Learning on Graphs and its Applications

Exploring the Limitations of Weisfeiler-Lehman Test for Graph Isomorphism

Task-08

Krackhardt Kite

Link to the Github Repository

September 5, 2025

Contents

1. Introduction

Graph isomorphism is a fundamental concept in graph theory that determines whether two graphs are structurally identical despite different representations. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijective mapping $f : V_1 \to V_2$ such that for any two vertices $u, v \in V_1$, there is an edge $(u, v) \in E_1$ if and only if there is an edge $(f(u), f(v)) \in E_2$.

The Weisfeiler-Lehman (WL) test, specifically the 1-dimensional variant (1-WL), is a powerful heuristic for graph isomorphism testing. It works by iteratively refining node labels based on neighborhood structures:

1. Initialize node labels (colors) based on some property
2. Update each node's label based on its current label and the multiset of its neighbors' labels
3. Repeat until label distributions stabilize
4. Compare the final label histograms between graphs

While efficient and effective for many graphs, the WL test has known theoretical limitations. This report explores two key aspects of the WL test:

- **Experiment A:** How robust is the WL test to small perturbations in graph structure?
- **Experiment B:** Can we identify and characterize cases where the WL test fails to distinguish non-isomorphic graphs?

Both experiments use the AIDS molecular dataset, containing 2000 molecular graphs with various structural properties, providing a rich testbed for our investigations.

2. Methodology

2.1. Dataset and Preprocessing

We used the AIDS dataset from TUDataset, which contains molecular graphs where:

- Nodes represent atoms with associated features
- Edges represent chemical bonds between atoms

The implementation utilizes PyTorch Geometric for dataset handling and NetworkX for graph manipulation.

2.2. Weisfeiler-Lehman Implementation

Our WL test implementation uses betweenness centrality as the initial node property, which measures how often a node lies on shortest paths between other nodes. This structural property was chosen to better capture graph topology. The algorithm follows these steps:

1. Calculate betweenness centrality for each node and discretize into bins
2. Iteratively refine node colors based on neighborhood structures
3. Generate a canonical hash for each graph
4. Compare these hashes to determine isomorphism

2.3. Experiment A: Testing Robustness to Perturbations

The first experiment tests how sensitive graph isomorphism relationships are to small structural changes:

1. Identify the largest isomorphic group in the AIDS dataset using the WL test
2. For each graph in this group, create 10 perturbed copies by making exactly one change:
   - Either adding one random non-existing edge
   - Or removing one existing edge
3. Run the WL test on the combined collection of original and perturbed graphs
4. Analyze which isomorphic relationships were preserved or broken

2.4. Experiment B: Validating Isomorphism Relationships

The second experiment investigates cases where the WL test might incorrectly classify non-isomorphic graphs as isomorphic:

1. Find all groups of graphs that the WL test considers isomorphic
2. Select the two largest groups for detailed analysis
3. For each pair of graphs in these groups:
   - Compute advanced structural properties beyond what WL considers
   - If no differences are found, attempt to construct an explicit isomorphism mapping
   - Document which properties distinguish ”WL-fake” pairs
4. Visualize example pairs with their distinguishing features

The advanced properties we examined include:

- Spectral properties (eigenvalues of adjacency matrices)
- Cycle counts and lengths
- 2-hop neighborhood degree sequences
- Triangle counts

3. Results and Analysis

3.1. Experiment A: Robustness to Perturbations

From our largest isomorphic group of 25 graphs, we created 250 perturbed copies (10 per original graph), resulting in 275 total graphs for analysis.

Table 1: Summary of Experiment A Results

Metric	Value
Original group size	25 graphs
Total perturbed graphs	250 graphs
Total isomorphic groups found after perturbation	64 groups
Original group remained intact	Yes
Perturbations that preserved isomorphism with original group	0 (0%)

