Graph queries for analyzing the coverage of requirements by test cases

Shingo Ariwaka, Hiroyuki Nakagawa, Tatsuhiro Tsuchiya
Graduate School of Information Science and Technology, Osaka University

Abstract

We study the applicability of graph queries to the coverage analysis of test cases for requirements specifications. First we show that when the similarity degrees between requirements specifications and test cases are available, they can be represented in the form of a graph. Then we identify several queries that are useful for extracting coverage information and show that all these queries can be written in the Cypher query language, a common graph query language. In a case study we apply these queries to data obtained from a real-world project in industry. The results of the case study show that coverage information can be retrieved in reasonable time. We also compare the graph queries with SQL queries with respect to conciseness and processing time.

1. Introduction

In this paper we study the applicability of graph queries in the field of software engineering. A graph query is a query for accessing a graph database which manages data with a graph structure. Graph databases and graph queries are becoming popular since they are specially tailored to handle data with graph structures, such as social networks [1]. On the other hand, applications of graph databases and graph queries to software engineering problems have not been studied sufficiently. This paper aims to investigate whether or not graph queries can be effectively used in analyzing the test coverage of requirements specifications in software system development. In our context, coverage means how many of the requirements specifications are tested by the test cases and the extent to which each of the requirements specification is tested.

There has already been a body of research that can be used to automatically compute the relevance or similarity between software artifacts, such as test cases and requirements specifications (for example, [2, 3]). In these previous studies, the similarity degree between two artifacts is estimated using natural language processing. The obtained similarity degrees can be used to, for example, infer the existence of traceability link between them.

In this paper, we assume that using some of these techniques similarity degrees have already been computed between test cases and requirements specifications. Under the assumption, we first identify several information items that are useful for system developers to perform coverage analysis. Then we show that these items can be naturally specified in the form of graph queries which are in turn used to retrieve coverage information from a graph database. Furthermore we demonstrate practical applicability of these graph queries through a case study using data obtained from a real-world industrial product.

The structure of this paper is as follows. In Section 2, we introduce graph databases and Cypher, a well-known query language for graph databases. In Section 3, we describe the basic assumptions about requirements specifications and test cases and show how they can be represented in the form of a graph. In Section 4, we list information items that can be useful for developers to perform coverage analysis. We also show that these items can be retrieved using Cypher queries. In Section 5, we show the results of a case study where a data set from an industrial product is used. In Section 6 we describe related research. In Section 7 we discuss potential threats to validity of the study. Finally, in Section 8, we summarize the paper and discuss future research directions.

2. Graph database and query languages

2.1. Graph database

A *graph database* is a database that manages data with a graph structure. In this paper we adopt the model of graph structures used by Neo4j, a well-known graph database [1]. In the Neo4j's model, a graph structure consists of *nodes*, *relationships*, and *properties*. Nodes refer to vertices and relationships refer to direct edges in graph theoretical terms. Properties are attributes assigned to nodes and relationships and can hold data such as numbers and strings in a key-value format. Furthermore, labels can be assigned to nodes and relationships. When searching the database, the labels are useful for specifying data items to be retrieved.

DOI 10.18293/SEKE2021-122

2.2. Cypher query language

In database management systems, accesses to databases are performed through execution of queries written in a query language. Cypher is a common query language for graph databases [1]. For example, Neo4j uses Cypher as its query language. Cypher allows for expressive querying of graph databases. Queries in Cypher create, read, update, and delete data items. In this paper we basically focus on read queries since our interest is in analysis over a set of requirements specifications and test cases that have already been provided. Read queries in Cypher start with keyword MATCH followed by a search pattern for finding nodes or relationships. Additional constraints to the pattern can be added using keyword WHERE.

Figure 1 shows a graph data representing relationships among three peoples which represents that Smith is known to Williams and Johnson. The nodes correspond to the people and represent their names by property `name`. For example, a query to retrieve a list of people who know Smith from this data can be written as follows.

```
1  MATCH (a:Person)-[:Knows]->(b:Person)
2  WHERE b.name = "Smith"
3  RETURN a
```

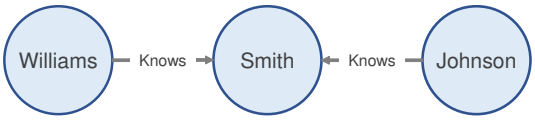

Figure 1: A graph data representing human relationships

3. Requirements specifications and test descriptions

3.1. Traceability data

In this study we assume that a set of requirements specifications, a set of test cases, and the similarity degrees between them are available. We collectively call these data *traceability data*. Below we formally describe the taceability data we assume and show that the data can be represented as a graph structure.

