

# Anomaly Detection in AI Knowledge Graphs: A Discrete Mathematical Framework Against Graph Poisoning

Made Branenda Jordhy - 13524026

Informatics Engineering Study Program

School of Electrical Engineering and Informatics

Bandung Institute of Technology, 10 Ganesha Street, Bandung

E-mail: [ethgalleryin@gmail.com](mailto:ethgalleryin@gmail.com) , [13524026@std.stei.itb.ac.id](mailto:13524026@std.stei.itb.ac.id)

**Abstract**—Knowledge graphs (KGs) are increasingly used in AI systems to organize factual information for reasoning and decision-making. However, their growing size and open nature make them vulnerable to a kind of attack known as *graph poisoning*, where malicious nodes or edges are added to distort meaning or mislead AI outputs. In this paper, we introduce a lightweight and interpretable framework for detecting such structural anomalies using discrete mathematics—particularly graph-theoretic concepts like node degree, clustering, and cut-points. Instead of relying on machine learning, our approach focuses on analyzing the graph's shape and connection patterns to flag suspicious entities. We tested the method on a synthetic knowledge graph with injected anomalies and found that even basic structural features can reveal hidden manipulations. The goal here isn't to replace ML-based methods, but to provide a simpler, explainable pre-processing step that can help safeguard the integrity of AI systems built on top of knowledge graphs.

**Keywords**—component; knowledge graphs; anomaly detection, graph poisoning, discrete mathematics; graph theory; AI security

## I. INTRODUCTION

### A. Background

Knowledge graphs (KGs) are widely used in today's AI systems to represent structured information about entities and the relationships between them. By organizing facts into a graph format, these systems can support advanced reasoning, improve search accuracy, and enable more intelligent decision-making. From search engines and personal assistants to biomedical and recommendation systems, KGs have become a core part of many modern applications.

However, just like any other system, knowledge graphs are not immune to security threats. One serious issue is graph poisoning, where attackers intentionally add, remove, or alter parts of the graph to distort its meaning. This could result in misleading conclusions, hidden misinformation, or failure of downstream AI models that rely on the graph's structure. Due to the interconnected nature of graphs, even a small

manipulation can propagate widely and cause significant impact.

### B. Research Objectives

Many existing approaches to detecting graph poisoning rely on machine learning models or embedding techniques. While effective in some cases, these methods often act as black boxes, they require large amounts of labelled data, and their outputs can be difficult to interpret or justify.

This paper takes a different approach. The goal is to develop a lightweight, interpretable framework for detecting structural anomalies in knowledge graphs using techniques from discrete mathematics and graph theory. By analysing features such as node degree, connectivity patterns, local clustering, and graph motifs, the system can flag suspicious nodes or edges without needing prior training data.

To demonstrate the feasibility of this approach, we simulate poisoning scenarios in example knowledge graphs and evaluate whether the proposed methods can detect anomalies effectively. The results show that even simple structural features can reveal hidden manipulations and serve as an early defence mechanism against graph-based attacks in AI systems.

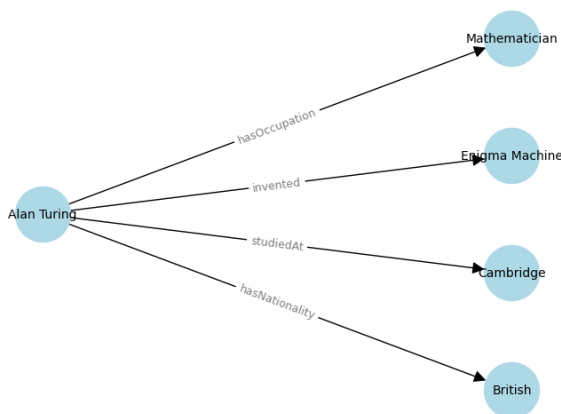
## II. THEORITICAL BACKGROUND

### A. Graphs and Knowledge Representation

Graphs are fundamental structures in mathematics and computer science that consist of nodes (vertices) and edges (connection between nodes). When used in Artificial Intelligence, graphs are powerful tools for modeling relationship between entities in the real world.

A knowledge graph (KG) is a special type of graph that represents information in the form of facts. These facts are typically stored as triples: subject, predicate, and object. For example, the fact "Alan Turing is a mathematician" would be written as (Alan Turing, hasOccupation, Mathematician). In graph form, this translates to a directed edge labeled "hasOccupation" pointing from the node "Alan Turing" to the node "Mathematician".

Most-large scale knowledge graphs, such as Wikidata or Google’s Knowledge Vault, are dynamic in nature. They are continuously updated and expanded with information from various sources, automated, human-curated, or even crowdsourced. This makes them flexible, but also vulnerable to inconsistencies and manipulation.



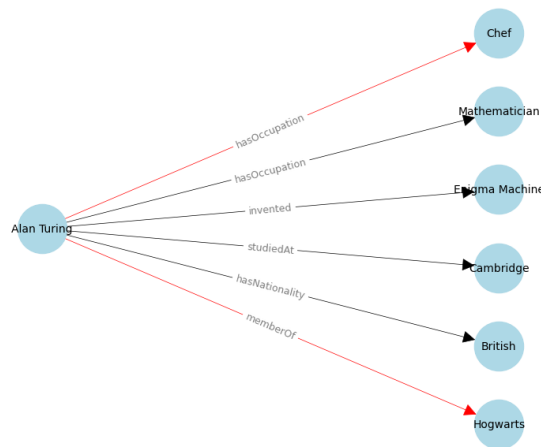
**Figure 1.** A simple knowledge graph representing facts as labeled directed edges. Each edge connects two entities with a specific relationship. (source: Author)

### B. Graph Poisoning Attacks

As knowledge graphs grow larges and are used in critical AI pipelines, they become more attractive targets for attackers. One of the main threats is *graph poisoning*, where the structure of the graph is deliberately manipulated. Attackers may insert fake relationship, remove important connections, or alter existing facts to distort downstream reasoning or predictions.

