BB-Graph: A New Subgraph Isomorphism Algorithm for Efficiently Querying Big Graph Databases

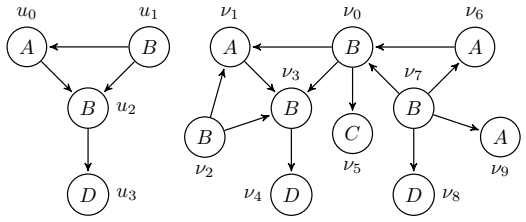
Merve Asiler*, Adnan Yazıcı†
Department of Computer Engineering
Middle East Technical University, Ankara, Turkey
{*asiler, †yazici}@ceng.metu.edu.tr

***Abstract*—With the emergence of the big data concept, the big graph database model has become very popular since it provides strong modeling for complex applications and fast querying, especially for the cases that require costly join operations in RDBMs. However, it is a big challenge to find all exact matches of a query graph in a big graph database, which is known as the subgraph isomorphism problem. Although a number of related studies exist in literature, there is need for a better algorithm that works efficiently for all types of queries since the subgraph isomorphism problem is NP-hard. The current subgraph isomorphism approaches have been built on Ullmann's idea of focusing on the strategy of pruning out the irrelevant candidates. Nevertheless, for some graph databases and queries, the existing pruning techniques are not adequate to handle some of the complex queries. Moreover, many of those existing algorithms need large indices that cause extra memory consumption. Motivated by these, we introduce a new subgraph isomorphism algorithm, namely BB-Graph, for querying big graph databases in an efficient manner without requiring a large data structure to be stored in main memory. We test and compare our proposed BB-Graph algorithm with two popular existing ones, GraphQL and Cypher of Neo4j. Our experiments are done on a very big graph database application (Population Database) and the publicly available World Cup graph database application. We show that our algorithm performs better than those that we use for comparison in this study, for most of the query types.**

I. INTRODUCTION

In the last decade, the big graph database models have widespreaded in a large variety of industry areas such as communications, logistics, Web/ISV, network management, social networks, mobile communication applications, data center management, bioinformatics, etc. The graph database model has recently been preferred to the other database models because it better fits into structure of complex and big data and provides higher performance than the others, especially RDBMs, for most cases [1], [2], [3], [4].

On the other hand, the subgraph isomorphism problem is one of the most frequently encountered challenges in big graph database applications. Subgraph isomorphism can be defined as follows: Given a query graph Q and a database graph G, find all matching instances of Q in G. To illustrate, in Figure I there exist two instances of Q in G, one is the subgraph consisting of the nodes ν_0, ν_1, ν_3, ν_4 and the relationships ${}_{\nu_0}e_{\nu_1}$, ${}_{\nu_0}e_{\nu_3}$, ${}_{\nu_1}e_{\nu_3}$, ${}_{\nu_3}e_{\nu_4}$ and the other one is the subgraph consisting of the nodes ν_1, ν_2, ν_3, ν_4 and the relationships ${}_{\nu_1}e_{\nu_3}$, ${}_{\nu_2}e_{\nu_1}$, ${}_{\nu_2}e_{\nu_3}$, ${}_{\nu_3}e_{\nu_4}$. The subgraph isomorphism problem is known as an NP-hard problem [5]. In almost all big graph database applications, there frequently occur queries directly or indirectly related to subgraph isomorphism problem; therefore, it is important to find an efficient solution for the subgraph isomorphism problem.



(a) query graph: Q (b) graph database: G

Fig. 1. An example query and a graph database with unlabeled relationships

Most of the subgraph isomorphism algorithms in the literature are based on one of the following two types of strategies: Feature indexing by using *filtering-and-verification* technique or candidate node checking by using *branch-and-bound* technique. Algorithms of the first type create an index of small graphs (features) and filter the database pieces which do not have some (or all) of the features included by the query graph. They mark the ones which succesfully passed the filtering to include an exact match(es) of the query, and try to verify them with respect to whether they really do or not. GraphGrep [6], GIndex [7], Labeled Walk Index (LWI) [8], Closure-Tree [9], Graph Decomposition Indexing [10], TreePi [11], TreeDelta [12] are examples to this type of algorithms. Although those algorithms are good for decreasing the number of candidate data sets, they cannot deduce all the isomorphisms of query graph. Moreover, they are applicable only for the databases consisting of many disconnected graph pieces. On the other hand, algorithms of the second type aim to find all the exact matches of a query graph without making an issue that the database is connected or disconnected. Starting off from Ullmann's idea [13], they extract all the candidates for each node in query and, incrementally, they match a query node with one of its candidates. In that way, they try to reach an exact match of query by branching the previous matches. VF2

[14], QuickSI [15], GADDI [16], GraphQL [17], SPath [18] belong to this type of algorithms. The mentioned algorithms are good in terms of finding all the exact matches of query graph; nevertheless, since they search the candidates globally along the database, most of their tryouts to find a relationship between the nodes are redundant due to the fact that the nodes are irrelevant. Therefore, they have low computational performance. Additionally, in order to prune the nodes which cannot be a candidate, some of those algorithms need large data structures, which results in an extra amount of memory consumption.