We consider traceability data that consists of a set $\mathcal{R}$ of requirements specifications, a set $\mathcal{T}$ of test cases, and the similarity degrees between them: The similarity degree is a real value ranging from 0 to 1 and is defined between any two requirements specifications and between any pair of a requirements specification and a test case. Formally,

Table 1: Example of traceability data consisting of seven requirements specifications and ten test cases. Entities with 0 similarity are omitted.

(a) Requirements specifications-pairs

$r0$	$r1$	0.14
$r0$	$r5$	0.08
$r0$	$r6$	0.59
$r1$	$r4$	0.97
$r1$	$r5$	0.84
$r2$	$r3$	0.2
$r2$	$r4$	0.94
$r2$	$r6$	0.34
$r4$	$r5$	0.93

(b) Test case and requirements specification-pairs

$t0$	$r0$	0.13
$t1$	$r0$	0.57
$t1$	$r1$	0.82
$t2$	$r2$	0.35
$t3$	$r2$	0.16
$t4$	$r2$	0.59
$t4$	$r4$	0.21
$t5$	$r4$	0.01
$t5$	$r5$	0.03
$t7$	$r1$	0.91
$t7$	$r2$	0.41
$t7$	$r4$	0.09
$t8$	$r0$	0.06
$t8$	$r1$	0.28
$t8$	$r2$	0.48
$t8$	$r5$	0.5
$t9$	$r1$	0.21
$t9$	$r2$	0.66
$t9$	$r5$	0.54

the similarity degree between a requirements specification pair is represented as the map $S_1 : \mathcal{R} \times \mathcal{R} \to [0, 1]$, and the similarity degree between a test case and a requirements specification is represented by the map $S_2 : \mathcal{T} \times \mathcal{R} \to [0, 1]$ where [0, 1] is the set of real numbers between 0 and 1. In this paper, we assume that $S_1(r, r') = S_1(r', r)$ for any requirements specifications $r, r' \in \mathcal{R}$

Table 1 shows a small example of traceability data, where $\mathcal{R} = \{r0, r1, \ldots, r6\}$ and $\mathcal{T} = \{t0, t1, \ldots, t9\}$.

3.2. Graph structure as traceability data

The traceability data can be represented in a graph structure as follows. The test cases and requirements specifications correspond to nodes. Nodes of test cases and nodes of requirements specifications are distinguished by assigning different labels to them. We call the node corresponding to a test case a *test case node* and the node corresponding to a requirements specification a *requirement node*. The similarity degree between requirements specifications pair or between test cases and requirements specifications is represented by a property assigned to the relationship defined between the requirement nodes or between the test case node and the requirement nodes. An exception is when the similarity degree between two nodes is 0, in which case no relationship is defined between them.

It should be noted that the directions of relationships are irrelevant to the traceability data we consider. Hence we set at most one relationship of either direction between two nodes. This is a standard treatment of undirected edges in the graph model of Neo4j.

Figure 2 shows a visualization of the above example, obtained using Neo4j's functions.

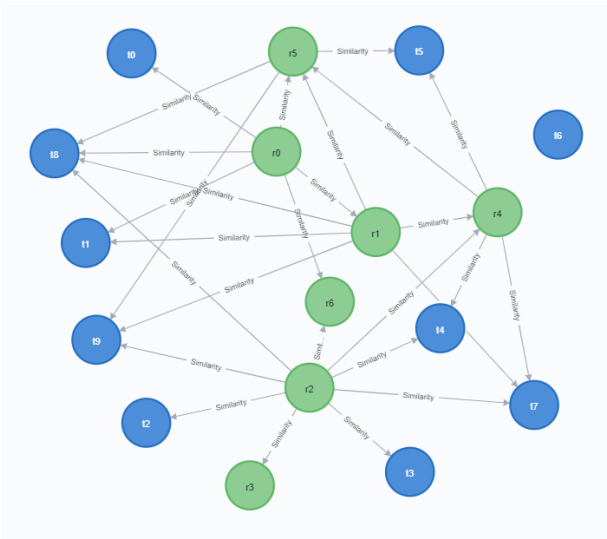

Figure 2: Visualization of the example traceability data

3.3. Table representation of traceability data

Compared to graph databases, relational databases have been much more widely used for a variety of applications. In a relational database, data are stored in tables which are rigorously designed to ensure data consistency. SQL queries are used to retrieve necessary information spanning across these tables. Here we explain how the traceability data can be represented as the form of tables here.

