

GraphMCS: Discover the Unknown in Large Data Graphs

Elena Vasilyeva¹

Maik Thiele²

Christof Bornhövd³

Wolfgang Lehner²

¹SAP AG
Dresden, Germany

elena.vasilyeva@sap.com

²Database Technology Group
Technische Universität Dresden, Germany

firstname.lastname@tu-dresden.de

³SAP Labs, LLC
Palo Alto, USA

christof.bornhoevd@sap.com

ABSTRACT

Graph databases implementing the property graph model provide schema-flexible storage and support complex, expressive queries like shortest path, reachability, and graph isomorphism queries. However, both the flexibility and expressiveness in these queries come with additional costs: queries can result in an unexpected, empty answer. To understand the reason of an empty answer, a user normally has to create alternative queries, which is a cumbersome and time-consuming task.

To address this, we introduce diff-queries, a new kind of graph queries, that give an answer about which part of a query graph is represented in a data graph and which part is missing. We propose a new algorithm for processing diff-queries, which detects maximum common subgraphs between a query graph and a data graph and computes the difference between them. In addition, we present several extensions and optimizations for an established maximum common subgraph algorithm for processing property graphs, which are the foundation of state of the art graph databases.

Categories and Subject Descriptors

H.2.4 [Systems]: Query processing

Keywords

Maximum Common Subgraph, Flexible Query Answering

1. INTRODUCTION

New kinds of data and their analysis increase the demand for flexible data models supporting data of different degrees of a structure. Graph databases implementing the property graph model [11] are a reasonable answer to this demand. They support diverse data with different degrees of a structure in the form of a graph. A diverse schema of vertices and edges is represented by an arbitrary number of attributes,

which can differ between vertices or edges of the same semantic type. A major advantage is that such systems do not require a predefined rigid database schema.

However, the flexibility provided by graph databases and the property graph model comes with additional costs. A user of graph databases typically has only limited knowledge about stored data, which complicates the creation of queries. He can overspecify queries that can result in an empty answer. Any empty response causes confusion on the user side, since its reason is unclear: was the query overspecified or are some data missing in the database? To answer this question, a user needs a possibility for explorative queries and guidance through the query answering process. To provide this, a system has to be able to give intermediate points in query processing, which describe the already discovered and still missing parts of a query graph. As a result, a user could discover overspecified query parts or conclude that some information is missing in a dataset and, therefore, has to be obtained from external data sources [8, 15].

Related Work.

If the result of a query does not meet the user's expectations, he can conduct "Why Not?" queries [3] to determine why the result set does not include the items of interest. It is assumed that a user cannot process the data manually because of their large volume and complexity. A user specifies items of interest with attributes or key values. Then a "Why Not?" query could be "Why are the items with predicate P not in the result set?"

There are several ways of answering "Why Not?" queries. On the one hand, the causes for an empty answer can be found, as done in [3], where a "Why Not?" query applies a set of manipulations to the original query. As an answer, the system provides an operator from the original query, which removes required items from the result set. This approach relies on manipulations of operators and derives an answer for a specific item. On the other hand, a provenance-based explanation can be delivered by computing the provenance of possible answers for SPJ queries, like for example in [9]. This is also possible to refine the query in such a way, that the items of interest appear in the result set. In this case, the explanation for a "Why Not?" query is based on an automatically generated query, which response consists of the original results and the items of interest [13].

In contrast, our problem is to find missing structural parts of a query that prevent the system from delivering a non-empty answer. At this point, we are not interested in specific attributes and items (which is done in the relational case).

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

To the best of our knowledge, the question of discovering a missing query part in a data graph has not been addressed in graph database research.

Contributions.

In graph databases, a query can be understood as a pattern that has to be sought in a large data graph. To tell a user which query parts were discovered in a data graph and which are missing, we propose (1) to find maximum common subgraphs in a data graph for a given query, and (2) to calculate the difference between them and a query graph. As a result, the system yields a list of discovered maximum common subgraphs as starting points and undiscovered parts of a query graph as a subject for the future explorative search.

As our contributions we present in this paper diff-queries, a new kind of graph database queries, which give an answer about existing and missing query parts. Diff-queries deliver discovered maximum common subgraphs of a data graph and corresponding missing parts of a query graph. We introduce an all-covering spanning tree allowing for the processing of a whole query graph and weakly connected graphs. This tree allows us to get larger subgraphs than the standard solutions for connected graphs. We also provide several optimizations for diff-queries to deliver a final result faster and to reduce the number of intermediate subgraphs.

The rest of the paper is structured as follows. We introduce the property graph model and basic algorithms for discovering maximum common connected subgraphs between two graphs in Section 2. We outline the processing of diff-queries in a graph database in Section 3. In Section 4 we describe challenges of multiple starts and weakly connected graphs for the standard algorithm and their solutions. We provide several optimizations for our proposed algorithm in Section 5 and evaluate our approach in Section 6.

