QLiG: **Q**uery **Li**ke a **G**raph For Subgraph Matching

Sumit Purohit, Patrick Mackey, Jeremy D Zucker, Ankur Bohra, Rahul D Deshmukh, George Chin
Pacific Northwest National Laboratory
Richland WA, USA, 99352
{sumit.purohit, patrick.mackey, jeremy.zucker, ankur.bohra, rahul.deshmukh, george.chin}@pnnl.gov

***Abstract*—A graph is a natural and flexible modeling approach to represent entities and relationships between them in the real world. A Knowledge Graph (KG) is a specialized graph with formal and structured representations of facts, relationships, annotated with semantic descriptions. Subgraph matching is one of the fundamental graph problems to identify relationships, interactions, and activities of interest within a large graph. A query specification is a collection of abstract components, operations, and constraints to express a pattern. The specification can be implemented in different ways based on the underlying data model. Various graph query specifications have been developed over the years that has led to the development of different open-sourced and vendor-specific query languages. Such specifications are modeled as an extension of relational algebra used in relational query languages such as SQL. Such approaches do not inherently support graph queries. There is a need to represent graph queries in terms of graph-based components to expedite the query construction by non-database experts. We present a graph-based query approach QLiG (pronounced *cleeg*), to perform subgraph matching in a Labeled Property Graph (LPG). QLiG provides required expressivity to represent a query graph in a natural way using high-level concepts such as path, structure, and constraints. We present the query specification, salient features, and a real-world use case to show functional examples.**

***Index Terms*—property graph, graph query, query specification, subgraph matching**

I. Introduction

The graph is a natural and flexible modeling approach to describe entities and relationships between them [1], [2]. Many real-world domains such as power-grid [3], social media [2], microbial interaction [4], the food web [5], and modeling adversarial activities [6] are shown to be modeled as graph. A Knowledge Graph (KG) is a formal and structured representation of facts, relationships, and semantic descriptions of a set of entities [7]–[9]. Graph Analytics involves a collection of algorithms and technologies to extract actionable knowledge of interest using graph-theoretic approaches. Subgraph matching is one of the fundamental problems in graph analytics of determining the presence and location(s) of a given small *query graph* in a large *target graph* [10]–[13]. Applications of subgraph matching have also led to the development of numerous open-source and proprietary graph databases [14]–[16]. These databases differ in their functionalities, implementations, and query mechanisms. A graph query specification provides an abstraction to explore the data stored as a graph. It is also used to perform pattern matching in the graph to extract the knowledge of interest. Similar to graph databases, different graph query specifications have been developed to explore underlying graphs.

Resource Description Framework (RDF) [17], [18] and Labeled Property Graph (LPG) [19] are two of the most used data models to encode information in a KG. SPARQL [20] is a W3C standard graph query language for RDF graphs. Labeled Property Graph (LPG) can be explored using various vendor-specific graph query languages [21]–[25]. Most widely used graph query languages are based on state-of-the-art relational technologies and provide features such as projection, aggregation, views, and recursion based on relational algebra. Such languages require a fine-grained description of each interaction to construct a large query pattern. Such a *bottom-up* query construction approach is tightly linked to the underlying graph schema because every query component uses the same syntax as the underlying graph. Additionally, these languages don't have the required expressivity to describe real-world scenarios. These limitations pose a challenge for domain experts to specify their pattern of interest as a graph query and search it in the large graph without the help of data modelers and database experts. This hinders the overall knowledge discovery and the adoption of graph-based technologies by subject matter experts (SMEs).

We present a novel query specification that defines graph-based components (e.g., node, edge, path, structure) as the basic building blocks of the query language. We present **QLiG** (pronounced *cleeg*), a ***Q**uery **Li**ke a **G**raph* approach to perform subgraph matching in a Labeled Property Graph (LPG). In contrast to the current *SQL-extended* graph query languages, QLiG focus on the subgraph matching problem and define graphs as the input and output of any query operation. QLiG allows SMEs to naturally describe domain-specific patterns using higher-level concepts such as group, path, and constraint. We present specifications of the query language including key graph-based components and different constraints specified on them. We present a real-world use case of a knowledge graph describing various biological assertions extracted from scientific publications. We present examples to showcase QLiG's expressivity to represent domain-specific complex questions as query patterns.

We present related work in section II and the QLiG specification is presented in section III. Section IV presents a real-world use case where QLiG is used to search biological patterns in a KG describing COVID-19 related biological claims and assertions. QLiG can also be used to evaluate the

Ankur Bohra performed this research when he was a Post Master Research Associate at PNNL.

quality of subgraph matching algorithms as shown in section V. We present our conclusion and future work in section VI.