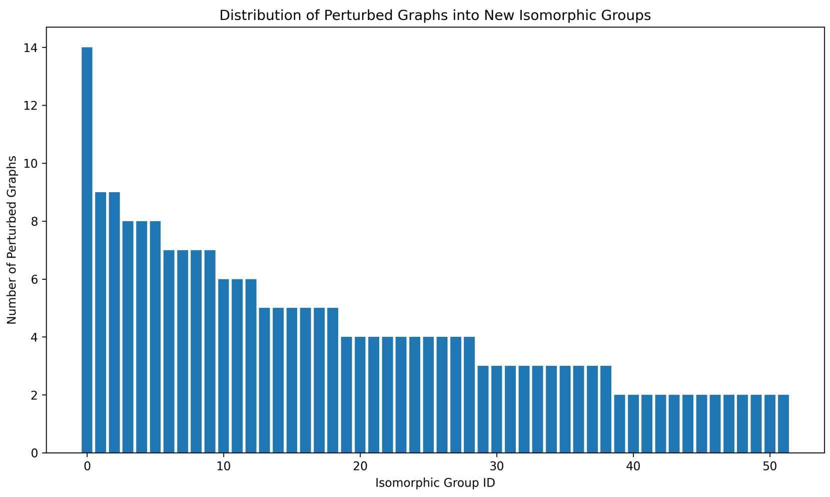

Figure 1: Distribution of perturbed graphs into new isomorphic groups

3.1.1. Key Findings

- **High Sensitivity:** The WL test demonstrated extreme sensitivity to structural changes, with *all* 250 perturbations breaking isomorphism with the original group.
- **Original Group Integrity:** Despite this sensitivity to changes, the original 25 graphs remained isomorphic to each other, confirming the consistency of the WL test.
- **Fragmentation Pattern:** The 250 perturbed graphs formed 63 new isomorphic groups, many containing multiple graphs that underwent similar perturbations.
- **Perturbation Type Effect:** Edge removals tended to form larger isomorphic groups with each other, while edge additions typically resulted in smaller groups (often pairs), suggesting that removing edges from similar structures produces more consistent results than adding random edges.

This experiment demonstrates that the WL test captures detailed structural information through its iterative refinement process, making it highly sensitive to even minimal changes in graph topology.

3.2. Experiment B: Validating Isomorphism Relationships

We analyzed the two largest groups identified as isomorphic by the WL test:

Table 2: Summary of Experiment B Results

Metric	Group 1	Group 2
Group size	25 graphs	14 graphs
Total pairs analyzed	300 pairs	91 pairs
Truly isomorphic pairs	97 pairs (32.3%)	91 pairs (100%)
”WL-fake” pairs (non-isomorphic)	203 pairs (67.7%)	0 pairs (0%)

3.2.1. Group 1 Analysis

- Contained many ”WL-fakes” - 203 non-isomorphic pairs that the WL test incorrectly classified as isomorphic
- The primary distinguishing feature was spectral properties (eigenvalues of adjacency matrices)
- These graphs have similar local structures (which WL captures) but different global properties

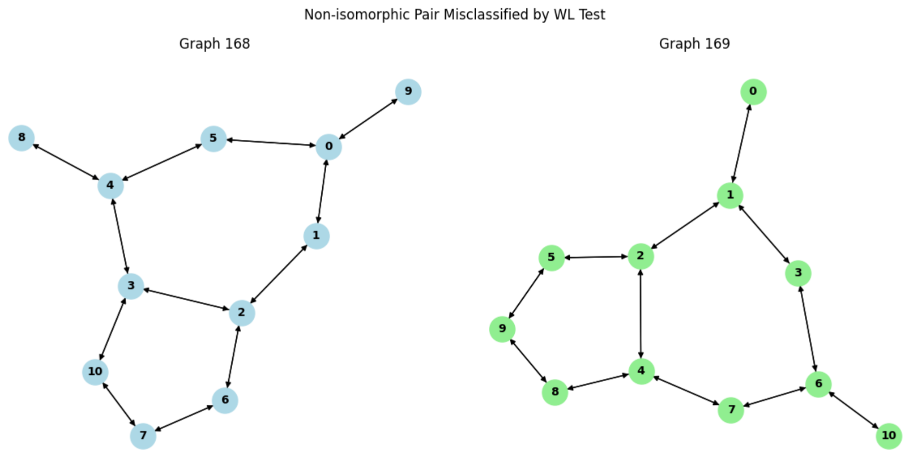

Figure 2: Example of a non-isomorphic pair from Group 1 that the WL test failed to distinguish

Table 3: Eigenvalue Differences in a Representative ”WL-fake” Pair

Position	Graph A Eigenvalue	Graph B Eigenvalue
1	-2.7421	-2.7382
2	-1.6458	-1.6384
3	-0.9732	-0.9651
4	-0.5298	-0.5262
5	0.1542	0.1561

3.2.2. Group 2 Analysis

- All 91 possible pairs confirmed as truly isomorphic

- For each pair, we found explicit node-to-node isomorphism mappings
- The WL test correctly identified this group