2. MAXIMUM COMMON CONNECTED SUBGRAPH DETECTION

As an underlying data model we use the property graph model [11]. It represents a graph as a directed multigraph, where vertices are entities and edges are relationships between them. Each edge and vertex can be described by multiple attributes and their values. The attributes can differ concerning edges and vertices – even if they are of the same semantic type. We define a **property graph** as a directed graph $G = (V, E, u, f, g)$ over attribute space $A = A_V \dot{\cup} A_E$, where: (1) V, E are finite sets of vertices and edges; (2) $u : E \rightarrow V^2$ is a mapping between edges and vertices; (3) $f(V)$ and $g(E)$ are attribute functions for vertices and edges; and (4) A_V and A_E are their attribute space.

A graph $G' = (V', E', u', f', g')$ is a **connected subgraph** of G , if $V' \subseteq V, E' \subseteq E, u' |_{u}, f' |_{f}$, and $g' |_{g}$.

Given a data graph G_d and a query graph G_q , the graph $G'_d = (V'_d, E'_d, u'_d, f'_d, g'_d)$ is a **common connected subgraph** of graphs G_d and G_q , if G'_d is a connected subgraph of G_d and G'_d is a connected subgraph of G_q . There may be multiple common connected subgraphs in a data graph G_d for a query graph G_q .

For property graphs, a **maximum common connected subgraph** G'_d is a common connected subgraph of a data graph G_d for a query graph G_q such that there exists a match S_{max} in G_d for G_q such that for any match S in G_d for G_q , $S \leq S_{max} : V \leq V_{max} \cup E \leq E_{max}$.

Finding Maximum Common Connected Subgraphs.

To tell, which part of a query can be found in a data graph, we have to find maximum common subgraphs in a data graph G_d for a query graph G_q . This can be done by maximum common connected subgraph algorithms. The computation depends on how a graph is stored and processed. A commonly used adjacency matrix or adjacency list allow the compact storing of graphs and their efficient processing [5]. For example, a matrix M consists of $n \times n$ elements, where n is the number of vertices in a graph. Each element of a matrix a_{ij} with a value 1 represents an edge between vertices i and j . A maximum common connected subgraph can be calculated by linear algebra operations. If a graph is a property graph, then its attributes can be stored in separated structures and can be used during prefiltering.

Ullmann in [14] describes a brute-force tree-search enumeration procedure, which efficiently eliminates successor vertices. It excludes some elements from a matrix M and, thereby, reduces the search space. The algorithm is commonly used for exact graph matching. Another backtracking algorithm – the McGregor algorithm [10] – also works on matrices and provides extension points for pruning techniques and prefiltering options.

Both methods rely on labeled graphs, which differ from our underlying property graph model [11]. To apply them to our use case, these algorithms have to be adapted to work on properties on edges and vertices.

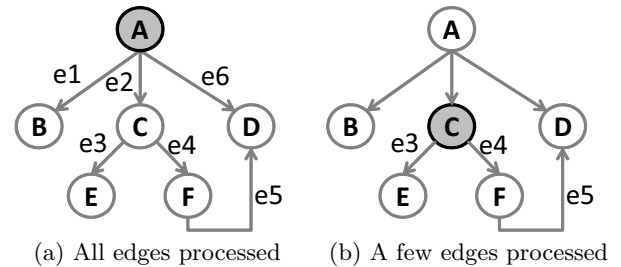


Figure 1: Depth-first search

Ullmann’s [14] and McGregor’s [10] algorithms are backtracking algorithms, which are a base for traversal operations in graph databases. Both methods implement a depth-first search that begins at the root and traverses the graph as far as possible along each branch before backtracking. Assuming the search starts from the grey vertex, in the example shown in Figure 1(a) we begin then from vertex A and explore all edges of the graph as follows: $e1, e2, e3, e4, e5, e6$. If we start from vertex C like in Figure 1(b), then only edges $e3, e4, e5$ are traversed. To ensure the discovery of all maximum common connected subgraphs, the search is conducted for each vertex of a query, and a data graph is treated as undirected. This makes the search NP-complete.

A maximum common connected subgraph problem can also be modified for the search of a maximum clique like in the Durand-Pasari algorithm [7] and in the Balas Yu algorithm [1]. These algorithms are also tree-search algorithms. Some of them work better with sparse graphs, others with dense graphs. According to [4], the McGregor algorithm shows good results in all cases and has the best space complexity. Based on these observations, we have chosen it as the base for our discovery of maximum common connected subgraphs.

