Institute of Computer Science
University of Bern

Bachelor Thesis

Matching Graphs with Enriched Node Labels using Weisfeiler-Lehmann Hashes

---

Raphael De Gottardi
18-122-929

Supervisors:
Prof. Kaspar Riesen
Anthony Gilloz

Oct. 2022

Abstract

Graphs are a commonly known data structure which can be used to represent all kinds of data as for example molecules, networks, routes or relationships. Their flexibility and ability to represent structure lets them stand out in comparison to statistical approaches (e.g. feature vectors). Recently, many applications which only had a statistical algorithm solution have been adapted to work with structural data too. An often used algorithm involving graphs is a dissimilarity measure called Graph Edit Distance (GED). It is based on assigning costs to the edit operations needed to transform a graph into another. This cost can then be used to perform Pattern Recognition tasks, an example is classification. This thesis explores the potential of a graph node label enrichment using Weisfeiler-Lehman hashes. These combine the labels of the adjacent node labels to a hash value which possibly can capture some structural information. This hash is computed and attached to each original label in order to enrich the data. The impact on the classification performance is evaluated using a k Nearest Neighbor (kNN) classifications. Most of the approaches of such enrichment lead to worse results, where the additions to the labels can be interpreted as simply adding noise. Some approaches in fact lead to a noticeable improvement in classification performance which underlines the findings made in the process. The findings allow a potential estimation of the method and paved the way for further experiments.

Contents

List of Figures

Chapter 1

Introduction

Problem solving is a main discipline in Computer Science and has seen many approaches for many different problems. A recent approach to problem solving is artificial intelligence, short AI. This field of research was founded in the 1950's, the birth of AI refers to the Darmouth workshop of 1956 [1]. AI grew rapidly since then and split up into many fields who are now well known. Examples for its subfields are Neural Networks, Natural Language Processing, Computer Vision as well as Machine Learning [2]. AI applications are studied as well as teached at the university of Bern and can be found in many different subject areas. An example is the newly founded masters degree in 'AI in medicine' at the CAIM [3]. The Institute of Computer Science hosts a group for Pattern Recognition as well as one for Machine Learning.

Machine Learning aims to use data to be able to solve a given task. In Machine Learning, two characteristics are important to split up the field into four categories: supervised vs unsupervised learning and continuous vs discrete data [4]. Supervised learning means that there is labeled data in advance which can be used for the learning. The discreteness of the data refers to the prediction one aims to get. For a yes/no question, one needs discrete methods whereas for an (e.g. age-) estimation, continuous models need to be chosen. This thesis will focus on classification, which is a form of supervised learning which leads to discrete results (assignment to classes). The other three fields are regression, dimension reduction and clustering, we will not further focus on those.

Classification can be used in the field of Pattern Recognition. Pattern Recognition aims at a variety of problems the human brain faces every day [5]. Understanding a word, reading a sign or recognising a friend are examples for recognition tasks. There are two approaches to such problems when using machines: structural and statistical.

The statistical approach uses feature vectors to represent data. This has the major advantage of a large mathematical framework of efficient operations. There is a large amount of algorithmic tools available for such data [6]. The downside of this approach is twofold. One is that the vector size must be predefined, regardless of the dynamics of the pattern the data represents. The other is that the simplicity of such vectors limits the possibility of describing higher order relationships in the data.

Structural pattern representation aims to overcome those limitations. The strengths and weaknesses are basically reversed. The structural approach is more flexible and can represent complex patterns. However, it increases the complexity of many al-

gorithms [5]. Structural approaches of Pattern Recognition use the arrangement of data to face recognition tasks [7]. Graphs are an example for structural data representation. They are the underlying structure behind the field of graph based Pattern Recognition, short GbPR.

Fig. 1.1 gives a brief overview of the subdivision of the discussed fields. It is important to notice that both the illustrated trees are highly overlapping. For example, Pattern Recognition often uses AI methods but can also be seen as a subfield of Machine Learning (ML). this thesis can be located at the intersection of both the fields and will use concepts from both.

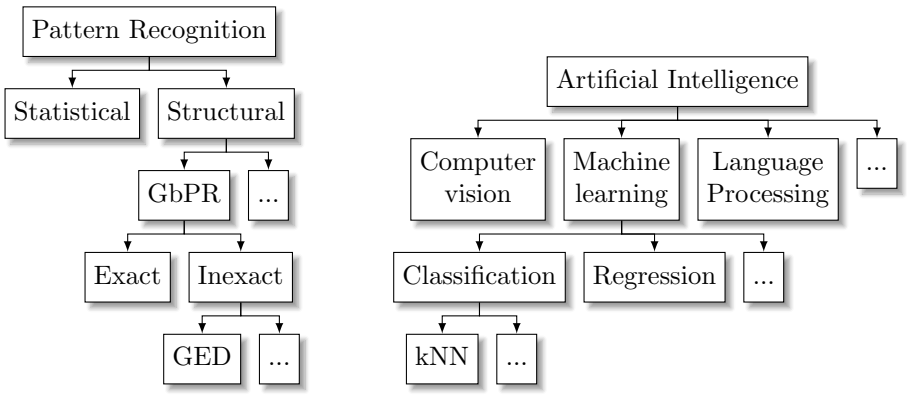


Figure 1.1: Hierarchical arrangement of subfields in PR and AI. Note that each level includes many more fields and that both trees are overlapping in practice.

A common Pattern Recognition task using ML is the classification of unknown samples into groups. For that, the kNN is a simple and frequently used algorithm. It uses a dissimilarity measure to assign an unknown sample to the group it is most similar to. For that, it compares the sample to the training set and computes a dissimilarity to each train-object. The sample is then classified to the group with the least dissimilarity. This is then done with a large amount of samples. As a result, it returns a classification accuracy which is highly dependent of the quality of the dissimilarity measure.

Graphs are well known structures in Computer Science and are used for structural Pattern Recognition tasks [8]. They consist of edges and nodes which can both be labeled. There are many ways to compare and classify graphs, in GbPR graph dissimilarity measures are developed. They are the foundation for many PR tasks. Over the past decades, many dissimilarity measures for graphs have been proposed and introduced. They can be subdivided into exact and inexact matching techniques [9].

Most of the exact graph matching algorithms are based on some form of tree search. The underlying idea is to iteratively expand a partial match until eventually it results in a complete match or can not be expanded anymore. The most popular example for such algorithms has been proposed by Ullmann in 1976 [10]. Exact graph matchings (with few exceptions) require exponential time in the worst cases. Our scope is on the inexact methods as they are less rigid and thus closer to practical applications.

Inexact methods assign costs to any pair of graphs, so the algorithm aims to find

a mapping which minimizes such cost. Suboptimal graph matchings find a minimum but are not able to ensure that it will be the global minimum. Usually the found (local) minimum and the global one are not far from each other (without guarantees) [9]. The advantage is that in return, they are substantially faster. Simultaneously to the exact matching methods, more practical inexact algorithms were developed. A big part of them is also based on tree search, where the cost can be assigned to the matching obtained so far. The first such algorithm was proposed by Tsai and Fu in 1979 [11]. A more recent tree search based algorithm uses the A* algorithm to find an optimum [12]. The use of A* leads to the concept of bipartite matching. for that, the edit operations of a transformation of a graph into another form a bipartite graph. This allows faster computation based on known algorithms for optimisation.

A commonly used inexact dissimilarity measure for graphs is the Graph Edit Distance GED. Between any two graphs one can find a list of edit operations to convert one into another. such edit operations are typically insertions, deletions and substitutions. Assigning a cost to the operations leads to a total cost for the conversion. This cost can then be used as a dissimilarity measure also called distance. The GED is found in applications such as image analysis, biometrics, bio/chem-inforatics [13].

As the exact computation of the GED is in NP, subobtimal methods must be chosen for applications. One is the bipartite Graph Edit Distance by K. Riesen et. al. [14]. Based on the bipartite GED, this thesis focuses on modifying the dissimilarity calculation by using Weisfeiler-Lehmann (WL) hashes.

The Weisfeiler-Lehmann algorithm is a long known isomorphism test for graphs [15]. For every node, it calculates a value which is based on the neighbor information of this node. The aim of the thesis is to evaluate if such values, which can be stored as a hash, can be used to enrich the GED computation (named WL-GED). Further, also to estimate if such a WL-GED can exceed the classification performance of the original GED. Nino Shervashidze proposed a method to include such hashes in the dissimilarity computation for kernels [16]. His Proposal for merging the WL-algorithm with kernels inspired this thesis. The idea is to enrich the original node labels with the hashes and to use the hashes to calculate a more specific dissimilarity measure. The results depend on how much structural information such hashes can encode and how well the GED algorithm can interpret such information. A kNN classification can reliably quantify the quality of the dissimilarity measure as its classification success is based on it.

There are many ways to assign a cost to such hashes and to embed this method into the GED. The remainder of this thesis describes the exploration of such cost models and their impact on the classification performance using a kNN classification. The most important findings are done in the pioneering experiments. Those form the basis for a final experiment which is later presented in the results section. To start, a detailed foundation of the most central concepts is given.

Chapter 2

Theoretical Background

The following definitions and explanatory notes serve as a foundation for understanding the concepts and methods used for the thesis. Graphs are formally defined in the same way as in the book from K. Riesen [5] as well as the Graph Edit Distance GED, which is the dissimilarity measure examined for this thesis. Its working principle is shown on a toy example using common molecules represented as graphs. This example is further used to show the concept of the Weisfeiler-Lehman algorithm and how it can be used to enrich the nodes of a graph.

Such an enriched graph opens the door for many possible adaptations of the GED (denoted as WL-GED) which will later be discussed in the methods section. To examine if such a node enrichment can lead to an improvement for tasks where such a distance is commonly used, the concept of kNN-classification is introduced, showing current state of research data and examples.

2.1 Graphs

First we properly define a graphs which consist of nodes, edges and corresponding labels.

