Assignment 1: Graph Representation and Kernel Methods in Large Graphs

ELTE University

by

Jacopo Manenti

Student Name	Student Number
Jacopo Manenti	G1R5RZ

Professor:	Guettala Walid
Faculty:	Computer Science, Budapest

0.1. Abstract

In this report, we propose an implementation of two famous kernels for graph analysis on small and large undirected graphs (subsection 0.3.1), namely the Graphlet and Weisfeiler-Lehman graph kernels [1]. The aim of this paper is to explore and evaluate these methods for efficiently comparing large graphs. By testing them, we seek to determine which approach offers superior performance in terms of efficiency and scalability. The experiments concluded that the WL method is the faster feature extraction algorithm between the two for testing graph isomorphism. Furthermore, in our experiments (subsection 0.3.6), we tested how its runtime scales with the increasing number of edges in the graphs and the length of the Weisfeiler-Lehman graph sequence to see the limitations of the WL kernel for large-scale graph comparisons.

0.2. Theoretical foundation

In graph, a graph kernel is a kernel function that computes an inner product of feature vectors of graphs. A **graph feature vector** $\phi(G)$ is a numerical representation that captures essential properties or structural information about the graph. Depending on how it is designed, we can have different approches.

0.2.1. Graphlet Kernel

Graphlet kernels are based on the idea of randomly sampling small subgraphs of size k (e.g., 3, 4, 5). These samples can then be used to construct the feature vector by counting the occurrences of these small subgraphs, known as **graphlets**. The nodes in a graphlet **do not need to be connected**, and the graphlets are **not rooted**, meaning their structure is analyzed without a designated starting point.

Given a graph G and a graphlet list $G_k = (g_1, g_2, \ldots, g_{n_k})$, i.e., a list that contains small connected subgraphs of k nodes (where k is the number of nodes considered), we define the graphlet count vector $f_G \in \mathbb{R}^{n_k}$ as:

$$f_G = \#(g_i \subseteq G) \quad \text{for } i = 1, 2, \ldots, n_k$$

Once we have computed the feature vector for each graph, we can apply the **Graph Kernel** to measure the similarity between the graphs based on the graphlets they contain.

Given two graphs G and G' of size $n \geq k$, the graphlet kernel K_g is defined as:

$$K_g(G, G') = f_G^T f_{G'}$$

To address the issue of size mismatch, we apply the kernel to the normalized version of the feature vectors:

$$h_G = \frac{f_G}{\sum(f_G)}$$

Thus, the final normalized graphlet kernel is:

$$K(G, G') = h_G^T h_{G'}$$

0.2.2. Weisfeiler-Lehman Graph Kernels

Weisfeiler-Lehman Graph Kernels rely on the Weisfeiler-Lehman test for graph isomorphism. This algorithm maps the original graph to a sequence of graphs by leveraging the neighborhood structure of nodes to iteratively refine the node representations (or colors) based on their neighbors.

Consider a graph G with a set of nodes V. The WL algorithm proceeds as follows:

1. **Initial Coloring**: Assign an initial color $c^{(0)}$ to each node $v \in V$.

[1]Note that each algorithm was implemented from scratch

2. **Iterative Color Refinement**: Refine the colors of the nodes iteratively using the following formula:

$$c^{(k+1)}(v) = \text{Hash}(\left\{c^{(k)}(v), \left\{c^{(k)}(u)_{u \in N(v)}\right\}\right\})$$

The Hash function maps different inputs to different colors. After K steps of color refinement, $c^{(k)}(v)$ summarizes the structure of k-hop neighborhood [2]. Once the color refinement is complete, the WL algorithm counts the number of nodes associated with each color. These counts are then used to construct the feature vector, where each element corresponds to the frequency of a specific color.

The WL kernel value is then computed as the inner product of the resulting color count vectors. To stabilize the similarity result and make it comparable, we first normalize each vector by its norm and the compute the kernel.

$$h_G = \frac{f_G}{||(f_G)||}$$

$$K(G, G^{'}) = h_G^T h_{G'}$$

This inner product effectively measures the similarity between the two graphs by comparing how many nodes share the same colors after the refinement process, reflecting their structural similarity.