Motivated by all of these mentioned above, we introduce a new *branch-and-bound* algorithm combined with backtracking strategy for graph isomorphism problem, called BB-Graph. Our algorithm removes the drawbacks mentioned above by conducting the search of candidates for other query nodes locally in the close neighbourhood of the firstly-matched database node. The main contributions of this paper are as follows:

1) A new subgraph isomorphism algorithm, BB-Graph, is introduced. Contrary to the other existing algorithms, BB-Graph follows a more efficient search strategy while matching graph elements (nodes and relationships) by using *branch-and-bound* technique combined with backtracking: It selects starting node candidates from all across the graph database and then, for each candidate isomorphism region potentially created by the starting node match, it selects the candidates of the other nodes locally by traversing every neighbour node and relationship. Later, it backtracks to check possible relationships and node combinations not handled previously.
2) BB-Graph uses the built-in data structures supplied by Neo4j. It does not consume extra memory for large indexing or data storing.
3) While many of other algorithms are experimented only on undirected graphs with hundreds of thousands of nodes by reading the input from a BLOB file, BB-Graph is tested and evaluated on a very big connected directed graph with almost 70 millions of nodes in addition to a directed graph with 45 thousands of nodes.
4) A number of experiments done for comparing our algorithm (BB-Graph) with Cypher and GraphQL show that BB-Graph has a better performance than these existing solutions for most of query types.

Organization of the rest of this paper is as follows: Section III introduces our BB-Graph approach and the main algorithms along with the pseudocodes of proposed solutions. Section IV shows the experimental results and comparison of BB-Graph with Cypher and GraphQL. Lastly, the conclusion and feature work are given in Section V.

II. Related Work

GraphQL [17], an algorithm for subgraph isomorphism problem, focuses on neigbourhood relations to filter candidates for a query node. If a query node u can be matched with a database node u', then for each query node u_k in k-neighbourhood of u, there must be a candidate node u'_k in k-neighbourhood of u'. Thus, GraphQL uses this fact to prune out false candidates of a query node, by scanning their k-neighbourhood upto a refinement level L, incrementally for each k where $1 \leq k \leq L$. Also, GraphQL follows an optimized node matching strategy by selecting the query node which is estimated to decrease the cost at each intermediate step and adjacent to set of already matched nodes. GraphQL compares neighbourhoods of query nodes with their candidates' neighbourhoods and tries to find a semi-perfect bipartite matching between the nodes in corresponding neighbouhoods but our experiments showed that this is already an exhaustive computation even when the refinement level is set to 1 and not very effective for reducing the candidate set size of some query nodes.

In their study, Lee et al. [19] compare five subgraph isomorphism algorithms; VF2, QuickSI, GADDI, GraphQL, and SPath on some real-world data sets on iGraph framework [20]. They work on small and large undirected graphs consisting of one or many pieces by testing with subgraph, clique and path queries. The experiments that they conduct show that there is not a perfect algorithm which works for all types of database queries efficiently. For instance, while QuickSI shows a good performance in many cases, it fails to return the answer in a reasonable time for NASA dataset. According to the experimental results that they have obtained, GraphQL is the only algorithm succeeding to respond in a reasonable time for all tests. They state that these start-of-the-art algorithms may perform poorly because of their ineffective matching order and the imbalance between efficiency and overhead of their pruning methods.

Although each of five algorithms, VF2, QuickSI, GADDI, GraphQL, and SPath, has its own defect, there is a common point that causes all of these algorithms to perform poorly. All start searching by trying to find and filter candidate nodes for each query vertex. The pruning rules that they use are generally effective for eliminating the nodes that can never be a representative of the query node for which it is selected as a candidate in a real exact match. However, since they don't apply the prunings by regarding each isomorphism as an independent one, the nodes belonging to different isomorphisms cannot be discriminated until their relationships are checked in matching phase. Therefore, the database nodes which are candidate for different query vertices and members of distinct isomorphisms seem available for being members of the same isomorphism at first sight. Hence, the main time loss occurs at this point while searching for an actually non-existing reasonable connection between those irrelevant nodes. In order to remove this kind of possible cases, after the start node candidates are taken from all across the database, candidate nodes for the rest of the query vertices should be selected depending on the start node match. In other words, each start node candidate potentially creates a distinct isomorphism region; therefore, the representatives for the other query vertices should be locally chosen from the close neighbourhood of the

start node for each distinct exact match.

III. BB-GRAPH ALGORITHM

In this paper, the notation $G : (V, E)$ represents the graph G with the vertex set V and edge set E. For vertex ν; L_ν denotes the label set of ν. Similarly, for edge e, L_e denotes the label set of e. The notation ${}_ue_\nu$ is used for the edge e starting from vertex u (outgoing w.r.t. u) and ending at vertex ν (incoming w.r.t. ν) when we need to state the end points of e. Otherwise, we just denote it as e.

For two graphs $G_1 : (V_1, E_1)$ and $G_2 : (V_2, E_2)$, G_1 is said to be *subisomorphic* to G_2, if there is a one-to-one and onto function $f : V_1 \longrightarrow V_2$ such that; $\forall \nu \in V_1\ L_\nu \subseteq L_{f(\nu)}$ and for any ${}_ue_\nu \in E_1\ {}_{f(u)}e_{f(\nu)} \in E_2$ satisfying that $L_{{}_ue_\nu} \subseteq L_{{}_{f(u)}e_{f(\nu)}}$. Given a query graph Q and a data graph G, the problem of finding all distinct subisomorphisms of Q in G is called as *Subgraph Isomorphism Problem*.