Definition 2.1.1 (Graph)**.** Let L_V and L_E be finite or infinite label sets for nodes and edges, respectively. A *graph* is a four-tuple $g = (V, E, \mu, \nu)$, where

- V is the finite set of nodes,
- $E \subseteq V \times V$ is the set of edges,
- $\mu : V \to L_V$ is the node labeling function, and
- $\nu : E \to L_E$ is the edge labeling function.

The size of a graph g is denoted by $|g|$ and is defined as the number of nodes. By assigning the empty label ϵ to all of the nodes, the graph is referred to as *unlabeled*. By assigning ϵ to every edge, we consider the edges as *unweighted*.Edges are given by pairs of nodes $(u, v) \in E$ where $u, v \in V$. This thesis only considers *undirected* graphs, which means that if $(u, v) \in E$ then also $(v, u) \in E$. Also, only fully labeled but unweighted graphs will be considered. This means the edge labeling ν always points at an empty label. To simplify the notation, a graph will consist only of three-tuple $g = (V, E, \mu)$ from now on.

Two nodes connected by an edge are called *adjacent*. A graph is termed *complete* if all pairs of nodes are adjacent. The *degree* of a node u is the number of adjacent nodes. Graphs can be used to represent a variety of structural data, examples of fields where graphs are used to represent data are chemoinformatics, animal social networks or infrastructure networks [17].

For demonstration purposes we will use the graph representation of a carbonic acid molecule H_2CO_3 as an example . Its representation is shown below (Fig. 2.1). For the sake of being able to address the individual nodes, atoms with multiple appearances are marked with apostrophes '. The type (single, double or others) of bound between the atoms is neglected so every edge represents some form of covalent bound.

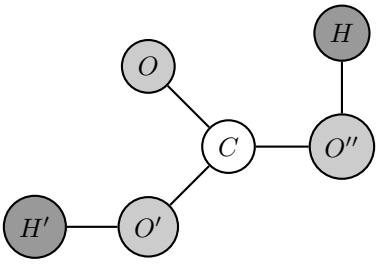


Figure 2.1: Carbonic acid H_2CO_3, an example of a graph of interest (unweighted, undirected, labeled)

By removing parts of the graph we obtain a *subgraph*

Definition 2.1.2 (Subgraph)**.** Let $g_1 = (V_1, E_1, \mu_1)$ and $g_2 = (V_2, E_2, \mu_2)$ be graphs. Graph g_1 is a *subgraph* of g_2, denoted by $g_1 \subseteq g_2$, if

1. $V_1 \subseteq V_2$
2. $E_1 \subseteq E_2$
3. $\mu_1(u) = \mu_2(u)$ for all $u \in V_1$

Subgraphs will be important for understanding the concept of the Weisfeiler-Lehmann algorithm and how it can be used. An example for a subgraph of carbonic acid could be CO_2.

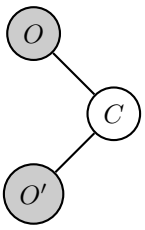


Figure 2.2: Example of a subgraph (CO_2) of carbonic acid (Fig. 2.1)

Given the basic definitions of graphs, the next step is to look for a dissimilarity measure for such graphs. The statistical approach to find such a dissimilarity, which can also be referred to as distance, is to use some form of Minkovski Metric (Eq.2.1). It defines a distance between two n-dimensional feature vectors x and y:

$$||x - y||_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \tag{2.1}$$

The order p can be chosen. p = 2 corresponds to the Euclidean distance. The search for a dissimilarity measure leads to the *graph comparison problem*

Definition 2.1.3 (Graph comparison problem)**.** Given two graphs g_1 and g_2 from a graph domain ζ the graph comparison problem is given by defining a function

$$d : \zeta \times \zeta \to \mathbb{R} \tag{2.2}$$

such that $d(g_1, g_2)$ quantifies the dissimilarity of g_1 and g_2.

Graph comparison is commonly solved via a particular *graph matching* algorithm, which maps similar substructures of g_1 to similar substructures of g_2. Using such an algorithm, a dissimilarity score $d(g_1, g_2)$ can be computed in various ways. Some have been mentioned in the Introduction.

Graph matching is topic of numerous recent studies [18]. It is usually split into two categories, exact and inexact graph matching. Exact matching uses the concept of graph or subgraph isomorphisms. An isomorphism is a bijective function defined below, its existence proves the identity of two graphs g_1 and g_2.

Definition 2.1.4 (Graph Isomorphism)**.** Let us consider two graphs g_1 and g_2 A graph isomorphism is a bijective mapping $f : V_1 \to V_2$ satisfying

1. $\mu_1(u) = \mu_2(f(u)) \forall u \in V_1$,
2. $e_1 = (u, v) \in E_1 \Rightarrow e_2 = (f^{-1}(u), f^{-1}(v)) \in E_2 \forall e_1 \in E_1$
3. $e_2 = (u, v) \in E_2 \Rightarrow e_1 = (f^{-1}(u), f^{-1}(v)) \in E_1 \forall e_2 \in E_2$

This family of problems however is known to be in NP. Subgraph isomorphism, whose solution is required for an exact dissimilarity measure, is shown to be NP-complete [19]. Subgraph isomorphism is a weaker form of node mapping but harder to solve than isomorphism as one has to check more than just whether some permutation of g_1 is identical to g_2. Based on maximum common subgraph (MCS) and minimum common subgraph (mcs), an exact dissimilarity measure d_{UGU} can be defined as proposed in [20].

$$d_{UGU}(g_1, g_2) = |MCS((g_1, g_2)| - |mcs(g_1, g_2)| \tag{2.3}$$

Exact graph matching has the main advantages of a stringent definition and solid mathematical foundation. However, such constraints can become too rigid for practical applications which lead to a large number of inexact graph matching methods in the last few decades [5]. Error tolerant (inexact) methods relax the concept of exact graph matching in three points:

- Mapping of nodes with different labels is possible
- Node mappings are allowed to violate the edge structure (while exact matching is edge-preserving)
- Error tolerant matching allows deletion (for the first graph) and insertion (for the second graph) of some nodes, which means that not all the nodes need to be involved in a bijective function.

This means that every node in g_1 is either uniquely mapped to a node in g_2 or to ϵ (which means it is deleted). Correspondingly, every node in g_2 is either a map of a node in g_1 or is inserted (map of ϵ). Major advantage of this method is that it allows

to map differently labeled nodes to each other and also to include a dissimilarity of nodes themselves. For example if the nodes are labeled with numbers, the absolute difference can be used to define a cost function for the substitution of nodes which have unequal labels.

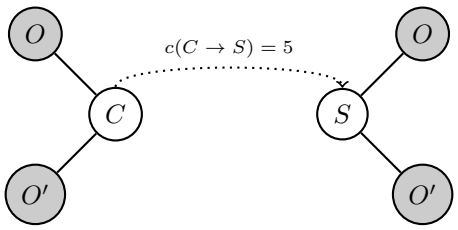


Figure 2.3: Example of a node substitution edit, transforming CO_2 into SO_2

The cost model can be chosen arbitrarily. An example is depicted in Fig. 2.3. This leads to many possible mappings f, who all have different costs $c(f)$ which then can be minimized. The optimization of the cost of an error-tolerant function as we described it is in turn again known to be NP-complete [19]. Yet, suboptimal algorithms have been proposed for solving this task [5] which are able to find a minimum in polynomial time. The risk with such algorithms is that they may find a local, but miss the global minimum of a function [9]. This leads to the Graph Edit Distance (GED) which is the concept aimed to be enriched in this thesis. It is properly introduced in the next subsection.

2.2 Graph Edit Distance GED

Given are two graphs $g_1 = (V_1, E_1, \mu_1)$ and $g_2 = (V_2, E_2, \mu_2)$ termed source and target graph respectively. The basic idea of Graph Edit Distance is to transform the source graph into the target graph by using edit operations and assigning a cost to them. Typically those are node substitutions ($u \to v$), insertions ($\epsilon \to v$) and deletions ($u \to \epsilon$). Note that editing a node may imply an edge insertion or deletion which can also be weighted in order to calculate a dissimilarity. A transformation of source to a target graph can be segmented into such edits, this leads to many possible edit paths for such a transformation.

Definition 2.2.1 (Edit Path)**.** A set $e_1, ..., e_k$ of k edit operations e_i that transform g_1 completely into g_2 is called a (complete) edit path $\lambda(g_1, g_2)$ between g_1 and g_2.

This definition corresponds perfectly to an error-tolerant graph matching. An example for an edit path is shown in Fig. 2.4. Note that this is not necessarily the minimum cost edit path but only an option for turning the H_2CO_3 graph into H_2CO_2.

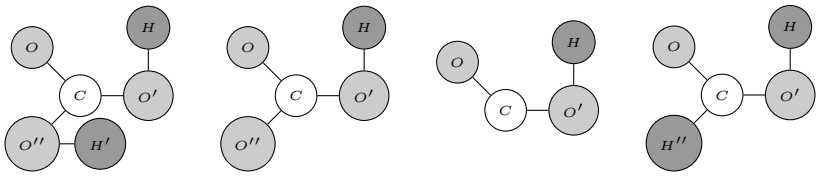


Figure 2.4: Example of an edit path from graphs representing H_2CO_3 to H_2CO_2

$$\lambda = \Big\{\{O \to O\}, \{H \to H\}, \{C \to C\}, \{O' \to O'\}, \{H' \to \epsilon\}, \{O'' \to \epsilon\}, \{\epsilon \to H''\}\Big\} \tag{2.4}$$

This particular edit path *implies* the following edge edit operations.

$$\Big\{\{(O'', H'') \to \epsilon\}, \{(C, O'') \to \epsilon\}, \{\epsilon \to (C, H'')\}\Big\} \tag{2.5}$$

All the possible edit paths which transform g_1 into g_2 form a set denoted as $\Lambda(g_1, g_2)$. By introducing a cost function $c(e_i)$ for every node edit and the edge edits implied by it, we can formalize a concept to measure a dissimilarity between two graphs by choosing the cheapest path of all.

