Enriched Weisfeiler-Lehman Kernel for Improved Graph Clustering of Source Code

Frank Höppner(✉) and Maximilian Jahnke

Department of Computer Science, Ostfalia University of Applied Sciences,
38302 Wolfenbüttel, Germany
f.hoeppner@ostfalia.de

Abstract. To perform cluster analysis on graphs we utilize graph kernels, Weisfeiler-Lehman kernel in particular, to transform graphs into a vector representation. Despite good results, these kernels have been criticized in the literature for high dimensionality and high sensitivity, so we propose an efficient subtree distance measure that is subsequently used to enrich the vector representations and enables more sensitive distance measurements. We demonstrate the usefulness in an application, where the graphs represent different source code snapshots, and a cluster analysis of these snapshots provides the lecturer an overview about the overall performance of a group of students.

1 Motivation

Graphs are a universal data structure and have become very popular over recent years in various domains with structured data (e.g. protein function prediction, drug toxicity prediction, malware detection, etc.). To apply existing clustering or classification techniques to graphs, either a distance (or similarity) measure is needed, or a transformation into a vector representation for which most clustering and classification algorithms were developed for. In this paper we are concerned about repeatedly clustering graphs to understand the evolution of student's source code. As will be explained in Sect. 2, we settle on Weisfeiler-Lehman (WL) graph kernels [9] to decompose the graph into subtrees and to define a similarity function over the number of common substructures across graphs. It has been criticized, however, that WL subtree kernels produce (a) many different substructures and thus only a few substructures will be common across graphs, which establishes (b) a tendency of *being only similar to itself*. In this paper we propose to include the subtree similarity in an efficient post-processing step to tackle both problems: We exploit the fact that many of the substructures may be formally distinct but actually quite similar. By enriching the vector representations we obtain positive effects for the overall graph similarity.

M. R. Berthold et al. (Eds.): IDA 2020, LNCS 12080, pp. 248–260, 2020.
https://doi.org/10.1007/978-3-030-44584-3_20

Algorithm 1. WLSK(G, l_{i-1})

Require: graph $G = (V, E)$, label function $l_{i-1} : V \to \Sigma^*$
Ensure: returns new label function $l_i : V \to \Sigma^*$
for $v \in V$ **do**
  store node label $l_{i-1}(v)$ in s
  for $w \in V, (v, w) \in E$ in (some lexicographical) order of $l_{i-1}(w)$ **do**
    append $l_{i-1}(w)$ to s
  end for
  compress $s \leftarrow h(s)$ by applying a hash function h
  assign new label to node $v : l_i(v) \leftarrow s$
end for
return l_i

2 Related Work

2.1 Measuring Similarity Directly

A common approach to compare graphs is to calculate the *edit distance* between graphs F and G: the minimal number of steps to transform G to F. For the special case of trees, these steps consists of node deletion, node insertion, and node relabelling. A survey on tree edit distance can be found in [1], an efficient algorithmic $O(n^3)$ solution, n being the maximal number of nodes in F and G, is proposed in [2]. To adapt a tree edit distance to a specific application, there are approaches to learn appropriate cost parameters [6]. With general graphs, the editing process becomes more complicated as additional operations need to be considered (edge insertion and edge deletion). A survey on graph edit distance is given in [3]. Its computation is exponential in the number of nodes and therefore infeasible for large graphs.

2.2 Measuring Similarity Indirectly

Instead of coping with the full graph, one may decompose the graph into a set of smaller entities and compare these sets instead of the graphs. These entities may be frequent subgraphs (e.g. [8]), walks (short paths), graphlets (e.g. [10]) or subtrees (e.g. [9]). Many *graph kernel* approaches explicitly construct a vector representation, where the i^{th} element indicates how often the i^{th} substructure occurs in the graph. From this vector a kernel or similarity matrix may be calculated. Recent approaches, such as `subgraph2vec` [5], use deep learning to translate graphs into such a vector representation.

This section particularly reviews the construction of a WL subtree kernel (following [9]), as it will be foundation of the next section. The subtree kernel transforms a graph into a vector, where a non-zero entry indicates the occurrence of a specific subtree in the graph. The total number of dimensions is determined by all subtrees that have been identified in the full set of graphs.

