Finding Logic Bugs in Graph-processing Systems via *Graph-cutting*

Qiuyang Mang [1] Jinsheng Ba [2] Pinjia He [1] Manuel Rigger [3]

[1]The Chinese University of Hong Kong, Shenzhen [2]ETH Zurich [3]National University of Singapore

Overview

Abstract Graph-processing systems, including Graph Database Management Systems (GDBMSes) and graph libraries, are designed to analyze and manage graph data efficiently. They are widely used in applications such as social networks, recommendation systems, and fraud detection. However, logic bugs in these systems can lead to incorrect results, compromising the reliability of applications. While recent research has explored testing techniques specialized for GDBMSes, it is unclear how to adapt them to graph-processing systems in general. This paper proposes GRAPH-CUTTING, a universal approach for detecting logic bugs in both GDBMSes and various algorithms in graph libraries. Our key idea is inspired by the observation that certain graph patterns are critical for various graph-processing tasks. Dividing graph data into subgraphs that preserve those patterns establishes a natural relationship between query results on the original graph and its subgraphs, allowing for the detection of logic bugs when this relationship is violated. We implemented GRAPH-CUTTING as a tool, GSLICER, and evaluated it on 3 popular graph-processing systems, NetworkX, Neo4j, and Kùzu. GSLICER detected 39 unique and previously unknown bugs, out of which 34 have been fixed and confirmed by developers. At least 8 logic bugs detected by GSLICER cannot be detected by baseline strategies. Additionally, by leveraging just a few concrete relationships, GRAPH-CUTTING can cover over 100 APIs in NetworkX. We expect this technique to be widely applicable and that it can be used to improve the quality of graph-processing systems broadly.

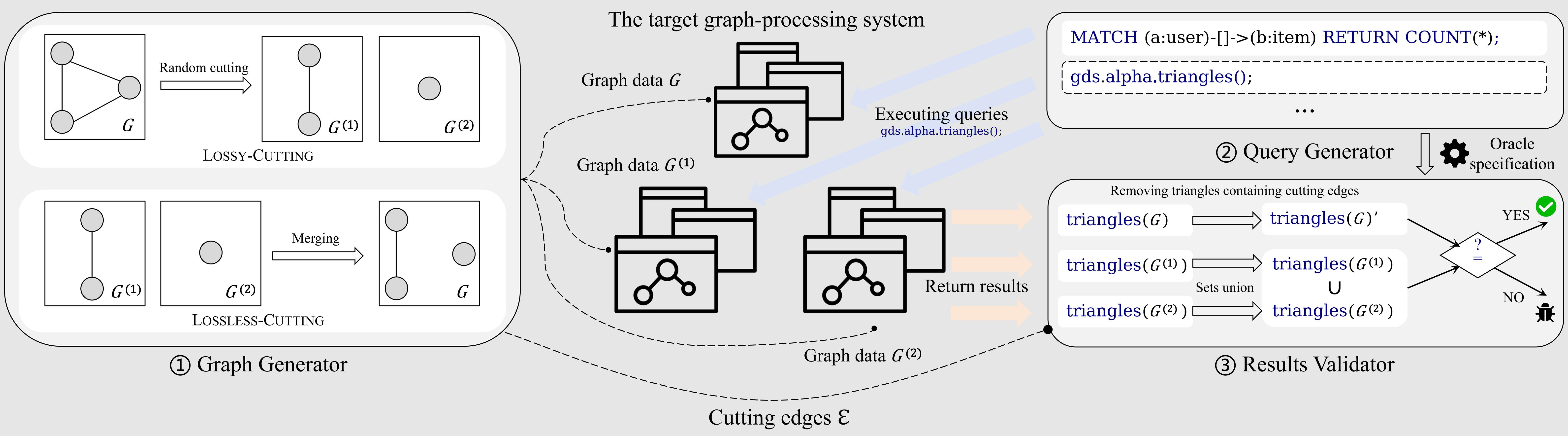
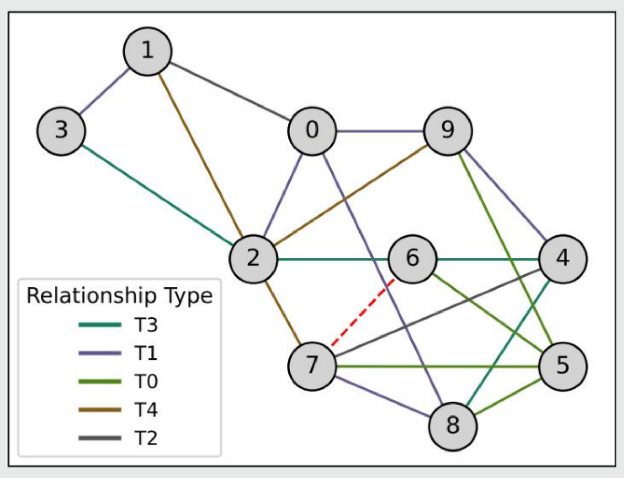