For two graphs $G_1 : (V_1, E_1)$ and $G_2 : (V_2, E_2)$, where G_1 is subisomorphic to G_2 under some isomorphism f, let $u \in V_1$ be matched with $u' := f(u) \in V_2$. Since each edge e adjacent to u must be matched with some edge e' adjacent to u', it follows that the degree of u must be less than or equal to the degree of u'. Moreover, since the matched edges must have the same label and direction, for each group of edges having the same label and direction degree of u must be also less than or equal to the degree of u'. As a result, the matching vertices in a subisomorphism must satisfy both label and degree rules deduced from the definition. In this paper, we call this as *matching node principal* for the vertices to be matched . We denote $u \overset{\text{m.n.p.}}{\equiv} u'$ when matching node principle holds for u and u'. Similarly, if e is matched with e' under f, then it means $Le \subseteq Le'$ and direction of e with respect to u must be the same with direction of e' with respect to u'. We call this as *matching relationship principal* for the relationships to be matched and denote $e \overset{\text{m.r.p.}}{\equiv} e'$.

Throughout the paper, we use the word *relationship* instead of the *edge* concept and the word *node* instead of *vertex* concept to make the contextual narrating compatible with Neo4j environment. Moreover, we denote the matching of query node u with database node u' by $< u, u' >$; similarly, the matching of query relationship r with the database relationship r' by $< r, r' >$.

A. Algorithm Overview

BB-Graph first chooses a query node to start matching, which we call it the *starting node*. Next, BB-Graph filters the database nodes to find candidates for the starting node. It does the filtering according to the *matching node principal* previously mentioned in this Section. After getting the candidate nodes, BB-Graph puts them into a list to match the starting node with one candidate at a time and search for query graph isomorphisms rooting from that match. In order to achieve this, the whole candidate list is walked through a loop such that at each time of iteration the next candidate node is picked from the list, matched with the starting node and the recursive isomorphism search begins from that point, Algorithm 1, lines 1-12.

Algorithm 1: BB-GRAPH SEARCH ALGORITHM

```
   Input: Q : Query graph with n vertices
          G : Database graph
   Output: M : Set of all exact matches of Q in G
1  begin
2      M := ∅
3      Choose the first node in the input as u_start
4      C_{u_start} := {u' | u' ∈ V_G and u' ≡(m.n.p.) u_start}
5      foreach u' ∈ C_{u_start} do
6          V_matched := ∅, E_matched := ∅, S := ∅
               // Reset the temporary storage
7          Push < u_start, u' > into S
8          Add < u_start, u' > into V_matched
9          SEARCH()
10     end
11     return M
12 end
13 void SEARCH()
14 begin
15     if S ≠ ∅ then
16         < u, u' >:= Pop S
17         if u has non-matched adjacent relationship then
18             BRANCHNODES(< u, u' >)
19         end
20         else
21             SEARCH()
22         end
23     end
24     else
25         Add the tuple (V_matched, E_matched) into M
               // An exact match found
26     end
27     return
28 end
```

In isomorphism search, BB-Graph uses the previously obtained node matches. Let $V_{matched}$ be the set of previously matched couples of nodes. Similarly, let $E_{matched}$ be the set of previously matched couples of relationships. For each node matching $< u, u' > \in V_{matched}$, where $u \in V_Q$ and $u' \in V_G$, BB-Graph applies the *reciprocal node branching process*, that is, the process of expanding partially matched graph piece in order to complete to an exact match by following the adjacent relationships of u and u' simultaneously. In order to decide which node matching is used in reciprocal node branching process at that moment, all the node matchings are kept in a stack which we denote by S. When BB-Graph pops a node matching $< u, u' >$ from S, it applies the reciprocal node branching as follows: Firstly, it detects the non-matched relationships adjacent to query node u. Next, for each of those relationships, it tries to find candidates among the relationships adjacent to u'. While determining

the candidates, it checks whether the *matching relationship principal* (previously mentioned in this Section) holds or not. After filtering, there may appear more than one candidate for a relationship. BB-Graph collects all the candidates in a separate list for each non-matched relationship adjacent to u, Algorithm 2, lines 2-4. A case that there is no candidate for a relationship never occurs because we always match a query node u with a database node u' provided that u' has a degree at least as u has for each different group of relationships.

Algorithm 2: BRANCHNODES

```
Input: global variables S, V_matched, E_matched
       < u, u' >: The node match currently being branched
Output: It affects the global variables S, V_matched, E_matched
1  begin
2      foreach non-matched relationship r_i adjacent to u do
3          C_{r_i} := {r'_i | r'_i is adjacent to u' and r'_i ≡(m.r.p.) r_i}
4      end
5      k := number of non-matched relationships adj. to u
6      MATCHRELATIONSHIP(1)
7  end
8  void MATCHRELATIONSHIP(i)
9  begin
10     foreach r'_i ∈ C_{r_i}, r'_i is not matched do
11         if CHECK(< r_i, r'_i >, < u, u' >) then
12             Add < r_i, r'_i > into E_matched
13             S* := S, V*_matched := V_matched
14             if i ≤ k then
15                 MATCHRELATIONSHIP(i++)
16             end
17             else
18                 SEARCH()
19             end
20             Remove < r_i, r'_i > from E_matched
                   // Backtracking
21             S := S*, V_matched := V*_matched
22         end
23     end
24 end
```