Given a graph $G = (V, E)$, a label function $l : V \to \Sigma^*$ yields for each node $v \in V$ a label over a finite alphabet Σ. The initial labels $l_0(v)$ are provided

together with the graph G (original labels). A new label function l_i is obtained by calling $WLSK(G, l_{i-1})$, which is shown in Algorithm 1: It constructs new labels by concatenating all child labels deterministically (by processing children in some lexicographic order). A series of n WLSK calls provides a sequence of n label functions $l_0, \ldots, l_n$, where a node label $l_i(v)$ takes all children of v up to depth i into account. A label $l_i(v)$ may thus serve as a kind of fingerprint of the neighbourhood of v (hashcode). Let $L_i = \{l_i^1, l_i^2, \ldots, l_i^{k_i}\} = l_i(V)$ be the set of all different l_i-labels in G. The final vector representation of a graph is obtained from

$$\Phi(G) = \left(\#l_0^1, \ldots, \#l_0^{k_0}, \#l_1^1, \ldots, \#l_1^{k_1}, \ldots, \#l_{n-1}^1, \ldots, \#l_{n-1}^{k_{n-1}}\right)$$

where $\#l_i^j$ denotes how many nodes received the label l_i^j. Originally this approach was proposed as a test of isomorphism [11], as isomorphic graphs exhibit identical substructures (labels).

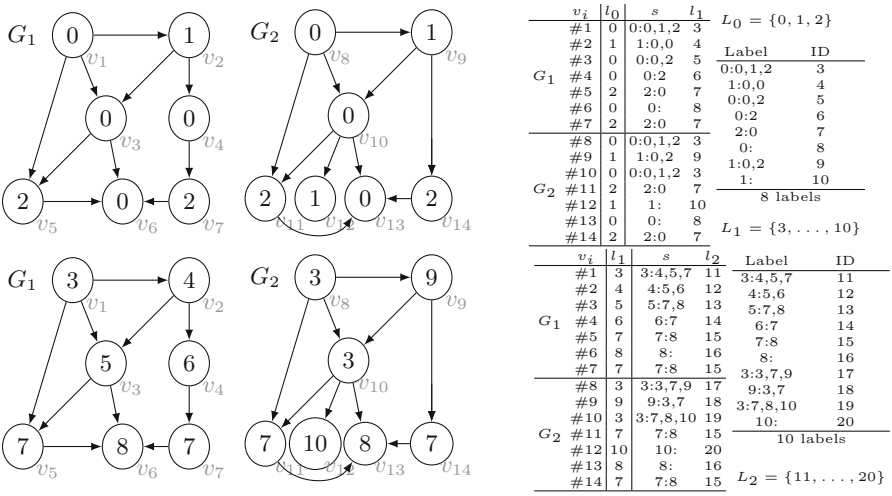
Figure 1 shows an illustrative example. On the top left we have two graphs G_1 and G_2 with nodes v_1–v_7 and v_8–v_{14}, resp. The (numeric) label is written in the node, the node identifiers are shown in gray. The table next to the graphs shows, for each node, how the new label s is constructed from the current node label and its successors. For instance, node v_1 of G_1 has label 0 and successors with labels $2, 0, 1$. Algorithm 1 creates new labels by appending the node label and the successor labels (in sorted order), which yields "$0 : 0, 1, 2$" for v_1. The rightmost table shows a dictionary, where each new label (here: $0 : 0, 1, 2$) gets a fresh ID (here: 3). Algorithm 1 refers to this step as hashing the node label into a new ID (or hashcode) – we use consecutive numbers just for illustrative purposes. Children need to be ordered deterministically to get the same hash for identical subtrees. The new label $l_1(v_1) = 3$ thus encodes a subtree of depth 1 with root 0 and children $0, 1, 2$. Once all new labels are determined (lower half of Fig. 1) the nodes v_1 and v_8 still have the same label: $l_1(v_1) = 3 = l_1(v_8)$, because their subtree of depth 1 was identical. After another WLSK iteration, however, the subtrees of depth 2 are no longer identical for v_1 and v_8, so their l_2-labels are no longer the same: $l_2(v_1) = 11 \neq 17 = l_2(v_8)$. The final vector representation for G_1 and G_2 (after 2 iterations) consists of counts for each label (from all depths):

$$\begin{aligned}
\Phi(G_1) &= (4,1,2,\ 1,1,1,1,2,1,0,0,\ 1,1,1,1,2,1,0,0,0,0) \\
\Phi(G_2) &= (\underbrace{3,2,2}_{L_0-},\ \underbrace{2,0,0,0,2,1,1,1}_{L_1-},\ \underbrace{0,0,0,0,2,1,1,1,2,1}_{L_2-\text{label counts}})
\end{aligned}$$

The vector representation $\Phi(G)$ enables us to construct a kernel matrix or apply standard clustering and classification directly.