Figure 1. An overview of GSLICER

A motivated example



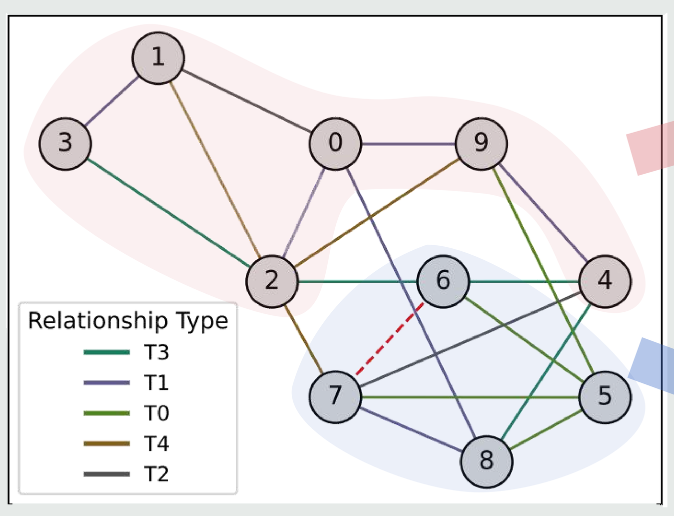

```
gds.alpha.triangles() --{
    line  nodeA  nodeB  nodeC
    0     0      1      2
               ......
    8     5      6      7 🐞
               ......
    11    5      7      8
}
```

Figure 2. A motivated logic bug: incorrect triangle listing in Neo4j when the input graph contains multiple relation types.

Challenges in automatically detecting such bugs in graph-processing systems

1. ***Effectiveness***. Existing methods rely on *query-mutation*, but they are not feasible for testing such functions due to the fixed input formats.
2. ***Generalizability***. Requires a general test oracle that can cover multiple functionalities with minimal engineering effort.

We addressed these challenges by mutating the *data* rather than the *query*.




```
gds.alpha.triangles() --{
    line  nodeA  nodeB  nodeC
    0     0      1      2
    1     1      2      3
    2     0      2      9
}   Missing the triangle of {5, 6, 7}
gds.alpha.triangles() --{
    line  nodeA  nodeB  nodeC
    0     5      7      8
}
```

Figure 3. An illustrative example of how GRAPH-CUTTING identifies the logic bug in Neo4j's triangle listing involves dividing the nodes into two sets: $\{0, 1, 2, 3, 4, 9\}$ and $\{5, 6, 7, 8\}$. The triangle $(5, 6, 7)$ in Figure 2 contains none of the cutting edges but is missing in both subgraph results, indicating a logic bug.

Scan the QR code to see the **bug report** ☺

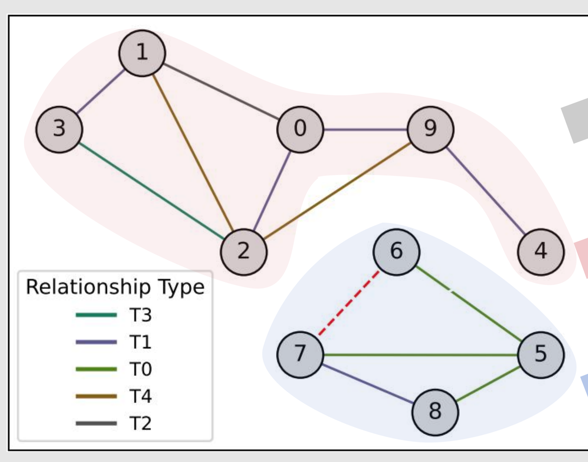
Behind the example

Given a graph $G = (V, E)$, we divide the vertices in G into two disjoint sets, $V^{(1)}$ and $V^{(2)}$. The corresponding induced subgraphs are $G^{(1)} = (V^{(1)}, E^{(1)})$ and $G^{(2)} = (V^{(2)}, E^{(2)})$. We then define the set of cutting edges, which belong to neither of the two subgraphs, as $\mathcal{E}$.