There are different types of poisoning attacks:

- **Injection attacks**, where fake nodes ad edges are added to insert misleading information.
- **Deletion attacks**, where key nodes or connections are removed to break context or isolate data.
- **Semantic modification**, where labels or properties are changed to cause misinterpretation (e.g., changing “hasOccupation” to “hasInterest”).



**Figure 2.** Illustration of a knowledge graph after poisoning. Malicious edges are injected to mislead inference or corrupt downstream reasoning. (source: Author)

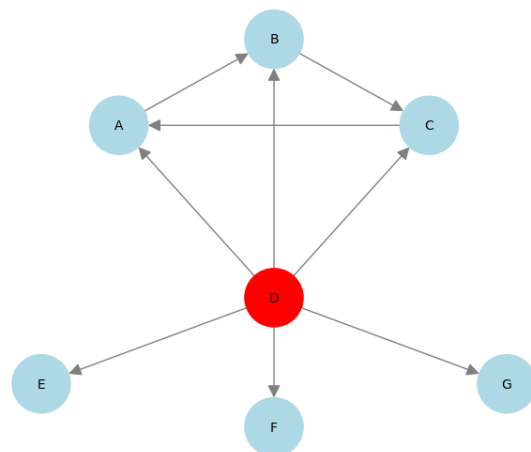
These changes can be subtle but impactful. A single edge injection in the right place can shift the behavior of recommendations systems, ranking algorithms, or fact-checking engines. What makes this problem harder is that poisoned graphs often still look structurally valid, anomalies hide in plain sight.

### C. Discrete Anomaly Detection Techniques

To identify poisoned elements in a knowledge graph, one effective approach is to analyze its structure using techniques from discrete mathematics, especially graph theory. Instead of relying on complex machine learning models, we can use simple graph-based features to detect unusual patterns.

Some common techniques include:

- **Node degree analysis** – detecting nodes that have too many or too few connections compared to others.

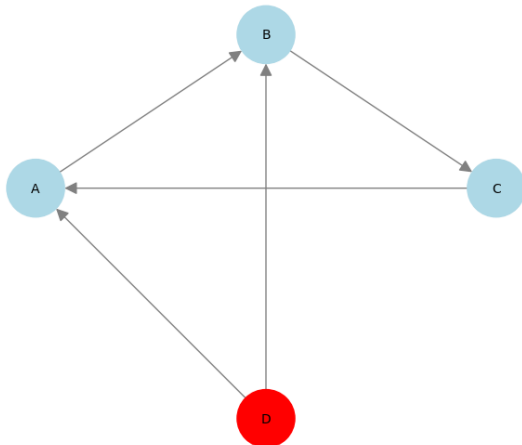


**Figure 3.** Node degree anomaly detection. The red node exhibits an unusually high number of connections compared to other nodes, which may indicate an injected

hub, spamming entity, or corrupted node in a knowledge graph. (source: Author)

Nodes with significantly higher or lower degree than their neighbors can be detected as structural outliers. In this example, node D connects to most of the graph despite being unrelated to the main triangle, indicating a potential anomaly due to manipulation or injection.

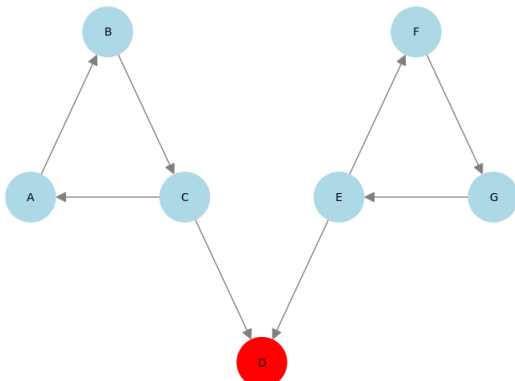
- **Clustering coefficient** – measuring how connected a node's neighbors are to one another.



**Figure 4.** Clustering coefficient anomaly detection. The red node connects to two densely linked nodes without forming a triangle, resulting in a low local clustering coefficient that may signal structural inconsistency. (source: Author)

Nodes embedded in dense subgraphs typically exhibit high clustering. When a node connects to clustered nodes but fails to form connections among its neighbors, it may indicate improper linkage or injected outlier nodes [4].

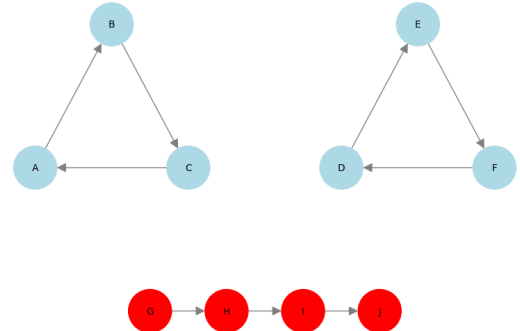
- **Betweenness centrality** – identifying nodes that sit on many shortest paths and might be artificially important.