2.3 Discussion

Measuring graph similarity indirectly is in general more efficient than direct approaches. Among the kernel approaches it has been pointed out that with some



	v_i	l_0	s	l_1
G_1	#1	0	0:0,1,2	3
	#2	1	1:0,0	4
	#3	0	0:0,2	5
	#4	0	0:2	6
	#5	2	2:0	7
	#6	0	0:	8
	#7	2	2:0	7
G_2	#8	0	0:0,1,2	3
	#9	1	1:0,2	9
	#10	0	0:0,1,2	3
	#11	2	2:0	7
	#12	1	1:	10
	#13	0	0:	8
	#14	2	2:0	7

$L_0 = \{0, 1, 2\}$

Label	ID
0:0,1,2	3
1:0,0	4
0:0,2	5
0:2	6
2:0	7
0:	8
1:0,2	9
1:	10

8 labels

$L_1 = \{3, \ldots, 10\}$

	v_i	l_1	s	l_2
G_1	#1	3	3:4,5,7	11
	#2	4	4:5,6	12
	#3	5	5:7,8	13
	#4	6	6:7	14
	#5	7	7:8	15
	#6	8	8:	16
	#7	7	7:8	15
G_2	#8	3	3:3,7,9	17
	#9	9	9:3,7	18
	#10	3	3:7,8,10	19
	#11	7	7:8	15
	#12	10	10:	20
	#13	8	8:	16
	#14	7	7:8	15

Label	ID
3:4,5,7	11
4:5,6	12
5:7,8	13
6:7	14
7:8	15
8:	16
3:3,7,9	17
9:3,7	18
3:7,8,10	19
10:	20

10 labels

$L_2 = \{11, \ldots, 20\}$

Fig. 1. Illustrative example of 2 WLSK iterations. left: initial labels l_0, middle: l_1, right: l_2

substructures, e.g. short paths (aka walks), many different graphs refer to the same point at the same point in the feature space (cf. [7]). Subtree kernels (and in particular WLSK) have been reported to be efficient[1] and well-performing in subsequent task (e.g. SVM classification). However, from the example in Fig. 1 we can also acknowledge the critique of the approach: Although G_2 has been obtained from G_1 by removing v_4 and adding v_{12} only, the vector representations are very different. Spotting differences early is good when checking for isomorphic graphs, but may be less desirable for similarity assessment (e.g. clustering). Despite the few changes, more than half of the labels occur exclusively in only one of the graphs (13 entries out of 21 that are zero in one of the two graphs). Continuous (rather than integer) features may help, as provided by some deep learning approaches, but deep learning requires a huge amount of training data, which makes them unsuitable for datasets of moderate size.

3 Enriching WL Subtree Kernels

Revisiting Fig. 1, node v_3 of G_1 and node v_{10} of G_2 differ only by a missing node labelled '1'. From the different l_1-hashcodes for both nodes (5 for v_3 and 3 for v_{10}) we cannot conclude what they have in common. Secondly, node v_2 of G_1 and v_9 of G_2 are similar in the sense that nodes labelled 0 and 2 can be reached, only in G_1 there is an intermediate node v_4. If we accept that node pairs (v_2, v_9) and (v_3, v_{10}) are somewhat similar, this should then positively affect the l_2-similarity of v_1 and v_8, too. We want to take this kind of similarity into account without

[1] The only necessary data structure is a hash table that collects how often each node label occurred.

sacrificing the efficiency of WLSK. Instead of integer features (subtree counts) we introduce continuous features to better reflect a partial matching of subtrees. We stick to the WLSK construction, but propose a post-processing step, which replaces the zero entries in the vector representation. As many subtrees (with different hashcodes) are in fact similar, we obtain highly correlating dimensions which are safe to remove and thus reduces the dimensionality. We optionally apply dimensionality reduction to arrive at a vector of moderate size.