0.2.3. Complexity of the two algorithms

- **Graphlet kernel**: Counting $size - k$ graphlets has a complexity of $O(n^k)$, which becomes computationally infeasible for large graphs due to the combinatorial explosion of subgraphs as n (the number of nodes) grows.

  When attempting to enumerate all possible graphlets of size k in a graph with n nodes, the number of such subgraphs grows combinatorially. Specifically, the number of possible graphlets of size k is $\binom{n}{k}$, which is proportional to $O(n^k)$. For example for $k = 4$, the number is $\binom{n}{4} \approx O(n^4)$.

  This **combinatorial complexity** becomes prohibitive for large n, as the process involves exploring every possible combination of k-node subgraphs in the graph.

- **Weisfeiler-Lehman Graph Kernels**: The runtime complexity of the 1-dimensional[3] Weisfeiler-Lehman algorithm, with h iterations, is $O(hm)$. The process begins with defining multisets for all nodes, which takes $O(m)$ time. Sorting each multiset also requires $O(m)$. Label compression involves a single pass over all strings and characters, also $O(m)$. Consequently, the total runtime across h iterations is $O(hm)$. So, the runtime scales only linearly in the number of edges of the graphs and the length of the Weisfeiler-Lehman graph sequence.

0.3. Experiments

0.3.1. Datasets

For our analysis, we utilized both synthetic and real undirected graph datasets. Specifically, we employed two small synthetic datasets (each with 100 nodes) and one large real-world dataset (approximately 4,000 nodes) for the computationally expensive Graphlet algorithm (Figure 1).

The datasets are as follows:

- **Synthetic Data:**
  - **Barabási-Albert Graph:** A scale-free graph representing real-world networks through preferential attachment.

[2]In a graph, the K-hop neighborhood of a node is the set of all nodes that can be reached from that node by traversing K or fewer edges. For example, a 1-hop neighborhood includes only the directly connected neighbors, while a 2-hop neighborhood includes those neighbors plus their neighbors

[3]In the 1-dimensional WL algorithm, each node in the graph is assigned an initial label (or feature). In a k-dimensional ($k > 1$) WL algorithm, each node might have multiple features, and the algorithm would have to account for these multiple dimensions in its label refinement and comparison processes.

- **Watts-Strogatz Small-World Graph:** A graph balancing randomness and regularity, capturing small-world properties.

- **Real Data:**
  - **Facebook Social Circles Network:** Comprising 'circles' (or 'friends lists') collected from survey participants via a Facebook app.

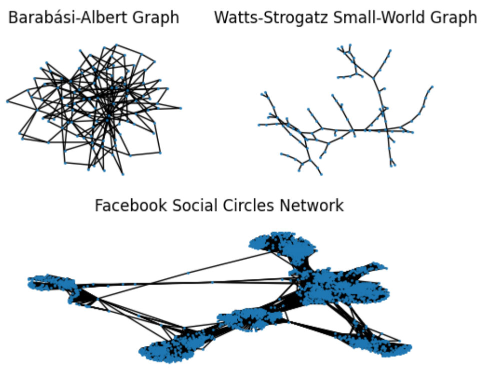

Barabási-Albert Graph
Watts-Strogatz Small-World Graph
Facebook Social Circles Network

Figure 1: A visualization of the graph datasets used for the Graphlet Kernel

Since Weisfeiler-Lehman (WL) kernel is supposed to be more efficient, we applied it to the **TUDataset** collection, focusing on the PROTEINS and REDDIT_BINARY datasets, which contain approximately 40,000 and 420,000 nodes, respectively (Figure 2).

Additionally, we applied both WL kernel and Graphlet kernel to a subset of 200 nodes of these TUDataset to evaluate the runtime. This comparison aims to illustrate the efficiency of the WL kernel in relation to the Graphlet kernel.