**Figure 5.** Betweenness centrality anomaly detection. The red node lies on many shortest paths between two dense regions, giving it high centrality and influence over graph flow despite a small degree. (source: Author)

Nodes with high betweenness can control or disrupt communication between communities. Such nodes may be critical or vulnerable points in knowledge graphs or social networks.

- **Rare motif detection** – spotting subgraph patterns that are unusual or don't normally exist in a healthy graph.



**Figure 6.** Rare motif detection. The red nodes form an uncommon linear substructure that deviates from typical dense or cyclic motifs, suggesting potential outlier behavior or injected structure. (source: Author)

Graphs often contain recurring structural patterns. The presence of rare or unfamiliar motifs may point to synthetic, corrupted, or misclassified data within the system.

- **Cut-point detection** – finding nodes whose removal would disconnect the graph. The red node shown on **Figure 5** acts as a single point of connection between two communities. Its removal would disconnect the graph, making it a structurally critical node. Cut-point often represent bridge, gatekeepers, or critical connections. Detecting such nodes is vital in assessing the robustness and vulnerability of graph-based system.

These techniques are lightweight, interpretable, and scalable. They work well even when no training data is available, making them ideal for real-time anomaly detection in growing knowledge graphs.

### III. PROBLEM STATEMENT

#### A. Motivation and Context

Knowledge graphs are a powerful way to represent facts and relationships in a structured format. They're used in all sorts of AI systems, from search engines to recommendation platforms, to help machines reason more effectively. As these graphs grow larger and more widely used, the risks that come with them also increase.

One of the key threats is graph poisoning. This happens when someone intentionally adds, removes or changes parts of the graph to mislead the system. These manipulations are often crafted to blend in with valid data. For instance, a fact like (*Alan Turing, hasOccupation, Mathematician*) could be quietly

accompanied by a misleading one like (*Alan Turing, hasOccupation, Chef*). Structurally, it looks fine, it connects two nodes using a familiar relation, but semantically, it introduces noise or even disinformation. Since most systems check only the structure, such poisoned links may go unnoticed.

Existing solutions often rely on machine learning or embedding-based techniques to detect anomalies. While powerful, these approaches typically require large, labeled datasets and often lack transparency. It's difficult to interpret why a certain node or edge is flagged, which becomes a problem in sensitive or safety-critical applications.

This paper explores a different path. Instead of learning from examples, it focuses on lightweight, interpretable techniques from discrete mathematics, especially graph theory. By studying basic graph features, like node degree, clustering patterns, centrality, or structural roles, we can identify suspicious elements based purely on structure. These methods don't require training data and produce results that are easier to understand and justify.

As shown in **Figure 2**, even a single poisoned edge can subtly distort a knowledge graph while escaping detection. Detecting this kind of anomaly requires more than just checking surface-level validity, it calls for deeper structural analysis. This motivates the need for practical detection tools grounded in graph structure itself, rather than relying on opaque black-box models.

### B. Problems Definition

The main problem addressed in this paper is how to detect structural anomalies in a knowledge graph that may indicate poisoning. Specifically, we aim to identify nodes or edges whose presence deviates from common patterns or weakens the semantic reliability of the graph, not based on contents or label, but purely from how the graph is shaped. We define the problem as follows:

Given a directed knowledge graph

$$G = (V, E)$$

where  $V$  is a set of entities and  $E \subseteq V \times R \times V$  is a set of labeled edges representing relationships, identify suspicious nodes or subgraphs based on structural properties. These properties include:

- **Node degree** – how many connections a node has, and whether it is unusually high or low
- **Clustering behavior** – whether a node's neighbors are well-connected among themselves
- **Betweenness centrality** – whether a node appears too frequently on shortest paths
- **Motif presence** – whether small subgraph patterns are rare or unexpected
- **Cut-point roles** – whether a node critically holds two parts of the graph together

Rather than relying on training data or semantic validation, our focus is on outliers from a structural standpoint. For

example, a single node that suddenly connects to many unrelated entities could raise suspicion, even if edge labels appear syntactically valid.

We assume that the graph is static at the time of analysis and that no prior poisoning labels are available. The methods used should work on any graph topology if basic structural information is accessible.

## IV. PROPOSED SOLUTION

### A. Overview

This paper introduces a lightweight and interpretable framework for detecting structural anomalies in knowledge graphs. Instead of using machine learning or graph embeddings, the framework relies on discrete mathematical techniques rooted in graph theory. The focus is on identifying structural irregularities, such as unexpected connections, unusual centrality, or rare graph motifs, that could signal potential poisoning.

Each anomaly is detected through simple, rule-based checks that evaluate the local or global structure of the graph. The approach is modular and can be extended or combined with other features as needed, depending on the context and the graph topology.

### B. Detection Workflow

The detection process is built as a modular pipeline that analyzes a knowledge graph's structural properties to uncover potential anomalies. The goal is to spot unusual behaviors, such as sudden connectivity spikes or structural bottlenecks, without relying on node labels or semantic interpretation.