3.1 Subtree Similarity

Given a graph $G = (V, E)$, let $L_i = l_i(V)$ be the set of all hashcodes for subtrees of depth i (cf. tables on the right of Fig. 1). The hashcodes compress the newly constructed node labels, but no longer contain any information about the subtree. So we track this information in tables: For all occurred hashcodes $h \in L_i$, we denote the root node label by $r_h \in L_{i-1}$ and the multiset of successor labels by $S_h \subseteq L_{i-1}$. (*Example:* For $h = 11 \in L_2$ in Fig. 1 we have $r_h = 3$ and $S_h = \{4, 5, 7\}$.)

Next we define a series of distance functions $d_i : L_i \times L_i \to \mathbb{R}$ to capture the distance between subtree hashcodes of the same depth i. We start with a distance d_0 for the original graph node labels. In absence of any background knowledge we use for the initial level

$$d_0(h, h') := \begin{cases} 0 & \text{if } h = h' \\ 1 & \text{otherwise} \end{cases}, \tag{1}$$

but generally assume that some background information can be provided to arrive at meaningful distances for the initial node labels.

For non-trivial subtrees (that is, $i > 0$) we recursively define distance functions $d_i(h, h')$. It is natural to define the distance as the sum of distances between root and child nodes. This requires to assign child nodes of h uniquely to child nodes of h', which is provided by a bijective function $f : S_h \to S_{h'}$:

$$d_i(h, h') := \underbrace{d_{i-1}(r_h, r_{h'})}_{\text{root node distance}} + \underbrace{\min_{f \in B(S_h, S_{h'})} \sum_{k \in S_h} d_{i-1}(k, f(k))}_{\text{distance of best subtree alignment}} \tag{2}$$

Here $B(S, T)$ denotes the set of bijective functions $f : S \to T$. The first term measures the distance between the root node labels and the second term identifies the minimal distance among all node assignments. Finding the assignment with minimal distance is known as the *assignment problem*, which has well-known solutions and we adopt the Munkres algorithm for this task [4].

We are likely to deal with unbalanced assignments, that is, different numbers of children for h and h'. A bijective assignment requires $|S_h| = |S_{h'}|$, so we add the necessary number of *missing nodes* (denoted by $\perp$) to the smaller multiset.[2]

[2] More formally $B(S, T)$ is the set of bijective functions $f : S' \to T'$ where $|S'| = k = |T'|$, $S \subseteq S'$, $T \subseteq T'$, S' has $k - |S|$ (and T' has $k - |T|$) additional $\perp$ elements.

d_0

	0	1	2	⊥
0	0.0	1.0	0.5	1.0
1		0.0	1.0	1.0
2			0.0	1.0
⊥				0.0

(i)

	0	0
0	0.0	0.0
2	0.5	0.5

(ii)

	0	2	⊥
0	0.0	0.5	1.0
1	1.0	1.0	1.0
2	0.5	0.0	1.0

d_1

-	3	4	5	6	7	8	9	10	⊥
3	0.0	2.5	1.0	.	2.5	.	2.0	.	4.0
4		0.0	1.5	.	2.0	.	0.5	.	3.0
5			0.0	.	1.5	.		.	3.0
6				.	1.0	.		.	2.0
7					0.0	.		.	2.0
8						.		.	1.0
9							0.0	.	3.0
10								.	1.0
⊥									0.0

(iii)

	4	5	7
3	2.5	1.0	2.5
7	2.0	1.5	0.0
9	0.5	0.0	1.0

Fig. 2. Left: A priori distances d_0 between labels of L_0. Case (i): Assignment matrix for d_1 distance of $l_1(v_2) = 4$ and $l_1(v_9) = 9$. Case (ii): Assignment matrix for d_1 distance of v_3 ($\{0, 2\}$) and v_{10} ($\{0, 1, 2\}$). Right: Derived d_1-distances from case (i) and (ii). Case (iii): Assignment matrix for d_2 distance of v_0 ($\{4, 5, 7\}$) and v_8 ($\{3, 7, 9\}$) (Color figure online)

We extend the distance d_0 to the case of missing nodes, which corresponds to an additional row/column in the d_0-matrix (see d_0 example matrix in Fig. 1(left)). Again, these $\perp$-distances may be an arbitrary constant or specifically provided for each label $h \in L_0$ using background knowledge. Then Eq. (2) extends naturally to $\perp$-values:

$$d_i(h, \perp) := d_{i-1}(r_h, \perp) + \sum_{k \in S_h} d_{i-1}(k, \perp) \tag{3}$$