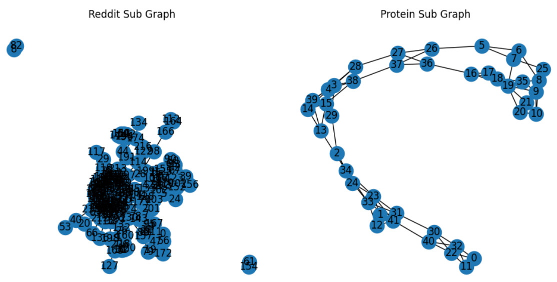


Figure 2: A visualization of the (sub) graph datasets (≈ 200 nodes) used for Weisfeiler-Lehman (WL) kernel

0.3.2. Datasets analysis

To illustrate the concept of isomorphism and similarity, let's examine the **degree distribution** of the graphs utilized in the Graphlet algorithm. This analysis will allow us to infer the similarity of the graphs produced by the algorithm.

In graph theory, the degree k of a node v is defined as the number of edges connecting that node to its neighbors. The degree distribution reflects how connectivity is distributed among the nodes in a graph, making it a crucial metric for comparing the structural properties of different graphs. When two graphs exhibit similar degree distributions, it often indicates that they possess comparable network structures or topologies.

As we can see from Figure 3, we can say that the *Barabási-Albert model graph* seems to be more similar to the *Facebook Social Circles Network graph*. Both show highly skewed distributions with many low-degree nodes and a long tail of high-degree nodes. The *Watts-Strogatz Small-World*

Graph has a more uniform degree distribution centered around a specific value, making it less similar (skewed) to the other two. So, the Watts-Strogatz model might have a different network topology with more evenly distributed connections.

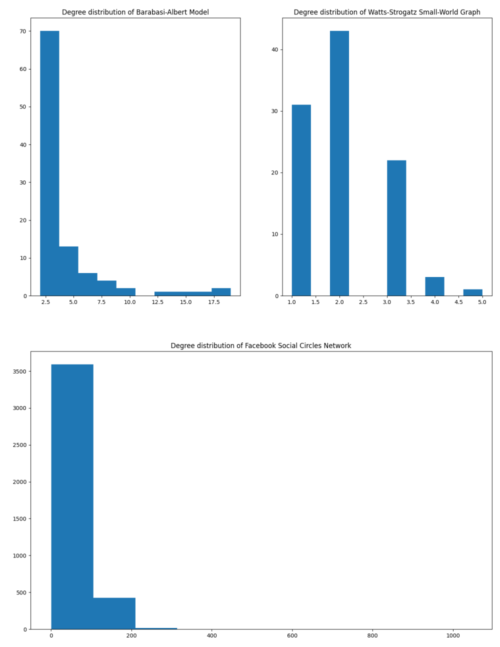


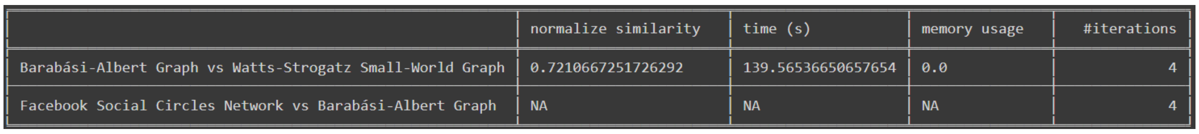
Figure 3: Degree distribution of dataset used for Graphlet kernel.

0.3.3. Graphlet Kernel

In Algorithm 1 we have reported our implementation of the Graphlet algorithm. Furthermore, in Figure 4 we have reported the result obtained by comparing Barabási-Albert vs Watts-Strogatz Small-World Graph and Barabási-Albert vs Facebook Social Circles Network.

Algorithm 1 Graphlet Kernel

```
Require: $k > 0$
 1: for Graph$_i$, $i \in \{1, 2\}$ do
 2:     initialize empty dictionary for $graphletsCount$
 3:     set $n =$ number of node in Graph$_i$
 4:     Pre-compute adjacency matrix
 5:     Initialize the feature vector with dimension $(k * (k - 1))//2 + 1$
 6:     for $verticesIdx \in$ combinations(range$(n), k)$ do
 7:         $submatrix \leftarrow$ subset the adjecency matrix with$verticesIdx$
 8:         $edges \leftarrow$ sum the edges
 9:         if $edges \notin graphletsCount$ then
10:             $graphletsCount[edges] \leftarrow 0$
11:         end if
12:         $graphletsCount[edges] \leftarrow graphletsCount[edges] + 1$
13:     end for
14:     for egdes, count $\in graphletsCount$ do
15:         $featurevector[edges] \leftarrow count$
16:     end for
17:     $total \leftarrow$ total number of graphlets in the feature vector
18:     $NormalizeFeatureVectorGraph_i \leftarrow$ normalize the feature vector by the $total$
19: end for
20: $GraphKernel \leftarrow$ dot product between $NormalizeFeatureVectorGraph_i$
        return $GraphKernel$
```