II. Related Work

A query language is a fundamental capability of a graph database that allows users to explore the data and perform analytics. Unlike RDF, which has SPQRQL as its *de-facto* query language, LPG does not have a single unique query language. The exponential adoption of graph technologies has given rise to multiple LPG databases and corresponding query languages [21]–[25]. Cypher [23] is one of the most widely used query languages for Neo4j graph databases. Cypher provides graph query capability that is closely aligned with SQL and supports path query, regular expression, aggregation, and result projection. OpenCypher [22] is an open-source initiative to standardize LPG query language based on Cypher. Gremlin [24] is another widely used graph query language. It is developed as a query specification by Apache TinkerPop [26]. Gramling specification is implemented by different graph database vendors such as Amazon Neptune [27], Jenus Graph [28], and Apache TinkerPop. These implementations vary in their support of complete specification. They also use different underlying data structures and algorithms to implement the query specification. All such query languages are used by application developers, database administrators, and data analysts. In contrast, these query languages are not widely used by subject-matter experts (SMEs) and domain experts in chemistry, biology, cyber-security, etc. because of a steep learning curve and limited expressivity to describe a pattern using higher-level concepts such as path, structure, and constraint.

Such query languages lead to piecemeal construction of a large query graph, using micro-level node and edge patterns, joined using basic query constructs such as *WHERE* and *CONSTRUCT* clauses. G-CORE [29] identifies challenges in this *bottom-up* approach and recommends *Composability* and *Paths as first-class citizen* as the new features for a future graph query language. Graph Query Language [21] is an ongoing effort that aims to develop a graph query standard using proven ideas from OpenCypher, G-CORE, etc. We further extend G-CORE ideas to also use graph sub-structure as the first-class citizen. Additionally, we present a novel query specification that uses graph-based components such (e.g., node, edge, path, structure) as the basic building blocks of the query language. We also present a mechanism to specify constraints on the query components that can be used to reduce the search space while discovering matches to the query graph. In contrast to existing query languages, QLiG has higher expressivity and the required accuracy to describe a complex graph pattern. QLiG also inherently specifies approximate subgraph matching, which is a required capability for a higher-level graph query language. In the following section, we present QLiG specification and its components.

III. QLiG Specification

QLiG is a "top-down" query approach that is targeted at users who are not experts in graph databases and data modeling. QLiG is used by subject matter experts (SMEs) in domains such as chemistry, biology, cybersecurity, etc. because it enables them to describe a graph query more naturally than a collection of relational operations. SMEs can define paths, structures. and constraints that accurately reflect the pattern of interest. Particularly, QLiG provides high expressivity and flexibility to define a query pattern to search in an LPG. A query graph in LPG has the following features:

- **Semantic**: Query graph is defined in terms of node types and edge types.
- **Structural**: Query graph describes an exact or approximate set of nodes and their interactions.
- **Attributed**: Nodes and Edges of the query graph have numerical, categorical, and temporal features.
- **Constrained**: Query graph enforces a set of conditions on the structure and attributes that must be matched while discovering the patterns.

A QLiG query is a graph structure at its core and encodes a domain-specific question in a constrained graph query language. QLiG provides a specification to describe a pattern. The vocabulary of QLiG specification is based on the following graph elements:

- *NodeDef*: It defines a node that should be searched in the large input KG. The *NodeDef* also specifies node attributes present in the LPG. These attributes are used to match a node in the large graph. The QLiG specifies *type* property that lists all the labels a matched node can have. Similarly, QLiG also specifies numerical and categorical node properties that should be matched. If a property is not specified in the query, any node with any values is considered a match.
- *EdgeDef*: It defines an edge that should be searched in the large input KG. The *EdgeDef* is defined in terms of source and destination nodes. Similar to the *NodeDef*, the *EdgeDef* also specifies edge attributes present in the LPG. These attributes are used to match an edge in the large graph. Similar to *NodeDef*, and *EdgeDef* can defines edge labels and attributes to specify an edge pattern.
- *StructDef*: It defines a set of connected subgraphs that should be searched in the large input KG. The *StructDef* also defines a set of node labels and edges labels that constitute this structure. QLiG also uses *Constraints* to further specify the structure to include specific nodes, edges, and attributes.
- *PathDef*: It defines a set of paths that should be searched in the large input KG. Similar to *StructDef*, *PathDef* also defines a set of node and edges labels that constitute the path and also uses *Constraints* to further specify the path.
- *Constraints*: In addition to graph-based components, QLiG also uses constraints as a first-class citizen to specify a graph query for subgraph matching. *Constraints* are the conditions applied to query components and provide

a powerful and flexible mechanism to construct a graph query. QLiG defines *unary* and *binary* constraints further divided into following categories: Structural, Semantic, and Attribute. Semantic constraints are defined in terms of node and edge labels. Structural constraints are defined using the number of nodes and edges in the structure. QLiG also uses *membership* constraint to associate a set of nodes or edges with a structure (or path). The attribute constraints use arithmetic operators such as $>, <, =$ and range to restrict the possible matches of a graph query.

- *GraphDef*: A novel contribution of QLiG is to define named graphs as a basic building block of the query. Every QLiG query is defined in a default graph that represents the outermost query component. Named graphs are also used to define nested queries where the nested query is defined in its own named graph. The named graph can also be referenced in a *pathdef* or *structdef*.

These are the first-class citizens of QLiG and every query is defined using these elements. In the next section, we demonstrate a real-world use case where QLiG is used to search biological patterns in an LPG, describing COVID-19 related biological claims and assertions.