#### 1) Graph Input & Construction

The system accepts a knowledge graph in the form of a directed labeled graph. This can be manually constructed (e.g., using Python dictionaries or NetworkX), or imported from external formats such as .ttl, .json, or triple-based CSV. Toy example, a graph is constructed with facts such as:

(Alan Turing, hasOccupation, Mathematician)  
(Alan Turing, studiedAt, Cambridge)  
(Alan Turing, invented, Enigma Machine)

A poisoned edge is then injected  
(Alan Turing, hasOccupation, Chef)

#### 2) Structural Feature Extraction

For each node and edge, structural metrics are extracted using standard graph-theoretic functions:

- Node degree (in-degree and out-degree)
- Clustering coefficient
- Betweenness centrality
- Motif frequency analysis
- Cut-point (articulation point) detection

These metrics form the basis for structural anomaly detection

### 3) Rule-Based Evaluation

Each metric is tested against heuristics or threshold rules. For example:

- A node with out-degree significantly higher than the graph's mean may be flagged as a hub anomaly
- A node with low clustering in a dense neighborhood may be a suspicious outlier
- Nodes with high betweenness or functioning as cut-points may indicate synthetic bridges

In our toy example, Alan Turing initially has a modest out-degree of 3. After the poisoned edge (Alan Turing → Chef) is added, his out-degree spikes. If this deviates beyond the normal range, the node is flagged. Each flagged element is annotated with the reason for detection and stored for later reporting.

### 4) Visualization

Anomalous nodes and edges are visualized using layout algorithms (e.g., `spring_layout`) with colored nodes (red for flagged anomalies), labeled edges, stylized arrows or dashed lines for suspicious links. These visualizations help in verifying the structural outliers and in presenting results clearly.

### 5) Output Summary

Finally, the system outputs:

- A list of flagged nodes and/or edges
- An explanation per flag (e.g., “high degree anomaly”, “rare motif”, “cut-point node”)
- A saved visualization or adjacency representation

This output can be reviewed by a human operator or fed into further pipeline components for mitigation or logging.

## V. IMPLEMENTATION AND EVALUATION

### A. Tools and Environment

The anomaly detection framework was implemented using Python 3.10, a versatile and widely adopted programming language for both academic research and production systems. The codebase was tested and run on a macOS machine (Apple M4 chip, 16 GB RAM), ensuring it performs well even on consumer-grade hardware. To ensure portability and reproducibility, all components were also tested on Google Colab, which allows for broader access and ensures consistent results across platforms.

- **NetworkX** (v3.2.1): Core library used to construct and analyze graph data structures. It supports operations like degree analysis, clustering, centrality measures, and articulation point detection, all crucial for rule-based anomaly detection.

- **Matplotlib**: Used to generate visual representations of the knowledge graph. This includes color-coded nodes and annotated edges, making it easier for humans to interpret anomalies.
- **Statistic (built-in) / NumPy**: Applied to compute key statistical values such as the mean and standard deviation, enabling us to define dynamic anomaly thresholds based on distribution rather than fixed heuristics.

Notably, the entire system does not rely on GPU acceleration, nor does it require large-scale machine learning libraries. This ensures that the solution is both lightweight and deployable on resource-constrained environments, a core design goal that favors transparency and explainability over complexity.

### B. Graph Construction and Dataset

To evaluate the framework, we designed a synthetic yet realistic knowledge graph that spans multiple semantic domains. At the core of this graph are historical scientific figures such as Alan Turing, Grace Hopper, Ada Lovelace, Charles Darwin, Gregor Mendel, and Rosalind Franklin. These entities are connected through accurate relationships like “hasOccupation”, “collaboratedWith”, and “studiedAt”.

The graph is composed of 25+ nodes and 30+ edges, and structured in three semantic clusters:

- Computer Science and Mathematics
- Biological and Genetic Sciences
- Abstract / Fictional Entities (used to simulate poison)

The initial “clean” graph encodes factual relationships:

```
# Valid facts
valid_edges = [
    # Cluster A
    ("Alan Turing", "Mathematician", "hasOccupation"),
    ("Alan Turing", "Enigma Machine", "invented"),
    ("Alan Turing", "Cambridge", "studiedAt"),
    ("Grace Hopper", "Compiler", "developed"),
    ("Grace Hopper", "Mathematician", "hasOccupation"),
    ("Alan Turing", "Grace Hopper", "collaboratedWith"),
    ("Ada Lovelace", "Algorithm", "pioneered"),
    ("Ada Lovelace", "Mathematician", "hasOccupation"),
    ("Ada Lovelace", "Charles Babbage", "collaboratedWith"),
    # Cluster B
    ("Charles Darwin", "Evolution", "proposed"),
    ("Gregor Mendel", "Genetics", "pioneerOf"),
    ("Gregor Mendel", "Charles Darwin", "inspired"),
    ("Gregor Mendel", "Monk", "hasOccupation"),
    ("Rosalind Franklin", "DNA Structure", "discovered"),
    ("James Watson", "DNA Structure", "collaboratedOn"),
    ("Francis Crick", "DNA Structure", "collaboratedOn"),
    ("James Watson", "Francis Crick", "collaboratedWith"),
    # Bridges
    ("Alan Turing", "Charles Darwin", "studiedIdeasOf"),
    ("Grace Hopper", "Rosalind Franklin", "collaboratedWith")
]
```

To emulate graph poisoning, we deliberately injected 10 syntactically valid but semantically incorrect edges. Examples include (“Alan Turing”, “Chef”, “hasOccupation”), (“Rosalind