	normalize similarity	time (s)	memory usage	#iterations
Barabási-Albert Graph vs Watts-Strogatz Small-World Graph	0.7210667251726292	139.56536650657654	0.0	4
Facebook Social Circles Network vs Barabási-Albert Graph	NA	NA	NA	4

Figure 4: The table presents the performance of Graphlet Kernels on two types of datasets: (**1° row**) two synthetic datasets ($\approx$ 100 nodes each) and (**2° row**) a synthetic dataset vs a real-world dataset ($\approx$ 4000 nodes). Due to limitations of the Colab free plan, we could not complete the experiment on the real dataset, as the process exceeded 6 hours and led to disconnection. Time complexity is measured in seconds, while memory usage is reported in MB.

In the table above, we observe that the algorithms computed a significant degree of similarity between the two synthetic datasets, yielding a similarity score of 0.72. This result contradicts our findings in Section 0.3.2. The high similarity score suggests that the networks may share the same graph structure, potentially indicating isomorphism. However, the node distribution analysis of the two datasets indicated otherwise, with the Barabási-Albert Graph displaying a skewed node distribution.

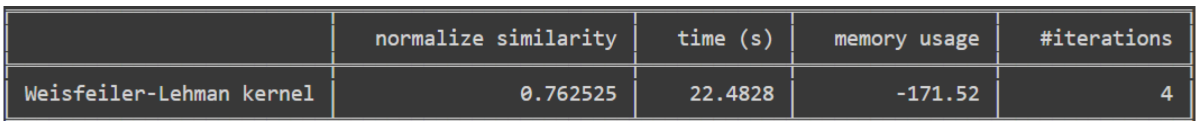
This discrepancy may be attributed to the limited number of nodes (100) used in the analysis. This choice was made to manage the time complexity constraints of the algorithm. Increasing the number of nodes (e.g., to 1000) would result in a computational complexity of $O(n^k) = O(1000^4)$, significantly increasing the number of required iterations. For this reason, we reported "NA" for both the normalized similarity score and time complexity when applying the kernel to the real dataset, which contains approximately 4000 nodes. With a complexity of $O(4000^4)$, we were unable to complete the run within a reasonable time.

0.3.4. Weisfeiler-Lehman Graph Kernels

In Algorithm 2 we have reported our implementation of the Graphlet algorithm. In Figure 5 we have reported the result obtained by comparing TUDatasets.

Algorithm 2 One iteration of the 1-dim. Weisfeiler-Lehman test of graph isomorphism

```
1:  procedure WeisfeilerLehman($G_1 = (V_1, E_1), G_2 = (V_2, E_2)$)
2:      kernel_vector_g1 ← {}
3:      kernel_vector_g2 ← {}
4:      for $v \in V_1$ do
5:          $l_1(v) \leftarrow 1$ (uniform color)
6:      end for
7:      for $v \in V_2$ do
8:          $l_2(v) \leftarrow 1$ (uniform color)
9:      end for
10:     for $i = 1$ to $k$ do
11:         for $v \in V_1$ do
12:             $M_1(v) \leftarrow$ Sort($\{l_1(u) : u \in N(v)\}$)
13:             $l'_1(v) \leftarrow$ Hash($l_1(v), M_1(v)$)
14:         end for
15:         for $v \in V_2$ do
16:             $M_2(v) \leftarrow$ Sort($\{l_2(u) : u \in N(v)\}$)
17:             $l'_2(v) \leftarrow$ Hash($l_2(v), M_2(v)$)
18:         end for
19:         labels ← $l'_1 \cup l'_2$
20:         for label ∈ labels do
21:             if label ∈ $l'_1$ then
22:                 kernel_vector_g1[label] ← $l'_1$[label]
23:             else
24:                 kernel_vector_g1[label] ← $0$
25:             end if
26:             if label ∈ $l'_2$ then
27:                 kernel_vector_g2[label] ← $l'_2$[label]
28:             else
29:                 kernel_vector_g2[label] ← $0$
30:             end if
31:             $l_1 \leftarrow l'_1$
32:             $l_2 \leftarrow l'_2$
33:         end for
34:     end for
35:     vector_g1 ← [value for value in kernel_vector_g1.values()]
36:     vector_g2 ← [value for value in kernel_vector_g2.values()]
37:     vector_g1_norm ← $\frac{\text{vector\_g1}}{\|\text{vector\_g1}\|}$
38:     vector_g2_norm ← $\frac{\text{vector\_g2}}{\|\text{vector\_g2}\|}$
39:     result ← vector_g1_norm · vector_g2_norm
        return result
40: end procedure
```