IV. Use Case

In this section, we present a real-world use case to demonstrate the expressivity of QLiG. We present concrete examples that use a biological knowledge graph describing claims about COVID-19. The COVID-19 Open Research Dataset (CORD-19) [30] contains over 60,000 scientific publication on COVID-19 and related historical coronavirus research. For every publication, it publishes metadata such as an identifier (i.e., PMID), title, abstract, publication time, journal name, etc. INDRA (the Integrated Network and Dynamical Reasoning Assembler) assembles [31] the natural language text about biochemical mechanisms into a common format. INDRA produces a casual graph about protein-protein interactions using the natural language processing of biomedical literature such as CORD-19. Each causal assertion in the INDRA knowledge graph is supported with evidence and a computed belief score representing the confidence in that assertion. We construct a large knowledge graph using the causal assertions related to SARS-CoV-2 infection as shown in Figure 1. The KG has 11344 nodes and 436291 edges with the following nodes types: Gene, Chemical, Protein, Disease, Biological Process, Reaction, Complex, and Abundance. The KG is serialized as a Semantic Property Graph (SPG) [32] which is an LPG-based approach to represent knowledge graphs. Figure 3 shows the degree distribution of constructed KG and the Figure 2 shows the node types distribution that exists in the KG.

Similarly, the KG has 158 edges types as shown in Figure 4. These edges are categorized into three different functional sets based on their impact on a biological pathway observed in the scientific publication. The majority of edges form an *Activation* functional group, where the source *activates* (or increase) the destination. Similarly in the *Inhibition* functional group, source node *inhibit*(or restrain) the destination group. The third function group defines a set of edges where the source is not observed to increase or decrease the abundance of the destination node. Notional visualization of three clusters is shown in Figure 5 where every element in the cluster is an edge-type present in the KG.

QLiG is used to construct a wide range of graph queries to search in the KG as shown below.

A. Singleton Query

QLiG can be used to uniquely identify a node or an edge in the KG using appropriate constraints. A *NodeDef* with attribute and membership constraints can match to a unique node in the KG irrespective of the adjacent edges. Similarly, an *EdgeDef* with source and destination *NodeDef* can match to a single edges in the KG.

B. Path Query

A *PathDef* is a first-class citizen in QLiG and can be defined using two endpoints and the path properties. Figure 6 shows a simple path query between a drug "Lopinavir" and "SARS-CoV-2". The query represents a biological pathway (i.e., a series of interactions) between "Lopinavir" and "SARS-CoV-2". The pathway is represented as an unbounded graph path with domain-specific constraints. Existing graph query languages have limited support for variable-length typed queries. Cypher supports variable-length path queries using range syntax. For example, $(a) - [*3..5] - > (b)$ represents a single path of minimum length 3, and a maximum length 5. It also supports named paths to reuse in larger Cypher queries. Such simple path query syntax lacks the required expressivity to represent the domain-dependent semantic constraints to a pathway, in addition to the variable path length.

Figure 6 shows an example of *Inhibition Pathway*, that represents a series of interaction between "Lopinavir" and "SARS-CoV-2". The overall impact of a *Inhibition Pathway* is such that the source (i.e., "Lopinavir") *Inhibits* (or restrains) the destination (i.e., "SARS-CoV-2"). Such a pathway can be found in the KG if it is constituted using only *Inhibition* or its inverse *Activation* edge types. Additionally, it also restricts a path with an odd number of *Inhibition* edges because even number of *Inhibition* edges (or odd number of *Activation* edges) lead to an *Activation* pathway instead. QLiG has required expressivity to construct such queries more naturally than state-of-the-art graph query languages. As shown in listing 1, a QLiG query is a graph described using the elements defined in section III. The pathway query is defined using two nodes (source and destination), one variable-length path, and a few constraints on the path. The length of the path is restricted using min_hops and max_hops attributes. The semantic constraints restrict the search space by specifying some filters on the path. QLiG provides $type_occurrence_filter$ to match only those paths that has a given number of *Inhibition* edge types.

C. Structural Query

Structural specifications of QLiG enable SMEs to query a graph database using more natural and easier syntax than