Franklin”, “Sith Lord”, “became”) and (“Atlantis”, “Wizard”, “inhabitedBy”).

```
# Poisoned facts
poisoned_edges = [
    ("Alan Turing", "Chef", "hasOccupation"),
    ("Grace Hopper", "Wizard", "hasOccupation"),
    ("Charles Darwin", "Middle Earth", "discovered"),
    ("Alan Turing", "Atlantis", "visited"),
    ("Atlantis", "Wizard", "inhabitedBy"),
    ("Middle Earth", "Hogwarts", "connectedTo"),
    ("Hogwarts", "Charles Darwin", "graduated"),
    ("Ada Lovelace", "Dragon Slayer", "hasOccupation"),
    ("Gregor Mendel", "Time Traveler", "isA"),
    ("Rosalind Franklin", "Sith Lord", "became")
]
```

These false links do not break the graph’s syntactic consistency, and in RDF-style triple stores would pass validation, yet they distort the knowledge graph semantically. This poisoning structure allows us to test whether rule based methods relying purely on topological features can successfully flag suspicious patterns without needing labeled data or supervision.

### C. Detection Techniques and Their Justification

The system integrates five structural anomaly detection techniques. The choice of methods reflects a balance between simplicity, explainability, and practical effectiveness [3].

- Node Degree Detection

Why it matters: Nodes that have been injected with multiple poisoned facts tend to have abnormally high out-degree. For instance, a node that originally had 2-3 valid relationships may suddenly have 6 or more due to false attachments.

```
# Node degree detection
def detect_high_degree_nodes(G, threshold_std=1.5):
    degrees = [G.out_degree(n) for n in G.nodes()]
    avg, dev = mean(degrees), stdev(degrees)
    return [(n, "High out-degree anomaly") for n in G.nodes() if G.out_degree(n) > avg + threshold_std * dev]
```

Implementation: We compute the Z-Score of each node’s out-degree and flag those above a threshold (e.g., +1.5 standard deviations). This catches degree inflation while tolerating natural variance in hubs like “Mathematician”.

- Clustering Coefficient Anomaly

Why it matters: Legitimate knowledge graphs tend to exhibit local clustering (e.g., coworkers, academic collaborators). Poisoned nodes often attach to isolated concepts or fictional entities, leading to a drop in clustering.

```
# Clustering coefficient anomaly
def detect_low_clustering(G, threshold=0.1):
    undirected = G.to_undirected()
    cc = nx.clustering(undirected)
    return [(n, "Low clustering anomaly") for n in cc if cc[n] < threshold and len(list(undirected.neighbors(n))) >= 2]
```

Critical Reflection: Clustering coefficient alone is a weak signal in sparse graphs. However, in entity

centric subgraphs (e.g., person, organization), it becomes much more expressive.

- Betweenness Centrality Check

Why it matters: A poisoned node might artificially bridge unrelated subgraphs. For example, connecting Charles Darwin to “Hogwarts”. Betweenness centrality identifies such nodes by measuring how often they appear on shortest paths.

```
# Betweenness Centrality Check
def detect_high_betweenness(G, top_percent=0.1):
    centrality = nx.betweenness_centrality(G)
    sorted_nodes = sorted(centrality.items(), key=lambda x: x[1], reverse=True)
    k = max(1, int(top_percent * len(G)))
    return [(n, "High betweenness anomaly") for n, _ in sorted_nodes[:k]]
```

Design Choice: We use percentile thresholding instead of fixed value, making the method adaptable to graphs of varying sizes.

- Cut-Point Detection

Why it matters: In graph security, cut-points can serve as failure points or injection gateways. Detecting them highlights weak spots, even if not directly manipulated.

```
# Cut-Point Detection
def detect_cutpoints(G):
    return [(n, "Cut-point node") for n in nx.articulation_points(G.to_undirected())]
```

Use Case: Flagging these can expose structural bottlenecks or high risk entry points.

- Rare Motif Detection

Why it matters: In real-world attacks, false facts are often added as leaf nodes. When too many leaves attach to a single node, they form a star motif, suspicious if unmatched in the rest of the graph.

```
# Rare Motif Detection
def detect_star_motif(G, min_children=3):
    flagged = []
    for n in G.nodes():
        if G.out_degree(n) >= min_children:
            children = list(G.successors(n))
            if all(G.out_degree(c) == 0 for c in children):
                flagged.append((n, "Suspicious star motif"))
    return flagged
```

Trigger: We flag nodes with 3+ leaf children that themselves have no further edges.

### D. Integration and Execution

Each detection technique is implemented as a pure function over the graph object. The system aggregates outputs from all modules, applies deduplication, and returns a unified report where each node may have multiple anomaly tags.

- Extensible: New rules can be plugged in with minimal code change.
- Traceable: Each flagged node is justified by specific topological reasoning.

- Portable: The system requires no training or fine-tuning and runs in seconds commodity hardware.

