Journal of Big Data

RESEARCH Open Access

HyGraph: a subgraph isomorphism algorithm for efficiently querying big graph databases

Merve Asiler[1], Adnan Yazıcı[1,3] and Roy George[2*]

*Correspondence:
rgeorge@cau.edu
[2] Department of Cyber-Physical Systems, Clark Atlanta University, 223 James Brawley Drive, Atlanta 30314, USA
Full list of author information is available at the end of the article

Abstract

The big graph database provides strong modeling capabilities and efficient querying for complex applications. Subgraph isomorphism which finds exact matches of a query graph in the database efficiently, is a challenging problem. Current subgraph isomorphism approaches mostly are based on the pruning strategy proposed by Ullmann. These techniques have two significant drawbacks- first, they are unable to efficiently handle complex queries, and second, their implementations need the large indexes that require large memory resources. In this paper, we describe a new subgraph isomorphism approach, the HyGraph algorithm, that is efficient both in querying and with memory requirements for index creation. We compare the HyGraph algorithm with two popular existing approaches, GraphQL and Cypher using complexity measures and experimentally using three big graph data sets—(1) a country-level population database, (2) a simulated bank database, and (3) a publicly available World Cup big graph database. It is shown that the HyGraph solution performs significantly better (or equally) than competing algorithms for the query operations on these big databases, making it an excellent candidate for subgraph isomorphism queries in real scenarios.

Keywords: Exact matching algorithm, Graph database, Neo4j databases, Subgraph isomorphism problem, Query graph search

Introduction

Big graph database models have seen widespread application in diverse areas including communications, logistics, social networks, data center management, bioinformatics, etc. It is preferred over other database models in demanding applications since its structure organically facilitates complex queries while providing higher performance, in comparison to other approaches, in particular Relational Database Management Systems (RDBMs) [1–4]. These applications frequently require the discovery of matching subgraphs, that are identical to a user supplied subgraph using the query mechanism-the subgraph isomorphism problem. Subgraph isomorphism is defined as follows: Given a query graph Q and a database graph G, find all matching instances of Q in G. Figure 1 illustrates this, where there exist two instances of Q in G, one is the subgraph consisting of the nodes v_0, v_1, v_3, v_4 and the relationships ${}_{v_0}e_{v_1}$, ${}_{v_0}e_{v_3}$, ${}_{v_1}e_{v_3}$, ${}_{v_3}e_{v_4}$ and the other one is

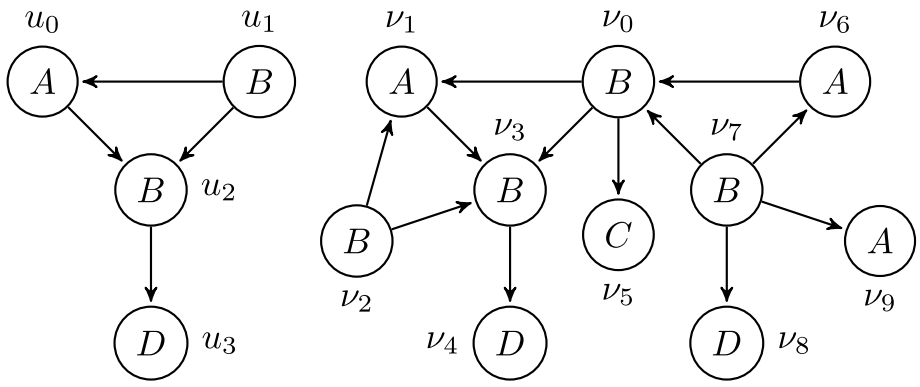

Fig. 1 An example query and a graph database with unlabeled relationships

the subgraph consisting of the nodes ν_1, ν_2, ν_3, ν_4 and the relationships ${}_{\nu_1}e_{\nu_3}$, ${}_{\nu_2}e_{\nu_1}$, ${}_{\nu_2}e_{\nu_3}$, ${}_{\nu_3}e_{\nu_4}$. This is a challenging problem since the user supplied subgraph could be arbitrarily complex, and is known to be NP-hard [5]. Since there is a requirement for subgraph isomorphism queries in these complex applications, it is important to find an efficient solution to this problem.

Subgraph isomorphism algorithms in the literature are primarily 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 out the ones which do not have some (or all) of the features included by the query graph. Items that successfully passed the filtering include exact match(es) of the query, which are then verified. GraphGrep [6], GIndex [7], Labeled Walk Index (LWI) [8], Closure-Tree [9], Graph Decomposition Indexing [10], TreePi [11], TreeDelta [12] are exemplars. Although these algorithms are good at decreasing the number of candidate data sets, they cannot retrieve all the isomorphisms of a query graph. Moreover, they are applicable only to the databases consisting of many disconnected graph pieces. The algorithms of the branch-and-bound type variety find the exact matches of a query graph without reference to whether the database is connected or disconnected. Using the Ullmann approach [13] or a variation, 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 a query by branching from the previous matches. VF2 [14], VF3 [15], QuickSI [16], GADDI [17], GraphQL [18], SPath [19] belong to this type of algorithms. The algorithms are good in terms of finding the exact matches of a query graph; however, since they search the candidates globally along the database, many of the attempts to find a relationship between the nodes are redundant since the nodes are mostly irrelevant to the query resulting in poor computational performance. Additionally, in order to prune the non-candidate nodes, these algorithms require large data structures, thereby needing large memory resources for execution.

Motivated by these considerations, we introduce a new algorithm, the HyGraph (**Hy**brid **Graph**), which combines the *branch-and-bound* algorithm with a *backtracking* strategy for subgraph isomorphism identification. In brief, HyGraph eliminates the problems seen with other approaches by conducting the search of candidates for other

query nodes locally in the neighborhood of the first-matched database node. We summarize the main contributions of this research as follows:

1. HyGraph utilizes an efficient hybrid search strategy matching graph elements (nodes and relationships) by using *branch-and-bound* technique combined with *backtracking*. Initially, candidates for the starting query node are selected from all across the graph database using node properties. A potential isomorphism region is created for each candidate, and candidates for the other query nodes are selected locally by traversing every neighbour node and relationship using branch-and-bound method. Finally, HyGraph backtracks to check possible relationships and node combinations not yet handled.
2. Many of the algorithms introduced previously have been evaluated only on small undirected graphs. HyGraph has been evaluated in an operationally realistic environment consisting of a directed graph with approximately 70M nodes. Additional validation of HyGraph was performed using two other directed graphs, one with 100K and the other with 45K nodes.
3. HyGraph is empirically compared with Cypher and GraphQL, on a queries of varying complexity. Results show that HyGraph has a better computational performance compared to the industrial best of the breed products for a majority of query types and equal to their performance in the rest of the cases.
4. The effect of a change in node matching order is illustrated with the experiments done during this study. The results show that developing an effective node matching order is definitely a criterion to increase the performance of the subgraph isomorphism algorithm. This research provides insights into the selection of node matching criterion.
5. HyGraph uses the built-in data structures in the Neo4j Graph Database system avoiding the memory needed for large indexes. The integration of HyGraph with Neo4j (an industry standard) makes it a suitable for deployment in an industrial setting.