We use a total of four tables: two *node tables* and two *edge tables*. One of the node tables stores the requirement nodes, while the other node table stores the test case nodes. Each row of the node tables consists of a unique ID and a name of the corresponding node. The edge tables store relationships and their attributes, including similarity. One of the tables stores the similarity degrees between requirements specifications, whereas the other manages those between test cases and requirements specifications. Each row of the edge table contains the IDs of the end nodes of the corresponding relationship, as well as the similarity. The end nodes' IDs must appear in the node tables and this constraint can be imposed with FOREIGN KEY constraints, which are usually supported by relational database management systems.

4. Database queries for coverage analysis

In this section, we list some information items that may be useful for developers to know how well the test cases test the requirements specifications. We wrote queries both in Cypher and SQL for retrieval of all these items. For space limitations we select two items as examples and present Cypher queries and SQL queries for extracting them. In addition the conciseness of the queries is evaluated in terms of character count.

4.1. Coverage information to be extracted

We identify a total of seven information items that can be useful for coverage analysis as follows:

1. List of test cases that directly test requirements specification R
2. List of test cases that directly and indirectly test requirements specification R
3. List of requirements specifications that are directly tested by test case T
4. List of requirements specifications that are directly or indirectly tested by test case T
5. List of requirements specifications that are directly tested by at least one test cases
6. List of requirements specifications that are directly or indirectly tested by at least one test cases
7. List of requirements specifications that are not tested

Here we say that a test case t *directly tests* a requirements specification r if t and r have similarity degree equal to or greater than the threshold X. Also we say that a test case t *indirectly tests* a requirements specification r_1 if r_1 and another requirements specification r_2 have the similarity degree equal to or greater than the threshold Y and r_2 is directly tested by at least α test cases including t.

In the following of the paper, we set X=0.5, Y=0.7, and α=2 and hard-code these values in queries for presentation simplicity.

4.2. Queries for retrieval of coverage information

For each of the above items, we created a Cypher query and an SQL query to extract it. Due to the limit of space, we only present the queries for the first and sixth items. Note that the $s and %s in the queries serve as formal parameters and are replaced with actual arguments at runtime.

Query 1, the query for the first item, receives the name of a requirements specification as an actual argument and

returns a list of test cases that directly test that requirements specification. When the query is executed with r1 being the actual argument for the running example shown in Section 3.1, the query should return two test cases, t1 and t7.

Below we show two queries written in Cypher and SQL. From the description of the query in Cypher, it is seen that the Cypher query very succinctly describes the pattern that matches relationships the target test cases must posses. On the other hand, the SQL implementation of Query 1 is slightly more complicated because it requires joining the node tables and the edge tables in order to obtain the similarity between test cases and the requirements specification.

Cypher query 1

```
MATCH (r:Requirement)-[s:Similarity]-(t
    :Testcase)
WHERE r.name = $s AND s.value >= 0.5
RETURN t.name AS test_name
ORDER BY test_name
```

SQL query 1

```
SELECT test.name AS test_name
FROM edge_req_to_test edge
JOIN node_req req
ON edge.from_id = req.id
JOIN node_test test
ON edge.to_id = test.id
WHERE req.name = %s AND edge.similarity
     >= 0.5
ORDER BY test_name;
```

Query 6 for the sixth item obtains a list of all requirements specifications that are tested directly or indirectly by at least one test case. For the running example, the query yields a set of five requirements specifications: r0, r1, r2, r4, and r5. In the Cypher implementation of this query, even more complex pattern matching than Query 1 is described in an intuitive way. On the other hand, the SQL query is long and difficult to understand, because many JOIN operation are performed and similar descriptions need to be repeated for matching requirements specifications.