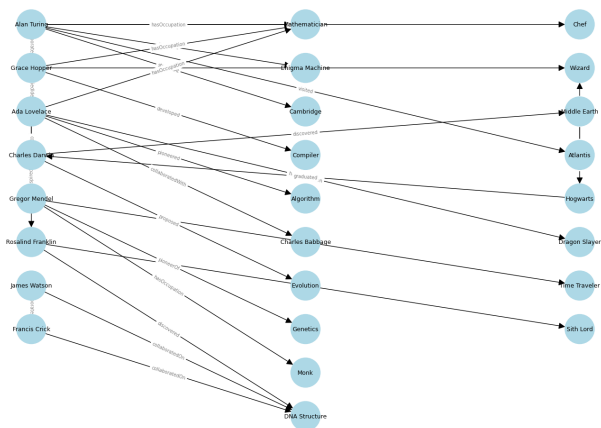
### E. Visual Interpretation and Output

To complement the detection process, this framework provides interpretable visualizations that allow analysts to quickly identify suspicious patterns without needing to inspect structural metrics manually. Nodes flagged as anomalous are colored red, immediately drawing attention to potential manipulations or misinformation. Additionally, all edges are labeled semantically (e.g., hasOccupation, collaboratedWith, developed), making it easier to trace how entities are connected and whether their relationships fall within a reasonable domain of knowledge.

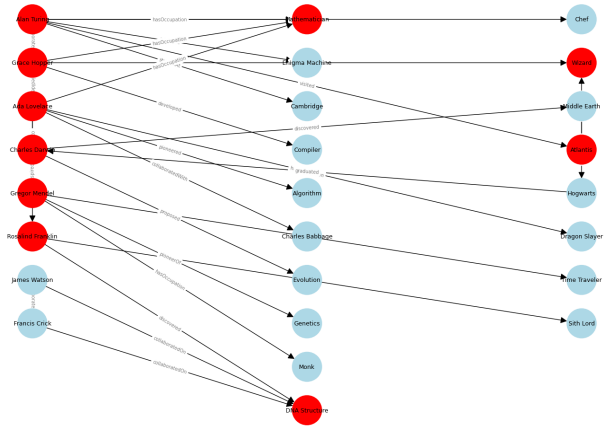
In our expanded test case, which now includes multiple semantic clusters such as historical scientists, computer scientist, and biological researchers, anomalies arise not only from a single entity (e.g., Alan Turing) but also from multiple high-profile nodes. These include:

- Alan Turing flagged due to a surge in out-degree caused by links to “Chef” and “Atlantis”.
- Rosalind Franklin flagged for the poisoned connection to “Sith Lord”
- Grace Hopper flagged due to an inflated neighborhood from injected relationships with “Wizard”.
- And several fictional or disconnected nodes with low clustering and no legitimate context.

These poisoned links result in various structural red flags, from degree inflation and star motifs to bridging anomalies and cut-points, which are all captured by our lightweight detectors. Compared side by side, the **Figure 7** presents the graph before any analysis, all nodes are treated equally. In **Figure 8**, the anomalies are visually highlighted, offering an immediate shift in interpretability. This visual feedback supports human in the loop auditing, which is essential for decision support systems, especially in domains where transparency and explainability are prioritized over raw predictive power.



**Figure 7.** Poisoned Knowledge Graph Before Detection (source: Author)



**Figure 8.** Poisoned Knowledge Graph After Detection: Multiple Nodes Flagged as Structurally Anomalous. (source: Author)

	Node	Anomaly Type
0	Alan Turing	High out-degree anomaly
1	Grace Hopper	High out-degree anomaly
2	Ada Lovelace	High out-degree anomaly
3	Gregor Mendel	High out-degree anomaly
4	Rosalind Franklin	Low clustering anomaly
5	Wizard	Low clustering anomaly
6	Atlantis	Low clustering anomaly
7	Charles Darwin	High betweenness anomaly
8	DNA Structure	Cut-point node
9	Mathematician	Cut-point node

**Figure 9.** Anomaly Detection Result Table. (source: Author)

The **Figure 9** summarizes all flagged nodes based on the five detection rules applied. Nodes such as Alan Turing, Grace Hoper, and Ada Lovelace were identified as having unusually high out-degree, likely due to injection of poisoned facts. Other nodes such as Rosalind Franklin and Wizard exhibit low local clustering, while Charles Darwin was flagged for high betweenness centrality. Additionally, “DNA Structure” and “Mathematician” function as cut-point nodes, indicating potential structural vulnerability.

This approach stands in contrast to opaque black box machine learning models. Instead of merely signaling that “something is wrong”, our method explains why a node is suspicious, whether due to structural abnormality, isolated behavior, or unexpected connections.