The organization of this paper is as follows: "Background and related work" section gives the necessary background and related work. "Method" section introduces the HyGraph approach and the main algorithms along with the relevant pseudocode. "Results" section shows the experimental results and comparisons of HyGraph with Cypher and GraphQL. "Discussion" section discusses the results and observations obtained through the experimental study. Conclusions and future avenues for research are given in "Conclusions" section.

Background and related work

Graph databases have gained popularity since they can efficiently handle complex queries on big database applications that require flexible modeling functionality. Several studies that compare graph and relational databases show that graph databases perform much better than RDBMs in several aspects [1–4, 20–23]. Primarily, the advantage lies in eliminating join operations, which are expensive in traditional RDBMs. Graph databases do not require joins, since they organically perform an analogical operation that directly traverses the nodes and relationships. In Fig. 2, the results of an experiment done with recursive queries that require many join operations are given [4]. The results of the comparison show that graph databases have higher performance than RDBMs for such complex queries [24]. Furthermore, it has

been shown that insert-delete-update operations on graph databases can be done much more easily and efficiently in comparison to relational databases.

Much of the previous work in this area, derives from Ullmann Algorithm [13], a search method developed to find isomorphic patterns of query graphs. Informally, it consists of 4 main steps:

- Filter candidates for each query node,
- Select a candidate for each query node, trying to match the node with that candidate by matching the corresponding relationships between the currently processed node and previously matched nodes,
- Replace the selected candidate with another one if the current match does not work, and
- Backtrack to try other candidates and find more isomorphic patterns of query graph.

To filter the candidates for each query node, The Ullmann Algorithm checks for the matching node principal (described in Sect. 3). After the filtering step, it starts with matching of the nodes in a recursive manner in order of the nodes given in the input. Matching two nodes u and u', it checks for each relationship between u and some previously matched query node v, if there is a corresponding relationship between u' and v' which is the database node matched with v. In other words, to be selected, u and u' must satisfy the condition for each relationship that ${}_{u}r_{v}$ (or ${}_{v}r_{u}$) where $< v, v' > \in V_{matched}$, there is a corresponding relationship ${}_{u'}r'_{v'}$ (or ${}_{v'}r'_{u'}$, respectively) where $r \overset{\text{m.r.p.}}{\equiv} r'$. Lee et al. [25], check the existence of this condition called ISJOINABLE. In Fig. 3, ISJOINABLE operation for the match of u_3 in graph A with u'_3 in graph B is illustrated. Each u_i is matched with u'_i where $0 \leq i \leq 2$ and the relationships ${}_{u_1}e_{u_0}$, ${}_{u_0}e_{u_2}$, and ${}_{u_1}e_{u_2}$ are matched with ${}_{u'_1}e_{u'_0}$, ${}_{u'_0}e_{u'_2}$, and ${}_{u'_1}e_{u'_2}$, respectively. When the matching turn comes to u_3 and u'_3, ISJOINABLE procedure searches candidates for the relationships between u_3 and previously matched nodes u_0 and u_2 (the relationship between u_3 and u_4 is not considered since u_4 is not matched yet), so it checks if there is any corresponding relationship between u'_0 and u'_3 and between u'_2 and u'_3 which satisfy the necessary matching relationship principals, respectively.

In the matching phase of query node u with database node u', if there does not exist any problem upto the end of ISJOINABLE procedure, $< u, u' >$ is added into the list of matched nodes. Then, isomorphism search continues by picking the next not-yet-matched query node in order to match that with one of its candidates. In the case that the ISJOINABLE procedure returns false for a query node and its matching node, the current match is cancelled and the next candidate is selected this time. In either case, when all the matched nodes result in an

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinishied	2.132	~800,000

Fig. 2 Comparison results for running time performaces of searching a recursive query with difference depths in RDBMs and Neo4j [4]

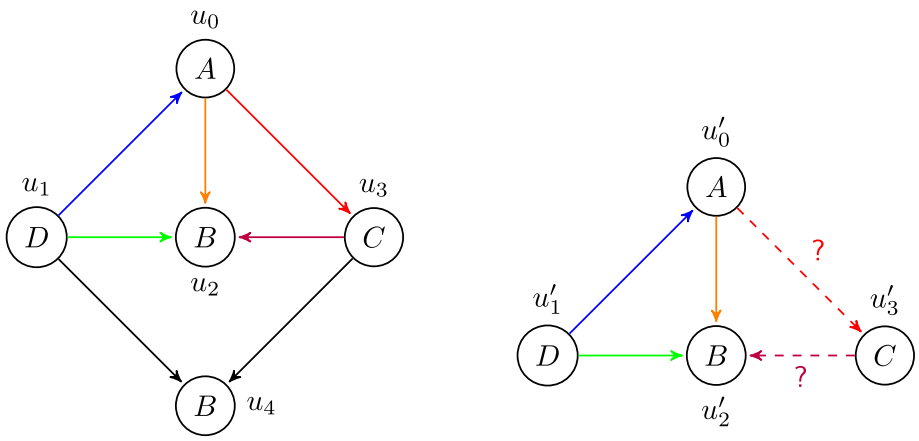

Fig. 3 IsJoinable procedure of u_3 and u'_3 while matching the graphs A and B

exact match of the query graph or a node matching results in a failure, the algorithm backtracks to try other candidates.

The Ullmann Algorithm is reasonably efficient for finding all isomorphic patterns and handling all possible node and relationship matching through backtracking. However, as several authors have also noted, that its performance may be improved with well-thought matching order strategies, effective pruning rules and some heuristics.

