher query to return the desired result.

Number of edges

- **User input/prompt**
  How many edges are there between the nodes with type <insert node type> (ex. Node_201511060000)?
- **Cypher Query (generated by LLM)**
  MATCH
  (n:Node_201511060000)-[:CONNECTS_201511060000]->(m:Node_201511060000)
  RETURN count(*) as edgeCount
- **Final Output**
  The number of edges between nodes is 139.
- **Findings**
  The LLM is easily able to find the number of edges between nodes in a graph. The LLM refines the user prompt well and can generate the correct Cypher query to return the desired result.

Isolated Nodes

- **User input/prompt**
  Are there any isolated nodes of type <insert node type> (ex. Node_201511060000) in the network? Can you mention their ids?
- **Cypher Query (generated by LLM)**
  MATCH (n:Node_201511060000)
  WHERE NOT (n)-[:CONNECTS_201511060000]-()
  RETURN DISTINCT n.id AS Unique_Isolated_Node_IDs
- **Final Output**
  Whatever you asked in the question does not exist.
- **Findings**
  The LLM is easily able to find that there are no isolated nodes in the graph. The LLM refines the user prompt well and can generate the correct Cypher query to return the desired result. However, the final output from the LLM does not seem to answer the question directly.

Indegree

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time name> (ex. t1). What is the indegree of node <insert node name> (ex. 22804) at <insert time name> (ex. t1)?
- **Cypher Query (generated by LLM)**
  Cypher Query is MATCH (n:Node_201511060000 {id:
  '22804'})<-[:CONNECTS_201511060000]-(connectedNode)
  RETURN count(connectedNode) as inDegreeT1
- **Final Output**
  The indegree of node 22804 of Node_201511060000 connected by CONNECTS_201511060000 at time t1 is 0.
- **Findings**
  The custom LLM is able to generate the correct Cypher query. Users must define the node type and relationship type in order for it to capture only one snapshot of the graph.

Change in Indegree

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time name> (ex. t1) and all nodes of type <insert node type> (ex.

Node_201511060800) and relationship as <insert relationship type> (ex. CONNECTS_201511060800) as graph at time <insert time name> (ex. t2). Between t1 and t2, what is the change in indegree for node <insert node name> (ex. 48479)?

- **Cypher Query (generated by LLM)**
  MATCH (n1:Node_201511060000 {id:'48479'})<-[r1:CONNECTS_201511060000]-()
  WITH COUNT(r1) AS indegree_time_t1
  MATCH (n2:Node_201511060800 {id:'48479'})<-[r2:CONNECTS_201511060800]-()
  WITH indegree_time_t1, COUNT(r2) AS indegree_time_t2
  RETURN indegree_time_t1, indegree_time_t2, ABS(indegree_time_t1 - indegree_time_t2) AS indegree_change
- **Final Output**
  The change in indegree for node 48479 in the graph database between time t1 and time t2 is 1.
- **Findings**
  The custom LLM is able to generate the correct Cypher query. Users must define the node type and relationship type in order for it to capture the distinct snapshot of the graph.

Outdegree

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time snapshot name> (ex. t1).What is the outdegree of node <insert node name> (ex. 22804) at <insert time snapshot name> (ex. t1)?
- **Cypher Query (generated by LLM)**
  Cypher Query is MATCH (n:Node_201511060800 {id:"22804"})-[r:CONNECTS_201511060800]->()
  RETURN COUNT(r) AS Outdegree_Node_22804_at_t2
- **Final Output**
  The outdegree of node 22804 of node type Node_201511060800 in the CONNECTS_201511060800 graph at time t2 is 1.
- **Findings**
  The custom LLM is able to generate the correct Cypher query. Users must define the node type and relationship type in order for it to capture the specific snapshot of the graph.

Change in Outdegree

- **User input/prompt**

Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time snapshot name> (ex. t1) and all nodes of type <insert node type> (ex. Node_201511060800) and relationship as <insert relationship type> (ex. CONNECTS_201511060800) as graph at time <insert time snapshot name> (ex. t2). Between t1 and t2, what is the change in outdegree for node <insert node name> (ex. 22804)?