Cypher query 6

```
CALL {
    MATCH (r:Requirement)-[s:Similarity
        ]-(t:Testcase) WHERE s.value >=
         0.5
    RETURN r.name AS req_name

    UNION

    MATCH (r1:Requirement)-[s1:
        Similarity]-(t:Testcase) WHERE
        s1.value >= 0.5
    WITH r1, count(t) AS count WHERE
        count >= 2
    MATCH (r1:Requirement)-[s2:
        Similarity]-(r2:Requirement)
        WHERE s2.value >= 0.7
    RETURN r2.name AS req_name
}
RETURN req_name
ORDER BY req_name
```

SQL query 6

```
(SELECT req.name AS req_name
FROM edge_req_to_test edge
JOIN node_req req
ON edge.from_id = req.id
JOIN node_test test
ON edge.to_id = test.id
WHERE edge.similarity >= 0.5
GROUP BY req.id
HAVING count(*) >= 1

UNION

SELECT req.name AS req_name FROM (
    SELECT id1, id2 FROM (
        SELECT edge.from_id AS id1,
            edge.to_id AS id2
        FROM edge_req_to_req edge
        WHERE edge.similarity >= 0.7

        UNION

        SELECT edge.to_id AS id1, edge.
            from_id AS id2
        FROM edge_req_to_req edge
        WHERE edge.similarity >= 0.7
    ) AS rr
    JOIN edge_req_to_test rt
    ON rr.id2 = rt.from_id
    JOIN node_test test
    ON rt.to_id = test.id
    WHERE rt.similarity >= 0.5
    GROUP BY rr.id1, rr.id2
    HAVING count(*) >= 2
) AS r
JOIN node_req req
ON r.id1 = req.id)

ORDER BY req_name;
```

Table 2 compares the Cypher and SQL queries in terms of character count. The queries in SQL are approximately 1.5 to 1.9 times longer than those in Cypher.

5. Case study

In this section, we describe the results of executing the queries shown in the previous section. For the experiment, we obtained traceability data by analyzing artifacts documented in a real-world project. The dataset is in a text format and contains 3,855 test cases and 260 requirements specifications.

The experiment was conducted on a Windows 10 PC

Table 2: Number of characters in each query

Query type	Cypher	SQL
Query 1	116	179
Query 2	415	724
Query 3	114	177
Query 4	413	795
Query 5	110	192
Query 6	325	599
Query 7	514	639

with an AMD Ryzen 5 3600 CPU and 16GB of memory. We used Neo4j graph database management system and PostgreSQL relational database management system.

To load the traceability data into these databases, we implemented Python scripts which parse given data and call database APIs to update the databases accordingly. This process required about 47 minutes for Neo4j and about 4 minutes for PostgreSQL.

Using the databases loaded with the traceability data, we measured the processing time of the queries. Table 3 shows the processing time for each query. Since Queries 1 to 4 take a test case or a requirements specification as input, we measured the processing times for all test cases or requirements specifications and averaged them.

For Neo4j, the longest execution time was observed when Query 7 was executed. This query obtains the list of specifications that are not tested at all. Considering the fact that the query has to exhaustively check all requirements specifications, the processing time, which was about 13 seconds, is sufficiently permissible. In addition, Queries 1 to 4, which are queries concerning a specific requirements specification or test case, were all executed in less than 0.6 seconds. For PostgreSQL, on the other hand, the longest execution time was observed when Query 4 was executed; but the time was only approximately 0.1 seconds. PoestgreSQL exhibited shorter processing time than Neo4j for all cases.

Table 3: Execution time (seconds)

Query type	Neo4j	PostgreSQL
Query 1	0.004	0.056
Query 2	0.054	0.075
Query 3	0.002	0.054
Query 4	0.047	0.125
Query 5	0.359	0.048
Query 6	5.086	0.098
Query 7	13.440	0.098

6. Related work

Studies that consider the applicability of graph queries in the area of software engineering include [4, 5, 6]. In reference [4], four query languages including SQL and Cypher were compared for test case traceability queries. The results show that Cypher is superior in terms of expressiveness and understandability. Although their work and ours both concern the applicability of graph queries, the sorts of data and the purposes of using the queries are significantly different. For example, [4] considers traversal of traceability links whereas ours concerns the coverage of requirements specifications by test cases. Reference [5] uses a network to represent the traceability links between requirements, code, and test cases and compares the conciseness of the SQL and Cypher representations of two types of simple queries. Their work also considers traversal of trasability links; they did not deal with the kind of coverage analysis we did. Reference [6] analyzed the performance of query processing for large-scale software artifact data: it is shown that querying a relational database running on Spark, which is a cluster computing framework, with SQL is more efficient than using Neo4j and Cypher. On the other hand, in this paper, we showed that even querying Neo4j running on a single computer exhibited sufficiently practical processing time for a real-world data set. The result does not conflict with ours, where PostgreSQL exhibited better performance than Neo4j on a single computer; but our results also show that graph queries can be executed sufficiently fast for a real-world data set.