	normalize similarity	time (s)	memory usage	#iterations
Weisfeiler-Lehman kernel	0.762525	22.4828	-171.52	4

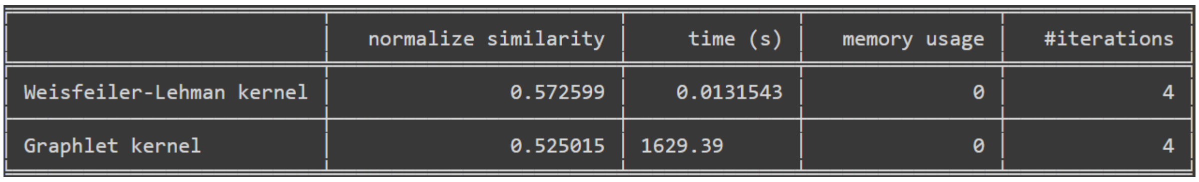
Figure 5: The table reports the performance of Weisfeiler-Lehman Graph Kernels on PROTEINS and REDDIT_BINARY (TUDatasets) graph datasets. Time complexity is measured in seconds, while memory usage is reported in MB.

As illustrated in the table above, the algorithms identified a significant degree of similarity between the two datasets, with a similarity score of 0.76. This score indicates a strong correlation, suggesting that the two graphs share numerous structural properties, such as connectivity patterns, node distributions, and other topological features.

Moreover, it is important to consider the scale of the datasets: PROTEINS contains approximately 40,000 nodes, while REDDIT-BINARY consists of around 240,000 nodes. Given this size, the time required for analysis is remarkably efficient, especially when compared to the graphlet algorithm, which takes approximately 22 seconds. This efficiency underscores the effectiveness of the methods employed in handling large-scale graph data while maintaining a robust similarity assessment.

0.3.5. A comparison of the two algorithms

In Figure 6 we compared the accuracy and computational cost of the Graphlet Kernel and Weisfeiler-Lehman (WL) Graph Kernel algorithms. Given that the former approach is time-intensive, we have evaluate the differences between the two algorithms using a subset of the TUDatasets (200 nodes each).



	normalize similarity	time (s)	memory usage	#iterations
Weisfeiler-Lehman kernel	0.572599	0.0131543	0	4
Graphlet kernel	0.525015	1629.39	0	4

Figure 6: Table showing the results of the comparison between Weisfeiler-Lehman Graph Kernels and Graphlet Kernel applyed on subsets (≈ 200 nodes each) of TUDataset. *Time complexity* is computed in seconds and *memory usage* is expressed in MB.

In comparing the two algorithms applied to the same graphs, we observe that both compute the same normalized similarity score and detecting somewhat degree of similarity, differing only by a negligible margin of 0.05. However, there is a substantial divergence in the computational time required by each algorithm. The second algorithm takes approximately 28 minutes to complete the analysis, while the Weisfeiler-Lehman (WL) algorithm achieves the same results in under 1 second when tested with the same number of iterations. This stark difference highlights the efficiency of the WL algorithm in processing large graphs [4].