Figure 2 shows an example. The leftmost table shows the d_0-distances between original node labels (cf. Fig. 1: $L_0 = \{0, 1, 2\}$), including the case of a missing label $\perp$. For the sake of illustration we assume a distance of $\frac{1}{2}$ for the label pair $(0, 2)$. Consider the comparison of v_2 and v_9 for depth-1 subtrees: $d_1(h, h')$ with $h = l_1(v_2)$, $h' = l_1(v_9)$. Both root nodes are identical ($r_h = r_{h'} = 0$), but the multisets of successors are not ($S_h = \{0, 0\}, S_{h'} = \{0, 2\}$). Matrix (i) shows the distance matrix for the assignment problem: all nodes of h' (rows) have to be assigned to a node of h (columns). As the child nodes represent l_0-hashcodes, we take the distances from the d_0 table. An optimal assignment is marked in red and we obtain a distance $d_1(h, h') = 0 + (0 + \frac{1}{2}) = \frac{1}{2}$. Matrix (ii) shows a second example for the d_1 comparison of v_3 vs v_{10}: As v_{10} has three children but v_3 only two, we introduce one $\perp$-element to obtain a square matrix. The optimal assignment is shown in red, the d_1-distance becomes 1.0. Both examples contribute two values to the d_1-distance (fourth matrix), from which we may then calculate, e.g., $d_2(l_2(v_1), l_2(v_8)) = 0 + (\frac{1}{2} + 1 + 0) = 1.5$ (matrix (iii)).

3.2 Updating Vector Representations

Once the WLSK algorithm has been executed, we determine all d_i-distances from the l_i-labels alone (without revisiting the graphs). Then we update the vector

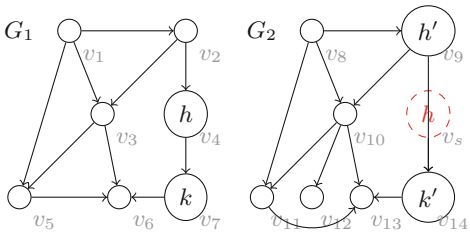


Fig. 3. Insertion of nodes to compensate side-effects of superfluous nodes. (Color figure online)

representations of all graphs, the zero entries in particular. Suppose $\mathbf{x}$ is a vector representation of G and $\mathbf{x}_h = 0$ for some $h \in L_i$, which means that subtree h is not present in G. Among the subtrees that *do occur* in G we can now find the one most similar to $h' \in L_i$ (smallest distance $d_i(h, h')$) and replace $\mathbf{x}_h$ by

$$\mathbf{x}_h \leftarrow k(d_i(h, h')) \cdot x_{h'}$$

where $k : \mathbb{R}^+ \to [0, 1]$ is a monotonically decreasing function that turns distances into similarities with $k(0) = 1$. The multiplication with $x_{h'}$ accounts for the fact that h' may occur multiple times in G. We used $k(d) = e^{-(d/\delta)^2}$, where δ is a user-defined threshold.

3.3 Compensating Superfluous Nodes

We say v is an *superfluous node* if it is just a *stopover* on the way to yet another node, but does not contribute to the graph structure itself, that is, if the in- and out-degree of v is 1. In Fig. 1 the node v_4 in G_1 is such a superfluous node. In some applications nodes with certain labels may occur occasionally, but do not carry any important information. Their existence/absence should therefore affect the graph similarity not too much.

The discussed distance measure can cope with such differences when comparing, e.g., the subtree of v_2 with that of v_9. But if we consider v_4 as an superfluous intermediate node, it brings another undesired effect: It may introduce completely new subtrees which are not present in other graphs. In the example of Fig. 1 the node v_4 introduces subtrees with hashcodes 6 (at depth 1) and 14 (at depth 2), which are not present in G_2. When measuring the similarity of G_1 and G_2, such subtrees make the graphs appear less similar.