Definition 2.2.2 (Graph Edit Distance GED)**.** Given two graphs $g_1 = (V_1, E_1, \mu_1)$(source) and $g_2 = (V_2, E_2, \mu_2)$ (target). The *Graph Edit Distance* $d_{\lambda_{min}}(g_1, g_2)$ between g_1 and g_2 is defined as

$$d_{\lambda_{min}}(g_1, g_2) = \min_{\lambda \in \Lambda(g_1, g_2)} \sum_{e_i \in \Lambda} c(e_i), \tag{2.6}$$

where λ_{min} represents the minimal cost edit path whose cost is then used as a dissimilarity measure.

This cost can be achieved by many possible paths so the minimal cost edit path is not unique while the cost for it is by definition. To limit the number of considered edit paths to a finite amount, we will introduce five weak conditions for the cost function.

1. Non-negativity:
   $c(e) \geq 0$ for all node and edge operations e.

2. Only substitutions are allowed to have zero cost:
   $c(e) > 0$ for all node and edge deletions and insertions e.

3. Prevention of unnecessary operations (triangle inequality):
   $c(u \to w) \leq c(u \to v) + c(v \to w)$
   $c(u \to \epsilon) \leq c(u \to v) + c(v \to \epsilon)$
   $c(\epsilon \to w) \leq c(\epsilon \to v) + c(v \to w)$

4. identity of indiscernible:
   $c(e) = 0$ for identical node or edge substitution

5. Symmetry:
   $c(e) = c(e^{-1})$ where e^{-1} describes the inverse edit operation to e.

Respecting those conditions, we only have to consider $|V_1| = n$ insertions, $|V_2| = m$ deletions and $|V_1| \times |V_2|$ substitutions for our possible edit paths. Still, this way we get $O(m^n)$ possible paths to compare. There are many different ways one can define a cost function, it makes sense to adapt the cost function to the type of label alphabets and their significance. the basic intuition is that the cost should be higher for substitutions of dissimilar nodes in respect to similar ones. Insertions and deletions may have fixed costs τ which for symmetry reasons should be equal.

Note that any substitution having a higher cost than 2τ can be safely replaced by a composition of a deletion followed by an insertion. A substitution cost function which guarantees to be in the interval $[0,2\tau]$ is the sigmoid function. An example of such a function is shown in Fig. 2.5.

$$c_{sigmoid}(u \to v) = \frac{1}{\frac{1}{2\tau} + exp(-\alpha||\mu_1(u) - \mu_2(v)||_p + \sigma)} \tag{2.7}$$

The two parameters α and σ can be used to adjust the gradient and left-right shift of the function. Note that (obviously) only positive dissimilarities are considered because of the non-negativity constraint.

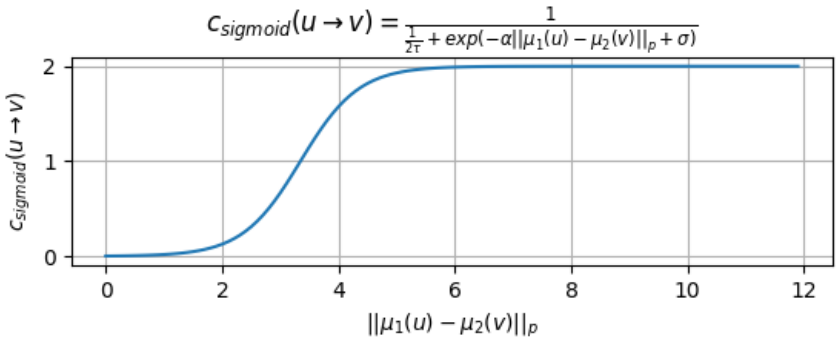


Figure 2.5: Example of a sigmoid function with parameters $\tau = 1$, $\alpha = 2$, $\sigma = 6$

Instead of numerical values, the labels can also consist of strings. A suggested cost function for such applications could be the string edit distance [14]. Another useful function is the dirac function, which returns zero for identical labels and a constant $\beta \in [0, 2\tau]$ else. The ability to change the cost function depending on the application makes the Graph Edit Distance a very flexible tool. The GED as defined above will be referred to as the 'original GED' in later formulations.

2.3 Bipartite Graph Edit Distance

To compute the exact Graph Edit Distance, an A*-based search algorithm is used. It organises the possible solutions in an ordered tree structure in which all the leaf nodes represent the end of a solution. This algorithm constructs an edit path and returns as soon as it finds a valid one. It is complete and admissible, which means that it always finds a solution and never overestimates the cost. Main drawback of the method is that it's time complexity is exponential in the number of nodes [5], in fact it belongs to the family of *quadratic assignment problems QAP* which belong to the class of NP-complete problems. This means that for now we have to assume that it is not possible to solve such a problem in polynomial time.

There exists an approximation algorithm for solving a QAP, which returns an upper and lower bound of the true solution. To apply it for the Graph Edit Distance, the problem must be rearranged to a QAP. Because a QAP only allows single (bijective) node assignments, many deletions (or insertions) are not allowed to be assigned to the same ϵ. For that, we need to extend our graphs by an appropriate number (m resp. n) of empty nodes ϵ_i. The augmented graphs then look the following:

$$V_1^+ = V_1 \cap \{\epsilon_1, ..., \epsilon_m\} \tag{2.8}$$

$$V_2^+ = V_2 \cap \{\epsilon_m + 1, ..., \epsilon_m + n\} \tag{2.9}$$

Having done that, we can generate a cost matrix C which includes all possible edit operations and their cost (see Fig. 2.10). Note that this matrix corresponds to the example in Fig. 2.4.

The upper left part represents all possible *substitutions* and their specific costs, defined by the chosen cost function. In this case it is arbitrarily chosen ($c(C \rightarrow any) = 1$, $c(O \rightarrow H) = 3$). On the upper right of the matrix, the diagonal corresponds to node deletions (cost = $\tau = 2$) and the remaining entries do not make sense within our constrains defined above (∞ cost). Same thing for the lower left part which corresponds to the node insertions. The value of τ is arbitrarily chosen (τ=2). The lower right part consists of zeroes only because it represents operations where an empty node is substituted by another empty node. The edit path discussed in Fig. 2.4 is bold. Note again that this is not the optimal (minimal cost) path.

$$\mathbf{C} = \begin{array}{c|ccccc|cccccc} & C & O & O' & H & H'' & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\ C & \mathbf{0} & 1 & 1 & 1 & 1 & 2 & \infty & \infty & \infty & \infty & \infty \\ O & 1 & \mathbf{0} & 0 & 3 & 3 & \infty & 2 & \infty & \infty & \infty & \infty \\ O' & 1 & 0 & \mathbf{0} & 3 & 3 & \infty & \infty & 2 & \infty & \infty & \infty \\ O'' & 1 & 0 & 0 & 3 & 3 & \infty & \infty & \infty & \mathbf{2} & \infty & \infty \\ H & 1 & 3 & 3 & \mathbf{0} & 0 & \infty & \infty & \infty & \infty & 2 & \infty \\ H' & 1 & 3 & 3 & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \mathbf{2} \\ \hline \epsilon & 2 & \infty & \infty & \infty & \infty & \mathbf{0} & 0 & 0 & 0 & 0 & 0 \\ \epsilon & \infty & 2 & \infty & \infty & \infty & 0 & \mathbf{0} & 0 & 0 & 0 & 0 \\ \epsilon & \infty & \infty & 2 & \infty & \infty & 0 & 0 & \mathbf{0} & 0 & 0 & 0 \\ \epsilon & \infty & \infty & \infty & 2 & \infty & 0 & 0 & 0 & 0 & \mathbf{0} & 0 \\ \epsilon & \infty & \infty & \infty & \infty & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{2.10}$$

Using this cost matrix, the following optimization problem can be stated.

$$(\varphi_1, ..., \varphi_{(n+m)}) = \underset{(\varphi_1,...,\varphi_{(n+m)}) \in P_{(n+m)}}{\arg\min} \left[\sum_{i=1}^{n+m} c_{i\varphi_i} + \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} c(a_{ij} \rightarrow b_{\varphi_i \varphi_j} \right] \tag{2.11}$$

Where $P_{(n+m)}$ refers to the set of all (n+m)! possible permutations of the integers 1,..n+m. This function consists of a linear term (first) and a quadratic term (second). Hence the problem exactly corresponds to a standard QAP. The linear term refers to the sum of node edit cost while the quadratic term corresponds to the implied edge edit costs. As the function returns a path $(\varphi_1, ..., \varphi_{(n+m)})$, the Graph Edit Distance (see def (2.2.2)) can be extracted by summing up the costs of each edit operation.

$$d_{\lambda_{min}} = \min_{(\varphi_1,...,\varphi_{(n+m)}) \in P_{(n+m)}} \left[\sum_{i=1}^{n+m} c_{i\varphi_i} + \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} c(a_{ij} \rightarrow b_{\varphi_i \varphi_j} \right] \tag{2.12}$$

The function can be subdivided into the computation of the Cost matrix considering vertex operations $C_\lambda^{\langle V \rangle}$ (linear) and one which contains the needed operations on the Edges $C_\lambda^{\langle E \rangle}$ (quadratic).

$$d_{\lambda_{min}} = \min_{(\varphi_1,...,\varphi_{(n+m)}) \in P_{(n+m)}} \left[C_\lambda^{\langle V \rangle} + C_\lambda^{\langle E \rangle} \right] \tag{2.13}$$

Up to here, we are still stuck on a QAP which is to time-comsuming for practical applications. For this reason we will introduce the concept of bipartite Graph Edit Distance, first presented by Kaspar Riesen. It allows the approximate computation of the GED in a substantially faster way than traditional methods on general graphs [21].

This is achieved by reducing the QAP to a *Linear Sum Assignment Problem LSAP*, which means that the optimisation is achieved using the linear term $C_\lambda^{\langle V \rangle}$ only. This means that the structural relationships between the nodes are neglected in order to make the algorithm faster. The best algorithms for solving such LSAP's have a cubic time complexity. As we need the knowledge of structural relationships for better performance of the algorithm, this information can be embedded into the Cost matrix by adding the minimum sum edit operation cost to every cost in the matrix. The resulting matrix will be referred as C^*.