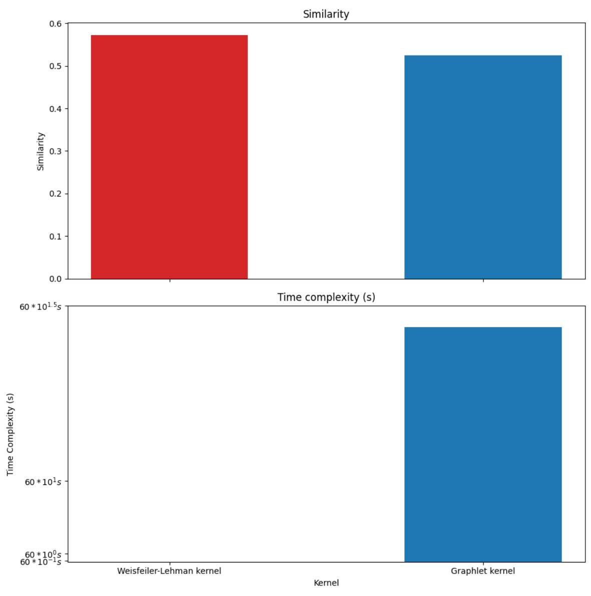


Figure 7: Comparison on accuracy and time complexity between Weisfeiler-Lehman Graph Kernels and Graphlet Kernel applyed on subsets (≈ 200 nodes each) of TUDataset with the same number of iterantion.

[4]That is also why we have made the comparison on subsets of the TUDatasets

0.3.6. Weisfeiler-Lehman Graph Kernels experiments

In this final section, we will evaluate the efficiency of the Weisfeiler-Lehman (WL) Graph Kernel by varying the number of K-hops and increasing the graph size (i.e., the number of nodes) in the TU datasets. We will analyze how these changes impact similarity, time complexity, and memory consumption.

In Figure 8, we focus on the effect of the number of K-hops, selecting seven different values that range from 3 to 100.

Subsequently, in Figure 9, we assess the efficiency of the algorithm as we increase the number of nodes in the graphs. For this analysis, we have chosen seven different sizes, ranging from 200 to 20,000 nodes. To ensure a robust evaluation, we sampled nodes from the TU datasets without replacement.

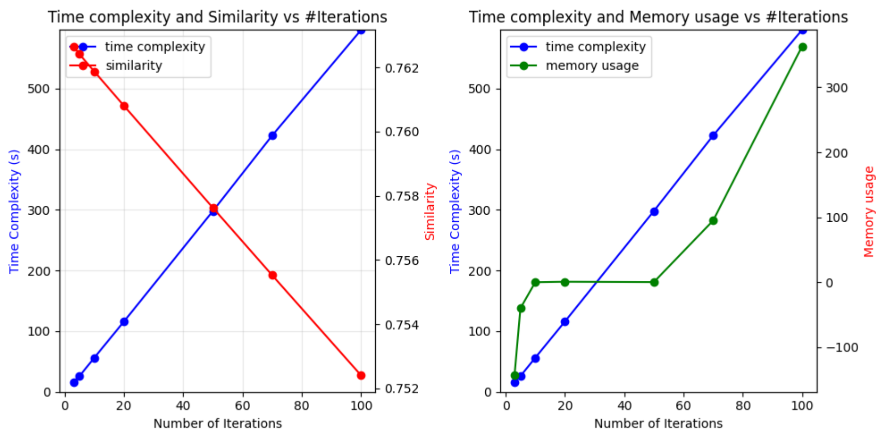


Figure 8: The graph shows the comparisons of Weisfeiler-Lehman (WL) graph kernel performance performance based on iterations on the large TUDatasets. **Left Graph**: on the Y-axis, Time Complexity (blue) and Similarity d have been tested; **Right Graph**: on the Y-axis, Time Complexity (blue) and Memory consumption (red) have been tested; For both graph, the X-axis corresponds to the number of iterations (ranging from 0 to 100).

As we can see from the graph above, the time complexity increases significantly with the number of iterations, showing a nearly exponential rise in the time taken as the number of iterations increases (left graph). However, similarity decreases as the number of iterations increase. This could mean that the WL kernel is becoming less effective in maintaining similarity between graphs over more iterations. The two lines intersect at around 50 iterations, possibly indicating an optimal balance point where time complexity and similarity are comparable.