In [3, 7] we developed an approach to automatically find traceability links between test cases and requirements specifications. This approach first estimates the similarity degree between two artifacts using natural language processing techniques and then infers the existence of traceability links using the estimates. The dataset used in the case study of this paper was obtained using the first step of this approach. The problem of measuring similarity or relevance between software artifacts have also been studied elsewhere, especially in the context of automatic construction of traceability links between artifacts. Examples of this line of studies include, for example, [8, 9, 2, 10, 11].

7. Threats to Validity

A major internal threat to validity stems from the expressiveness of Cypher and SQL. In general queries in these languages have different representations; thus the queries we presented here might have more intuitive or concise alternatives. Another threat lies in how to compare the conciseness of the queries. In this paper we measured the conciseness in terms of character count; but another measure, for example, the number of tokens, might be more appropriate.

An external threat of validity concerns the representativeness of the data we used in the case study. Although the data was obtained from one of the largest projects lead by our industrial partner, there should be projects that need to manage data of larger size. Other characteristics of data, especially, the distribution of similarities can also vary from projects to projects. In view of these, a care should be taken when generalizing the findings about query processing performance.

8. Conclusion

In this paper we discussed the applicability of graph databases and graph queries to coverage analysis between test cases and requirements specifications. We showed that traceability data can be represented as a graph database and that coverage information can be easily retried from the database with graph queries. We also demonstrated that the graph queries, which are written in the Cypher query language, are often more concise than those in SQL, while the processing performance is comparable between the graph database and the SQL database.

Future research include many possible directions. The case study of this paper concerned a single product. Data sets from other systems, particularly larger ones, should be considered in future. Extending the list of queries for coverage analysis also deserves further research. To this end we plan to interview developers from industry to find out other queries that are useful in practice.

Acknowledgment This work was supported by JSPS KAKENHI Grant Number JP20K11747.

References

[1] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*, 2nd ed. O'Reilly Media, Inc., 2015.

[2] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best practices for automated traceability," *Computer*, vol. 40, no. 6, pp. 27–35, 2007.

[3] H. Nakagawa, T. Hasegawa, S. Matsui, and T. Tsuchiya, "Visualization of specification coverage: A case study of a web application development in industry," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 77–80.

[4] M. Rath, D. Akehurst, C. Borowski, and P. Mäder, "Are graph query languages applicable for requirements traceability analysis?" in *Joint Proceedings of REFSQ-2017 Workshops, Doctoral Symposium, Research Method Track, and Poster Track*, ser. CEUR Workshop Proceedings, vol. 1796, 2017, p. (unassigned).

[5] R. Elamin and R. Osman, "Implementing traceability repositories as graph databases for software quality improvement," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 269–276.

[6] J. Lin, Y. Liu, and J. Cleland-Huang, "Supporting program comprehension through fast query response in large-scale systems," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 285–295.

[7] H. Nakagawa, S. Matsui, and T. Tsuchiya, "A visualization of specification coverage based on document similarity," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. IEEE Press, 2017, p. 136–138. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.117

[8] F. Erata, M. Challenger, B. Tekinerdogan, A. Monceaux, E. Tüzün, and G. Kardas, "Tarski: A platform for automated analysis of dynamically configurable traceability semantics," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1607–1614. [Online]. Available: https://doi.org/10.1145/3019612.3019747

[9] A. Goknil, I. Kurtev, and K. Van Den Berg, "Generation and validation of traces between requirements and architecture based on formal trace semantics," *Journal of Systems and Software*, vol. 88, pp. 112 – 137, 2014.

[10] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 95–104.

[11] C. Mills, J. Escobar-Avila, and S. Haiduc, "Automatic traceability maintenance via machine learning classification," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 369–380.