Theorem 1 (Graph-Cutting for triangle listing). For any graph G and its division $G^{(1)}, G^{(2)}, \mathcal{E}$, let $\texttt{Triangles}(G)$ be the set of triangles (aka 3-cliques) in G We have

$$\texttt{Triangles}(G) = \texttt{Triangles}(G^{(1)}) \cup \texttt{Triangles}(G^{(2)}) \setminus \text{triangles containing edges in } \mathcal{E}.$$

Deliberately dividing the graph without any cutting edges (*i.e.*, $\mathcal{E} = \emptyset$) if we cannot infer the last term from the results, *e.g.*, testing triangle counting.




```
MATCH (a)-[]-(b)-[]-(c) RETURN
COUNT(*); --{COUNT(*): 5 🐞}

MATCH (a)-[]-(b)-[]-(c) RETURN
COUNT(*); --{COUNT(*): 3 ✓}

MATCH (a)-[]-(b)-[]-(c) RETURN
COUNT(*); --{COUNT(*): 1 ✓}
```

Generalization

1. Triangle Listing ⇒ Graph Pattern Matching ⇒ GDBMS Query
2. Testing graph libraries (*e.g.*, NetworkX) through automatic enumeration of graph-cutting oracles or manual design with minimal engineering effort.

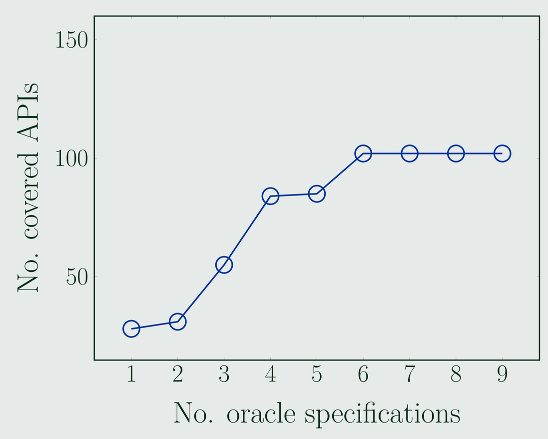


Figure 4. The relationship between the number of GRAPH-CUTTING oracles and the number of APIs in NetworkX covered by them.

Results

Table 1. Bugs status in graph-processing systems.

Graph-processing system	Fixed	Confirmed	Duplicate	Unconfirmed	Sum
Neo4j	3	0	1	0	4
NetworkX	10	12	1	2	25
Kùzu	5	4	1	0	10
Sum	18	16	3	2	39

Table 2. Classification of fixed or confirmed bugs.

Graph-processing system	Logic bugs	Crashes/Exceptions	Hang bugs	Sum
Neo4j	2	1	0	3
NetworkX	4	18	0	22
Kùzu	5	3	1	9
Sum	11	22	1	34

Bug example

```
CREATE NODES, RELS from tinysnb dataset
MATCH (a:person)-[:knows]->(b:person),
(b)-[:knows]->(a), (a)-[:knows]->(b)
RETURN COUNT(*) --{16} 🐞 --{12} ✓
```

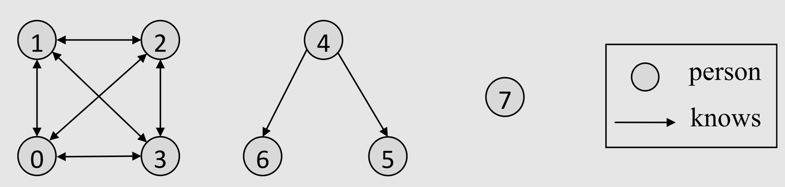


Figure 5. Incorrect pattern matching in Kùzu caused by erroneous planning of Worst-Case-Optimal Join

Scan the QR code to see the **PR** ☺

Conclusion

1. `Graph-cutting` is a **general black-box** testing method for various graph-processing tasks. It is simple, intuitive, and easy to apply.
2. `Graph-cutting` is **complementary** to existing query-mutation approaches by executing the same query over different graph structures.
3. `Graph-cutting` can be applied to test both GDBMS **queries** and **APIs**.

Code Bug List

Trustworthy Engineering of Software Technologies Lab

 qiuyangmang@link.cuhk.edu.cn