The **VF2** [14] matches the nodes in an increasing order of number of their labels regarding a specified query graph by following their strategy. It selects the next query node from the set of nodes that are connected to at least one of the previously matched query nodes with a relationship. In this way, it eliminates more candidates during the ISJOINABLE stage. Moreover, the VF2 algorithm refines the candidates before passing to ISJOINABLE step by comparing degrees of already matched and not-yet-matched neighbour nodes. It divides the not-yet-matched adjacent nodes into two, as the ones in first-degree-neighbourhood of the already matched nodes and the ones not in that area. It then makes degree comparison separately for both of the sets adjacent to query node u and their correspondents that are adjacent to candidate.

Shang et al [16] describe **QuickSI** as a variant of the Ullmann algorithm. The key aspect of this method is that it defines a data structure named as QI-Sequence which provides efficient pruning; and thus, resulting in low-cost processing. QI-Sequence is a minimum spanning tree created based on edges weighted according to the number of each node label and the number of each <start node label - relationship type - end node label> triple in database graph. This structure ensures that there are less possibilities to test and in this way QuickSI is able to decrease the number of recursive calls. As its pruning strategy, QuickSI applies the ISJOINABLE procedure over a query node u by beginning the checks from the relationship between u and its parent node in QI-Sequence (in case that u is not the root node).

GADDI [17] is based on the refinement of the candidate nodes using a distance based indexing- they are eliminated by examining their k-neighbourhood to detect their match with the k-neighbourhood of a query node. Using the neighborhood comparison, the discriminating ones that occur with different frequencies in common k-neighbourhood

of sample pairs of a database node are picked. GADDI uses 3 different pruning rule to refine the candidates: For a query node u and a candidate node u', firstly for each node v in k-neighborhood of u, it tries to find a candidate v' in k-neighbourhood of u' by comparing labels. Secondly, for the common k-neighbourhood of each (u, v) pair, it counts the number of discriminated fragments in this area and prunes out u' if there are less number of occurrences of a specific fragment in the corresponding region obtained by (u', v'). Thirdly, for each v it compares the length of shortest path, say t, between u and v with the one, t', between u' and v' and u' is eliminated in the case $t < t'$ for at least one v. Furthermore, GADDI applies these pruning rules in reverse manner for each candidate v' in neighbourhood of u'. Lastly, as matching order, GADDI selects the first node randomly, the rest are selected by depth first search.

GraphQL [18] focuses on neighborhood 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 $1 \leq k \leq l$. GraphQL follows an optimized node matching strategy by selecting a query node which is estimated to decrease the cost at each intermediate step and adjacent to set of already matched nodes. It compares neighbourhoods of query nodes with their candidates' neighbourhoods and tries to find a semi-perfect bipartite matching between the nodes in corresponding neighbourhoods. However, experiments show that this is an exhaustive computation even when the refinement level is set to one, and is therefore not effective towards reducing the candidate set size of query nodes.

The **SPath** algorithm [19] uses candidate paths by matching more than one node on a linear sequence at a recursive call through the ISJOINABLE procedure for each node on the path. The SPath algorithm filters the candidate vertices by checking the number of each node label in their k-neighbourhood, where k is a parameter for the radius of neighbourhood. It applies a rule which is as follows: For each node label L, the total number of occurrences of L in the neighbourhood up to kth level of query node u must be less than or equal to the total number of occurrences of L in the neighbourhood up to the kth level of database node u' where u' is a candidate for u. While matching the paths, the algorithm follows a decreasing order of path selectivity defined as a metric based on the size of candidate node sets.

The algorithms summarized above have performance issues in cases where the query is complex and when graph is large (for example, the case study described in this paper consisting of 70M nodes). Each algorithm mentioned above has some drawbacks: The pruning techniques of VF2 are not powerful enough, and the matching order it follows is effective only when database graph has similar node label statistics with query graph. QuickSI has to evaluate the whole database to deduce the information about label and <label-relationship type-label> triple count used in edge weighting. It needs an additional B+-Tree index structures to track all the nodes it has visited. GADDI creates a large index that keeps the number of discriminative fragments in the intersected k-neighbourhood of each node pair in the whole database, requiring exhaustive pre-computation. Futhermore, the pruning rules of GADDI are ineffective and time-consuming. GraphQL compares neighbourhoods of query nodes with their candidates' neighbourhoods and tries to find a semi-perfect bipartite matching between the nodes in corresponding neighbourhoods. However, experiments show that this is

an exhaustive computation even when the refinement level is set to one and thus is not effective for reducing the candidate set size of some query nodes. SPath needs a data structure including the number of each label with shortest distance i from u' for each database node u' and for each i from 1 to k; which requires a long pre-computation time and large storage. The experiments in [25] show that the ordering based on path selectivity does not provide a good performance for searching the graph in database.

Lee et al. [25] compare five subgraph isomorphism algorithms; VF2, QuickSI, GADDI, GraphQL, and SPath on real-world data sets on iGraph framework [26]. It is shown that these algorithms only work on undirected graphs consisting of one or many pieces by testing with subgraph, clique and path queries. The experiments show that there is not a satisfactory algorithm that 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 the NASA dataset described in [25]. According to the experimental results, GraphQL is the only algorithm succeeding to respond in a reasonable time for all tests on that dataset. They state that these start-of-the-art algorithms perform poorly because of their ineffective matching order and the trade-off between efficiency and overhead of their pruning methods.

In [27], **LAD-filtering** is introduced which targets to decrease the number of backtracks by applying a strong filter. That is, they guarantee to find neighbourhood of each query node with noncoincident matchings in the database graph. Although they are successful at reducing the search space, their computational cost is high and their experimental work is proved only on much smaller size of databases compared to ours. On the other hand, the **RI** algorithm given in [28] focuses on efficient and early pruning by avoiding complex computations. They handle this by following a static matching order, contrary to LAD-filtering's dynamic matching, which is based on query graph density and neighborhood characteristics.

Also, there are some studies which are conducted in biochemistry area.

On further analysis, there is a common denominator that causes all of these algorithms to perform poorly. While the pruning rules that they use are generally effective for eliminating the candidate nodes, they do not apply the pruning by regarding each isomorphism as independent, and so the nodes belonging to different isomorphisms cannot be discriminated until their relationships are checked in matching phase. Therefore, the database nodes which are candidates for different query vertices and members of distinct isomorphisms seem available for taking place in the same subgraph isomorphism at first glance. The computational time complexity of the algorithms primarily occurs at this point - searching for a non-existing reasonable connection between those irrelevant nodes. In order to remove such 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 starting node candidate potentially creates a distinct isomorphic region; therefore, the representatives for the other query nodes should be chosen from the close neighbourhood of the starting node through a *local* region scanning instead of a *global* one for each distinct exact match. The HyGraph algorithm is formulated to specifically overcome the problems presented.