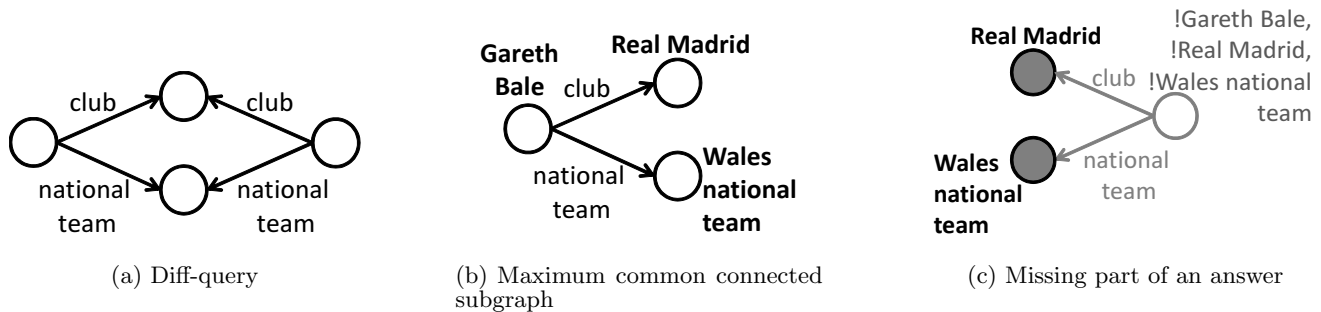


Figure 2: A diff-query and its answer: which two vertices play in the same national team and club?

Difference Graphs.

With a maximum common connected subgraph algorithm we can determine, which query part has an answer in a data graph. To detect, which structural part is missing we need to compute the difference graph – the difference between discovered maximum connected subgraphs and a query graph.

A difference graph includes those query vertices and edges, which were not discovered during query processing, and the instances of query vertices adjacent to a maximum common connected subgraph.

For property graphs we define a **difference graph** as a graph $G'_q = (V'_q, E'_q, u'_q, f'_q, g'_q, V'_d(adj), C)$, where $V'_q \subseteq V_q, E'_q \subseteq E_q, u'_q |_{u_q}, f'_q |_{f_q}, g'_q |_{g_q}, V'_d(adj)$ are adjacent vertices, and C is a set of non-adjacent discovered vertices to be excluded from the further explorative search.

In [6] the authors compute difference graphs from conceptual graphs. The first graph is transformed into one of its large common subgraphs and the set of applied operations is stored. Then the large common subgraph is transformed into the second graph, and the set of applied operations is stored. Finally, both stored sets are concatenated in a difference graph. Operations include standard insert and remove operations and specific operations for conceptual modeling like a generalization or a specialization of a concept.

In our work we do not provide any conceptual analysis and reasoning for “non-existing” edges from a hierarchical taxonomy. Moreover, the first step is redundant: we record the discovered parts of a graph during the graph traversal.

Diff-queries.

If a user gets an empty response to his query, he can conduct a **diff-query** that shows, which query part is addressed in data and which part is missing. For this purpose, it detects maximum common connected subgraphs and computes their corresponding difference graphs, which prevent a system from the delivery of a non-empty answer to a user.

Assuming we search for two soccer players from the same national team and the same club. Then the query graph could be represented as in Figure 2(a). A possible answer to this diff-query would consist of a maximum common connected subgraph G'_d as shown in Figure 2(b), and a missing part of a query with constraints G'_q as in Figure 2(c). The first part includes all discovered instances of vertices and edges like “Gareth Bale”, “Real Madrid”, and “Wales national team”. The second part consists of instances of discovered adjacent vertices (dark grey), missing query vertices and edges (grey), and constraints for vertices (grey).

3. PROCESSING OF DIFF-QUERIES IN GRAPH DATABASES

In this section we shortly describe the graph database we use in our prototype and outline the diff-query computation process. The processing of diff-queries consists of two steps: the detection of a maximum common connected subgraph by using an extended version of the McGregor algorithm [10] for property graphs, and the computation of a difference graph between the discovered maximum common connected subgraph and a query graph.

3.1 Storage Representation

In our prototype system the property graph model is implemented as a graph abstraction on top of a RDBMS, which uses separate tables for vertices and edges. Vertices are described by a set of columns for their attributes, and edges are stored as simplified adjacency lists in a table. Each edge can have multiple attributes, which are stored together with its description. All edges and vertices have unique identifiers.

To process such a graph efficiently, we use an in-memory column database, which supports optimized flexible tables (new attributes can efficiently be added and removed) and provides advanced compression techniques for sparsely populated columns like in [2, 12]. This abstraction allows us to store graphs with an arbitrary number of attributes without a predefined rigid schema.

The graph database provides the following operations: insert, delete, update, filter based on attribute values, aggregation, and graph traversal in a breadth-first manner. Traversal along directed edges is possible in both directions with the same performance.

Queries to the database are represented via graphs, where vertices describe entities and edges describe connections between them. Each description of vertices and edges can include predicates for attribute values. A specific vertex is represented by its identifier in a query graph.

3.2 Detection of Maximum Common Connected Subgraphs

To detect the maximum commonality between a query and a data graph, we have chosen the McGregor maximum common connected subgraph algorithm [10] presented in Algorithm 1, which uses a depth-first search (Figure 1).