- **Cypher Query (generated by LLM)**
  Cypher Query is MATCH (n1:Node_201511060000 {id: '22804'})-[r1:CONNECTS_201511060000]->()
  WITH COUNT(r1) AS outdegree_t1
  MATCH (n2:Node_201511060800 {id: '22804'})-[r2:CONNECTS_201511060800]->()
  WITH outdegree_t1, COUNT(r2) AS outdegree_t2
  RETURN outdegree_t2 - outdegree_t1 AS outdegree_change
- **Final Output**
  The change in outdegree for node 22804 between the graph at time t1 and t2 is 0.
- **Findings**
  The custom LLM is able to generate the correct Cypher query. Users must define the node type and relationship type in order for it to capture the distinct snapshot of the graph.

Path existence

- **User input/prompt**
  Does a path exist from node <insert node name> (e.g -28625) to node <insert node name>(e.g - 2828) for nodes of type <insert node type> (e.g - node_201511060000)? If so what is the path in terms of node id?
- **Cypher Query (generated by LLM)**
  MATCH path=(n:Node_201511060000 {id: "28625"})-[:CONNECTS_201511060000*]->(m:Node_201511060000 {id: "2828"})
  RETURN path
- **Final Output**
  The path from node 28625 to node 2828 is as follows: Node 28625 connects to node 23148, which in turn connects to node 2828. All nodes are of type node_201511060000.
- **Findings**
  The custom LLM is able to generate the correct Cypher query to determine the path and output the results in a user readable format.

Path length

- **User input/prompt**
  What are the paths from <insert node name> (e.g -1251) to node <insert node name>(e.g - 4844) for nodes of type <insert node type> (e.g - node_201511060000)? Also, return the path length of each path.
- **Cypher Query (generated by LLM)**
  MATCH
  path=(start:Node_201511060000)-[:CONNECTS_201511060000*]->(end:Node_201511060000)
  WHERE start.id = '1251' AND end.id = '4844'
  RETURN path, length(path) AS path_length
- **Final Output**
  Here are all the paths from node 1251 to 4844, including only nodes of type Node_201511060000, along with their corresponding path lengths:
  - Path: 1251 > 28625 > 12989 > 6762 > 3356 > 2914 > 4844, Path Length: 6
  - Path: 1251 > 28625 > 12989 > 6762 > 2914 > 4844, Path Length: 5
  - Path: 1251 > 28625 > 12989 > 7473 > 4844, Path Length: 4
  - Path: 1251 > 28625 > 12989 > 12956 > 3356 > 2914 > 4844, Path Length: 6
  - Path: 1251 > 28625 > 262589 > 6762 > 3356 > 2914 > 4844, Path Length: 6
  - Path: 1251 > 28625 > 262589 > 6762 > 2914 > 4844, Path Length: 5
- **Findings**
  The custom LLM is able to generate the correct Cypher query to determine if the path exists and the path length of each existing path. Also, the final output is in a user readable format.

Change in Path

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time snapshot name> (ex. t1) and all nodes of type <insert node type> (ex. Node_201511060800) and relationship as <insert relationship type> (ex. CONNECTS_201511060800) as graph at time <insert time snapshot name> (ex. t2). Between t1 and t2, have the paths from <insert node name > (e.g 28625) to <insert node name> (e.g 4608) changed? If so, how did it change?
- **Cypher Query (generated by LLM)**
  MATCH path_t1 = (n1:Node_201511060000 {id: '28625'})-[:CONNECTS_201511060000*..]->(n2:Node_201511060000 {id: '4608'})
  MATCH path_t2 = (n3:Node_201511060800 {id: '28625'})-[:CONNECTS_201511060800*..]->(n4:Node_201511060800 {id: '4608'})
  RETURN path_t1, path_t2, path_t1 <> path_t2 AS change_detected
- **Final Output**

At time t1, the path between node 28625 and 4608 goes through the following node sequence: 12989 - 4637 - 1221. At time t2, there are multiple paths detected with different node sequences, as follow:

Path 1: 12989 - 4637 - 1221

Path 2: 12989 - 6762 - 3356 - 4637 - 1221

Path 3: 12989 - 6762 - 174 - 3356 - 4637 - 1221

Path 4: 12989 - 12956 - 3356 - 4637 - 1221

Path 5: 12989 - 262589 - 6762 - 3356 - 4637 - 1221

Path 6: 12989 - 262589 - 6762 - 174 - 4637 - 1221

Yes, there have been several changes in the paths between the two timestamps. The path has become more complex at time t2 compared to time t1, possibly indicating changes in the routes between node 28625 and 4608. Significant changes are the inclusion of new nodes 6762, 174, 3356, 12956, and 262589 in the paths at time t2 compared to time t1.