An example of a C^* matrix corresponding to the one shown in Eq. 2.14 is shown below. It has the same values as C but the difference in adjacent nodes is added on top of any cost (e.g for deletions the value added corresponds to the number of neighbors). This requires us to solve a LSAP for every element in the matrix. This procedure allows us to handle the complexity of the problem while staying in polynomial time complexity.

$$\mathbf{C}^* = \begin{array}{c} \\ C \\ O \\ O' \\ O'' \\ H \\ H' \\ \hline \epsilon \\ \epsilon \\ \epsilon \\ \epsilon \\ \epsilon \end{array} \left[\begin{array}{cccccc|cccccc} C & O & O' & H & H'' & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\ \underline{\mathbf{0}} & 3 & 2 & 3 & 3 & 5 & \infty & \infty & \infty & \infty & \infty \\ 3 & \underline{\mathbf{0}} & 1 & 3 & 3 & \infty & 3 & \infty & \infty & \infty & \infty \\ 2 & 1 & \underline{\mathbf{0}} & 4 & 4 & \infty & \infty & 4 & \infty & \infty & \infty \\ 2 & 1 & 0 & 4 & 4 & \infty & \infty & \infty & \underline{\mathbf{4}} & \infty & \infty \\ 3 & 3 & 4 & \underline{\mathbf{0}} & 0 & \infty & \infty & \infty & \infty & 3 & \infty \\ 3 & 3 & 4 & 0 & \underline{0} & \infty & \infty & \infty & \infty & \infty & \mathbf{3} \\ \hline 5 & \infty & \infty & \infty & \infty & \underline{\mathbf{0}} & 0 & 0 & 0 & 0 & 0 \\ \infty & 3 & \infty & \infty & \infty & 0 & \underline{\mathbf{0}} & 0 & 0 & 0 & 0 \\ \infty & \infty & 4 & \infty & \infty & 0 & 0 & \underline{\mathbf{0}} & 0 & 0 & 0 \\ \infty & \infty & \infty & 3 & \infty & 0 & 0 & 0 & 0 & \underline{\mathbf{0}} & 0 \\ \infty & \infty & \infty & \infty & \mathbf{3} & 0 & 0 & 0 & 0 & 0 & \underline{0} \end{array}\right] \tag{2.14}$$

The minimum cost edit path solution in this example is given by the underlined values. As we suspected, the path suggested in the example is not the optimal one.

2.4 Nearest-Neighbor Classification kNN

The traditional [5] approach to Graph Edit Distance-based Pattern Recognition is given by the *k-Nearest Neighbor* classification (kNN). It is a straightforward algorithm which uses training data to classify new examples based on their neighborhood. The principle works as follows: the training data serves a a map, a two dimensional (x,y) example using two classes (circles and squares) is shown in Fig.2.6. For classifying a (unknown) test point q, the algorithm is divided into two steps. First, the chosen metric is used to calculate the distance to every point in the training data. Then, the parameter k chooses how many Nearest Neighbors are considered for the classification.

In the example we choose k=3 and decided using the Euclidean Metric (see Eq.2.1 with p=2), all of the considered neighbors are circles. Thus q will be labeled as a circle. For k=1, only the circle labeled 1 would be considered.

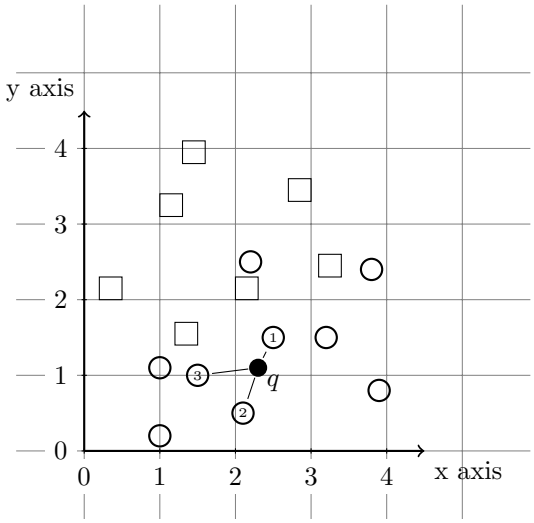


Figure 2.6: Example of a NN-classification with two classes

The decision can be made by majority which is the simplest choice. Another way to decide is by distance weighted vote, where a 'vote' is weighted by the inverse distance to the neighbor [22]. By choosing a low k, one risks to classify based on an outlier in the training data, by choosing a k too large, the scope of the method gets to wide which leads to missclassification [23], i.e. the model looses accuracy. Note that the classification performance is strictly based on the quality and quantity of the training dataset and the appropriateness of the distance measure [24].However the larger the training data the poorer the performance, because all the work is done at runtime [22].

To find the best-suited parameters for an application, the parameters are often optimized using a validation set. The best performing cases are then again examined using a test dataset. This prevents a choice of parameters based on overfitted examples. Another approach is to optimize the parameters by using many different sets and averaging them to minimize the risk of overfitting. The latter has been done for the final evaluation of this thesis.

2.5 Current State of Research

The code used for this thesis is an adaptation to the current bipartite graph matching framework. It is written in cython and contains methods to load and parse data as well as perform kNN classifications. The dataset used for all the experiments is called Enzymes [25]. It has appropriate size and complexity of graphs to do exploratory experiments. A visualization (by Anthony) of graphs from this dataset is depicted in Fig. 2.7.

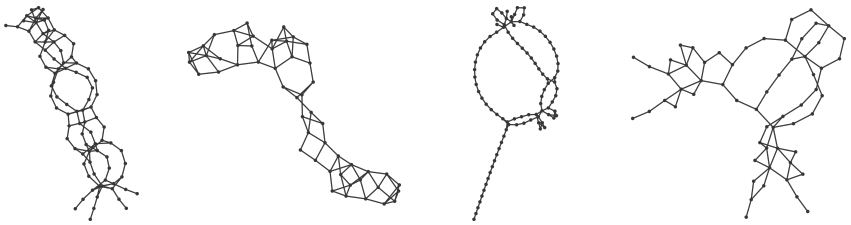


Figure 2.7: Visualization of graphs No. 7,111,292 and 483 from the Enzymes dataset

The dataset consists of 600 graphs which are shuffled into train (360 graphs), validation (120 graphs) and test (120 graphs) sets. The graphs are subdivided into 6 classes. For the evaluation, 10 different shuffles are used. The result is then the accuracy of the classification, averaged over twenty (10x val + 10x test) classified sets. The results of such a kNN classification using the original GED and for different k's are shown in Fig. 2.8, the standard deviation is shown as error-bars. The final experiments of this thesis have the same conditions.

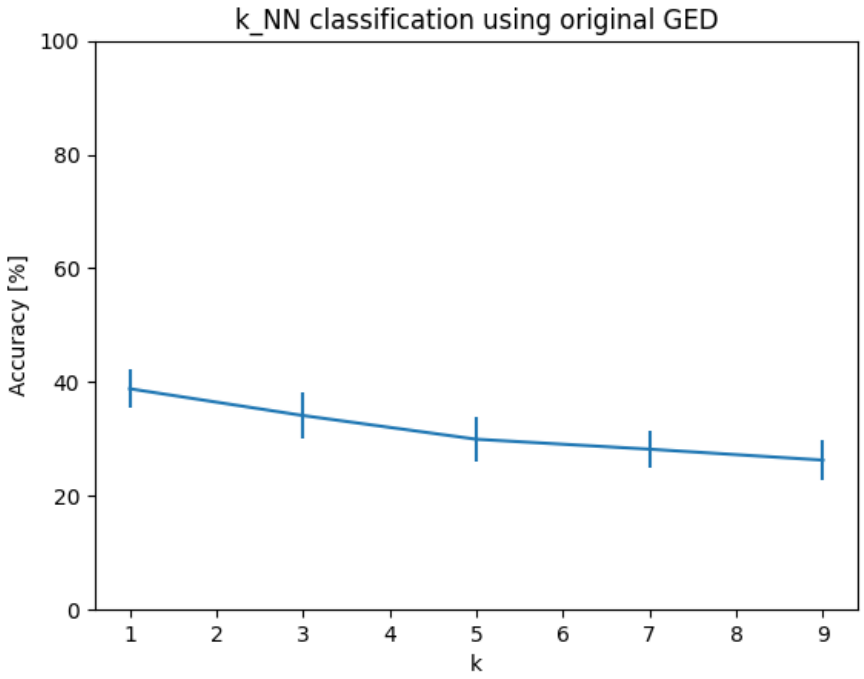


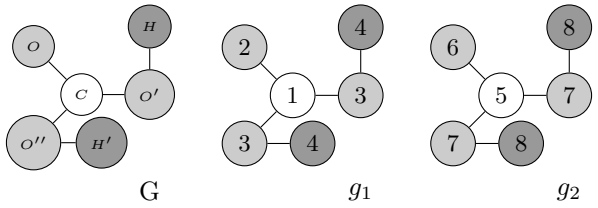
Figure 2.8: Classification performance of the original GED on the Enzymes dataset

2.6 The Weisfeiler-Lehman Algorithm

The method used to enrich the labels of the nodes is based on an algorithm first published in 1968, referred to as the (1-dim) Weisfeiler and Lehman test of isomorphism [15].

The algorithm supplements every node label R of both the graphs with the labels of its adjacent nodes. This tuple is then compressed into a new label. This short label is meant to encode the information of the labels of the node and of its adjacent nodes $R, N...N$. A hash function $h(R, N...N)$ is used for collision avoidance. This results in a new graph with same nodes and edges but updated labels which change every iteration i: $G_i = (V, E, \mu_i)$. This procedure is then repeated until the labels of the two graphs differ (non isomorphic) or the number of iterations i reaches a threshold n (assumption of isomorphism).