To leverage the McGregor algorithm for property graphs, the edges and vertices tables of our graph database have to be processed. First, the projection on a vertices table reduces the number of start vertices at line 4. Second, each

Algorithm 1 The MCCS algorithm for a property graph

```
1: function MCCSSEARCH(query graph  $G_q$ )
2:    $graphs, tmp$ 
3:   for all edge  $q_i$  in  $G_q$  do
4:      $sources_i = getSourceVertices(q_i)$ 
5:      $graph = DFS(sources_i, q_i, true, graphs_i)$ 
6:      $graphs.addGraph(graph)$ 
7:   for all  $graphs_i$  do
8:     if  $graphs_i > tmp$  then  $tmp = graphs_i$ 
9:   return  $tmp$ 
10: /*depth-first search*/
11: function DFS( $sources, edge, isStart, graph$ )
12:   for all  $sources_j$  do
13:     if  $isStart$  then  $edge = getNextEdge(edge)$ 
14:     if  $noNextEdge$  then return  $graph$ 
15:      $targets = traverse(edge)$ 
16:      $filterTargets(targets)$ 
17:     for all  $targets_d$  do
18:        $graph.addEdge(edge, sources_j, targets_d)$ 
19:        $graph = DFS(targets_d, edge, false, graph)$ 
20: return  $graph$ 
```

step is traversed by the graph traversal operator at line 15. Finally, the target vertices are filtered according to their predicates (see line 16). To ensure that the algorithm finds a maximum common connected subgraph, it is started multiple times from all query vertices as starting points at line 5. The maximum common connected subgraph is stored for each starting point in a set. After all runs the best subgraph is chosen from the collected set (see lines 7-9).

3.3 Computation of a Difference Graph

To compute a missing part of a query, we use a query graph and a discovered maximum common connected subgraph. The process consists of two steps: (1) the split of discovered and undiscovered vertices and edges, and (2) the completion of an undiscovered part with attributes or vertices conditions.

In our first step, during processing we store the mapping between data edges and query edges, data vertices and query vertices in temporary tables. The difference graph consists of query edges and vertices, which are not presented in these temporary tables. Some edges in the difference graph will have only single vertices at their ends, because other end vertices have already been traversed. Therefore, we have to include the discovered edges' ends into the difference graph in the second step – the completion of the difference graph with attributes or vertices conditions.

In the second step we detect, which conditions have to be applied to the graph discovered in the first step. We study the table with discovered vertices and a query description and assign conditions to the difference graph according to several rules: If a query edge is not discovered, but at least one of its end vertices has already been found, then this is a positive condition. It means we include a discovered end vertex into the difference graph. In the example presented above the two dark grey vertices represent such conditions (see Figure 2(c)). Such vertices are included into a difference graph and can be used as starting points for a future explorative search. If a query vertex and all its query edges



Figure 3: Weakly connected graphs

(incoming and outgoing) are discovered in a data graph, then this vertex is a negative condition, and its instance has to be excluded from the non-discovered query vertices. In our example this can be “Gareth Bale” (see Figure 2(c)), which does not have to be considered in a future explorative search.

4. PROBLEMS OF MULTIPLE STARTS AND WEAKLY CONNECTED GRAPHS

The general version of the McGregor algorithm [10] takes all vertices of a query as starting points and iterates through them. So, the system traverses the same data edges multiple times. On the one hand, this ensures that no edge is left out and all maximum common connected subgraphs are discovered. On the other hand, this generates large intermediate results and increases the response time dramatically.

We figure out two problems, which solution can increase the performance of the algorithm, find larger graphs, and reduce the number of runs. First, the algorithm works only with connected graphs, therefore, only one-directed search for directed graphs is done. This can be solved by the extension of the search for weakly connected graphs. Second, we can miss some maximum common connected subgraphs by start from a single vertex. This can be solved by a restart strategy for non-traversed edges.

Processing of Weakly Connected Graphs.

The McGregor maximum common connected subgraph algorithm, which is a base for our algorithm, processes the directed graph only in a forward direction. This can limit the size of discovered subgraphs and deliver subgraphs of potentially smaller size than could be determined. To ensure the discovery of a maximum subgraph, we have to choose that vertex as a root, where all vertices can be reached from. Because the algorithm works only in a forward direction, it is not always possible to find the best root vertex.

For example, the query presented in Figure 3 does not have any ideal root. This is a weakly connected graph: it is connected, if directions of edges are not considered. For this query the McGregor algorithm can discover subgraphs only with two edges and three vertices (ABC or BCD). Therefore, we need to modify the algorithm to also consider unreachable components.

To process queries with unreachable components, we introduce an all-covering spanning tree that has the following characteristics. If the whole query graph is available in data, then the all-covering spanning tree is able to cover all query vertices and edges in a single run. An edge can be included into the search in forward or backward directions. In case of a backward direction, an edge is marked with a flag “back”. This can be done without additional effort because of the underlying data model and the graph traversal operator provided by the database like in [12]. Another way would be to make a graph basically undirected with duplicated data or double table scans, which is less efficient.