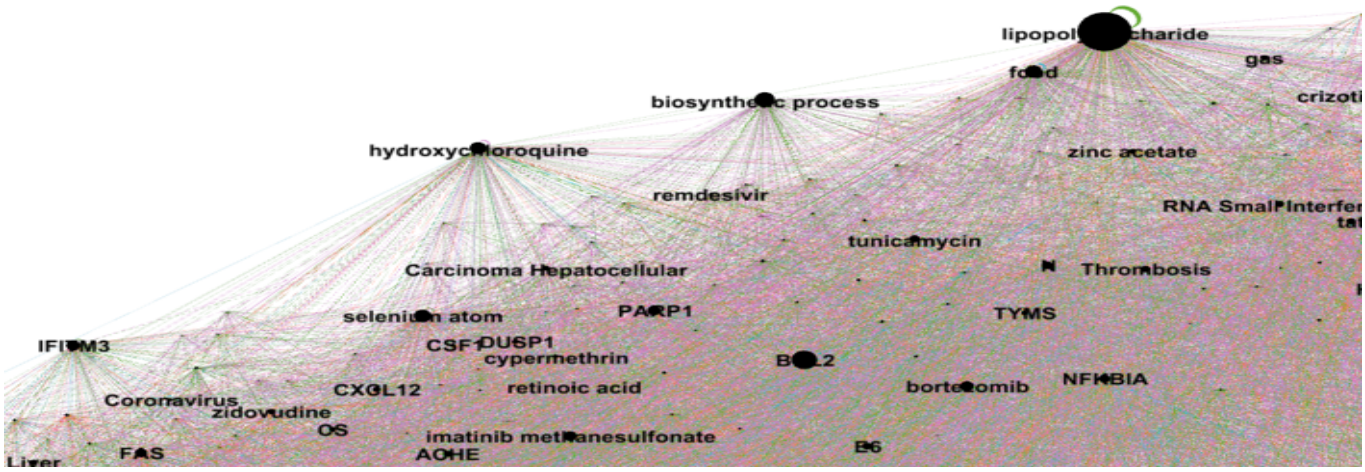

Fig. 1. COVID-19 Knowledge Graph Example

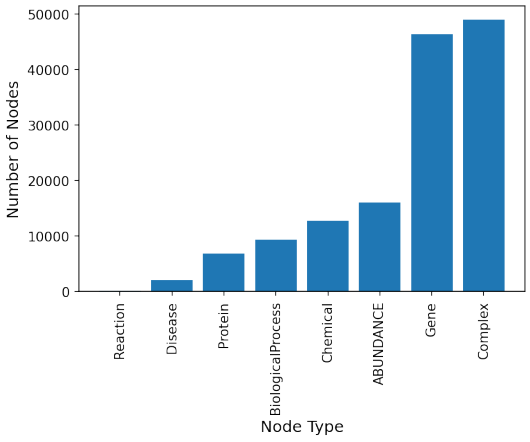

Fig. 2. COVID-19 Knowledge Graph node type distribution

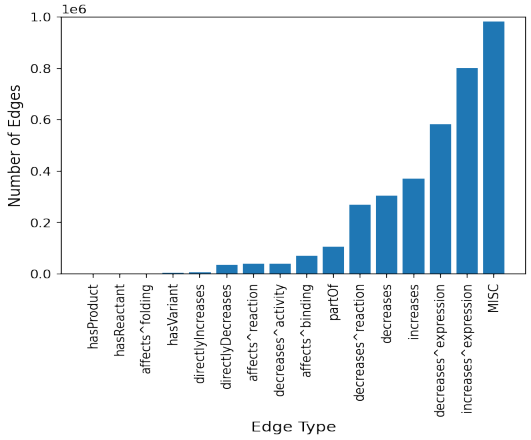

Fig. 4. COVID-19 Knowledge Graph edge type distribution

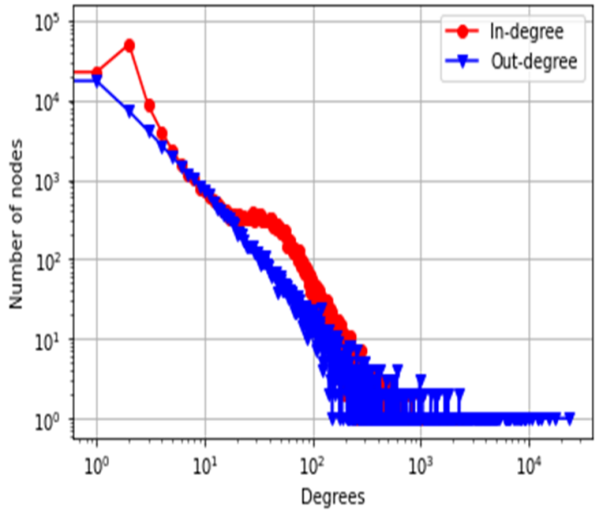

Fig. 3. COVID-19 Knowledge Graph degree distribution

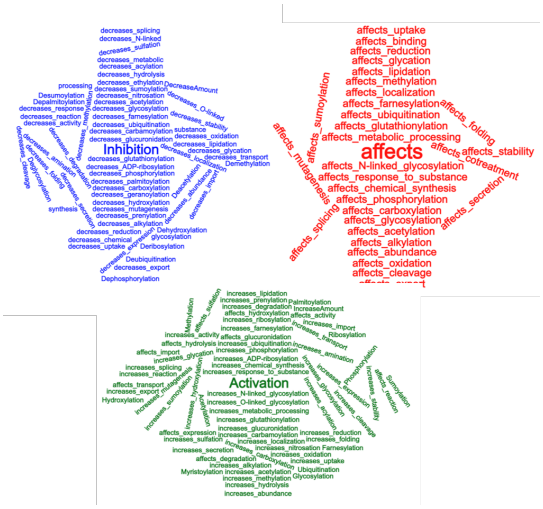

Fig. 5. COVID-19 Knowledge Graph edge clusters

Fig. 6. Simple Path Query Example

```
{
  "name": "template0",
  "summary": "Query to support the claim:
  Lopinavir inhibits SARS-CoV-2",
  "nodedef": [
  {
    "template_id": "node_0",
    "node_type": ["ABUNDANCE"],
    "textValue": ["Lopinavir"]
  },
  {
    "template_id": "node_1",
    "node_type": ["BiologicalProcess"],
    "textValue": ["SARS-CoV-2"]
  }],
  "pathdef": [
  {
    "description": "Path between node_0, node_1",
    "template_id": "path_0",
    "source": ["node_0"],
    "target": ["node_1"]
  }],
  "pathConstraints": [
  {
    "description": "A constraint on the path
    between node_0 and node_1",
    "template_id": "pc_path_0",
    "path_id": ["path_0"],
    "min_hops": 1,
    "max_hops": 5,
    "type_occurrence_filter": {
      "edge_type": ["Inhibition"],
      "values": [1,3,5]
    }
  }]
}
```

Listing 1: QLiG Path Query Example

the piecemeal query construction process required by existing graph query languages. QLiG focuses on the SME-driven query construction and uses *StructDef*, *PathDef*, and *Constraints* to define a query using such higher-order graph element than a finer level description of each node and edge in the query graph.

Figure 7 shows a graph query describing the process of IL6 amplification. It is observed in COVID-19 patients that the simultaneous activation of **NF-kB** and IL6-induced **STAT3** causes a positive feedback loop called IL6-Amplifier, which can lead to cytokine storms. Cytokine storm is a biological event when an infection triggers the