Method

The problems with previous approaches is related to the inefficient mechanisms for restricting the possible graph isomorphisms. The algorithm developed, HyGraph, overcomes this drawback through the use of efficient pruning mechanisms. It first chooses

a query node to start the matching process (*starting node*). It then filters the database nodes to find possible candidate matches for the starting node using the *matching node principal* mentioned previously. HyGraph puts the candidate nodes into a list and searches for query graph isomorphisms rooting from that match. The whole candidate list is walked through a loop such that at each iteration the next candidate node is picked from the list, matched with the starting node and the recursive isomorphism search begins from that point, as shown in Algorithm 1, lines 1–12.

Algorithm 1: HyGraph 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, HyGraph 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 $u' \in V_G$, HyGraph 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 HyGraph 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 Sect.) holds or not. After the filtering, there may be more than one candidate for a relationship. HyGraph collects all the candidates in a separate list for each non-matched relationship adjacent to u, as specified in 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, HyGraph 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' are matched before. Thus, HyGraph has to check whether the other end points of r_i and r'_i, say v and v', match, as shown in Algorithm 3. It can be one of the following three cases: First, if v and v' are already matched with each other, then there is no conflict. Second, if v is matched with some node other than v', or vice versa, it is reported as a conflict. Third, it may be the case that both v and v' 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 the 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 HyGraph 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. At this point, HyGraph 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 HyGraph 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, as in Algorithm 1, lines 9–25.

When HyGraph 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, as it is given in Algorithm 2, lines 21–22.

Since the 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, HyGraph continues its schedule with backtracking. When HyGraph completes the iterative relationship matching part successfully, all the relationships adjacent to u are matched with some relationship adjacent to u'. This completes 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 matching, as in Algorithm 1, lines 14–29.

In order to illustrate how HyGraph works, a sample running schedule is given in Fig. 4. 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 Fig. 1.

Complexity of the HyGraph algorithm

The running time complexity (computational complexity) analysis and space complexity analysis of HyGraph for the worst case scenarios is evaluated in the following discussion. The number of nodes (vertices) in query graph Q is denoted 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

HyGraph starts with the filtering part. It finds candidates for the starting node by filtering the database nodes depending on the *matching node principal* given in Sect. 3. To find all the candidates, label and degree properties of each database node must be checked. The empirical evaluation of the algorithm uses Neo4j, which stores the information of database nodes grouped by label. The filtering with respect to node labels takes $\mathcal{O}(1)$. In the worst case, all database nodes may have labels of the starting

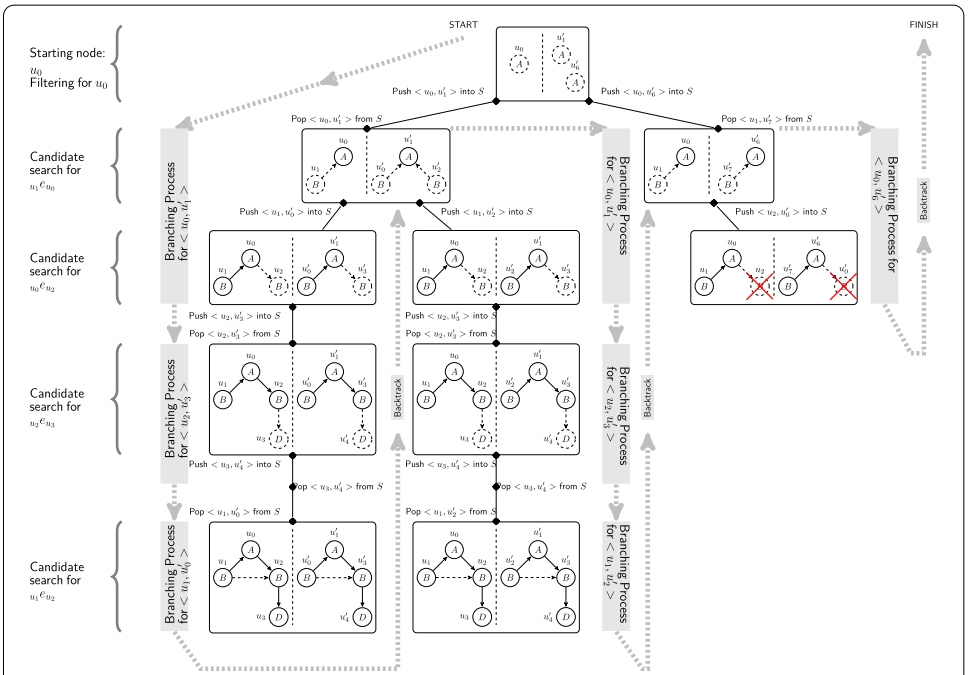

Fig. 4 A sample execution for the query graph and graph database given in Fig. 1. Each level of the diagram represents the candidate relationship search for edge matching. At the rectangle boxes of each level the found candidates are shown in dashed lines whereas the already matched nodes and relationships are shown in non-dashed lines. 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

node, which means that there exist $|V_G|$ number of candidate nodes that are going to be filtered with respect to their adjacent relationships. Neo4j directly provides the information about the different adjacent relationships of each type for every candidate node. Therefore, this part takes $\mathcal{O}(|V_G| \times deg_Q^{max})$, which also gives the worst case complexity of the total filtering stage. In the worst case, it is possible that there are $|V_G|$ number of candidates for the starting node.

An isomorphism search is conducted for each candidate of the starting node. The time complexity of this search is $|V_G| \times \mathcal{O}(\text{Search}())$. Since, 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 analyse the computation cost of the operations done in one recursive depth of the branching procedure.