Assume that r_1, r_2, ..., and r_k are the non-matched relationships adjacent to u and C_{r_1}, C_{r_2}, ... and C_{r_k} are the lists which contain corresponding candidate relationships adjacent to u'. In the next stage, BB-Graph picks a candidate relationship $r'_i \in C_{r_i}$ to match with r_i iteratively for each i. At each step of iteration, it is checked whether the prospective match $< r_i, r'_i >$ causes a conflict related with the relationship end points, or not. It can easily be seen that if two relationships are matched with each other, then their corresponding end points must also be matched with each other. Here, it is already known that u and u' had been matched before. Thus, BB-Graph has to check whether the other end points of r_i and r'_i, say ν and ν', match, Algorithm 3. It can be one of the following three cases: First, if ν and ν' are already matched with each other, then there is not any conflict. Second, if ν is matched with some node other than ν', or vice versa, it is reported as a conflict. Third, it may be the case that both ν and ν' are not-yet-matched nodes. At this point, it is checked whether the nodes satisfy the *matching node principal*, or not. If they satisfy, then they are matched with each other and the new match $< v, v' >$ is added into $V_{matched}$ and pushed into S to apply *reciprocal node branching process* for them later. Otherwise, it is reported as a conflict. As a result, we guarantee that there does not occur an empty candidate list for any relationship in candidate filtering part. After checking the end points of relationships, if there does not exist any reported conflict, then the match $< r_i, r'_i >$ is approved and added into $E_{matched}$, and BB-Graph goes on the relationship matching procedure from the next iteration for r_{i+1}. During the checks, if there is a reported conflict, then the next candidate in C_{r_i} is picked and the end point checks are repeated for the prospective matching of r_i with its new candidate this time. Nevertheless, it may happen that all the candidate relationships for r_i are tried but resulted in a problem. If such a case occurs, then it is understood that there are wrong decisions in the previous matchings, in fact. At this point, BB-Graph backtracks to the former relationship matching, say $< r_{i-1}, r'_{i-1} >$, and until getting a non-conflicting match it tries the next candidates for r_{i-1} this time. If one is obtained, then BB-Graph continues to its normal schedule from the next stage as usual. Otherwise, it backtracks again and applies the same procedure, and goes on in this way, Algorithm 2, lines 9-25.

When BB-Graph backtracks from stage i to $i-1$, all the global data structures are returned to their old version at stage $i-1$; that is, the last relationship and node matchings are cancelled and removed from $V_{matched}$ and $E_{matched}$. Also S is stored with its old content by doing the opposites of push and pop operations done at stage i, Algorithm 2, lines 21-22.

Since our aim is to find all exact matches of the query graph, backtracking is needed not only when a contradiction occurs, but it is also necessary to try all candidate options for each query graph node and relationship with all possible combinations. For that reason, when an isomorphism of the query is found, instead of terminating or restarting, BB-Graph continues its schedule with backtracking.

When BB-Graph completes the iterative relationship matching part successfully, all the relationships adjacent to u are matched with some relationship adjacent to u'. This finishes the reciprocal branching process of two nodes u and u'. The rest of algorithm maintains the isomorphism search by recursively repeating the *reciprocal node branching process* for the newly obtained node matchings, Algorithm 1, lines 14-29.

In order to illustrate how BB-Graph works, a sample running schedule is given in Figure 2. The whole recursive process of reciprocal node branching, relationship matching and backtracking are shown step-by-step for the sample query and the graph database given in Figure I.