We address such cases by considering the insertion of a superfluous node in our distance calculation. Figure 3 shows the situation once more: To enrich the vector representation of G_2 we seek a closest match for label h. According to Sect. 3.1 we consider, amongst others, the node v_9 with label h' as a candidate. With both nodes having a single child only, finding the optimal bijective assignment f is trivial ($f(k) = k'$) and Eq. (2) boils down to $d_{i-1}(r_h, r_{h'}) + d_{i-1}(k, k')$. Now we additionally consider the *insertion* of a superfluous node v_s with the

same label as v_4, as shown in Fig. 3 (red). Note that a hashcode $l_i(v_s)$ for the newly inserted node was not necessarily generated earlier. How would the distance between a node v_4 and v_s evaluate? According to (2) we have

$$d_i(l_i(v), l_i(v_s)) = d_{i-1}(l_{i-1}(v), l_{i-1}(v_s)) + d_{i-1}(k, k')$$

The second part consists of a single term because both nodes have a single child only. Note that it does not depend on v_s. Substituting the first term repeatedly by its definition eventually leads us to

$$d_i(l_i(v), l_i(v_s)) = \underbrace{d_0(l_0(v), l_0(v_s))}_{0 \text{ by construction}} + \sum_{j=0}^{i-1} d_j(l_j(k), l_j(l)) \tag{4}$$

The level-0-distance to the newly inserted node is 0 by construction, however, we replace it by a penalty term $d_I(l_0(v))$ to reflect the fact that we had to insert a new node. As with $d_0(\cdot,\cdot)$ we assume that $d_I(\cdot)$ can be derived meaningfully from the application context: If, for instance, nodes with a certain label h are optional, we choose a low insertion distance $d_I(h)$ and may otherwise set $d_I(h) = \infty$ to prevent undesired insertions.

We thus arrive at a distance $d_i^*(h, h')$ for the insertion of a superfluous node

$$\min \begin{cases} \min\{d_I(r_h), d_I(r_{h'})\} + \sum_{j=0}^{i-1}(l_j(k), l_j(k')) & \text{if } S_h = \{k\} \wedge S_{h'} = \{k'\} \\ \infty & \text{otherwise} \end{cases} \tag{5}$$

which yields ∞ if the prerequisites of a superfluous nodes are not given and considers node insertion on both sides (inner min-term). The original distance (2) may then be replaced by $\min\{d_i(h, h'), d_i^*(h, h')\}$ to reflect the occurrence of superfluous nodes appropriately. These changes can be handled during the precalculation of the distance matrices, the vector enrichment remains unchanged.

3.4 Complexity

Enriching the vector representations requires two steps: (1) The calculation of all distance matrices d_i requires to calculate $\sum_i |L_i|^2$ entries. For each entry we have to solve an assignment problem, which is $O(d^2 \log d)$ where d is the maximal node degree. The method is therefore unattractive for highly connected graphs. But many applications with large graphs have a bounded node degree. (2) Secondly, the vector representations $\mathbf{x}$ of all n graphs need to be enriched. This takes $O(m_z \cdot m_{nz})$ for each graph, where m_z (resp. m_{nz}) is the number of entries in $\mathbf{x}$ with zero (resp. non-zero) entries: for each 0-entry in $\mathbf{x}$ we have to find the most similar 1-entry. The number of all labels from all graphs ($m = \sum_i |L_i|$) is much larger than the number of nodes in a single graph, whereas m_{nz} is bounded by the number of nodes in a single graph. With $m_{nz} \ll m_z$ we may consider m_{nz} as a constant (max. no. of nodes) and arrive at $O(n \cdot m)$ for the vector enrichment.

Exercise: Write a function to count the number of entries in an integer array having a 3 at the last digit.

```
public static int count3(int [] x){
  int count=0;
  int i=0;
  int zaehler=0;
  while (i < x.length) {
    zaehler=x[i] % 10;
    if (zaehler == 3) {
      ++count;
} } }
```

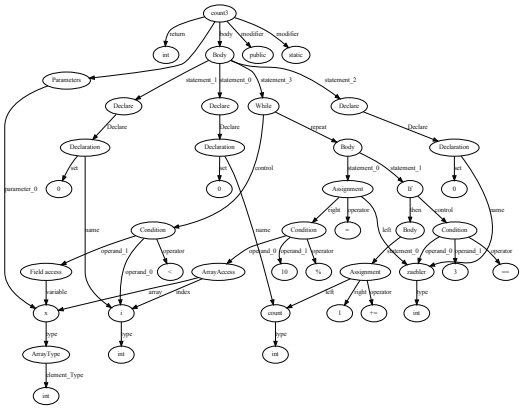


Fig. 4. Example of a source code snapshot and its graph representation. The student has not yet finished the solution at this stage/snapshot, the return statement is still missing.