Algorithm 3: CHECKCONFLICT

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 ≡^{m.n.p.} ν'_i 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
```

In the branching procedure (Algorithm 2), initially, all non-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 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} 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 $(deg_G^{max})^{(x_i)}$ 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 a 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 CHECKCONFLICT() function (Algorithm 3). In CHECKCONFLICT(), 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 CHECKCONFLICT function is called x_i, namely deg_Q^{max} at most and 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 be equal to the number of query nodes, $|V_Q|$, the overall complexity of the 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 HyGraph equals to the summation of the cost of the filtering part and the cost of the 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 the filtering part is negligible when compared to the cost of the searching part. As a result, the running time complexity of HyGraph is $\boldsymbol{\mathcal{O}(|V_G| \times |V_Q| \times (deg_G^{max})^{(deg_Q^{max})} \times (deg_Q^{max})^2)}$.

Space complexity

For the filtering part, HyGraph 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 (that is, all the database nodes), there is a need for a list of size $|V_G|$ in the worst case. In addition, for each query relationship, there is a constructed 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 worst case is when the recursive computation reaches to the deepest value and requires 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 the storage needed. Additionally, there exists global variables consuming some memory- two global hashing maps used to hold the currently matched query graph and database graph items; one maps for the matched nodes and the other maps for the matched relationships. These two maps include information of which query node or relationship is 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. Finally,

the stack which is used to hold node matches, which are not yet sent to *reciprocal node branching process* needs $2 \times |V_Q|$ units of storage, that is, the space required for the case that starting node is adjacent to all the remaining query nodes. Throughout the whole execution of HyGraph, all the other variables minimal space and can be neglected. Consequently, the space complexity of HyGraph equals to summation of all the mentioned parameters appeared in the worst case, which is $\mathcal{O}(|V_G| + |V_Q| + |E_Q| \times deg_G^{max})$.

Results

We compare the performance of HyGraph, with two graph query languages - Cypher and GraphQL emperically. Cypher is a declarative graph query language that permits efficient querying and updating of a property graph introduced as a query language for Neo4j, though subsequently released as open source. GraphQL is an open-source data query and manipulation language for application programming interfaces (APIs) which permits clients to define the structure of the data needed from data sources thereby minimizing the amount of data returned. Both Cypher and GraphQL are widely used on massive scale web graph data in industry, and the comparison with HyGraph is used to bring out the relative performance advantages of the latter.

The experimental datasets consist of the 2018 WorldCup Database, a Bank Database, and a Population Database consisting of 45K, 100K, and 70M nodes, respectively. These datasets provide the scale and complexity required to compare HyGraph with Cypher, and GraphQL. In Table 1, features of Population, Bank and WorldCup Databases are given in detail. It may be noted that the Population Database is several orders larger than the examples used in [25], as shown in Table 1.

For the experiments, we use 10 real-world queries of varying complexity for Population Database and 5 real-world queries for each of WorldCup and Bank Databases where each query has different number of nodes and relationships and also has different types of node labels and relationships. For each query, HyGraph and Cypher experiments are repeated 10 times, and the averages of the elapsed times are used. For all GraphQL experiments refinement-level is adjusted to one (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.

Table 1 General Features of Population, Bank and WorldCup Databases

	WorldCup DB	Bank DB	Population DB
Size	20 MB	510.23 MB	10.32 GB
# of graphs	1	1	1
# of nodes	45348	105085	70422787
# of relationships	86577	107898	77163109
# of distinct node labels	12	15	14
# of distinct relationship types	17	18	18
Avg. # of labels per node	1	1	1

Setup

The experiments were conducted on a 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. HyGraph and GraphQL are implemented in Java on Eclipse and Cypher queries are called in Java.

Experimental analysis

We group the experimental results according to the query type constructed by node matching orders used in HyGraph. There exist 3 types of queries that we use in our experiments: First, if a query includes cyclic paths on it, we call that as a *complex query*. Second, if a query itself is a single path and HyGraph starts matching from one of its end nodes, we simply call that as a *path query*. Third, if a query does not contain any cycles and HyGraph matches its nodes in a tree manner, we call that as a *tree query*. Figures 5, 6 and 7 show a sample complex query, a sample path query and a sample tree query, respectively.

Complex queries

Table 2 shows the experimental results obtained for distinct complex-type query graphs executed over Worldcup, Bank and Population Databases.

As it can be clearly seen from Table 2, HyGraph performs much better than both Cypher and GraphQL for most complex-type queries. Generally, HyGraph has higher performance than Cypher for the queries that we have tested, except the queries Q-11 and Q-14, for which Cypher performs a little better than HyGraph.

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

Table 2 Query Response Times for Complex Queries

Query	Database	Number of Nodes	Number of Relationships	Number of Cycles	Peformance of GraphQL (sec.)	Performance of Cypher (sec.)	Performance of HyGraph (sec.)
Q-1	WorldCup	4	4	1	1112.5	10.1	1.9
Q-2	WorldCup	7	8	2	17.9	7.8	3.9