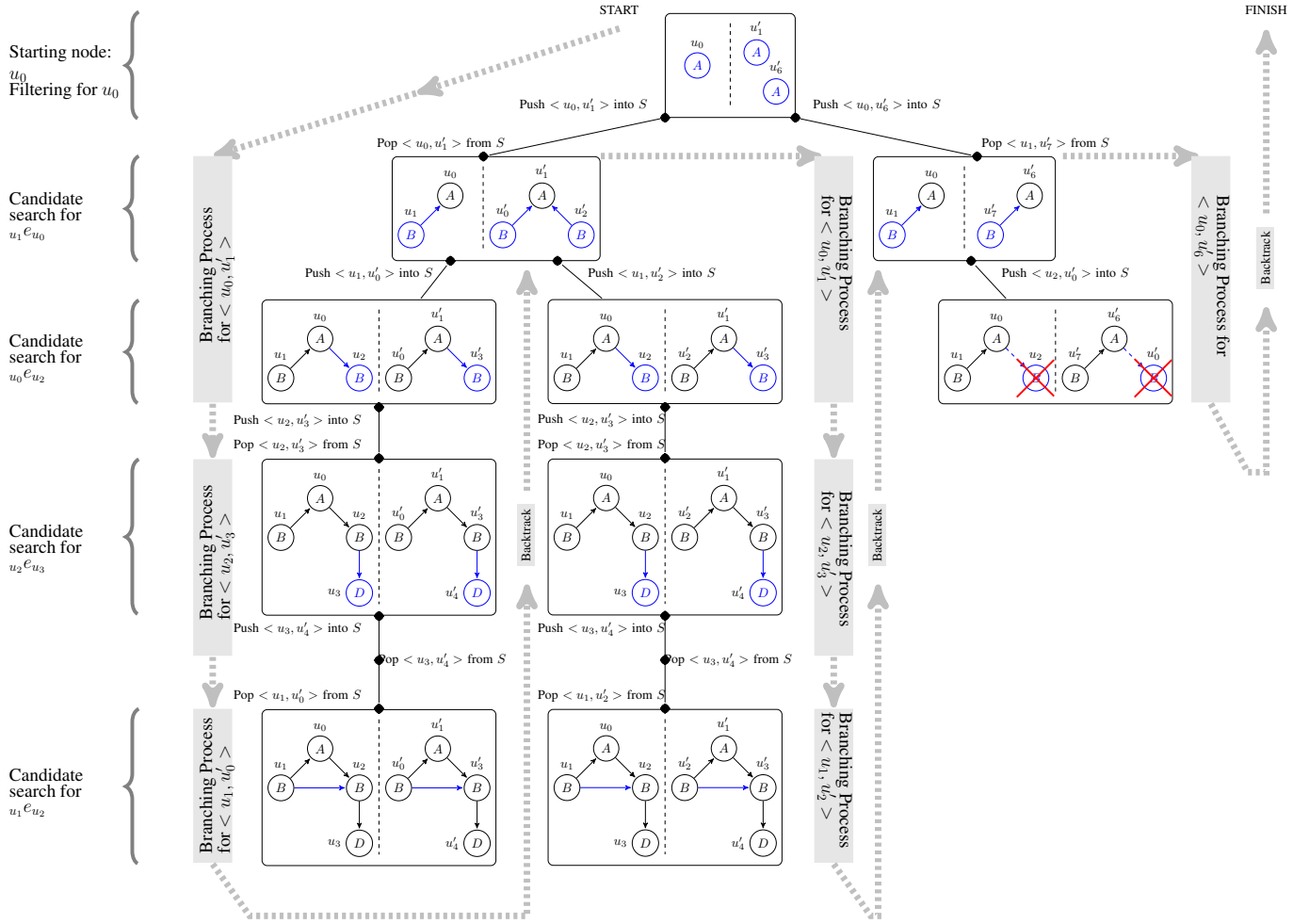


Fig. 2. A sample execution for the query graph and graph database given in Figure I. Each level of the diagram represents the candidate relationship search for edge matching. At the rectangle boxes of each level the found candidates are shown in blue whereas the already matched nodes and relationships are shown in black. At each rectangle box, operations in query part and database part of the reciprocal node branching process are shown in left and right parts of the box, respectively. The execution flows in the arrow direction.

B. Algorithm Complexity

In this part, we give running time complexity (computational complexity) analysis and space complexity analysis of BB-Graph for the worst case scenarios. In the analyses, we denote the number of nodes (vertices) in query graph Q by $|V_Q|$, number of nodes (vertices) in graph database G by $|V_G|$ and number of relationships (edges) in query graph Q by $|E_Q|$. Also, we use deg_Q^{max} and deg_G^{max} to denote maximum node degree in query graph and maximum node degree in database graph, respectively.

Time Complexity:

BB-Graph starts with the filtering part. It tries to find candidates for the starting node by filtering the database nodes depending on the *matching node principal* given in Section III. In other words, to find all the candidates, label and degree properties of each database node must be checked. Since we work in Neo4j, which holds the information of database nodes grouped by label, filtering with respect to node labels takes $\mathcal{O}(1)$. In the worst case, all database nodes may have labels of the starting node, which means that there exist $|V_G|$ number of candidate nodes that are going to be filtered with respect to their adjacent relationships. Since Neo4j directly gives the information about adjacent reationships of each type for every node, it just remains to ask for this information for every candidate node regarding each relationship of different type adjacent to starting node. Therefore, this part takes $\mathcal{O}(|V_G| \times deg_Q^{max})$, which also gives the worst case complexity of total filtering stage. In the end of it, there may be $|V_G|$ number of candidates for the starting node in the worst case.

For each candidate of the starting node isomorphism search is conducted. For that reason, the complexity of this part is going to be $|V_G| \times \mathcal{O}(\text{SEARCH}())$. Because the function SEARCH() and the function BRANCHNODES() work in a mutual recursive manner, the cost of SEARCH() actually equals to the cost of BRANCHNODES(). To be able to calculate the complexity of BRANCHNODES, we should analyse the computational cost of the operations done in one recursive depth of branching procedure.

In the branching procedure (Algorithm 2), initially, all non-

Algorithm 3: CHECK

Input: global variables S, $V_{matched}$, $E_{matched}$
$< r_i, r'_i >$: The relationship match to be checked
$< u, u' >$: The node match currently being branched
Output: Boolean, depending on the existence of any contradictory situation

```
1  begin
2      ν_i := The end point of r_i other than u_i
3      ν'_i := The end point of r'_i other than u'_i
4      if ∃v_x ≠ ν_i  s.t.  < ν_x, ν'_i > ∈ V_matched then
5          return false
6      end
7      if ∃ν'_x ≠ ν'_i  s.t.  < ν_i, ν'_x > ∈ V_matched then
8          return false
9      end
10     if < ν_i, ν'_i > ∉ V_matched then
11         if ν_i ≡ ν'_i (m.n.p.) then
12             Push < ν_i, ν'_i > into S
13             Add < ν_i, ν'_i > into V_matched
14         end
15         else
16             return false
17         end
18     end
19     return true
20 end
```