4 Application

We demonstrate the usefulness of the proposed modification in an application from computer science education. The increase in the number of CS students over the last years calls for tools that help lecturers to assess the stage of development of a whole group of students – rather than inspecting the solutions one by one. Our dataset consists of editing streams from the students source code editor (for selected exercises of an introductory programming course using Java). In our preliminary evaluation we have about 30–50 such streams per task. We extract snapshots of the code whenever a student starts to edit a different code line than before. (Many snapshot thus do not represent compileable code.) The goal is to compare editing paths against each other, for instance, to identify the most common paths or outliers. We replace the textual representation of the source snapshot by a graph capturing the abstract syntax tree and the variable usage, as can be seen in the example of Fig. 4. We want to cluster the snapshots and to construct a new graph where nodes correspond to clusters (of code snapshots) and edges indicate editing paths of students. For the experiments we applied some preprocessing (e.g. variable renaming in the graph) and assigned low insertion costs to expression- and declaration-nodes, because students may phrase conditions quite differently. Our use case for superfluous nodes (Sect. 3.3) are code blocks ({ }), which are optional if the code within the block consists of a single statement only (e.g. the `++count` in Fig. 4).

4.1 Effect on Distances

To measure the effect of the enriched kernel we have manually subdivided a set of snapshots into *similar* and *dissimilar snapshots*. In a clustering setting we want

Table 1. Effect of vector enrichment on distances.

Kernel depth d	σ	Standard vector	Enriched vector
		f	f
2	$3 \cdot d$	$\frac{8.43-4.76}{2.44} = 1.50$	$\frac{7.03-1.75}{0.97} = 5.44$
3	$3 \cdot d$	$\frac{9.48-6.43}{3.06} = 0.99$	$\frac{11.18-4.24}{2.36} = 2.93$
4	$3 \cdot d$	$\frac{9.85-7.82}{3.46} = 0.58$	$\frac{14.67-6.63}{3.99} = 2.01$
5	$3 \cdot d$	$\frac{9.91-8.52}{3.64} = 0.38$	$\frac{15.07-8.34}{4.54} = 1.48$

the modification to carve out clusters more clearly. We therefore compare the mean distance μ_w (and variance σ_w) *within* the group of similar graphs against the mean distance μ_b (and variance σ_b) *between* both groups. By the factor f we denote the size of the gap between both means in multiples of the within-group standard deviation σ_w, that is, $f = \frac{\mu_b - \mu_s}{\sigma_s}$. The factor f may be considered as a measure of separation between the cluster of similar graphs and the remaining graphs. From Table 1 we find that the enriched representation consistently yields higher values of f for the enriched than for the standard vector representation.

4.2 Dimensionality

New node labels are introduced for every new subtree, which introduces a high dimensional vector representation that has been identified as problematic in the literature (Sect. 2.3). Enriching the vector representation can help to overcome this problem, because labels with minor changes will receive similar (enriched) entries. For instance, a dataset with 718 code snapshot graphs generated as many as 5179 different subtree labels (depth 3). After enrichment we identified the number of attributes that might be removed from the dataset because it contains a highly correlating attribute already. This leads to a substantial reduction in the number of columns: Depending on the Pearson correlation threshold of 0.9/0.95/0.99 as much as 77%/68%/55% of the attributes can be discarded.

4.3 Code Graph Clustering

To reduce the dimensionality further, a principal component analysis (PCA) may be applied. Figure 5 shows the scatter plot of the principal components (PC) #2 against PC #1, #3 and #4 for the standard representation (top) and the enriched vectors (bottom). The colors indicate cluster memberships from a mean shift clustering over 4 principal components. Note that, by construction of the dataset, we do not expect the source code snapshots to fall apart completely in well separated clusters, because the data represents the evolution towards a final solution, snapshots differ by incremental changes only. In the standard case the data scatters more uniformly and less structured (left; PC1 vs PC2), while the enriched data shows two long-stretched clusters that reflect a somewhat linear code evolution for two different approaches to solve the exercise, which

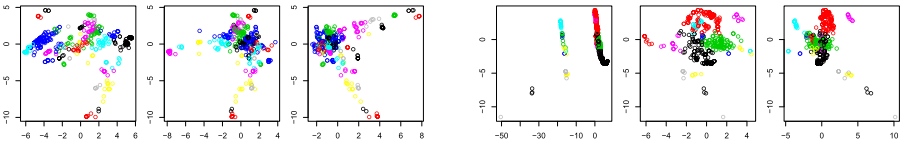


Fig. 5. Principal component #2 versus principal component #1, #3 and #4 for standard (left) and enriched (right) vectors. (Color figure online)