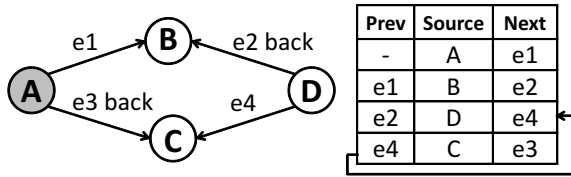


Figure 4: All-covering spanning tree and the backtracking procedure

We adapt Algorithm 1 to work with an all-covering spanning tree. From now on, we consider all edges for each vertex. Outgoing edges have priority over incoming edges and are processed first. After all outgoing edges are traversed, incoming edges are considered.

To guarantee a correct search, we maintain the all-covering spanning tree as a temporary table and refer to it during the backtracking procedure. It records the mapping between previously traversed and next edges. The all-covering spanning tree has three columns: a previous edge, a source vertex, and a next edge. To save space, we can use a Boolean identifier of a traversed edge instead of an identifier for a source vertex. For ensuring the simplicity of explanation, we use vertices in the following example.

Assuming the search for a query presented in Figure 4 begins from vertex A . At initialization, the spanning tree is empty. Vertex A has two outgoing edges. After we have followed edge $e1$, we add the following entry into the table: no previous edge, source vertex is A , next edge is $e1$ ($-;A;e1$). Now we are at vertex B without any outgoing edges. We take incoming edge $e2$, mark it with “back”, and add an entry into the table ($e1; B; e2$). We repeat the process and traverse edges $e4, e3$. Finally, we are at vertex A without any non-traversed edges and start the backtracking.

The backtracking procedure is done according to the created all-covering spanning tree. The last traversed edge is $e3$. We check its entry (column “Next”) in the mapping table, take its previous edge $e4$ and go to source vertex C . There are no other non-traversed edges for vertex C , and so we continue the backtracking. The predecessor of $e4$ is $e2$ with vertex D , so we move to it. The procedure continues until it gets to source vertex A , where no further non-traversed edges exist.

A graph database gives the possibility of changing the direction through suitable storing and processing of edges. All edges are stored in a forward direction – “from a source to a target”. In case of the backward traversal, the graph traversal operator changes the order of columns to be searched: “from a target to a source”. Therefore, we need only to change the direction of an edge in the query description and pass it to the traversal operator.

Restart Strategy.

With the all-covering spanning tree, we can construct a traversal path, which includes all vertices and edges, and process weakly connected graphs. Hereby, we solve the first problem of the one-directed search. Now we do not need to iterate through all vertices multiple times. We just take one vertex and search from it. This approach works well if all edges are represented in a data graph. The absence of some edges in the data graph can split a query graph into several subgraphs, which are unreachable from each other.

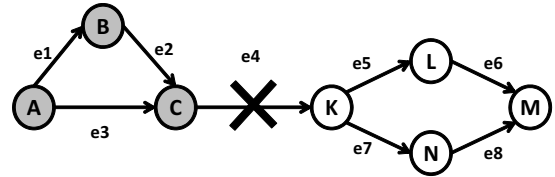


Figure 5: Only white or grey part is traversed

In this case if we start with a vertex from a smaller subgraph, we will miss a maximum common connected subgraph from another subgraph. Thereby, we come to the second problem of the algorithm, which can be solved by a restart strategy.

If we start a search from a single node that is located in the smaller connected subgraph of a query graph, we can potentially miss the larger subgraph, provided by another subgraph. The problem can be explained with a query graph containing a bridge. If a query has a bridge (see Figure 5), which is not addressed in the data graph, then only a subset of vertices and edges is traversed. In our example query edge $e4$ does not have any matching data edges. In this case, a maximum common connected subgraph found by our algorithm would be the white or the dark-grey part. Therefore, if we do a single run in the dark area the maximum common connected subgraph will be missing, which is located in the white area. To solve this problem, we can resume the search with the edges, which were not traversed. The final maximum common subgraph would be unconnected and would contain all discovered maximum common connected subgraphs.

We maintain a list of traversed edges of a query graph. After the first set of maximum common connected subgraphs is returned, we remove those edges from the list that have already been traversed. The next step is taken from this set. This strategy ensures the discovery of a maximum common subgraph for a given start vertex, if an all-covering spanning tree was constructed.

For example, at the beginning a query graph in Figure 5 has an empty list of traversed edges. Assuming we start from the edge $e1$ and find edges $e1, e2, e3$, but the edge $e4$ is missing from a data graph. We add all four edges into the list of traversed edges and remove them from the list of start edges. We choose the next start among the edges $e5, e6, e7, e8$. The search from any of them will find the same subgraph of four edges. This is the maximum common connected subgraph. If we concatenate it with the first discovered maximum common connected subgraph, then we will get a maximum common unconnected subgraph. So, as a maximum common subgraph we will get a set of unconnected parts. This reduces the number of intermediate results and gives a notion to a user about which edges should exist to complete the graph. Such a methodology can potentially return larger subgraphs than the strategy for connected subgraphs.