- **Findings**
  The custom LLM is able to generate the correct Cypher query to list down all the paths. Users must define the node type and relationship type in order for it to capture the distinct snapshot of the graph. There is no change in paths between the two nodes, however the LLM misinterpreted the multiple paths as a change in path and gave an incorrect result. We could however improve this by specifying a type of path like the shortest path.

Change in Shortest Path

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time snapshot name> (ex. t1) and all nodes of type <insert node type> (ex. Node_201511060800) and relationship as <insert relationship type> (ex. CONNECTS_201511060800) as graph at time <insert time snapshot name> (ex. t2). Between t1 and t2, have the shortest paths from <insert node name > (e.g 31133) to <insert node name> (e.g 4777) changed? If so, how did it change?
- **Cypher Query (generated by LLM)**
  MATCH (a:Node_201511060000{id:"31133"}), (b:Node_201511060000{id:"4777"}),
  p=shortestPath((a)-[:CONNECTS_201511060000*..15]->(b))
  WITH p as path_t1

MATCH (c:Node_201511060800{id:"31133"}), (d:Node_201511060800{id:"4777"}),
p=shortestPath((c)-[:CONNECTS_201511060800*..15]->(d))
WITH path_t1, p as path_t2
RETURN path_t1, path_t2, CASE WHEN path_t1 = path_t2 THEN FALSE ELSE TRUE END AS hasChanged

- **Final Output**
  Yes, the shortest paths from node 31133 to node 4777 have indeed changed between time t1 and t2. At t1, the path was 31133 → 3356 → 2497 → 4777, with nodes being in countries Russia, U.S., Japan, and Australia respectively. While at t2, the path changed to 31133 → 174 → 2914 → 2497 → 4777, representing a route through Russia, U.S., U.S., Japan, and Australia respectively. Hence, there were modifications in the path network.
- **Findings**
  The custom LLM is able to generate the correct Cypher query to list down the shortest paths and find the difference between them at the two snapshots. Users must define the node type and relationship type in order for it to capture the distinct snapshot of the graph.

Change in Number of Paths

- **User input/prompt**
  Consider all nodes of node type <insert node type> (ex. Node_201511060000) and relationship as <insert relationship type> (ex. CONNECTS_201511060000) as graph at time <insert time snapshot name> (ex. t1) and all nodes of type <insert node type> (ex. Node_201511060800) and relationship as <insert relationship type> (ex. CONNECTS_201511060800) as graph at time <insert time snapshot name> (ex. t2). Between t1 and t2, are there any change in number of paths from <insert node name > (e.g 28625) to <insert node name> (e.g 50300) ?
- **Cypher Query (generated by LLM)**
  MATCH p1=(n1:Node_201511060000 {id: '28625'})-[:CONNECTS_201511060000*]->(m1:Node_201511060000 {id: '50300'})
  WITH count(p1) as t1_paths
  MATCH p2=(n2:Node_201511060800 {id: '28625'})-[:CONNECTS_201511060800*]->(m2:Node_201511060800 {id: '50300'})
  RETURN t1_paths as Total_Paths_in_T1, count(p2) as Total_Paths_in_T2
- **Final Output**
  In graph t1, defined by node type Node_201511060000 and relationship CONNECTS_201511060000, there are a total of 9 paths from node 28625 to node 50300. Meanwhile, in graph t2, defined by node type Node_201511060800 and relationship CONNECTS_201511060800, there are a total of 7 paths between the same nodes. Comparing the two, there are more paths present in graph t1 than in graph t2.
- **Findings**

The custom LLM is able to generate the correct Cypher query to find the number of paths and the difference in that number between the two snapshots. Users must define the node type and relationship type in order for it to capture the distinct snapshot of the graph. Note: On repeated testing, it has been noticed that sometimes the answer would be incorrect as the LLM returns the query to find path length or existence of any path instead of number of paths.