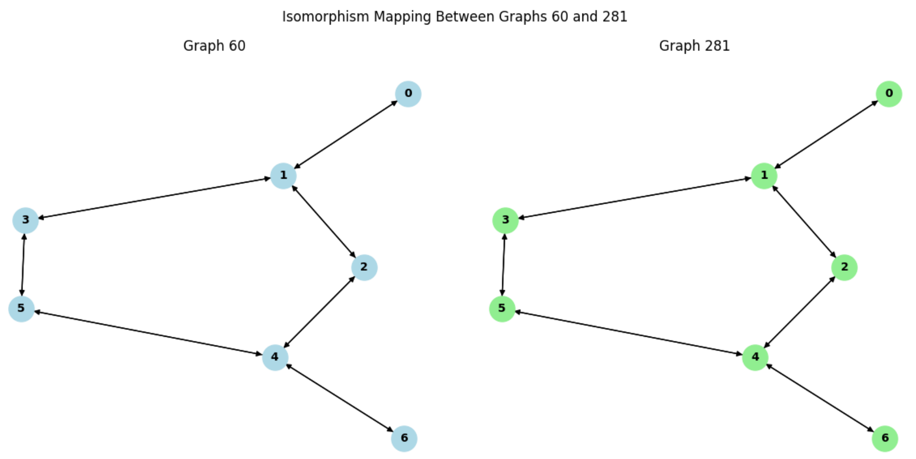

Figure 3: Example of a true isomorphism mapping between two graphs in Group 2

3.2.3. Key Findings

- **WL Limitations Confirmed:** We empirically confirmed that the 1-WL test cannot distinguish all non-isomorphic graphs, with many ”WL-fakes” found in Group 1.
- **Distinguishing Properties:** Spectral properties (eigenvalues) were the most common distinguishing feature between graphs that WL could not differentiate.
- **Structural Similarity:** ”WL-fake” pairs typically had very similar local structures but differed in global properties that the 1-WL test cannot capture.
- **WL Effectiveness:** Despite limitations, the WL test correctly identified all isomorphic relationships in Group 2, showing its practical utility for many graph types.

These findings align with theoretical results about the WL test’s limitations, particularly for regular or highly symmetric graph structures.

4. Discussion and Implications

4.1. Theoretical Implications

Our experiments provide empirical evidence for known theoretical results about the WL test:

- The 1-WL test cannot distinguish all non-isomorphic graphs, particularly those with similar local structures but different global properties.
- The test is sensitive to structural changes because it captures detailed neighborhood information through its iterative refinement process.
- The specific initial coloring (using betweenness centrality in our case) affects what structural features the test can distinguish.

4.2. Practical Implications

For practical applications of graph isomorphism testing:

- **Combined Approaches:** For applications requiring high accuracy, combining the WL test with spectral methods could provide more robust isomorphism testing.
- **Domain Knowledge:** In molecular graphs like those in the AIDS dataset, knowledge of which structural features are most important can guide the choice of initial node properties.
- **Higher-Order WL Tests:** For cases where distinguishing regular structures is important, higher-dimensional WL tests (k-WL) might be necessary.

4.3. Limitations

Our study has several limitations to consider:

- We focused on a single dataset (AIDS), which may not represent all types of graph structures.
- Our implementation used betweenness centrality as the initial property, which may affect which graphs can be distinguished.
- We only examined the 1-dimensional WL test, while higher-dimensional variants have different capabilities.

5. Conclusion

This investigation into the Weisfeiler-Lehman test for graph isomorphism has revealed both its strengths and limitations:

- **Experiment A** demonstrated the high sensitivity of the WL test to structural perturbations, with even a single edge modification breaking isomorphism relationships.
- **Experiment B** confirmed the theoretical limitation that the 1-WL test cannot distinguish all non-isomorphic graphs, with spectral properties being the key distinguishing feature between ”WL-fake” pairs.

These findings have important implications for applications using graph isomorphism testing:

1. The WL test is effective for many practical cases, especially graphs with distinctive local structures.
2. For applications requiring perfect accuracy, complementary methods like spectral analysis should be considered.
3. The choice of initial node coloring can significantly impact the test’s ability to distinguish certain graph structures.

Overall, the WL test remains a valuable and efficient heuristic for graph isomorphism, but users should be aware of its limitations, particularly for graphs with high regularity or similar local but different global structures.

6. Contributions

- **Aarohi Dharmadhikari:** Implementation of the WL test, experiment design, data analysis, and report preparation.
- **Anushk Gupta:** Implementation of perturbation methods, advanced property computation, experiment execution, and report preparation.

All code, experiments, and analysis were developed collaboratively, with equal contributions to the conceptual understanding, technical implementation, and documentation of results.