immune system to flood the bloodstream with inflammatory proteins called cytokines. It is suspected that SARS-CoV-2 can activate **NF-kB** and IL6-induced STAT3 through independent mechanisms. Figure 7 shows an example of piecemeal construction of the complete biological mechanism to understand the impact of **NF-kB** and IL6-induced **STAT3**. Expressing such domain-specific ques-

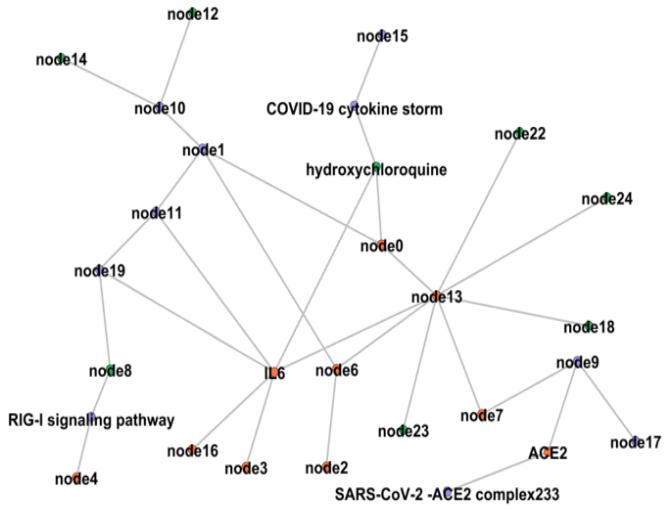

Fig. 7. Graph Query describing IL6-Amplification

Fig. 8. QLiG Query describing IL6-Amplification

tions into a general-purpose graph query language of major graph databases is extremely difficult, time-consuming, and error-prone. Instead of specifying every possible interaction in the biological process, an SME can specify the query in terms of core entities, structures, and interactions involved.

Figure 8 shows a notional structure that describes the SME intended query using QLiG syntax. This graph query can be decomposed into two different pathways and three key nodes along the pathways. The notional structure is an extremely simple query to visualize and search in the KG. Listing 2 shows the JSON serialization of the QLiG query that defines three nodes and two pathways.

In addition to the linear pathways described above, QLiG also describes non-linear queries using structural elements. Figure 9 shows a COVID-19 disease pathway explaining *Coagulation Cascade*. The coagulation cascade is a series of events that lead to damaged blood vessels and clots. COVID-19 patients have been observed to show cytokine-induced inflammation and virus-induced epithelial tissue damage in the lungs that leads to a coagulation cascade. Identifying the key biological processes, genes, proteins, etc. are of utmost importance to expedite drug discovery for COVID-19 patients with such symptoms. A more natural and efficient way to search for this process in a large KG is to define it in terms of key entities and pathways as shown in the notional Figure 10.