Regarding the Memory usage, it fluctuates more (right graph). It starts low, rises steeply between 10 to 40 iterations, and then dips before rising again around 70-100 iterations. This suggests that memory usage varies more unpredictably with different iterations, possibly due to changes in graph structure or operations being performed during the WL kernel iterations.

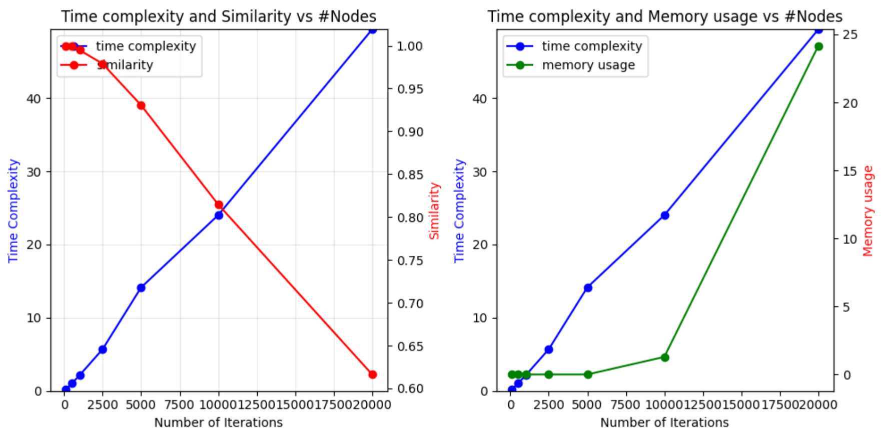


Figure 9: The graph shows experiments using the Weisfeiler-Lehman (WL) graph kernel, comparing performance with respect to the number of nodes on the large TUDatasets. **Left Graph**: on the Y-axis, Time Complexity (blue) and Similarity have been tested; **Right Graph**: on the Y-axis, Time Complexity (blue) and Memory consumption (red) have been tested; For both graph, the X-axis corresponds Number of iterations, with node counts ranging from 0 to approximately 20,000.

In the figure above, we experimented with the WL algorithm by changing the number of nodes. The results obtained are similar to those from the previous experiment. *Time complexity* increases as the number of nodes grows. The blue line starts low and rises sharply, especially as the number of nodes exceeds around 10,000, suggesting a near-exponential rise in computational cost with more nodes. *Similarity* decreases steadily as the number of nodes increases. The decreasing similarity score suggests that, with more nodes, the WL kernel becomes less effective at preserving graph similarities, possibly due to increasing structural complexity.

Regarding *Memory Usage*, it rises much more gradually. The green line starts nearly flat up until 5,000 nodes, then climbs steadily, but less steeply than time complexity. This suggests that memory usage increases at a slower rate than time complexity as the number of nodes grows. Both lines exhibit steep rises around 15,000–20,000 nodes, indicating that for larger graphs, both time and memory consumption become considerable factors.

0.4. Conclusions

During the analysis we have seen that both algorithms compute similar similarity scores and are effective isomorphism detectors for graph databases. However, as we have seen from the results in Figure 6 and Figure 7, the WL algorithm is more efficient when dealing with large graph datasets. Thus, WL represents a better Graph Kernel when it comes to scaling the database.

From the results in Figure 8, we have analyzed the trade-offs between *complexity*, *similarity*, and *memory usage*: Increasing the number of iterations in WL kernels increases computational time but may reduce the similarity score, indicating that more iterations do not always yield better results. Furthermore, memory usage doesn't increase as linearly as time complexity, possibly suggesting optimization opportunities in managing memory during iterations.

Finally, we have also investigated the trade-offs between node size and kernel performance in Figure 9: As the number of nodes increases, the time complexity grows rapidly, which is expected because larger graphs require more computational resources for kernel comparisons. The similarity decreases, possibly because larger graphs become more difficult to compare using the WL kernel's iterative approach. Memory usage increases at a slower rate compared to time complexity, suggesting that while memory is a concern, computational time is the dominant factor when scaling up the number of nodes.

In conclusion, both experiments suggest that beyond a certain point (around 7,500–10,000 nodes and 50 iterations), the performance of the WL kernel degrades significantly in terms of similarity and efficiency. This provides insight into the limitations of the WL kernel for large-scale graph comparisons.