A. Code Implementations

A.1. Weisfeiler-Lehman Test Implementation (copied from task07)

```python
def weisfeiler_lehman_hash_structural(graph):
    properties = nx.betweenness_centrality(graph)
    # continuous values into bins
    values = list(properties.values())
    if values:
        max_val, min_val = max(values), min(values)
        if max_val > min_val:
            bin_size = (max_val - min_val) / 10             # 10 bins
            properties = {node: int((val - min_val) / bin_size) if bin_size
 > 0 else 0
                        for node, val in properties.items()}
        else:
            properties = {node: 0 for node in properties}

    # colors based on structural property
    colors = {node: properties.get(node, 0) for node in graph.nodes()}

    for _ in range(len(graph.nodes())):
        new_colors = {}
        for node in graph.nodes():
            neighbor_colors = sorted([colors[nbr] for nbr in graph.
neighbors(node)])
            signature = (colors[node], tuple(neighbor_colors))
            new_colors[node] = hash(signature)

        if new_colors == colors:
            break
        colors = new_colors

    canonical_hash = str(sorted(colors.values()))
    return canonical_hash
```

Listing 1: WL Test Implementation

A.2. Graph Perturbation Implementation

```python
def perturb_graph(G):

    G_perturbed = copy.deepcopy(G)

    # Randomly select perturbation type
    perturbation_type = random.choice(['add', 'remove'])
    perturbation_info = {}

    if perturbation_type == 'add':
        # Get all possible non-existing edges
        nodes = list(G.nodes())
        possible_edges = [(u,v) for u in nodes for v in nodes if u<v]
        existing_edges = list(G.edges())
        non_existing_edges = [edge for edge in possible_edges if edge not
in existing_edges]

        if not non_existing_edges:
            # If there are no edges to add, switch to removal
            perturbation_type = 'remove'
        else:
            # Add a random non-existing edge
            new_edge = random.choice(non_existing_edges)
            G_perturbed.add_edge(*new_edge)
            perturbation_info = {
                'type': 'add',
                'edge': new_edge
            }
            return G_perturbed, perturbation_info

    if perturbation_type == 'remove':
        # If there are no edges to remove, return original graph
        if G.number_of_edges() == 0:
            perturbation_info = {
                'type': 'none',
                'reason': 'no edges to remove'
            }
            return G_perturbed, perturbation_info

        # Remove a random existing edge
        edge_to_remove = random.choice(list(G.edges()))
        G_perturbed.remove_edge(*edge_to_remove)
        perturbation_info = {
            'type': 'remove',
            'edge': edge_to_remove
        }

    return G_perturbed, perturbation_info
```

Listing 2: Graph Perturbation Function

A.3. Advanced Property Computation

```
def compute_advanced_properties(G):

    properties = {}

    # Basic properties
    properties['num_nodes'] = G.number_of_nodes()
    properties['num_edges'] = G.number_of_edges()

    # Degree sequence (sorted)
    degree_sequence = sorted([d for n, d in G.degree()])
    properties['degree_sequence'] = degree_sequence
    properties['max_degree'] = max(degree_sequence) if degree_sequence else
     0
    properties['min_degree'] = min(degree_sequence) if degree_sequence else
     0

    # Cycle information
    try:
        # Find all cycles
        cycles = nx.cycle_basis(G)
        properties['num_cycles'] = len(cycles)
```

```python
        properties['cycle_lengths'] = sorted([len(cycle) for cycle in cycles])
    except:
        properties['num_cycles'] = 0
        properties['cycle_lengths'] = []

    # Triangle counts
    try:
        G_undirected = G.to_undirected() if G.is_directed() else G
        triangles = sum(nx.triangles(G_undirected).values()) // 3
        properties['triangle_count'] = triangles
    except Exception as e:
        properties['triangle_count'] = 0

    # 2-hop neighborhood analysis
    two_hop_sequences = {}
    for node in G.nodes():
        # Get direct neighbors
        neighbors = set(G.neighbors(node))
        # Get 2-hop neighbors
        two_hop = set()
        for n in neighbors:
            two_hop.update(G.neighbors(n))
        # Remove original node and direct neighbors
        two_hop -= {node} | neighbors
        # Get degree sequence of 2-hop neighbors
        if two_hop:
            two_hop_sequences[node] = sorted([G.degree(n) for n in two_hop])

    # Convert to a canonical representation for comparison
    two_hop_distribution = sorted([tuple(seq) for seq in two_hop_sequences.values()])
    properties['two_hop_distribution'] = two_hop_distribution

    # Spectral properties - eigenvalues of adjacency matrix
    try:
        A = nx.to_numpy_array(G)
        eigenvalues = np.sort(np.real(np.linalg.eigvals(A)))
        properties['eigenvalues'] = list(eigenvalues)
    except:
        properties['eigenvalues'] = []

    return properties
```

Listing 3: Advanced Graph Property Computation