Q-3	WorldCup	10	12	3	8.1	26.7	7.9
Q-4	Bank	4	4	1	238.9	1.5	1.4
Q-5	Bank	5	5	1	339.0	2.1	2.0
Q-6	Bank	6	6	1	630.8	3.6	2.7
Q-7	Bank	9	9	1	> 1800.0	8.9	6.0
Q-8	Bank	12	12	1	> 1800.0	226.8	16.0
Q-9	Population	6	12	7	> 1800.0	189.9	138.1
Q-10	Population	5	7	4	> 1800.0	24.2	22.3
Q-11	Population	5	5	1	> 1800.0	14.8	17.3
Q-12	Population	6	6	1	> 1800.0	92.4	54.5
Q-13	Population	7	11	5	> 1800.0	14.4	8.5
Q-14	Population	7	11	4	> 1800.0	13.7	14.0
Q-15	Population	13	25	13	> 1800.0	> 1800.0	502.7
Q-16	Population	10	14	3	> 1800.0	226.7	73.4

The timings are given in terms of seconds (sec.)

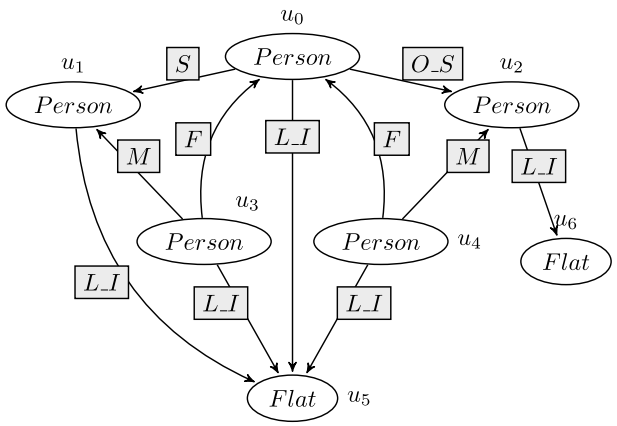

Fig. 5 An example for a complex query: "*Find the families consisting of a man, a wife, at least two children, one from an ex-wife, who live at the same address in Population Database*". The relationship "*S*" stands for "*spouse of*", "*O_S*" stands for "*old spouse of*", "*M*" stands for "*has mother*", "*F*" stands for "*has father*" and "*L_I*" stands for *"lives in"*

2 cycles and 10 nodes and 12 relationships with 3 cycles, respectively, HyGraph 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 HyGraph for Q-3. We think that it is probably caused by the fact that the existence of a very unique node label in Q-3 made the number of candidate nodes very small. Therefore, the entire search does not take a long time.

In Bank Database, despite the fact that GraphQL succeeds in returning the results for Q-4 consisting of 4 nodes and 4 relationships with 1 cycle, Q-5 consisting of 5 nodes and 5 relationships with 1 cycle and Q-6 consisting of 6 nodes and 6 relationships with 1 cycle, its performance clearly falls behind the performances of Cypher and HyGraph. On the other hand, for the other two queries in Bank Database, which are Q-7 consisting of 9 nodes and 9 relationships with 1 cycle and Q-8 consisting of 12 nodes and 12 relationships with 1 cycle, GraphQL could not terminate in 30 minutes. Lastly, for all of the five queries run in Bank Database, as it can be seen from Table 2, HyGraph performs better than Cypher.

From those queries executed in Population Database, for Q-9, Q-10, Q-12, Q-13, Q-15 and Q-16 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, HyGraph performs much better than Cypher. For Q-15, which is the most complex and the largest query among all in our experiments, Cypher could not finish its execution in a reasonable time. Interestingly, although Q-13 and Q-14 have the same number of nodes, relationships and even the node labels, the performance of HyGraph changes. The reason for that as follows: While both of those queries have the same number of items, they have different structuring such as the neighbourhoods, node degrees and the number of cycles, which also specialize a graph. For that reason, performance of HyGraph changes with the number of false candidates that depends on how much a query graph is specialized with its characteristics. Lastly, in

Population Database, GraphQL could not terminate in 30 minutes for any of these complex queries executed.

Path queries

Table 3 shows the experimental results obtained for distinct path-type query graphs executed over Worldcup and Population Databases.

From Table 3, it can be seen that HyGraph has the best performance for 2 path-type queries out of 3. The first path query, Q-17, consists of 5 nodes and 4 relationships executed over WorldCup Database. For this query, HyGraph shows 4 times better performance than Cypher. For Q-17, GraphQL shows rather low performance when compared to HyGraph's and Cypher's performances. On the other hand, this is the only path query that GraphQL could complete its execution in 30 seconds. Since Population Database is much more complex, dense and larger than the WorldCup Database, GraphQL exceeds a reasonable time limit, which we set as 30 minutes, for the other two path queries. The second path query, Q-18, consists of 8 nodes and 7 relationships executed over Population Database. This query is a recursive query, with the depth of 4. For Q-18, again HyGraph performs much better than Cypher. On the other hand, for Q-19, which is the third path query consisting of 4 nodes and 3 relationships, Cypher peforms almost the same as HyGraph.

Tree queries

Table 4 shows the experimental results obtained for distinct tree-type query graphs executed over Worldcup and Population Databases.

Table 3 Query Response Times for Path Queries

Query	Database	Number of Nodes	Number of Relationships	Peformance of GraphQL (sec.)	Performance of Cypher (sec.)	Performance of HyGraph (sec.)
Q-17	WorldCup	5	4	28.5	1.6	0.4
Q-18	Population	8	7	> 1800.0	26.2	2.2
Q-19	Population	4	3	> 1800.0	12.5	18.9

The timings are given in terms of seconds (sec.)

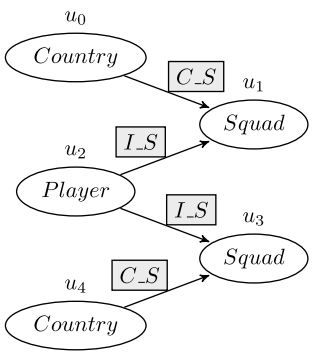

Fig. 6 An example for a path query: *"Find the players who join the squad of different countries in WorldCup Database"*. The relationship "*C_S*" stands for "*creates squad*" and "*I_S*" stands for "*in squad*"

Table 4 Query Response Times for Tree Queries

Query	Database	Number of Nodes	Number of Relationships	Peformance of GraphQL (sec.)	Performance of Cypher (sec.)	Performance of HyGraph (sec.)
Q-20	WorldCup	10	9	> 1800.0	26.9	35.7
Q-21	Population	4	3	> 1800.0	12.5	24.1

The timings are given in terms of seconds (sec.)

We have executed 2 tree-type queries; one in WorldCup Database and the other in Population Database. The first tree query, Q-20, consists of 10 nodes and 9 relationships on which there exist 3 identical paths including 3 nodes such that all the 3 paths are connected to a common node and construct a tree with 3 branches. Similarly, the second query, Q-21, consists of 4 nodes and 3 relationships on which there exist 2 identical paths including 2 nodes such that the paths are connected to each other with an edge. It can be seen from Table 4 that Cypher has better performance than HyGraph for this type of queries. We think that the reason for the Cypher's good performance is that Cypher applies an efficiency rule for the queries which include identical patterns by conducting a search only for one of those patterns instead of searching for each. Since we do not analyze the query graphs in terms of their analytical features which have such short cuts, performance of HyGraph falls somewhat behind the performance of Cypher in those cases. As a future work, HyGraph can be enhanced in order to increase the performance for such cases as well.