Therefore, with all-covering spanning trees and restart strategies we can limit the number of restarts and find a maximum common unconnected subgraph, which then can be used for the further explorative search and the integration of missing data. In the following we use maximum common connected and unconnected subgraphs and refer to them jointly as maximum common subgraphs.

5. OPTIMIZATION STRATEGIES

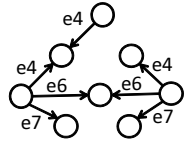
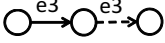
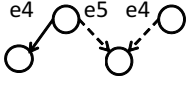
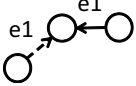
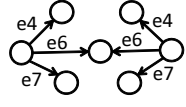
Experiment	Configuration	Topology: Path	Topology: Zigzag	Topology: Star	Optimization
Figure	Figure 6	Figure 7(a)	Figure 7(b)	Figures 7(c)-7(d)	Figures 7(e)-7(f)
Query					
Edge Types	e4, e6, e7	e3	e4, e5	e1	e4, e6, e7

Table 1: Diff-query templates used in the evaluation

To increase the efficiency of the proposed algorithm, we have developed several optimizations for start and restart vertices, and early termination conditions.

5.1 Choice of Start and Restart Edges

The general McGregor algorithm [10] processes a graph from all query vertices, and then the biggest graph is chosen and delivered as a maximum common connected subgraph. With all-covering spanning tree and multiple restarts as proposed in Section 4, we can ensure that from each query vertex the whole query can potentially be traversed in the best case. The question is: Which query vertex should be taken as a start? The size of the intermediate results strongly depends on the cardinality of the processed edges and vertices. If a query graph is described very generally, then it will result in a large amount of intermediate results. To decrease it, we extend our algorithm with several strategies to select a start vertex, a start edge, and a next branch to traverse. For example, we can make decisions based on the cardinality of predicates or on the degree of a vertex.

Number of Incoming/Outgoing Edges.

The order of edges can be chosen according to the number of their previous or next edges. A vertex with the maximal degree is selected as the starting point. For a vertex with a higher degree, more edges need to be processed, and, therefore, we can discover a maximum common subgraph earlier. This strategy can potentially reduce the number of restarts.

Edge and Vertex Cardinality.

Before executing a query, the system calculates the cardinality for all vertices and edges in a query. It then sorts them separately according to the number of items, which should be returned by a system in an ascending order. We choose the edge with the lowest cardinality as the start edge. If we use an all-covering spanning tree, then we can also choose a search direction, based on the cardinality of a source and a target. Otherwise, we use forward processing as default. The same strategy can be applied to restarts, but only the cardinality for edges is considered. In addition, this method has the advantage that if an edge has cardinality = 0 then it is discarded from the search. This reduces the number of table scans and makes the search more efficient.

5.2 Threshold-based Termination Condition

In general, the search stops when no more edges are found and a backtracking procedure returns to start. In addition, there can be cases, when a system can stop the search earlier.

A threshold can be an estimated size of a maximum common subgraph or the number of discovered maximum common subgraphs, which could be derived from a data graph. To calculate these numbers, we can reuse the above presented cardinality of a query. If a query graph has N edges, and for M edges the $cardinality(M) > 0$, where $M \in N$, then the maximum common subgraph can have only M edges. After M edges are found, the search can be safely stopped. Similar rules can be formulated for sources and targets.

Assuming we have a query with four vertices and three edges with the following predicate cardinalities: $card_{edge1} = 5$, $card_{edge2} = 2$, $card_{edge3} = 0$, then the maximum common subgraph can only consist of up to two edges, and we can have a maximum of five graphs like this. We can terminate our search, after the first subgraph with two edges has been discovered.

6. EVALUATION

In this section we evaluate diff-queries and proposed optimization techniques. We describe the evaluation setup in Section 6.1. Then we discuss the scalability of the best configuration, derived in Section 6.2, for different query topologies in Section 6.3. Finally, we evaluate start and restart strategies in Section 6.4.

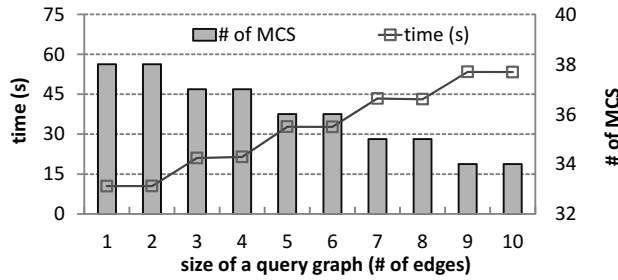
6.1 Evaluation Setup

We have implemented our algorithm and its optimizations in an in-memory column database, which provides the graph abstraction as described in Section 3.1. We have created a property graph from DBpedia RDF triples, where labels represent attribute values of entities. It has about 20K vertices and 100K edges. The evaluated queries are presented in Table 1. We have tested each case for each query ten times and have taken the average as a measure.