### F. Full Implementation

To ensure complete transparency and reproducibility, the entire anomaly detection pipeline is implemented in pure Python, using only standard graph and visualization libraries. Each component, from graph construction and poison injection to anomaly detection and visualization, is built as modular and testable code. No machine learning models, or external datasets are needed.

```

import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from statistics import mean, stdev

G = nx.DiGraph()

# Valid Facts
valid_edges = [
    # Cluster A
    ("Alan Turing", "Mathematician", "hasOccupation"),
    ("Alan Turing", "Enigma Machine", "invented"),
    ("Alan Turing", "Cambridge", "studiedAt"),
    ("Grace Hopper", "Compiler", "developed"),
    ("Grace Hopper", "Mathematician", "hasOccupation"),
    ("Alan Turing", "Grace Hopper", "collaboratedWith"),
    ("Ada Lovelace", "Algorithms", "pioneered"),
    ("Ada Lovelace", "Mathematician", "hasOccupation"),
    ("Ada Lovelace", "Charles Babbage", "collaboratedWith"),
    # Cluster B
    ("Charles Darwin", "Evolution", "proposed"),
    ("Gregor Mendel", "Genetics", "pioneered"),
    ("Gregor Mendel", "Charles Darwin", "inspired"),
    ("Gregor Mendel", "Monk", "hasOccupation"),
    ("Rosalind Franklin", "DNA Structure", "discovered"),
    ("James Watson", "DNA Structure", "collaboratedOn"),
    ("Francis Crick", "DNA Structure", "collaboratedOn"),
    ("James Watson", "Francis Crick", "collaboratedWith"),
    # Bridges
    ("Alan Turing", "Charles Darwin", "studiedIdeasOf"),
    ("Grace Hopper", "Rosalind Franklin", "collaboratedWith")
]

# Poisoned Facts
poisoned_edges = [
    ("Alan Turing", "Chef", "hasOccupation"),
    ("Grace Hopper", "Wizard", "hasOccupation"),
    ("Charles Darwin", "Middle Earth", "discovered"),
    ("Alan Turing", "Atlantis", "visited"),
    ("Atlantis", "Wizard", "inhabitedBy"),
    ("Middle Earth", "Hogwarts", "connectedTo"),
    ("Hogwarts", "Charles Darwin", "graduated"),
    ("Ada Lovelace", "Dragon Slayer", "hasOccupation"),
    ("Gregor Mendel", "Time Traveler", "isA"),
    ("Rosalind Franklin", "Sith Lord", "became")
]

# Add all edges
for u, v, lbl in valid_edges + poisoned_edges:
    G.add_edge(u, v, label=lbl)

# Anomaly detection functions
def detect_high_degree_nodes(G, threshold_std=1.5):
    degrees = [G.out_degree(n) for n in G.nodes()]
    avg, dev = mean(degrees), stdev(degrees)
    return [(n, 'High out-degree anomaly') for n in G.nodes() if G.out_degree(n) > avg + threshold_std * dev]

def detect_low_clustering(G, threshold=0.1):
    undirected = G.to_undirected()
    cl = nx.clustering_coefficient(undirected)
    return [(n, 'Low clustering anomaly') for n in cl if cl[n] < threshold and len(list(undirected.neighbors(n))) >= 2]

def detect_high_betweenness(G, top_percent=0.1):
    centrality = nx.betweenness_centrality(G)
    sorted_nodes = sorted(centrality.items(), key=lambda x: x[1], reverse=True)
    k = max(1, int(top_percent * len(G)))
    return [(n, 'High betweenness anomaly') for n, _ in sorted_nodes[:k]]

def detect_cutpoints(G):
    return [(n, 'Cut-point node') for n in nx.articulation_points(G.to_undirected())]

def detect_star_motif(G, min_children=3):
    flagged = []
    for n in G.nodes():
        if G.out_degree(n) >= min_children:
            children = list(G.successors(n))
            if all(G.out_degree(c) == 0 for c in children):
                flagged.append((n, 'Suspicious star motif'))
    return flagged

def run_all_anomaly_checks(G):
    results = {}
    for func in [detect_high_degree_nodes, detect_low_clustering, detect_high_betweenness, detect_cutpoints, detect_star_motif]:
        results[func.__name__] = func(G)
    seen = set()
    return [(n, r) for n, r in results if not (n in seen or seen.add(n))]

# Layout for visualization (Neural Network Style)
layer_0 = ["Alan Turing", "Grace Hopper", "Ada Lovelace", "Charles Darwin", "Gregor Mendel", "Rosalind Franklin", "James Watson", "Francis Crick"]
layer_1 = ["Mathematician", "Enigma Machine", "Cambridge", "Compiler", "Algorithms", "Charles Babbage", "Evolution", "Genetics", "Monk", "DNA Structure"]
layer_2 = ["Chef", "Wizard", "Middle Earth", "Atlantis", "Hogwarts", "Dragon Slayer", "Time Traveler", "Sith Lord"]

pos_neural = {}
for i, node in enumerate(layer_0):
    pos_neural[node] = (-2, -i)
for i, node in enumerate(layer_1):
    pos_neural[node] = (0, -i)
for i, node in enumerate(layer_2):
    pos_neural[node] = (2, -i)

# Visualize graph
def visualize_graph_with_layout(G, pos, anomalies=None, title=""):
    plt.figure(figsize=(14, 10))
    node_colors = ['red' if anomalies and n in [a[0] for a in anomalies] else 'lightblue' for n in G.nodes()]
    nx.draw(G, pos, with_labels=True, node_size=2000, node_color=node_colors, arrows=True, arrowsize=20, font_size=9)
    edge_labels = nx.get_edge_attributes(G, 'label')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='gray', font_size=7)
    plt.title(title)
    plt.axis('off')
    plt.tight_layout()
    plt.show()

# Run detection and show
anomalies_detected = run_all_anomaly_checks(G)
visualize_graph_with_layout(G, pos_neural, anomalies=anomalies_detected)

# Show anomaly table
df = pd.DataFrame(anomalies_detected, columns=["Node", "Anomaly Type"])
print(df)

```

**Figure 10.** Full Code Implementation. (source: Author)

This core implementation constructs a large scale directed graph (networkx.DiGraph) modeling real and fictional knowledge, injects syntactically valid but semantically false edges that simulates real world poisoning attempts, applies five

independent rule-based detectors, and aggregates all detection results into a structured report and overlays the finding on the graph. This code supports visual comparison before and after detection, layered layout resembling neural architectures for better readability, and tabular summary of all flagged anomalies (e.g., nodes + reasons).