Matching order analysis for HyGraph

Table 5 shows the effect of different node matching orders for some of the queries executed by HyGraph in Population Database.

As it can be seen from Table 5, when we change the node matching order followed by HyGraph Algorithm (by just changing the starting node and then the rest is determined by the algorithm itself), the performance considerably changes. For Q-18, which is the path query explained above, the reason of the change in performance is the *N*-1, 1-1 or *N*-*M* property of relationship types between the node labels. Clearly, for a *N*-1 type of relationship, it is better to match the node in '*N*' side first, and then the node in '1' side next, instead of matching those in the reverse order because the number of candidate nodes automatically decreases to 1 in the first case, whereas there may occur *N* candidates in the second case. Thus, for Q-18, when we change the path following direction from 1-*N* to *N*-1, the performance increases almost 20 times. For Q-21, which is a tree query, the performance increases when we match its nodes as a path query that corresponds to Q-19. Additionally, for Q-11 and Q-12, the increase in performance occurs when HyGraph starts matching from the node that is rarer due to its semantics appearing with its neighbourhood. As a result, since the node matching order may result in a change in the number of false candidates for query nodes, it highly affects the performance of HyGraph.

Discussion

The overall study shows that when the size of the candidate node search space gets smaller, performance of the graph query algorithms gets quicker. Throughout the experiments, we observed that the number of candidate nodes can be reduced either

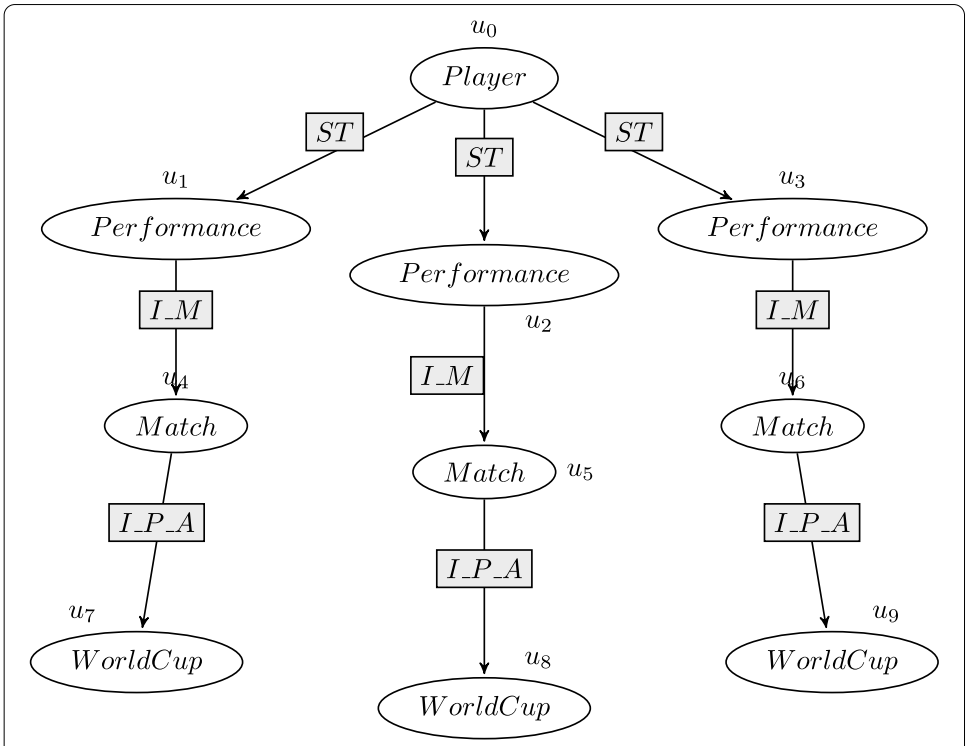

Fig. 7 An example for a tree query: *"Find the players who participate in at least 3 different World Cup Tournaments in WorldCup Database"*. The relationship "*ST*" stands for "*started*", "*I_M*" stands for "*in match*" and "*I_P_A*" stands for "*is played at*"

Table 5 Query Response Times for Different Matching Orders in BB-Graph

Query	Database	Number of Nodes	Number of Relationships	Performance of HyGraph (sec.) Matching Order-1	Performance of HyGraph (sec.) Matching Order-2
Q-18	Population	8	7	41.4	2.2
Q-19, Q-21	Population	4	3	24.1	18.9
Q-11	Population	5	5	33.3	17.3
Q-12	Population	6	6	101.4	54.5

The timings are given in terms of seconds (sec.)

by focusing on the relationships between the relevant candidates or by following a smart matching order strategy. In order to handle the relevancy, applying robust pruning techniques on the candidates can provide benefit to some extent. Nonetheless, it may not be not sufficient as we have seen during the experiments done with GraphQL. It is also necessary to preserve the dependency between the candidates of distinct query nodes, which was our main concern in the production phase of HyGraph algorithm. The experiments done with HyGraph shows that iterative matching of each query node with the candidates connected to previous matchings increases the performance significantly. In addition to these, the order determining which node is matched before or after the other is very critical since it affects the size of the candidate set which is created dynamically at each step. For the queries having

the $N-1$ type of relationships it is easy to decide on from which node to start due to the fact that one side can have only one candidate for the desired adjacency whereas the other side has N-many. However, for most of the queries, it is really difficult to detect which node will create an advantageous state in the next steps since it depends on many aspects like total number of distinct types of nodes, distinct types of relationships, neighborhood structures etc., which results in an N^{th} order equation with multiple variables. In these cases, it would be very beneficial to find the node having the "leading role" in the query such that matching it first discriminates the true matches from the false candidates. The leading node may be determined through a wide range of approaches from the basic statistics such as selecting the most visited node on the shortest paths between the other nodes to complex machine learning algorithms. Since it is out of scope of this paper, we leave it as a future work.

Conclusions

This paper introduces a new algorithm, HyGraph, for the Subgraph Isomorphism Problem. HyGraph uses a hybrid technique to solve the problem by matching each node and relationship of a query graph with its candidates and backtracking for the other possible candidates. As differentiated from the current algorithms, HyGraph does not find candidates of each query graph node all across the graph database. After matching the first query node, HyGraph 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 real-world databases, a Population Database 70M individuals, the 2018 WorldCup Database, and one simulated Bank Database, to obtain realistic scenarios of usage. We evaluate HyGraph experimentally with two graph query languages, GraphQL and Cypher, widely used in industry to obtain practical comparisons. The experimental results reveal that GraphQL is not scalable for querying very big graph databases such as Population Database, though it shows comparable performance with HyGraph for some types of queries in medium size databases. In general it may be stated that GraphQL performs poorly in comparison to Cypher and HyGraph due to its strategy of obtaining candidate nodes from the entire database and trying to match all nodes even irrelevant ones. In comparing HyGraph with Cypher, our experimental results show that HyGraph performs better than Cypher for most of the query types, for both large and small databases. From these results it can be concluded that the HyGraph algorithm provides a query mechanism that outperforms current industry standard query languages, making HyGraph suited to the big graph database scenarios required by social media applications.