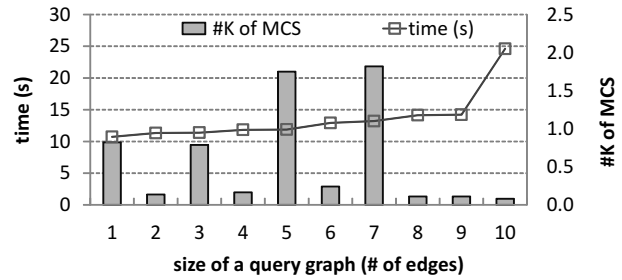
6.2 Configuration

In this section we study several configurations of the algorithm: multiple starts from all edges without the all-covering spanning tree (only for connected components), with the all-covering spanning tree (for weakly connected components), and restart strategy (also for unconnected components).

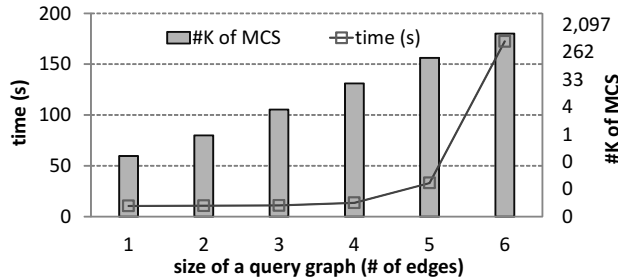
As we can see in Figure 6, the restart strategy discovers larger graphs with shorter response times and less intermediate and final results. Although the method with the all-covering spanning tree has a longer response time (because of the tree construction), it discovers larger graphs. The response time and the size of the maximum common subgraph (MCS) are the best for the restart strategy with the tree construction.



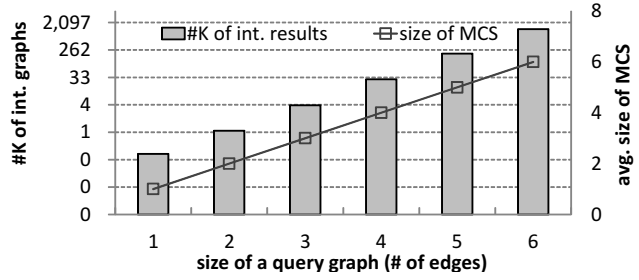
(a) Topology: path



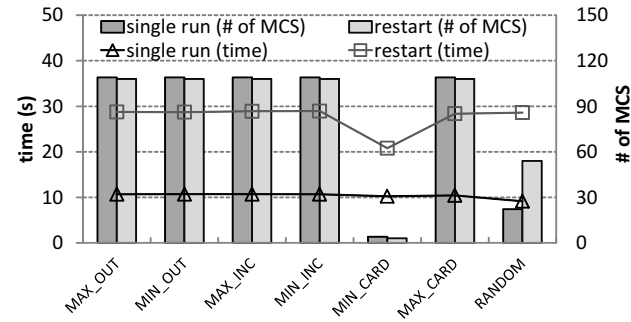
(b) Topology: zigzag



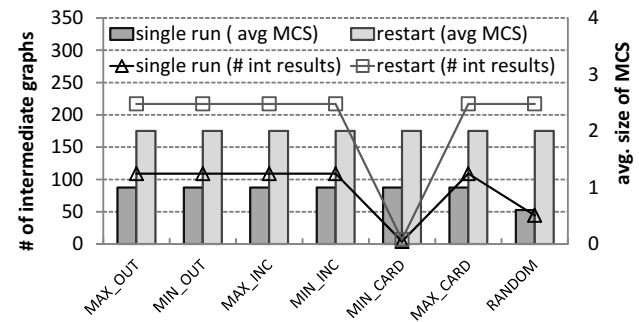
(c) Topology: star



(d) Intermediate results for a star



(e) Optimization: response time evaluation



(f) Optimization: intermediate results

Figure 7: Evaluation of different topologies and optimization strategies

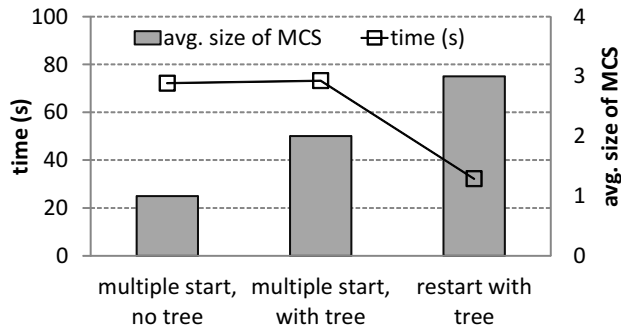


Figure 6: Evaluation of configurations

6.3 Topology

In this section we use the best configuration of the previous step: restart with the all-covering spanning tree. For its evaluation on different graph topologies, we have const-

rueted several queries, which consist of the edges of similar semantic (for a specific topology). The star and path topologies use a single type of edges, while the zigzag evaluates queries with two edge types. The evaluation results are presented in Figures 7(a)-7(d).