This full stack implementation enables research and practitioners due to audit graphs interactively, extend detection rules for custom use cases, and integrate anomaly feedback into downstream reasoning or filtering systems. All outputs, including visual graphs and DataFrames, can be saved or embedded directly into reports and dashboards. The framework is designed to be both useful (for teaching graph theory and adversarial robustness) and practically applicable (for early stage KG integrity audits in production).

## VI. CONCLUSION AND FUTURE WORK

### A. Conclusion

This study demonstrates that structural anomaly detection in knowledge graphs can be effectively through lightweight, interpretable techniques rooted in graph theory. By modeling a poisoned knowledge graph and applying rule based detectors, including node degree, clustering coefficient, betweenness centrality, cut-point, and motif analysis, we show that it is possible to detect anomalies without relying on machine learning or semantic validation.

The detection results align with intuitive expectations. Key nodes such as Alan Turing, Grace Hopper, and Ada Lovelace were flagged for out-degree inflation, while entities like Rosalind Franklin and Wizard were identified for low clustering. Moreover, structural roles like cut-points and bridging nodes were successfully highlighted. The use of visual overlays (red-highlighted nodes) reinforces the method's explainability, making the results accessible even to non-technical auditors.

Compared to black-box ML systems, this approach offers a critical advantage. Each anomaly is structurally traceable and explainable by design. The output is not just a binary flag, but a reflection of how the node's topology deviates from expected behavior. In short, this work supports the thesis that discrete mathematical heuristics are sufficient to reveal meaningful structural anomalies in knowledge graphs, especially in early-stage or zero-training contexts.

### B. Limitations

Despite its strength, the system has several limitations:

- **Context-agnostic:** Structural detectors do not incorporate semantic nuance. A node with high out-degree may not be poisoned in a real world context, but flagged nonetheless.
- **Scalability:** While efficient for small to medium graphs ( $\leq 100$  nodes), some techniques (e.g., betweenness centrality) scale poorly to massive graphs without approximation.



- No repair mechanism: The system flags anomalies but does not offer automated correction or mitigation.

These limitations underscore the need to treat this framework as a first stage filter, a preprocessor or early warning system, rather than a complete security pipeline.

### C. Future Work

This project opens several directions for further exploration:

- Hybrid Models: Integrating graph structure with lightweight semantic embeddings could combine the best of both worlds, interpretability and expressivity.
- Real World Datasets: Applying the system to real RDF knowledge (e.g., DBpedia, Wikidata) will test its practical viability under noisy and heterogeneous conditions [7].
- Interactive Auditing Tools: Packaging the system into a web-based UI with graph interactivity could empower domain experts to manually verify and edit poisoned facts.
- Explainability Metrics: Developing quantitative measures for explanation clarity (e.g., “why was node X flagged?”) could strengthen trust and usability.

Ultimately, while simple, the framework provides a strong foundation for transparent, explainable knowledge graph integrity checks, especially in scenarios where trust and auditability matter as much precision.

### APPENDIX

*Video link and PowerPoint slides:*

[https://drive.google.com/drive/u/2/folders/1aohEzlyuEWX\\_pNmBqLki2F3g3SbJjHHI](https://drive.google.com/drive/u/2/folders/1aohEzlyuEWX_pNmBqLki2F3g3SbJjHHI)

*Full code:*

<https://github.com/ethj0r?tab=repositories>

### ACKNOWLEDGMENT

The author would like to thank the lectures and teaching assistants of IF1220 Discrete Mathematics, School of Electrical Engineering, Bandung Institute of Technology, for the valuable foundational material that supported the development of this paper. Appreciation is also extended to the open-source contributors of NetworkX and Matplotlib for enabling rapid prototyping and visualization of graph-based anomaly detection techniques.

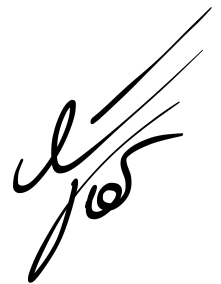
### REFERENCES

- [1] Rinaldi Munir, *Graf (Bagian 1 dan 2)*, Lecture Notes IF1220 Matematika Diskrit, STEI ITB, 2024.
- [2] NetworkX Developers, “NetworkX Documentation (v3.2.1),” [Online]. Available: <https://networkx.org/documentation/stable/>
- [3] J. Leskovec, A. Rajaraman, J. Ullman, *Mining of Massive Datasets*, 3<sup>rd</sup> ed., Cambridge University Press, 2020.
- [4] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, “Summarizing and understanding large graphs,” *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 8, no. 3, pp. 183–202, 2015.
- [5] A. Akoglu, H. Tong, and D. Koutra, “Graph-based anomaly detection and description: A survey,” *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 626–688, 2015.
- [6] Y. Zhang and H. Chen, “A survey on graph neural network-based knowledge graph completion,” *ACM Computing Surveys*, vol. 54, no. 5, pp. 1–37, 2021.
- [7] DBpedia Project, “DBpedia Knowledge Graph,” [Online]. Available: <https://www.dbpedia.org>
- [8] M. Newman, *Networks: An Introduction*, Oxford University Press, 2010.
- [9] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs,” *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

### STATEMENT

I hereby declare that the paper I have written is entirely my own work. It is not an adaptation, a translation of another person’s work, nor a product of plagiarism.

Bandung, June 19, 2025



Made Branenda Jordhy 13524026