Comparison between Current and Shortest Path

- **User input/prompt**
  If the current path from <insert node name> (e.g 1251) to <insert node name> (e.g 4844) is <insert path sequence> (e.g -1251-28625-12989-6762-3356-2914-4844) for node type <insert node type> (e.g -Node_201511060000),
  What is the shortest path possible and what is the difference?
- **Cypher Query (generated by LLM but incorrect)**
```
MATCH (startNode:Node_201511060000 { id: '1251' }),(endNode:Node_201511060000
{ id: '4844' }),
path = shortestPath((startNode)-[:CONNECTS_201511060000*]-(endNode))
RETURN path as ShortestPath, length(path) as ShortestPathLength

MATCH currentPath=(s:Node_201511060000
{id:'1251'})-[:CONNECTS_201511060000*]->(n1:Node_201511060000
{id:'28625'})-[:CONNECTS_201511060000*]->(n2:Node_201511060000
{id:'12989'})-[:CONNECTS_201511060000*]->(n3:Node_201511060000
{id:'6762'})-[:CONNECTS_201511060000*]->(n4:Node_201511060000
{id:'3356'})-[:CONNECTS_201511060000*]->(n5:Node_201511060000
{id:'2914'})-[:CONNECTS_201511060000*]->(endNode:Node_201511060000
{id:'4844'})
RETURN length(currentPath) as CurrentPathLength

RETURN ShortestPathLength - CurrentPathLength as DifferenceInPathLength
```
- **Cypher Error**
```
{code: Neo.ClientError.Statement.SyntaxError} {message: Variable `ShortestPathLength` not defined (line 8, column 8 (offset: 716))
"RETURN ShortestPathLength - CurrentPathLength as DifferenceInPathLength"
       ^}
```
- **Corrected Cypher Query (generated by LLM)**
```
MATCH (startNode:Node_201511060000 { id: '1251' }),(endNode:Node_201511060000
{ id: '4844' }),
shortestPath = shortestPath((startNode)-[:CONNECTS_201511060000*]-(endNode))
```

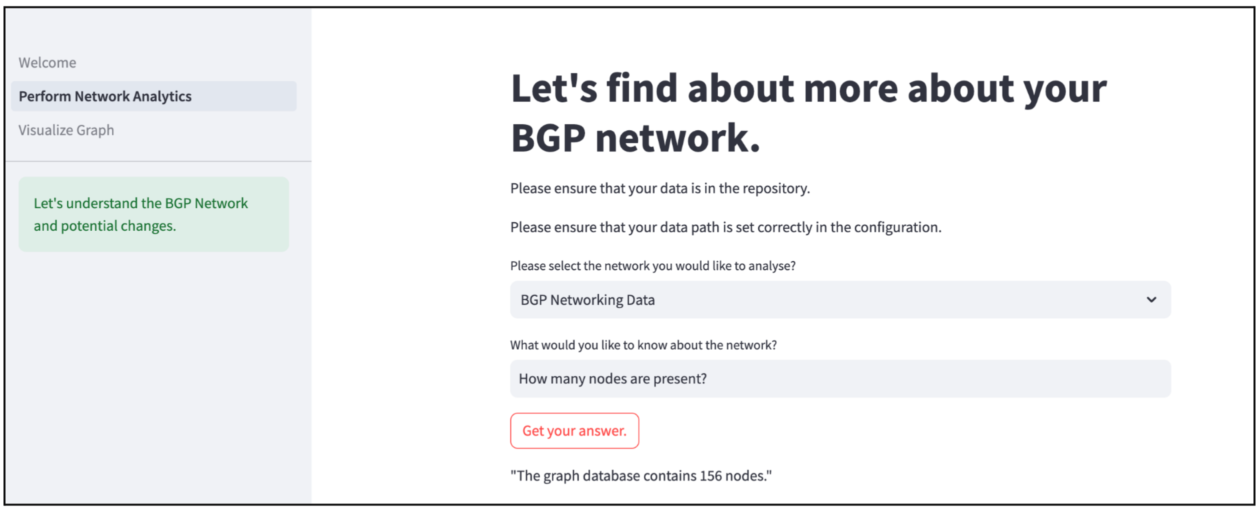
```
MATCH currentPath=(s:Node_201511060000
{id:'1251'})-[:CONNECTS_201511060000*]->(n1:Node_201511060000
{id:'28625'})-[:CONNECTS_201511060000*]->(n2:Node_201511060000
{id:'12989'})-[:CONNECTS_201511060000*]->(n3:Node_201511060000
{id:'6762'})-[:CONNECTS_201511060000*]->(n4:Node_201511060000
{id:'3356'})-[:CONNECTS_201511060000*]->(n5:Node_201511060000
{id:'2914'})-[:CONNECTS_201511060000*]->(endNode:Node_201511060000
{id:'4844'})

WITH startNode, endNode, shortestPath, currentPath
RETURN length(shortestPath) - length(currentPath) as DifferenceInPathLength
```