The experimental results reveal that the performance of HyGraph algorithm does not depend on just one factor such as the number of nodes or relationships. Other features, including the frequency of query node labels and query relationship types in the database graph, the number of cycles in query graph, structure of query graph (in other words, whether it is a path or something more complex) and the semantic design of relationships like $1-N$ or $N-N$, affect the computation time. A future study would further improve the performance of HyGraph further by methodically evaluating the features of the query and graph database mentioned above.

Abbreviations
HyGraph: Hybrid graph; RDBM: Relational Database Management System; w.r.t.: With respect to; m.n.p.: Matching node principal; m.r.p.: Matching relationship principal.

Acknowledgements
Not applicable.

Author contributions
MA developed the research approach and the computer code for this work, analyzed the results, and wrote the paper. AY provided in-depth guidance on the research approach, analyzed the results and was a major contributor to writing the paper. RG analyzed the results and contributed to writing the paper. All authors read and approved the final manuscript.

Funding
This research is funded in part by NSF, and NGC under Grant Numbers FAIN-1901150, and 2017–2007 respectively. Any opinions, findings, and conclusions expressed here are those of the author(s) and do not reflect the views of the sponsor(s).

Availability of data and materials
Datasets used in this paper are the following: (1) a country-level population database, (2) a simulated bank database, and (3) World Cup big graph database. The datasets (1) and (2) that support the findings of this study were provided by Kale Yazılım Industrial and Commercial Corp., Ankara and used under license for the current study. These data sets are currently not available publicly. The dataset (3) used in the current study is available in the WorldCup repository, http://worldcup.neo4j.org/Dataset.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
The authors grant consent for this paper to be published.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer Engineering, Middle East Technical University, Dumlupınar Bulvarı, 06800 Ankara, Turkey. [2]Department of Cyber-Physical Systems, Clark Atlanta University, 223 James Brawley Drive, Atlanta 30314, USA. [3]Present Address: Department of Computer Science, School of Science and Technology, Nazarbayev University, Dastana, Kazakhstan.

Received: 14 April 2020 Accepted: 28 March 2022
Published online: 21 April 2022

References

1. Kolomičenko V. Analysis and Experimental Comparison of Graph Databases [Master Thesis]. Prague: Charles University Department of Software Engineering; 2013.
2. Batra S, Tyagi C. Comparative analysis of relational and graph databases. Int J Soft Comput Eng. 2012;2(2):509–12.
3. Vicknair C, Macias M, Zhao Z, Nan X, Chen Y, Wilkins D. 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; 42.
4. Bitnine. Bitnine, editor. Relational Database vs Graph Database. http://bitnine.net/rdbms-vs-graph-db/; 2016. Accessed Oct 2016.
5. Abboud A, Backurs A, Hansen TD, Williams VV, Zamir O. Subtree isomorphism revisited. In: Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms. SIAM; 2016;1256–1271.
6. Giugno R, Shasha D. 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;112–115.
7. Yan X, Yu PS, Han J. Graph indexing: a frequent structure-based approach. In: Proceedings of the 2004 ACM SIGMOD, International Conference on Management of Data. ACM; 2004;335–346.
8. Srinivasa S, Maier M, Mutalikdesai MR, Gowrishankar K, Gopinath P. LWI and Safari: A New Index Structure and Query Model for Graph Databases. In: COMAD; 2005;138–147.
9. He H, Singh AK. Closure-tree: An index structure for graph queries. In: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06). IEEE; 2006;38.
10. Williams DW, Huan J, Wang W. Graph database indexing using structured graph decomposition. In: IEEE 23rd International Conference on Data Engineering (ICDE 2007). IEEE; 2007;976–985.
11. Zhang S, Hu M, Yang J. Treepi: A novel graph indexing method. In: IEEE 23rd International Conference on Data Engineering (ICDE). IEEE; 2007;966–975.
12. Zhao P, Yu JX, Yu PS. Graph indexing: tree+ delta<= graph. In: Proceedings of the 33rd International Conference on Very Large Data Bases. VLDB Endowment; 2007;938–949.
13. Ullmann JR. An algorithm for subgraph isomorphism. J ACM. 1976;23(1):31–42.

14. Cordella LP, Foggia P, Sansone C, Vento M. A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell. 2004;26(10):1367–72.
15. Carletti V, Foggia P, Saggese A, Vento M. Introducing VF3: A new algorithm for subgraph isomorphism. In: International Workshop on Graph-Based Representations in Pattern Recognition. Springer; 2017;128–139.
16. Shang H, Zhang Y, Lin X, Yu JX. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proc VLDB Endowment. 2008;1(1):364–75.
17. Zhang S, Li S, Yang J. 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;192–203.
18. He H, Singh AK. 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;405–418.
19. Zhao P, Han J. On graph query optimization in large networks. Proc VLDB Endow. 2010;3(1–2):340–51.
20. Lange C, Sneed HM, Winter A. Comparing graph-based program comprehension tools to relational database-based tools. In: Proceedings 9th International Workshop on Program Comprehension (IWPC). IEEE; 2001;209–218.
21. Wycislik L, Warchal L. A performance comparison of several common computation tasks used in social network analysis performed on graph and relational databases. In: Man-Machine Interactions 3. Springer; 2014;651–659.
22. Miller JJ. Graph database applications and concepts with Neo4j. In: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA. vol. 2324; 2013;134–140.
23. Nayak A, Poriya A, Poojary D. Type of NOSQL databases and its comparison with relational databases. Int J Appl Inform Syst. 2013;5(4):16–9.
24. Küçükkeçeci C, et al. A Graph-Based Big Data Model for Wireless Multimedia Sensor Networks. In: INNS Conference on Big Data. Springer; 2016;205–215.
25. Lee J, Han WS, Kasperovics R, Lee JH. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: Proceedings of the VLDB Endowment. vol. 6. VLDB Endowment; 2012;133–144.
26. Han WS, Lee J, Pham MD, Yu JX. iGraph: a framework for comparisons of disk-based graph indexing techniques. Proc VLDB Endow. 2010;3(1–2):449–59.
27. Solnon C. Alldifferent-based filtering for subgraph isomorphism. Artif Intell. 2010;174(12–13):850–64.
28. Bonnici V, Giugno R, Pulvirenti A, Shasha D, Ferro A. A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform. 2013;14(S7):S13.

Publisher's Note