Listing 3 shows a partial QLiG query to describe the coagulation cascade process. In addition to the *Pathdef*, QLiD also defines *StructDef*, *structuralConstraints*, and *membership-*

```
{
  "summary": "Structural Template to describe IL6
  amplification.",
  "nodedef": [
  {
    "template_id": "node_0",
    "node_type": ["Protein"],
    "textValue" : "SARS-CoV-2-ACE2-Complex"
  },
  {
    "template_id": "node_1",
    "node_type": ["Gene"],
    "textValue" : "ADAM17"
  },
  {
    "template_id": "node_2",
    "node_type": ["Protein"],
    "textValue" : "Cytokine Storm"
  }],
  "pathdef": [
  {
    "template_id": "path_0",
    "source" : ["node_0"],
    "target" : ["node_1"],
    "node_type" : ["Gene", "Protein"]
  },
  {
    "template_id": "path_1",
    "source" : ["node_1"],
    "target" : ["node_2"],
    "node_type" : ["Gene", "Protein", "Chemical"],
    "importance" : 1
  }],
  "pathConstraints":[
  {
    "template_id" : "pc_0",
    "path_id" : ["path_0"],
    "min_hops" : 1,
    "max_hops" : 5,
    "type_occurrence_filter" : {
      "edge_type" : ["Inhibition",
      "DecreaseAmount"],
      "values" : [2,4]
    }
  },
  {
    "template_id" : "pc_1",
    "path_id" : ["path_1"],
    "min_hops" : 1,
    "max_hops" : 5,
    "type_occurrence_filter" : {
      "edge_type" : ["Inhibition",
      "DecreaseAmount"],
      "values" : [2,4]
    }
  }]
}
```

Listing 2: QLiG Query Specification for IL6-Amplification

Constraints that can be used to describe a group structure in a query pattern. As shown in the listing 3, a textitstructuralConstraints can define a range of number of nodes and edges in a structure. Similarly, a *membershipConstraints* can define an association between a named node and a structure (or a path) that must be matched at the query time.

D. Nested Query

QLiG supports nested queries using the named graph defined above. The *GraphDef* is used to define inner queries as

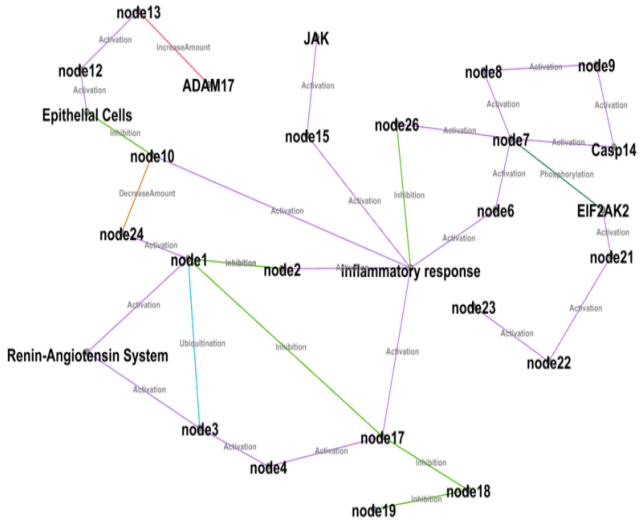

Fig. 9. Graph Query describing Coagulation Cascade in COVID-19 patients

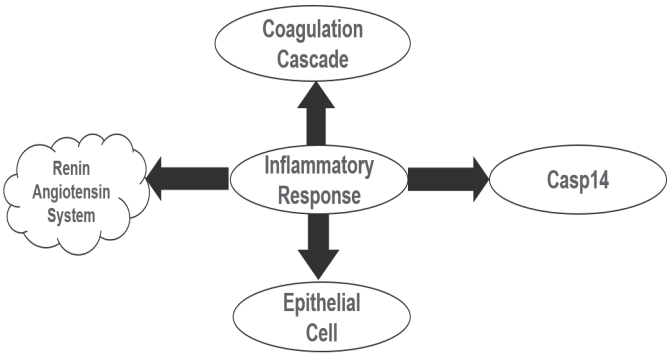

Fig. 10. QLiG Query describing Coagulation Cascade in COVID-19 patients

named graph. An Inner query can be defined as an independent graph query using all the QLiG elements. In contrast to existing graph query languages, QLiG uses reusable constraints such as *membershipConstraints* to specify a set of edges that connects nodes in an inner query to the outer query. For the real-world COVID-19 use case, we use QLiG to search for the "confounders" of a pathway. We extend the path query defined in section IV-B, to search for nodes that have a direct path to two additional nodes along the pathway. Figure 11 shows a notional block diagram of the query. The figure shows that the inner pathway query must be explored before the pathways related to *confounder* nodes are matched.

V. Query Evaluation

Subgraph matching algorithms are used to find matches of a given query graph in a large graph. State-of-the-art graph databases (e.g., Neptune, Neo4J) return a matched pattern in different formats including a named graph, JSON, and tabular data. These formats do not provide a *self-contained* instances required to assess the accuracy of the subgraph matching algorithm. A *self-contained* instance combines the input pattern and a collection of matched nodes and edges.

```
{  "structuralConstraints":[
   {
     "description" : "Structure to describe renin
     angiontensin system triad",
     "template_id" : "sc_0",
     "struct_id" : ["struct_0_triad"],
     "min_num_nodes" : 3,
     "max_num_nodes"  : 3
   },
   {
     "description" : "Structure to describe cell
     decay process involving Epithelial and
     Endothelial cells",
     "template_id" : "sc_1_cell_decay",
     "struct_id" : ["struct_1_cell_decay"],
     "min_num_nodes" : 3,
     "max_num_nodes"  : 5,
     "min_num_edges" : 5,
     "max_num_edges" : 7
   },
   {
     "description" : "Structure to describe
     sub-pathway involving Interferon and possible
     drugs (chemicals)",
     "template_id" : "sc_2_interferon_casp",
     "struct_id" : ["struct_2_interferon_casp"],
     "min_num_nodes" : 5,
     "max_num_nodes"  : 9,
     "min_num_edges" : 5,
     "max_num_edges" : 11
   }],
   "membershipConstraints" : [
   {
     "template_id" : "mc_1_Renin_triad",
     "component_id" : ["struct_0_triad"],
     "nodeset" : ["node_0"]
   },
   {
     "template_id" : "mc_2_Cell_decay",
     "component_id" : ["struct_1_cell_decay"],
     "nodeset" : ["node_2", "node_6"]
   },
   {
     "template_id" : "mc_3_interferon_casp",
     "component_id" : ["sc_2_interferon_casp"],
     "nodeset" : ["node_3","node_7"]
   }]
}
```