A detailed example on the graph we have seen before (H_2CO_3) is shown in fig 2.9 and the exact algorithm can be found in [16](page 2544). The order of the neighbors is not relevant as they are sorted (alphabetically or by number) before being compressed. For the isomorphism test, the procedure shown in Fig. 2.9 is done on two graphs simultaneously where after each iteration, the sorted label lists are checked for equality.



$h(R, N...N)$	1	2	3	4	5	6	7	8
$R, N...N$	C,OOO	O,C	O,CH	H,O	1,233	2,1	3,14	4,3

Figure 2.9: Weisfeiler-Lehman Algorithm on a single g G: H_2CO_3 with hash table

This table (2.1) shows the present labels as well as the sorted label list, which is then compared to other graphs to check for isomorphism. note that each graph iteration has unique labels so only graphs in the same iterations should be compared.

Iteration	G	g_1	g_2
labels	C,H,O	1,2,3,4	5,6,7,8
sorted label list	CHHOOO	123344	567788

Table 2.1: Table of iterations of the WL-algorithm

The runtime complexity is $O(nm)$ for n iterations and m nodes per graph. An important observation is that the compressed labels $h(R, N...N)$ each correspond to a subtree with root node R. The subtree for node O' in the example is shown below:

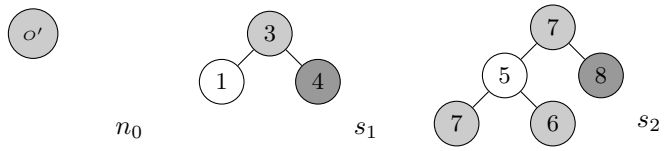


Figure 2.10: Subtree structure implied by the WL-algorithm with root node O'

The tree has the height of the iteration number i. The idea is that this trees show how much neighboring label information one could possibly encode. However, this procedure does not fully encode the structural information of the graph. Still it is sufficient to be a valid isomorphism test for almost all graphs [26]. Finally, we will focus on how to combine the two concepts presented (GED and WL-algorithm).The following method of combining a dissimilarity measure with the WL-algorithm is proposed in [16] for kernels. Despite that the GED is strictly speaking not a kernel, The important features are similar so the application of such a procedure is reasonable. Analogous to the definition for kernels, we define the Weisfeiler-Lehman-GED as follows:

Definition 2.6.1 (Weisfeiler-Lehman-GED)**.** Let GED be the original GED defined in section 2.2, then the *WL-GED with n iterations* is defined as

$$GED_{WL,h}(g_0, g_0') = GED(g_0, g_0') + \beta_1 GED(g_1, g_1') + ... + \beta_n GED(g_n, g_n') \quad (2.15)$$

Where the non-negative real weights β_i have been added to obtain a more general definition. For this thesis, the combination of the WL-hashes with the GED computation has been embeded in the cost computation. For that, the cost of such a hash is directly added to the cost matrix and for substitutions only. This is describe in more detail in the next Chapter.

Chapter 3

Method

3.1 Objective and Hypothesis

Following the example of Nino Shervashidze on a number of kernels [16], the goal of this thesis is to enrich the nodes of graphs using the Weisfeiler-Lehman algorithm and including the hashes in the original GED computation. The hypothesis is that the enrichment of the nodes using the Weisfeiler-Lehman algorithm can lead to an improvement of classification accuracy if the the GED is modified to harness the added information.

The reasoning behind this is based on the structural information that should be encoded into the WL-hashes. The risk is that this procedure possibly results in adding noise only and the 'enrichment' consists of information that the original GED is already using. The GED can be modified in many ways, including the choice of the distance metric for the hashes. There are also many other parameters to be considered.

This thesis focuses on optimizing k for the kNN classification as well as many possibilities for choosing the arising weights. As a dataset, the Enzymes dataset is chosen. It will be introduced below. This section will be structured as a description of the journey of finding out if this approach can have a positive impact in classification tasks. The final evaluation is then shown in the Results section.

3.2 Procedure

To perform own experiments, I use the windows subsystem for linux WSL environment [27] which i installed on my personal device. This allows using linux-based code from Anthony without having the overhead of a virtual machine. The code is written in cython [28], which combines the strengths of the C/C++ language with the python syntax. This makes execution faster and allows the usage of many libraries. Computationally more demanding programs, e.g. the kNN classification, are performed on UBELIX `http://www.id.unibe.ch/hpc`, the HPC cluster at the University of Bern.

As first steps, some basic examples and according tests are recreated. Those are re-ran when having changed the code. Examples are the check that $GED(g_i, g_i) = 0$ or $GED(g_i, g_j) \geq 0$ for $i \neq j$.

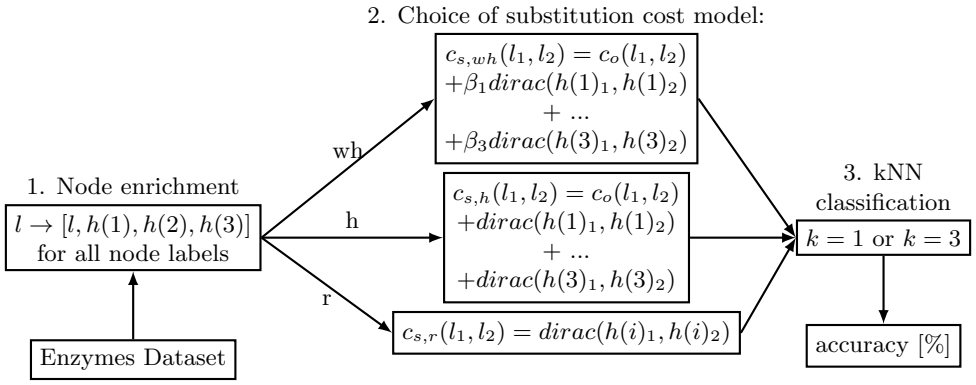


Figure 3.1: Computational pipeline for 3 possible cost models: replacement (r), handicapped (h) and weighted handicap (wh)

Fig. 3.1 shows the three major tasks to be performed. For every task, the procedure and findings are described. Starting by extending the node labels using the WL-algorithm as described before (1.). Then the embedding of the additional labels into the GED computation is done by integrating them in the cost function (2.). This can be done is many different ways as will be shown. Lastly, the performance of such an approach is evaluated using kNN framework which then returns the classification accuracy of the system (3.).

3.3 Node Label Enrichment

The starting point is a set of graphs from the Enzymes dataset. Enzymes is a set of protein tertiary structures obtained from Borgwardt et al. [29] consisting of 600 Enzymes from BRENDA enzyme database [25]. Every node of such a graph is labeled with one of three possible labels. The set is subdivided in different classification-groups: (test,train,val) with sizes 360,120,120 and different shuffles. A shuffle is a distribution of the graphs in the set into the classification-groups (taking into account some conditions). The enzyme graphs are labeled and can be assigned to one of six classes. They are stored in .graphml format. We use the Weisfeiler-Lehman algorithm to calculate the hashes of every node for n iterations and then store them in the graph representation as label extension.

Luckly the networkx library contains the algorithm by Shervashidze et al. [16]. This function takes a graph, the iteration number n and the hash size (in our case 8) as inputs and then returns a list of n 8 bit long hashes for every node in the graph.A typical extended node is shown in Fig.3.2. "key d0" contains the original label, "key d1" contains the extended label. Note that the labels are converted into 1/0 vectors which improves the performance of the algorithm.

```
<node id="0">
  <data key="d0">[1.0, 0.0, 0.0]</data>
  <data key="d1">['[1.0, 0.0, 0.0]', '0f80ff95bbe1ad57', '67f59b8cdc5aa298', '0a10d44c52d3f3bc', 'df48828685f9f0b0',
```

Figure 3.2: Example Node (extended)

The original label is then appended to the list, then the list is converted into a string. This is because the GED framework does only accept one single string label

per node. A simple script ran the described procedure for every node in every graph of the dataset (see dark arrow in Fig. 3.3). I used $n = 10$ as iteration number to perform some first experiments. The evaluation showed that the high iteration hashes are extremely unlikely to match so 10 is already a quite high choice.

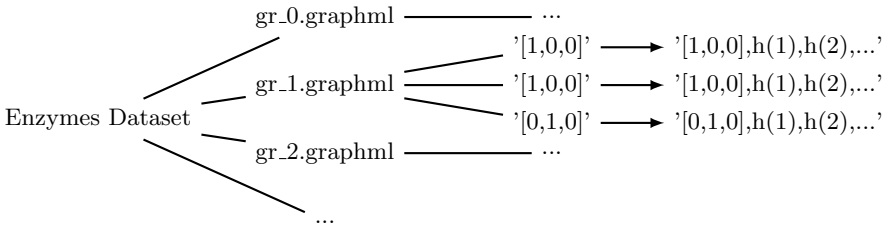


Figure 3.3: Node label extension

The generated labels can then be used for any kind of parameters (unused labels can be neglected) of the later chosen cost function. Also they can easily be generated before runtime and thus be reused for all the experiments.

3.4 GED Framework Adaptation

There are two ways of embedding the computed WL- hashes into the distance calculation. The first is to calculate the GED once with the original label and once with the hash values and then add the distances up according to def. 2.15. As described in the theory section, the other is to add the cost of the labels directly into the distance calculation, means modifying the substitution cost (upper left of the cost matrix). The later version is chosen because it prevents having the costs of deletions and insertions multiple times, facilitates the comparison of cost matrices and fitted better in the present code.

For an original node label the substitution cost is calculated based on the Euclidean Metric. As the original labels are converted in arrays filled with zeroes and a single one, the Euclidean Metric is very similar to the dirac function as for equal labels the cost is zero and for every dissimilar combination the cost is $\sqrt{2}$. A generic example is shown below (Eq. 3.1). Finally the constraint that the cost (c(deletion)+c(insertion)) does not exceed the limit of 2τ is checked.

$$euclidean([1,0,0,..,0],[0,..,1,0,0]) = \sqrt{1^2+1^2} = \sqrt{2} = const \tag{3.1}$$

As shown in Fig. 3.3 the enriched label is not a single 0,1-array anymore but has become a single string containing the original label (array shaped) and many 8-char hashes. Thus a parser which converts the string into an array of labels had to be included into the loading framework. When initialized, the original GED function took two graphs as input parameters and provided a float number which determines the distance as output.