In the path each second edge is missing, so the number of MCS decreases. The star tends to increase the number of solutions, because all edges have the same starting point and larger graphs are combined from smaller graphs. The number of MCS for the zigzag evolves dramatically, because we use edges of two types, otherwise, the behavior would be similar to behavior of the path. With the size of a query graph, the response time is growing linearly, except the star, which is explained by the growing intermediate results (see Figures 7(c)-7(d)).

Comparing the results of the evaluation on three topologies, we conclude that, first, the response time dependency on the number of MCS is linear. Second, for the star topology, the size of a result set grows if edges of the same semantic type are used. Third, the response time depends on both factors: size of intermediate results and size of a query

graph. Fourth, to ensure the linear dependency, optimization strategies have to be used, which reduce the number of intermediate results.

6.4 Optimization Strategies

We evaluate start strategies for a single start and for restart (see Figures 7(e)-7(f)). We observe that with the restarts we can increase the average size of MCS. Regardless of the strategy, we get MCS of the same size by using the restart configuration. The response time for the search can be reduced by using an appropriate optimization strategy. For example, the “maximum cardinality” strategy reduces the number of intermediate results, the number of MCS, and the processing time. Although the “random strategy” gives less intermediate results, on average it discovers smaller MCS.

Comparing the results of this evaluation, we conclude that with the restart strategy we can find bigger common subgraphs without starting with each edge multiple times. Optimizations can reduce the number of intermediate results, the number of MCS, and the response time. If characteristics of edges (degree, predicate) are similar, all strategies provide similar results. The strategies of cardinality can be even more efficient, if after the edge selection the direction of its processing is chosen according to the vertices’ cardinality.

With the evaluation we show that the restart configuration and all-covering spanning tree can be used without the start from each query edge. They facilitate to find bigger maximum common unconnected subgraphs with less response time. Optimizations can also decrease the response time, but they will give less MCS.

7. CONCLUSION

To express graph queries correctly is a complicated task, because of the diversity and schema flexibility of a data graph. If a query derives an empty answer, a user requires support to understand, what the reason was: an overspecified query or missing data. In this paper we introduce *diff-queries*, a new kind of graph queries, that support a user in such cases. The response to a diff-query describes the parts of a query graph that are addressed and those that are missing in a data graph. The processing of a diff-query consists of two steps: the discovery of a maximum common subgraph and the computation of a difference graph. As a base algorithm we take the McGregor maximum common connected subgraph algorithm. We adapt it for directed weakly-connected property graphs with an all-covering spanning tree and reduce the number of lookups with the restart strategy, which searches from a single edge, does restarts, if some of the edges were not processed, and delivers a maximum common unconnected subgraph. We show that this can be improved by the choice of a start and restart vertex and edge. After the answer is delivered to a user, he can do explorative search of missing data in external sources or modify the query according to the derived difference graph.

Although our method shows good results for our use case, there is an open challenge for the future: the number of intermediate and final results is still very large. We want to develop strategies for reducing and ranking them. In addition, we did not study, how to present and to rate answers according to a given specification. For this purpose, we propose assigning priorities to specific subgraphs of a diff-query and conducting a user study to qualify solutions. Also, we would

like to introduce a similarity measure to quantify vertices, edges and their predicates, and to enhance the algorithm by discarding the backtracking part and by introducing more sophisticated strategies for the choice of a start vertex to decrease the number of restarts.

8. ACKNOWLEDGMENT

This work has been supported by the FP7 EU project LinkedDesign (grant agreement no. 284613).

9. REFERENCES

- [1] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068, Nov. 1986.
- [2] C. Bornhövd, R. Kubis, W. Lehner, H. Voigt, and H. Werner. Flexible Information Management, Exploration and Analysis in SAP HANA. In *DATA*, pages 15–28, 2012.
- [3] A. Chapman and H. V. Jagadish. Why not? In *Proc. of ACM SIGMOD*, pages 523–534, New York, NY, USA, 2009. ACM.
- [4] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 11(1):99–143, 2007.
- [5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.
- [6] H. S. Delugach and A. D. Moor. Difference graphs. In *In Contributions to ICCS 2005*, pages 41–53, 2005.
- [7] P. J. Durand, R. Pasari, J. W. Baker, and C.-c. Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17):1–16, 1999.
- [8] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. DrillBeyond: enabling business analysts to explore the web of open data. *Proc. VLDB Endow.*, 5(12):1978–1981, 2012.
- [9] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1):736–747, Aug. 2008.
- [10] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.
- [11] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Inf. Science and Technology*, 36(6):35–41, 2010.
- [12] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The Graph Story of the SAP HANA Database. In *BTW*, pages 403–420, 2013.
- [13] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc of ACM SIGMOD*, pages 15–26, New York, NY, USA, 2010. ACM.
- [14] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.
- [15] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner. Leveraging flexible data management with graph databases. In *GRADES*, pages 12:1–12:6, New York, NY, USA, 2013. ACM.