Listing 3: QLiG Query Specification describing Coagulation Cascade in COVID-19 patients

```
MATCH
(src)-[r:Inhibition]->(hop)-[r2:Activation]->(dst)
RETURN *
```

Listing 4: Neo4j graph query

Figure 12 shows a graph-based visualization of a notional query described in listing 4.

Instead of an individual pattern that matches, it shows a combined view of all the matched subgraph instances. Similarly, the text-based result in Figure 13 does not show the input query pattern. In contrast, the QLiG uses a *graph in, graph out* approach and supports *self-contained* query result that combines both the input pattern and its matched nodes and edges. This combined approach is useful for evaluation and benchmark workflow as it facilitates automated assessment of matching accuracy, in the context of

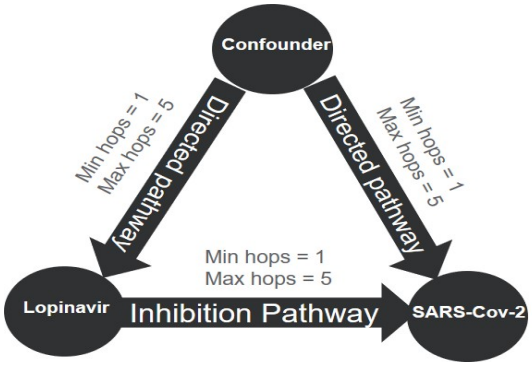

Fig. 11. QLiG Query describing a Confounder Query

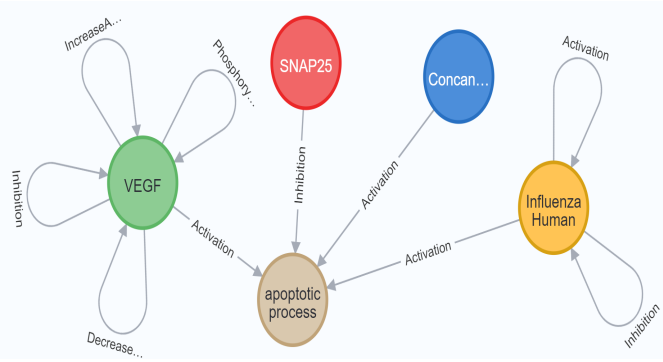

Fig. 12. Neo4j graph-based query result

input query patterns. QLiG defines *submission_nodeset* and *submission_edgeset* for all the graph elements defined in section III. QLiG uses a *in-place* result submission approach using as shown in listing 5.

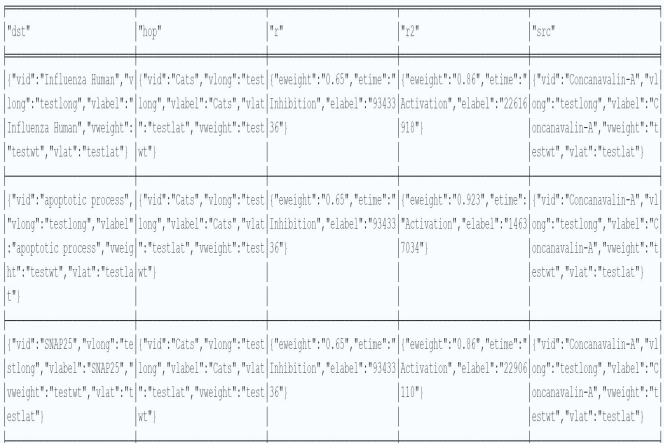

Fig. 13. Neo4j tabular query result

VI. Conclusion and Future Work

We present a novel graph query specification QLiG, that provides graph-based components to construct a query pattern. QLiG allows subject matter experts to formulate a complex

```
{
"pathConstraints": [
{
  "description": "A constraint on the path
  between node_0 and node_1",
  "template_id": "pc_path_0",
  "path_id": ["path_0"],
  "min_hops": 1,
  "max_hops": 5,
  "type_occurrence_filter": {
    "edge_type": ["Inhibition"],
    "values": [1,3,5]
  },
"submission_nodeset": [],
"submission_edgeset": []
}]}
```

Listing 5: QLiG-based evaluation

graph query without specifying a pattern in detail. QLiG uses higher-level concepts such as path and structure, and three types of constraints (semantic, structural, and attribute). Initial implementation of QLiG specification is used in subgraph matching algorithms developed for DARPA MAA program [33]. Future work will focus on standardization of the specification, development of a user interface to auto-generate QLiG syntax, and development of novel capabilities to have interoperability with existing query languages. Future work will also develop performance metrics to measure the impact of QLiG syntax on subgraph matching runtime.