- **Final Output**
  The shortest path from Node_201511060000 1251 to 4844 is two hops shorter than the current path: 1251-28625-12989-6762-3356-2914-4844.
- **Findings**
  The custom LLM is able to generate the correct Cypher query to find the shortest path and compare it with the given (e.g existing) path. It can also be noticed that the LLM was able to correct the initial incorrect query using the generated error statement.

Product Description

The developed functions have been modified into a modularised format for easy implementation. A prototype for a user interface has also been developed using Streamlit. We have integrated the capabilities of the custom functions into the Streamlit application.

The application allows the user to input a prompt. This prompt is sent to the LLM that generates an appropriate Cypher query. This is sent to the Neo4j database to retrieve the information. Finally, this information is translated to human readable language and displayed on the Streamlit website.



Snapshot of Streamlit Application

Conclusion

The developed custom function incorporating LLMs is found to be effective in performing graph operations like detecting change in in- degree, isolated nodes, shortest paths, change in path number etc. The function is also able to find path changes, however can misinterpret the output when there are multiple interconnected paths between ASNs. However when a specified path is provided, the function is able to detect the change in paths.

The future of this project could ensure to fix the limitations of the current version of the application. The OpenAI gpt-4 model has a toke limit of 8192, which forces the function to truncate the Neo4j output if it exceeds a certain limit (This is the case with langchain as well). This can be improved by instructing the LLM to modify its output to shorten it, in case there is a need for truncation. This feedback loop can be incorporated into future versions. A timeout and recheck functionality can also be included in case the LLMs take a lot of time to generate queries. The current application also uses the free instance of Neo4j Aura where only 1 DB instance can be created and is limited to 200,000 nodes. The professional version or other graph databases could also be explored. However, the current prototype of the network analysis application provides a novel approach for addressing the specific problem at hand of analyzing BGP networks. All the codes used can be found in the github repo https://github.com/HenryLiang-123/Amazon_Knowledge_Graph. (It is currently a private repo).

The Network Analysis Application seamlessly combines the power of natural language processing, graph database queries, and an intuitive front end to provide users with a robust platform for studying and understanding changes in networking data. Whether for network administrators, researchers, or anyone keen on exploring network evolution, this application can offer a sophisticated and user-friendly solution.

References

1. Detailed information about BGP: [BGP_SLIDES]
2. P. Moriano, R. Hill, L.J. Camp, Using bursty announcements for detecting BGP routing anomalies, Computer Networks, vol. 188, 2021, 107835, ISSN 1389-1286
3. G. Comarela, E. Terzi, M. Crovella, Detecting Unusually-Routed ASes: Methods and Applications, in: Proceedings of the 2016 Internet Measurement Conference (IMC '16), November 2016, pp. 445–459
4.M. Cheng, Q. Xu, J. L.V., W. Liu, Q. Li and J. Wang, "MS-LSTM: A multi-scale LSTM model for BGP anomaly detection," 2016 IEEE 24th International Conference on Network Protocols (ICNP), Singapore, 2016, pp. 1-6, doi: 10.1109/ICNP.2016.7785326.
5. Wang, Heng, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. "Can Language Models Solve Graph Problems in Natural Language?." *arXiv preprint arXiv:2305.10037* (2023).
6. Chen, Zhikai, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang et al. "Exploring the potential of large language models (llms) in learning on graphs." *arXiv preprint arXiv:2307.03393* (2023).
7. Zhang, Jiawei. "Graph-ToolFormer: To Empower LLMs with Graph Reasoning Ability via Prompt Augmented by ChatGPT." *arXiv preprint arXiv:2304.11116* (2023).
8. https://ris.ripe.net/docs/