matched relationships of the current query node are detected and for each of them candidate relationship sets are constructed. If we assume that there are x_i many number of non-matched relationships for the current node in the depth i of the recursive process, complexity of candidate relationship set construction becomes $\mathcal{O}(x_i)$. For each non-matched relationship, there can be at most deg_G^{max} many number of candidate relationships (If there are more than one different type of non-matched relationships adjacent to the current node, then it is certain that the candidate set size is less than deg_G^{max}. Nevertheless, we take deg_G^{max} as an upper bound for the candidate set size of each non-matched query relationship). This means, there occurs at most $(deg_G^{max})^{(x_i)}$ many number of different combinations of relationship matching for a query node. Since a query node can have at most deg_Q^{max} non-matched relationships, the upper bound for number of combinations of relationship matching becomes $(deg_G^{max})^{(deg_Q^{max})}$. At each candidate relationship selection, it is checked whether there occurs any conflict with the match through CHECK() function (Algorithm 3). In CHECK(), it is checked whether the end nodes of both relationships given by the function argument exist among the already matched query and database nodes. Since we use hash tables to understand which nodes are matched, the only thing that costs in this function is checking *matching node principle* between nodes (line 10). For that reason, complexity of this function becomes $\mathcal{O}(deg_Q^{max})$. Since *Check* function is called x_i, namely deg_Q^{max} at most, many times for each combination of relationship matching, at each depth of recursive branching process the cost becomes $\mathcal{O}((deg_G^{max})^{(deg_Q^{max})} \times deg_Q^{max} \times deg_Q^{max})$. Since the maximum depth of the recursive branching can equal to the number of query nodes, $|V_Q|$, the overall complexity of branching procedure becomes $\mathcal{O}(|V_Q| \times (deg_G^{max})^{(deg_Q^{max})} \times (deg_Q^{max})^2)$.

To conclude, the computational cost of BB-Graph equals to the summation of cost of filtering part and cost of searching part, which is, $\mathcal{O}(|V_G| \times deg_Q^{max}) + \mathcal{O}(|V_G| \times |V_Q| \times (deg_G^{max})^{(deg_Q^{max})} \times (deg_Q^{max})^2)$. However, the cost of filtering part is negligible when it is compared to the cost of searching part. As a result, the running time complexity of BB-Graph becomes $\boldsymbol{\mathcal{O}(|V_G| \times |V_Q| \times (deg_G^{max})^{(deg_Q^{max})} \times (deg_Q^{max})^2)}$.

Space Complexity:

For the filtering part, BB-Graph uses a list to hold the candidate database nodes for the starting node. Since there can be maximum $|V_G|$ number of candidates for the starting node (in other words all the database nodes), there is a need for a list of size $|V_G|$ in the worst case. Also, for each query relationship, there is constructed a set of candidate relationships. Since there can be maximum deg_G^{max} number of candidates for a query relationship, size of a set of candidate relationships becomes deg_G^{max}. The case in which the recursive computation reaches to the deepest value is the worst case requiring the largest storage for the sets of candidate relationships. Since all of the query relationships are in the process in the highest depth of the recursion, sets of candidate relationships consume $|E_Q| \times deg_G^{max}$ units of space at most. When the algorithm backtracks to the previous depth of recursion, the space reserved for containing sets of candidate relationships constructed at that depth is released. Therefore $|E_Q| \times deg_G^{max}$ is the maximum value of storage needed by them. Additionally, there exists global variables consuming some memory. There are two global hashing maps used to hold already matched query graph and database graph items; one map for already matched nodes and one map for already matched relationships. These two maps include information of the fact that which query node or relationship was matched with which database node or relationship, respectively. For that reason, they can consume at most $2 \times |V_Q|$ and $2 \times |E_Q|$ units of storage, respectively. Lastly, the stack which is used to hold node matchings which are not yet sent to *reciprocal node branching process* needs $2 \times |V_Q|$ units of storage, that is the space required in the case that starting node is adjacent to all the remaining query nodes. Throughout the whole execution of BB-Graph, all the other variables consume quite little space and can be neglected. Consequently, the space complexity of BB-Graph equals to summation of all the mentioned values appeared in the worst case, which is $\boldsymbol{\mathcal{O}(|V_G| + |V_Q| + |E_Q| \times deg_G^{max})}$.

IV. EXPERIMENTS AND EVALUATION

We compare the performances of BB-Graph, Cypher and GraphQL on Population Database supplied by Kale Yazılım which consists of a directed graph with 70 millions of nodes, and WorldCup Database supplied public by Neo4j Team which consists of a directed graph with 45 thousands of nodes. In Table I, features of Population Database and WorldCup Database are given in a detailed way. We should note that Population Database is much bigger than the ones used in [19], as shown in Table I.

For the experiments, we use 10 real-world queries for Population Database and 5 real-world queries for WorldCup Database where each query has different number of nodes and relationships and also has different types of node labels and relationships. For each query, BB-Graph and Cypher experiments have been repeated 10 times, and the averages of the elapsed times are used. For all GraphQL experiments refinement-level was adjusted to 1 and the refining process has been repeated as many times as the number of nodes in a query graph. In the experiments of these three algorithms, all the exact matches are found in a continuous time interval without any break. The total elapsed time is calculated in order to compare the performances of the algorithms.