Acknowledgment

We thank the DARPA Modeling Adversarial Activity (MAA) program for funding this project. The associated PNNL project number is 69986. A portion of the research was performed using PNNL Institutional Computing (PIC) at the Pacific Northwest National Laboratory.

References

[1] A.-L. Barabási, R. Albert, and H. Jeong, "Scale-free characteristics of random networks: the topology of the world-wide web," *Physica A: statistical mechanics and its applications*, vol. 281, no. 1-4, pp. 69–77, 2000.

[2] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *Link mining: models, algorithms, and applications*. Springer, 2010, pp. 337–357.

[3] C.-C. Chu and H. H.-C. Iu, "Complex networks theory for modern smart grid applications: A survey," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 7, no. 2, pp. 177–191, 2017.

[4] X. Shen, X. Gong, X. Jiang, J. Yang, T. He, and X. Hu, "High-order organization of weighted microbial interaction network," in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2018, pp. 206–209.

[5] J. Klaise and S. Johnson, "The origin of motif families in food webs," *Scientific reports*, vol. 7, no. 1, pp. 1–11, 2017.

[6] J. A. Cottam, S. Purohit, P. Mackey, and G. Chin, "Multi-channel large network simulation including adversarial activity," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3947–3950.

[7] H. Paulheim, "Knowledge graph refinement: A survey of approaches and evaluation methods," *Semantic web*, vol. 8, no. 3, pp. 489–508, 2017.

[8] L. LiuQiao, L. DuanHong *et al.*, "Knowledge graph construction techniques," *Journal of computer research and development*, vol. 53, no. 3, p. 582, 2016.

[9] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A large ontology from wikipedia and wordnet," *Journal of Web Semantics*, vol. 6, no. 3, pp. 203–217, 2008.

[10] Q. Yang and S.-H. Sze, "Path matching and graph matching in biological networks," *Journal of Computational Biology*, vol. 14, no. 1, pp. 56–67, 2007.

[11] H. Dai, C. Li, C. W. Coley, B. Dai, and L. Song, "Retrosynthesis prediction with conditional graph logic network," *arXiv preprint arXiv:2001.01408*, 2020.

[12] C. McCreesh, P. Prosser, and J. Trimble, "The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants," in *International Conference on Graph Transformation*. Springer, 2020, pp. 316–324.

[13] Z. Lou, J. You, C. Wen, A. Canedo, J. Leskovec *et al.*, "Neural subgraph matching," *arXiv preprint arXiv:2007.03092*, 2020.

[14] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 217–218.

[15] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan *et al.*, "Amazon neptune: Graph data management in the cloud." in *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018.

[16] "Blazegraph," https://blazegraph.com/, accessed: 2021-09-02.

[17] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[18] M. Frank, M. Eric, and M. Brian, "Rdf 1.1 primer," https://www.w3.org/TR/rdf11-primer/, 2020.

[19] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, 2010.

[20] "SPARQL," https://www.w3.org/TR/sparql11-query/, accessed: 2021-09-02.

[21] "GQL," https://www.gqlstandards.org/, accessed: 2021-09-02.

[22] "Open Cypher," https://opencypher.org/, accessed: 2021-09-02.

[23] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1433–1445.

[24] "Gremlin," https://tinkerpop.apache.org/gremlin.html, accessed: 2021-09-02.

[25] "GQL," https://graphql.org/, accessed: 2021-09-02.

[26] "Apache Tinker Pop," https://tinkerpop.apache.org/, accessed: 2021-09-02.

[27] "AWSNeptuneGremlin," https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-gremlin.html, accessed: 2021-09-02.

[28] "JenusGraph," https://docs.janusgraph.org/, accessed: 2021-09-02.

[29] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda *et al.*, "G-core: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1421–1432.

[30] L. L. Wang, K. Lo, Y. Chandrasekhar, R. Reas, J. Yang, D. Eide, K. Funk, R. Kinney, Z. Liu, W. Merrill *et al.*, "Cord-19: The covid-19 open research dataset," *ArXiv*, 2020.

[31] B. M. Gyori, J. A. Bachman, K. Subramanian, J. L. Muhlich, L. Galescu, and P. K. Sorger, "From word models to executable models of signaling networks using automated assembly," *Molecular systems biology*, vol. 13, no. 11, p. 954, 2017.

[32] S. Purohit, N. Van, and G. Chin, "Semantic property graph for scalable knowledge graph analytics," *arXiv preprint arXiv:2009.07410*, 2020.

[33] E. Hovy, "Modeling Adversarial Activity (MAA)," https://www.darpa.mil/program/modeling-adversarial-activity.