After having modified it so it can consider the hashes, it newly takes an array of weights $[\beta_0, \beta_1, \beta_2, ..., \beta_i]$ which determines how many iterations i (max 10) of the WL-hashes should be considered and how they should be weighted. This array of weights

is used for the substitution cost computation which will be discussed below. It does not interfere with the original syntax which still works if no additional parameters are given.

3.5 Choice of substitution cost model

There are many ways to use the hashes from the WL-algorithm in the cost computation. The thesis focuses on three approaches, these are shown in Fig. 3.1 (see a), b), c)). Those ways are tested and then compared to the original GED to examine if this label enrichment has the potential to improve the method.

The way the hashes are compared is using a dirac function. A first iteration hash comparison cost (between graphs x and y) has the form $dirac(h(1)_x, h(1)_y)$. For simplicity the first iteration hash cost may be denoted as h(1), the second as h(2) and so on. The original (Euclidean) cost will be denoted as o. A more complex dissimilarity measure is not considered because the check for equality if the only comparison of hashes that makes sense. They don't share any other similarity (per definition). thus, any other way to compare the subtrees of the nodes would exclude the WL-algotrithm. An appropriate cost function is searched using three different approaches:

1. **replace (r)** the original labels by the hashes
   $c_{s,r}(l_1, l_2) = dirac(h(i)_1, h(i)_2) = h(i)$
   The first approach taken is to replace (thus r) the original with the new hashes cost function. Only one iteration i of hashes is checked for equality.

2. **handicapping (h)** the substitutions where the hashes do not match

$$\begin{aligned} c_{s,h}(l_1, l_2) &= c_o(l_1, l_2) + dirac(h(1)_1, h(1)_2) + ... + dirac(h(i)_1, h(i)_2) \\ &= o + h(1) + h(2) + ... + h(i) \end{aligned}$$

   Next, the sum of the distances is used and examined up to an iteration number i. The idea behind this approach is to keep the original GED framework but handicap the substitutions of nodes with dissimilar neighbours. The longer a source node has the same subtree as a target , the less it is handicapped. For high i this gets harder to achieve. This means that if a substitution edit can keep low cost it is very likely to be a good match. On the other side the handicap can add unwanted noise to the cost or even kick it over the 2τ limit. Which would then lead to performance drops. The concept of this approach is visualized below in Fig. 3.4.

3. Adding **weights** to the **handicaps (wh)**

$$\begin{aligned} c_{s,wh}(l_1, l_2) &= c_o(l_1, l_2) + \beta_1 dirac(h(1)_1, h(1)_2) + ... + \beta_3 dirac(h(3)_1, h(3)_2) \\ &= o + \beta_1 h(1) + \beta_2 h(2) + ... + \beta_i h(i) \end{aligned}$$

Where l stands for the original label of a node. Testing the cost functions proposed above leads to this final formula for the cost function. It results from good results for low iteration numbers as well as using the original cost function. Additionally, weights β have been added so the handicap can be regulated. It is also important to check for some conditions to make sure the altered cost does work as expected: If all the weights are set to zero then the distance of the original function and the modified GED are equivalent. And if the hashes are included, the modified GED must be larger than the original. A graphs GED to itself must be zero in any case.

The concept of handicapping unfavourable substitution operations is implemented in the cost matrix. There, any substitution for nodes with non-matching hash values is handicapped by receiving an added cost. The matrix form the example shown in Fig. 3.5 is shown below with its corresponding added weights. The added cost for unequal hashes in the first iteration ($i = 1$) is represented as a red plus. A difference in hashes for in the second iteration is represented as a blue plus. for this example, only two iterations are considered. Substitution operations of nodes with identical subtrees are not affected by the handicapping and are thus more favourable to be chosen in the minimal cost edit path computation.

$\mathbf{C} =$

	C^{+}	O^{++}	O'^{++}	H^{++}	H''^{++}	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
C^{++}	$\mathbf{0}^{++}$	1^{++}	1^{++}	1^{++}	1^{++}	2	∞	∞	∞	∞	∞
O^{++}	1^{++}	$\mathbf{0}^{+}$	0^{++}	3^{++}	3^{++}	∞	2	∞	∞	∞	∞
O'^{++}	1^{++}	0^{++}	$\mathbf{0}^{+}$	3^{++}	3^{++}	∞	∞	2	∞	∞	∞
O''^{++}	1^{++}	0^{++}	0^{+}	3^{++}	3^{++}	∞	∞	∞	**2**	∞	∞
H^{++}	1^{++}	3^{++}	3^{++}	**0**	0^{++}	∞	∞	∞	∞	2	∞
H'^{++}	1^{++}	3^{++}	3^{++}	0	0^{++}	∞	∞	∞	∞	∞	**2**
ϵ	2	∞	∞	∞	∞	**0**	0	0	0	0	0
ϵ	∞	2	∞	∞	∞	0	**0**	0	0	0	0
ϵ	∞	∞	2	∞	∞	0	0	**0**	0	0	0
ϵ	∞	∞	∞	2	∞	0	0	0	0	**0**	0
ϵ	∞	∞	∞	∞	**2**	0	0	0	0	0	0

Figure 3.4: Modified cost matrix with '+' representing handicap values

3.6 kNN Classification

To determine how well a distance helps classifying, we use a kNN classification as described in the theory section. This is the common application for the GED and can be used as a comparison tool for different cost models. The classification returns an accuracy for each of the twenty shuffles available. The final result is then the mean value. For pioneering experiments only single shuffles are examined because many

possible β's can be considered as well as different k's for the kNN. This does not deliver the most exact results but helps finding patterns in the choice of parameters. The findings from the experiments are presented in the following section.

3.7 Pioneering Experiments

Using the replacement (r) cost model for the iteration numbers $i = 1$ to 8 some first tendencies are found. The results from the classification (using a single test shuffle) are shown in table 3.1. The original GED result is also presented for comparison. A clear tendency that the results get worse towards higher i's can be seen. Also, the convergence of the results for $i > 4$ shows that for most graphs the WL-hashes do not lead to an information benefit for the method. The last relevant iteration is dataset specific but it is important to notice that it is reached after few iterations. This will be discussed in the later sections.

The limit of a 16.67% accuracy is a minimum because for 6 classes, a completely random projection would (on average) be equivalent. Having found that, the next experiments would include the original cost as well as hashes from early iterations. Because those are the ones that appear to help finding similarities between the graphs. The original labels are more likely to be similar which leads to low substitution costs. Low costs, especially zeroes, are expected to lead to better classification. Later iterations are more unlikely to be identical because they represent full subtrees (see Fig. 2.10). This leads to low similarity between graphs and thus low information for the classification.

Replacement Model (r)									
k	O	h(1)	h(2)	h(3)	h(4)	h(5)	h(6)	h(7)	h(8)
1	35.83	35.00	32.50	28.33	26.67	25.83	25.00	24.17	24.17
3	40.83	26.67	29.17	25.00	24.17	23.33	25.00	25.00	24.17
5	30.00	25.00	25.83	28.33	25.83	25.00	25.00	25.00	25.00
7	27.50	23.33	20.83	21.67	20.83	20.00	20.83	20.83	20.00

Table 3.1: accuracies using the replacement model (r) for different k and i

Not using the original label is worse than using it, it still contains important information. In those pioneering experiments.

The next model is inspired by the proposed method by Shervashidze (Eq. 2.15) and just adds the hashes on top of the original cost. The idea already mentioned above is to keep similar substitutions cheap while handicapping those where the source and target have dissimilar neighbours.

The results however are worse that the original as shown in table 3.2. The classification does not work in that way because most of the costs are set to 2τ. This can be confirmed by observing the cost matrices. We showed that in those experiments, the information gained for the substitutions where hashes do match is overruled by the number of substitutions where it does not. A confirmation of this hypothesis is found by examining the cost matrix (see Section 3.8). To regulate that, low weights for $h(i)$ should be chosen.

Handicap Model (h)					
k	O+h(1)	O+h(1)+h(2)	O+...+h(3)	O+...+h(4)	O+...+h(5)
1	34.17	28.33	23.33	20.00	19.17
3	28.33	27.50	24.17	20.83	19.17
5	25.83	20.83	23.33	17.50	18.33
7	21.67	18.33	17.50	16.67	16.67

Table 3.2: Classification accuracies [%] for handicap (h) cost model

Adding weights to the cost lead to the first results where the modified GED outperformed the original one. This is a first sign that the label enrichment has the potential of improving the method. However, those experiments are done on a single shuffle and thus have not enough expressiveness to be valued as results. Those pioneering results are shown for the weights 0.5 and 0.125 (for all iterations) in tables 3.3 and 3.4.

Weighted Handicap Model (wh) using $\beta_i = 1/2$			
k	O+h(1)/2	O+(h(1)+h(2))/2	O+(h(1)+h(2)+h(3))/2
1	35.00	33.30	30.00
3	37.50	30.00	32.50
5	30.00	28.33	23.33
7	25.83	23.33	23.33

Table 3.3: Classification accuracies [%] for $\beta_i = 1/2$

Weighted Handicap Model (wh) using $\beta_i = 1/8$				
k	O	O+h(1)/8	O+(h(1)+h(2))/8	O+(h(1)+h(2)+h(3))/8
1	35.83	47.50	45.00	40.83
3	40.83	30.00	35.00	35.00
5	30.00	29.17	24.17	28.33
7	27.50	24.17	21.67	24.17

Table 3.4: Classification accuracies [%] for $\beta_i = 1/8$

3.8 Cost Matrix Analysis

As mentioned above in the context of the handicap model, the cost matrix can be useful for understanding how and why the algorithm works as it does. Having many 2τ values in the cost matrix leads to poor performance because it does not distinguish between substitution and delete-insert edits. Our Hypothesis is that the zero values are important for finding the right node pairs to be substituted so we examined the number and reduction of zeroes in such matrices. The quantitative results are shown in the results (Chapter 4).