	WorldCup	**Population**
Size	20 MiB	10.32 GiB
# of graphs	1	1
# of nodes	45348	70422787
# of relationships	86577	77163109
# of distinct node labels	12	14
# of distinct relationship types	17	18
Avg. # of labels per node	1	1

TABLE I
GENERAL FEATURES OF POPULATION AND WORLDCUP DATABASES

A. Setup

All the experiments were conducted on the same server with Intel Octa Core 2.27GHz, 8 GB of main memory, and 100 GB hard disk, running Debian GNU, Linux 7.8 (wheezy). The Neo4j version used is 2.3.1. BB-Graph and GraphQL were implemented in Java on Eclipse and Cypher queries were called in Java.

B. Experiments

We group the experimental results according to the type of query graphs used in BB-Graph. For a query which include cyclic paths on it, we call that as a *complex query*, and for a query which is itself a single path, we simply call that as a *path query*. In Figures 3 and 4, a sample complex query and a sample path query are shown, respectively.

Complex Queries:

Table II shows the experimental results obtained for distinct complex-type query graphs executed over Worldcup and Population Databases.

As it can be clearly seen from Table II, BB-Graph performs so much better than Cypher and GraphQL for a large majority of complex-type queries. Generally, BB-Graph has higher performance than Cypher for the queries that we have tested, except the two, Q-6 and Q-9, which performs as nearly good as Cypher for those although it falls a little behind Cypher. For the complex queries executed in WorldCup Database which are Q-1, Q-2 and Q-3 consisting of 4 nodes and 4 relationships with 1 cycle, 7 nodes and 8 relationships with 2 cycles and 10 nodes and 12 relationships with 3 cycles, respectively, BB-Graph has the best performance among all the algorithms. Although GraphQL has the worst performance for Q-1 and Q-2, it achieves a very close performance to BB-Graph for Q-2. We think that it is probably caused by the fact that the existence of a very unique node label in Q-2 made the number of candidate nodes very small. Therefore, the entire search does not take a long time. On the other hand, GraphQL could not terminate under 30 minutes for any of complex queries executed over Population Database. From those queries, for Q-4, Q-5, Q-7, Q-8, Q-10 and Q-11 consisting of 6 nodes and 12 relationships with 7 cycles, 5 nodes and 7 relationships with 4 cycles, 6 nodes and 6 relationships with 1 cycle, 7 nodes and 11 relationships with 5 cycles, 13 nodes and 25 relationships with 13 cycles and 10 nodes and 14 relationships with 3 cycles, respectively, BB-Graph performs much better than Cypher. For Q-10, which is the most complex and the largest query among all in the experiments, Cypher could not complete its running in a reasonable time. Interestingly, although Q-8 and Q-9 have the same number of nodes, relationships, cycles and even the node labels, the performance of BB-Graph changes. The reason for that, while both of those queries have the same number of items, they have different structuring such as the neighbourhoods and degrees of the node which also specializes a graph. For that reason, performance of BB-Graph changes accodingly with the number of false candidates that depends on how much a query graph is specialized with its characteristics.

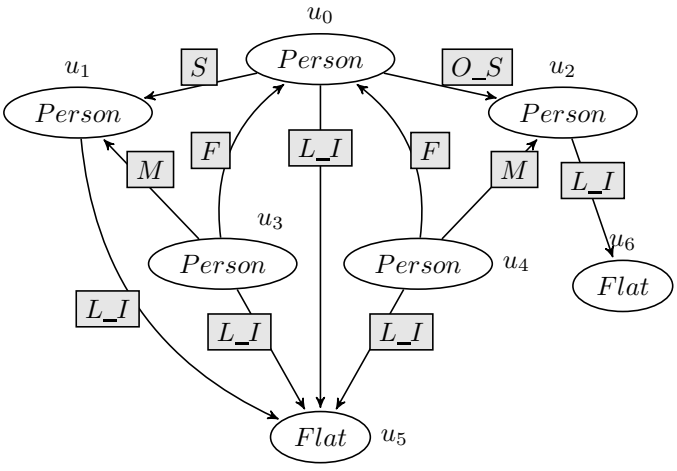


Fig. 3. An example for a complex query: *"Find the families (a man, a wife, at least two children, one from an ex-wife), who live at the same address in Population Database"*

Query	Database	Number of Nodes	Number of Relationships	Peformance of GraphQL (sec.)	Performance of Cypher (sec.)	Performance of BB-Graph (sec.)
Q-1	WorldCup	4	4	1112.5	10.1	1.9
Q-2	WorldCup	7	8	17.9	7.8	3.9
Q-3	WorldCup	10	12	8.1	26.7	7.9
Q-4	Population	6	12	$>$ 1800.0	189.9	138.1
Q-5	Population	5	7	$>$ 1800.0	24.2	22.3
Q-6	Population	5	5	$>$ 1800.0	14.8	17.3
Q-7	Population	6	6	$>$ 1800.0	92.4	54.5
Q-8	Population	7	11	$>$ 1800.0	14.4	8.5
Q-9	Population	7	11	$>$ 1800.0	13.7	14.0
Q-10	Population	13	25	$>$ 1800.0	$>$ 1800.0	502.7
Q-11	Population	10	14	$>$ 1800.0	226.7	73.4

TABLE II
QUERY RESPONSE TIMES FOR COMPLEX QUERIES

Path Queries:

Table III shows the experimental results obtained for distinct path-type query graphs executed over Worldcup and Population Databases.