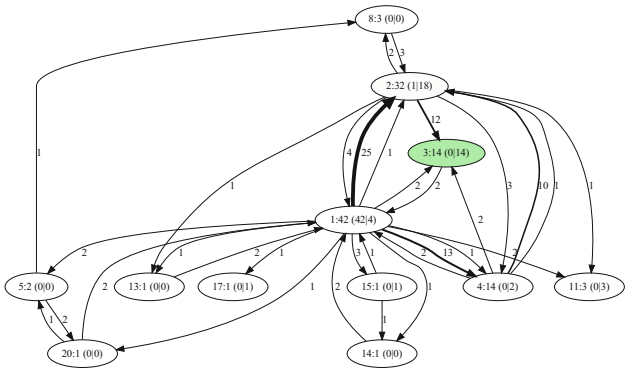


Fig. 6. Snapshot evolution for a group of students: Nodes represent clusters, edges represent snapshot transitions. (Color figure online)

corresponds much better to our expectation. When taking additional component into account (PC3), the scatterplot in the middle (PC2 vs PC3) offers a clearer structure for the enriched data (e.g. the separation of the curved red cluster at the top) than the original data.

Figure 6 shows how the clusters are used in the context of our application. Each cluster (like those in Fig. 5, but for a different exercise) corresponds to a node in this graph. Whenever a student changes the code and thereby moves to a different cluster, a (directed) edge is inserted. The number of students who have followed a path is written nearby the edge. Clusters that have only one incoming and one outgoing edge are not shown for the sake of brevity. The green color indicates the degree of unit-test fulfillment. The node labels $a : b(c|d)$ carry information about the cluster id a, number of students b that came across this node, number of students c (resp. d) who started (resp. ended) in this node. From this example the lecturer can immediately recognize that 42 students start in cluster #1, from where most students (25) transition to cluster #2 and 10 more students reach the same cluster via cluster #4 as an intermediate step. Cluster #2 does not yet correspond to a perfect solution, but only 12 students manage to reach the green cluster #3 from cluster #2. Other clusters and edges have much smaller numbers, they cover exotic solutions or trial-and-error approaches. The graph provides a good overview about the students performance as a group.

5 Conclusions

Weisfeiler-Lehman subtree kernels can be used to transform graphs into a meaningful vector representation, but suffer from high dimensionality and sparsity, such that the similarity assessment is limited. We overcome both problems by taking the subtree distances into account – which are simpler to assess than general tree distance, because only subtrees of equal depth need to be considered. Based on the subtree distance we enrich the zero entries of graph vectors and improve the similarity assessment. A removal of highly correlating attributes reduces the dimensionality considerably. The modifications turned out to be advantageous in a use case of source code snapshot clustering.

References

1. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. **337**, 217–239 (2005)
2. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. In: International Colloquium on Automata, Languages, and Programming, pp. 146–157 (2007)
3. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Anal. Appl. **13**(1), 113–129 (2010)
4. Munkres, M.: Algorithms for the assignment and transportation problems. J. Soc. Ind. Appl. Math. **5**(1), 32–38 (1957)
5. Narayanan, A., Chandramohan, M., Chen, L., Liu, Y., Saminathan, S.: subgraph2vec: learning distributed representations of rooted sub-graphs from large graphs. In: Workshop on Mining and Learning with Graphs (2016)
6. Paassen, B.: Metric learning for structured data. Ph.D. thesis, Bielefeld University (2019)
7. Ramon, J., Gärtner, T.: Expressivity versus efficiency of graph kernels. In: Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences, pp. 65–74 (2003)
8. Seeland, M., Girschick, T., Buchwald, F., Kramer, S.: Online structural graph clustering using frequent subgraph mining. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010. LNCS (LNAI), vol. 6323, pp. 213–228. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15939-8_14
9. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. J. Mach. Learn. Res. **12**, 2539–2561 (2011)
10. Shervashidze, N., Vishwanathan, S., Petri, T.H., Mehlhorn, K., Borgwardt, K.M.: Efficient graphlet kernels for large graph comparison. In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS) (2009)
11. Weisfeiler, B., Lehman, A.: A reduction of a graph to a canonical form and an algebra arising during this reduction. Nauchno-Technicheskaya Informatsia **2**(9), 12–16 (1968)