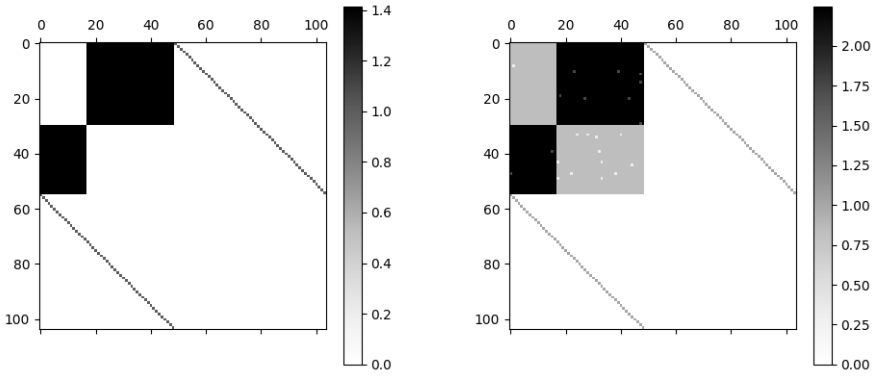


Figure 3.5: Original cost matrix (l) and for $c = c_o + h(1)/2 + h(2)/3$ for two randomly chosen graphs. Bright parts mean a low cost (handicap) because of matching subtrees.

For this evaluation, τ is set to one and infinite values (for insertions and deletions) are plotted white. This example of two cost matrices from two randomly chosen enzyme graphs makes clear that for most of the substitutions (top left of the matrix), the hashes are not equal and thus the chosen weights are added. It is visible that only few substitutions stay at low cost after addition of the WL-hash costs.

Chapter 4

Results

The findings described in the methods section have paved the way for designing a GED in which the WL-hashes can be included in a meaningful way. The insight that the original label must be considered as well as that early iteration hashes are more valuable for the classification is made. Further weights are added to the distances so they don't exceed the 2τ limit. These weights can be chosen in many ways and need optimisation. As already seen, low weights lead to better results because of the 2τ limit.

Classification experiments are time consuming and there are many parameters to optimize so the scope is limited to only few weights. For the sake of keeping the overview only the first three hashes are considered for the cost calculation. Those are the best performing in the pioneering experiments (see tables 3.1 and 3.2). The original label cost is not weighted. The final cost ($c_{s,f}$) function examined correspond to the (wh) model using up to 3 hashes. It is defined as follows:

$$c_{s,f}(l_1, l_2) = c_o(l_1, l_2) \begin{array}{l} +\beta_1 dirac(h(1)_1, h(1)_2) \\ +\beta_2 dirac(h(2)_1, h(2)_2) \\ +\beta_3 dirac(h(3)_1, h(3)_2) \end{array} \tag{4.1}$$

4.1 Accuracies using the WL-GED

As final experiment, a weight optimisation for β_1,β_2 and β_3 as well as for k=1 and k=3 is attempted. For the optimisation, every weight combination for the weights [0,0.1,0.2,0.3] is tested in order to find the ones that have the highest classification accuracy. Higher cost have, in intermediate experiments shown to result in worse classification performance. Note that for (β_1,β_2,β_3) = $(0, 0, 0)$ this corresponds to the original GED function.

In order to have quantitative significance, this experiment is done on twenty shuffles for each pair of parameters, then the mean value is calculated as well as the standard deviation shown as error bar. For the examined weights, the method can lead to much worse results but also outperform the original GED in some cases. The best and worst accuracy results as well as the result for the original GED for k = 1 are shown in the bar plot in Fig. 4.1.

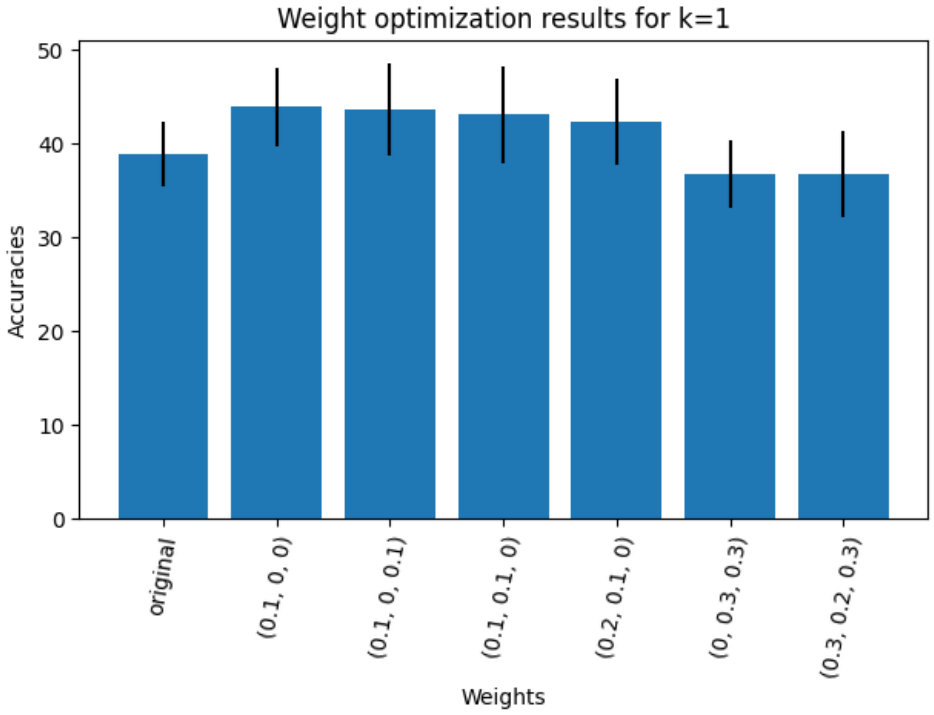


Figure 4.1: Bar plot of best and worst performing weight combinations, compared with the original algorithm

4.2 Experiments on C-matrix

We have seen that the number of zeroes in the cost matrix always reduces, which makes sense because a term is added to the cost. Using already two hash iterations, only few substitution operations stay without a handicap. To quantify this measure, the zero values are counted and the reduction rate is reported. For comparison, the cost matrix for 100 randomly chosen graph pairs is calculated in the original way and using the WL-hashes. As for a dataset of 600 graphs the number of pairings

is high (600^2) and the results are already stable using 100 graph pairs, this sample is considered to be enough. Also, some rare pairs do not include any zeroes for the substitution. This leads to an infinite reduction which can be seen as an outlier. This happens when the labels are completely different for all the nodes (e.g. a graph only consisting of '[1,0,0]' and another only of '[0,1,0]'). Those pairs are neglected because they would distort the results.

The reduction of zero values is calculated for all 100 pairs and then the mean is taken which is the result of the analysis. Only hash values h(1) and h(2) are considered because for h(3), the reduction is already negligible. The weights are both chosen as $\beta_1 = \beta_2 = 1/2$, note however that for this analysis the weights (if > 0) do not play any role.

Zeroes orig_C/wl_C	h(1)	h(2)
mean	0.014	0.000076
std	0.016	0.00046

Table 4.1: Zeroes in the WL-cost matrix divided by original Cost matrix (only substitutions considered). Higher iterations result in negligibly low values.

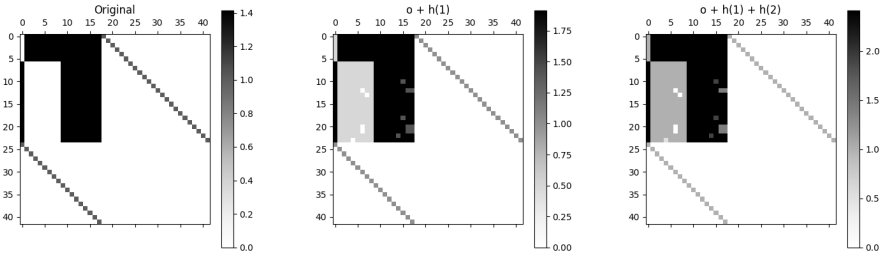


Figure 4.2: C matrices for the different weights for randomly chosen Graphs (255 and 210)

The cost matrices shown in Fig. 4.2 show the costs for edit operations of two graphs with size 17 and 23 (square matrix with 41^2 fields). The first graph (left side of the matrix) has two types of node and the second has 3 types. After having added the costs for h(1), few substitutions are left with the original cost. All the others are raised by the chosen weight. For h(2), none of the substitutions is left unchanged, which corresponds to adding noise only.

Chapter 5

Discussion

The development of a framework for embedding the WL-algorithm into the GED computation has lead to findings for further development. It also helped to gain knowledge about the GED algorithm and how to edit and then interpret it. The results are a collection of findings followed by a qualitative exploration of the potential of the method. The conclusion of this thesis is followed by a proposal for future work.

5.1 Conclusion

As a first exploration of the method, a lot of knowledge is gained in the process of developing an appropriate cost function. The approach of replacing (r) the original labels by the hash values is discarded because of poor results and the discovery, that the original label cost is indispensable. This showed that the direct similarity between the node labels is important for finding similar structures and thus compute an expressive distance between graphs. The original labels remain the best way to compare graphs, whereas the hashes have the potential to play a supporting role which could add detail to the original method. This means that the original nodes must be kept and only low weights should be used for the hash comparisons.

Next we have seen that it is important that the cost values stay in the limits of 2τ. This also underlines the importance of keeping the weights low. Everything over this limit will be neglected and does not contribute to a similarity detection and thus to an efficient dissimilarity measure. Overshooting this limit lead to the worst results as to many deletions followed by insertion are chosen for a minimal path. Having a to high cost ($> 2\tau$) can be compared to ignoring a possible substitution which could have helped providing a meaningful dissimilarity measure. As a consequence, the distance consists mostly out of the difference in graph size.