From Table III, it can be seen that BB-Graph has the best performance for 2 path-type queries out of 3. The first path query, Q-12, consists of 5 nodes and 4 relationships executed over WorldCup Database. For this query, BB-Graph shows 4 times better performance than Cypher. For Q-12, GraphQL shows rather low performance when compared to BB-Graph's and Cypher's performances. On the other hand, this is the only path query that GraphQL could complete its execution under 30 minutes. Since Population Database is many times more complex, dense and bigger than WorldCup Database, GraphQL exceeds our reasonable time limit, which is 30 minutes, for the other two path queries. The second path query, Q-13, consists of 8 nodes and 7 relationships executed over Population Database. This query is a recursive query, with the depth of 4. For Q-13, again BB-Graph shows much better performance than Cypher. On the other hand, for Q-14, which is the third path query consisting of 4 nodes and 3 relationships, Cypher peforms a little better than BB-Graph.

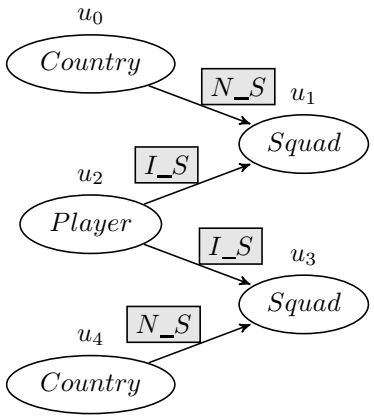


Fig. 4. An example for a path query: *"Find the players who join the squad of different countries in WorldCup Database"*

V. CONCLUSION

In this paper, a new algorithm, BB-Graph, for Subgraph Isomorphism Problem is introduced. BB-Graph uses branch-and-bound technique to match each node and relationship of query graph with its candidate and uses backtracking approach for the other possible candidates. Different from the current algorithms, BB-Graph does not find candidates of each query graph node all across the graph database. After matching the first query node, BB-Graph searches candidates for the other query graph nodes and relationships in local region of the first-matched database node. Our experiments are conducted with different types of queries on two different real-world databases, namely Population Database and WorldCup Database. According to our experimental results, GraphQL is not scalable enough for querying in very large databases like Population Database. Although GraphQL shows close performance to BB-Graph for one of the queries in WorldCup Database, generally GraphQL performs worse than Cypher and BB-Graph due to its strategy of getting the candidates from the whole database and trying to match even some irrelevant ones. On comparing BB-Graph with Cypher, our experimental results show that BB-Graph has better performance than Cypher for most of the query types, for both large and small databases.

According to our experimental results, we observe that the performance of BB-Graph algorithm does not depend on just one factor like the number of nodes or relationships. Actually, many other features, such as frequency of query node labels and query relationship types in the database graph, number of cycles in query graph, structure of query graph (whether it is a path or something more complex) and also

Query	Database	Number of Nodes	Number of Relationships	Peformance of GraphQL (sec.)	Performance of Cypher (sec.)	Performance of BB-Graph (sec.)
Q-12	WorldCup	5	4	28.5	1.6	0.4
Q-13	Population	8	7	> 1800.0	26.2	2.2
Q-14	Population	4	3	> 1800.0	12.5	18.9

TABLE III
QUERY RESPONSE TIMES FOR PATH QUERIES

the semantic design of relationships like $1 - N$ or $N - N$, affect the computation time. As a future work, it may inspire the researchers to improve the performance of BB-Graph by taking all the query and graph database features mentioned above into consideration.

ACKNOWLEDGMENT

We like to present our thanks to KALE YAZILIM for sharing their resources, especially the real-world database, and giving the opportunity to work with them.

REFERENCES

[1] V. Kolomičenko, "Analysis and experimental comparison of graph databases," Master Thesis, Charles University Department of Software Engineering, Prague, 2013.

[2] S. Batra and C. Tyagi, "Comparative analysis of relational and graph databases," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 2, pp. 509–512, 2012.

[3] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, 2010, p. 42.

[4] "Bitnine Company Website, Relational Database vs Graph Database," http://bitnine.net/rdbms-vs-graph-db/, accessed: Oct. 2016.

[5] A. Abboud, A. Backurs, T. D. Hansen, V. V. Williams, and O. Zamir, "Subtree isomorphism revisited," pp. 1256–1271, 2016.

[6] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *Proceedings of the IEEE 16th International Conference on Pattern Recognition (ICPR)*, vol. 2. IEEE, 2002, pp. 112–115.

[7] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD, International Conference on Management of Data*. ACM, 2004, pp. 335–346.

[8] S. Srinivasa, M. Maier, M. R. Mutalikdesai, K. Gowrishankar, and P. Gopinath, "Lwi and safari: A new index structure and query model for graph databases." in *COMAD*, 2005, pp. 138–147.

[9] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 38–38.

[10] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *IEEE 23rd International Conference on Data Engineering (ICDE 2007)*. IEEE, 2007, pp. 976–985.

[11] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," in *IEEE 23rd International Conference on Data Engineering (ICDE)*. IEEE, 2007, pp. 966–975.

[12] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree+ delta¡= graph," in *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 2007, pp. 938–949.

[13] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[15] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[16] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 2009, pp. 192–203.

[17] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, pp. 405–418.

[18] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[19] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," in *Proceedings of the VLDB Endowment*, vol. 6, no. 2. VLDB Endowment, 2012, pp. 133–144.

[20] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "igraph: a framework for comparisons of disk-based graph indexing techniques," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 449–459, 2010.