Further we have seen that the high iteration hashes (i.e. higher than $i = 3$) are not very useful, especially compared to the first two. The 3. iteration hash is (for the Enzymes dataset) the last which could lead to an improvement. The first iterations only consider the closest neighbors and thus have the highest potential to be similar. Such similarities are what the algorithm aims to find. For later iterations, the probability of having an identical subtree is not compensated by the gained information. This is also confirmed in the cost matrix analysis experiments. The number of 0 cost substitutions reduces very fast (see tab. 4.1) with iteration number i whereas the number of high-cost substitutions rises rapidly.

We are still convinced that the zeroes in the cost matrix are of major importance for good classification performances. However this must not mean that the high iteration hashes need to be useless. This evaluation depends highly on the Enzymes dataset and also does not rule out more efficient techniques for integrating the hashes.

The final choice for the design of the cost function has the shape of Eq. 4.1 (or Fig. 3.1(wh)). It consists of the original cost, which is handicapped using dirac functions if the hashes do not match. Also the handicap is weighted by $\beta_1, \beta_2, \beta_3$ which must be chosen at low magnitude (< 0.3). The extensive evaluation on the search for appropriate weights has lead to some weight combinations who lead to an improvement in classification performance. The best performing weight combinations however do not show a clear pattern for best results besides of keeping the magnitude of the weights low. Also, the degree of improvement is not high enough to make further conclusions.

The WL-hashes have shown to have the potential to lead to a benefit in classification performance and we have seen some examples where it does so. A general solution to improve the classification was not found because of many parameters and dataset specific measures e.g. graph size and number of different node kinds can have an influence in the classification performance.

5.2 Future work

As mentioned in the methods section, the approach of adjusting the cost computation is not exactly the one proposed by Shervashidze (see Eq. 2.15). His approach would compute the complete GED for every iteration, including the edge costs. It would be interesting to test if such a method would lead to better results. However, the calculations involved correspond to the very first experiments done for this thesis (replacement model). Only that for our purposes we have not added the distances. The finding of this pioneering experiment is that the classification success and thus the expressive power of the method drops rapidly for higher iteration hashes. Also none of the hash iterations exceeded the original GED. Still this does not exclude success for this approach.

The chosen cost function still has potential and ways to be altered. An example would be to also weight the original cost using a parameter β_0. The approach in the thesis was chosen in the belief that only the ratio between the single costs would have an impact. For example that for weights $(\beta_0, \beta_1, \beta_2, \beta_3) = (1, 0.2, 0.2, 0.2)$ one would get the same results as for $(0.5, 0.1, 0.1, 0.1)$. This must not be the case as there is another important measure, τ. The ratio depends on this value, which means that for further experiments, I would choose to include a weight β_0 which also allows to keep the cost magnitude low. This will lead to many possible parameter combinations, to reduce those, one could start by ignoring the 3. iteration of hashes as it seems to be the less valuable. Note, that the magnitude of one has been chosen for the original GED (in appropriate ratio to τ) which motivated the decision of keeping the magnitude. For further exploration, a recommendation would be to reduce the magnitude of the cost for original label substitutions.

The dirac function, as described in section 2.2, is the only metric where converting hash dissimilarities into a distance is meaningful. However, there are other ways to represent the subtrees of a node. An example could be as a list of labels. This would allow to introduce a more specific metric. For special cases even database-specific. Such a metric can be chosen arbitrarily so it will again lead to a broad list of parameters which are hard to optimise because it is difficult to quantify their direct impacts. Such an approach could also rapidly lead to quite high time complexity, which is what the original algorithm tried to avoid.

The most interesting findings were the bright, low cost spots in the cost matrix (fig. 3.8). Such spots suggest strongly that the substitution should be chosen for the optimal edit path. Instead of going the way of handicapping the cost matrix, one could also try to directly embed this information into the path computation. This could be done by fixing some substitutions based on their hash values. This will again leads to an optimisation problem because one wants to fix as much values as possible (low iteration). But higher iteration matches are more promising to be a 'correct' match. The price is that they are more rare.

Lastly, to find quantitative results, many datasets must be considered and also compared. We have seen that for the Enzymes dataset this method can have a positive impact in classification performance. This however is only a qualitative insight. Other datasets can give insight in underlying mechanisms as well as general knowledge about finding an appropriate method. Still the growing complexity in influences by graph specific measures is not to be underestimated.

Besides all the possibilities of enriching the GED, the time consumption of the algorithm should still be highly weighted when rating new algorithms. Thus the complexity of the method should be kept as low a possible. This also allows it to be used in many different applications and thus become a general tool for graph comparison.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetzes vom 5. September 1996 über die Universität zum Entzug des aufgrund dieser Arbeit verliehenen Titels berechtigt ist.

Datum, Ort & Unterschrift:

07.11.22, Bern R. De Gottardi

Bibliography

[1] Stephanie Dick. "Artificial Intelligence". In: *Harvard Data Science Review* 1.1 (July 2019). https://hdsr.duqduq.org/pub/0aytgrau.

[2] Neelam Tyagi. *6 Major Branches of Artificial Intelligence (AI) — Analytics Steps*. en. URL: https://www.analyticssteps.com/blogs/6-major-branches-artificial-intelligence-ai (visited on 09/22/2022).

[3] *Master in Artificial Intelligence in Medicine (AIM)*. eng. Mar. 2021. URL: https://www.caim.unibe.ch/education/msc__aim/index_eng.html (visited on 09/22/2022).

[4] Valentina Boeva. *Machine Learning for Biological Use Cases - CompbioZurich*. URL: https://compbiozurich.org/courses/UZH-BIO390/2022-10-11-Valentina-Boeva/ (visited on 10/15/2022).

[5] Kaspar Riesen. *Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications*. Advances in Computer Vision and Pattern Recognition. Springer, 2015. ISBN: 978-3-319-27251-1. DOI: 10.1007/978-3-319-27252-8. URL: https://doi.org/10.1007/978-3-319-27252-8.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification, 2nd Edition*. Wiley, 2001. ISBN: 9780471056690. URL: https://www.worldcat.org/oclc/41347061.

[7] Theodosios Pavlidis. *Structural pattern recognition*. Vol. 1. Springer, 2013.

[8] Mario Vento. "A long trip in the charming world of graphs for Pattern Recognition". In: *Pattern Recognition* 48.2 (2015), pp. 291–301. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2014.01.002. URL: https://www.sciencedirect.com/science/article/pii/S0031320314000053.

[9] Donatello Conte et al. "Thirty Years Of Graph Matching In Pattern Recognition". In: *Int. J. Pattern Recognit. Artif. Intell.* 18.3 (2004), pp. 265–298. DOI: 10.1142/S0218001404003228. URL: https://doi.org/10.1142/S0218001404003228.

[10] Julian R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1 (1976), pp. 31–42. DOI: 10.1145/321921.321925. URL: http://doi.acm.org/10.1145/321921.321925.

[11] Wen-Hsiang Tsai and King-Sun Fu. "A Pattern Deformational Model and Bayes Error-Correcting Recognition System". In: *IEEE Trans. Syst. Man Cybern.* 9.12 (1979), pp. 745–756. DOI: 10.1109/TSMC.1979.4310126. URL: https://doi.org/10.1109/TSMC.1979.4310126.

[12] Adrie C. M. Dumay et al. "Consistent inexact graph matching applied to labelling coronary segments in arteriograms". In: *11th IAPR International Conference on Pattern Recognition, ICPR 1992. Conference C: Image, Speech and Signal Analysis, The Hague, Netherlands, August 30-September 3, 1992*. IEEE, 1992, pp. 439–442. DOI: 10.1109/ICPR.1992.202019. URL: https://doi.org/10.1109/ICPR.1992.202019.

[13] Michael Stauffer et al. "A Survey on Applications of Bipartite Graph Edit Distance". In: *Graph-Based Representations in Pattern Recognition - 11th IAPR-TC-15 International Workshop, GbRPR 2017, Anacapri, Italy, May 16-18, 2017, Proceedings*. Ed. by Pasquale Foggia, Cheng-Lin Liu, and Mario Vento. Vol. 10310. Lecture Notes in Computer Science. 2017, pp. 242–252. DOI: 10.1007/978-3-319-58961-9_22. URL: https://doi.org/10.1007/978-3-319-58961-9%5C_22.

[14] Kaspar Riesen, Michel Neuhaus, and Horst Bunke. "Bipartite Graph Matching for Computing the Edit Distance of Graphs". In: *Graph-Based Representations in Pattern Recognition, 6th IAPR-TC-15 International Workshop, GbRPR 2007, Alicante, Spain, June 11-13, 2007, Proceedings*. Ed. by Francisco Escolano and Mario Vento. Vol. 4538. Lecture Notes in Computer Science. Springer, 2007, pp. 1–12. DOI: 10.1007/978-3-540-72903-7_1. URL: https://doi.org/10.1007/978-3-540-72903-7%5C_1.

[15] Boris Weisfeiler and Andrei Leman. "The reduction of a graph to canonical form and the algebra which appears therein". In: *NTI, Series* 2.9 (1968), pp. 12–16.

[16] Nino Shervashidze et al. "Weisfeiler-Lehman Graph Kernels". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2539–2561. DOI: 10.5555/1953048.2078187. URL: https://dl.acm.org/doi/10.5555/1953048.2078187.

[17] Ryan A. Rossi and Nesreen K. Ahmed. "An Interactive Data Repository with Visual Analytics". In: *SIGKDD Explor.* 17.2 (2016), pp. 37–41. URL: http://networkrepository.com.

[18] Junchi Yan et al. "A Short Survey of Recent Advances in Graph Matching". In: *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval, ICMR 2016, New York, New York, USA, June 6-9, 2016*. Ed. by John R. Kender et al. ACM, 2016, pp. 167–174. DOI: 10.1145/2911996.2912035. URL: https://doi.org/10.1145/2911996.2912035.

[19] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[20] Mirtha-Lina Fernández and Gabriel Valiente. "A graph distance metric combining maximum common subgraph and minimum common supergraph". In: *Pattern Recognit. Lett.* 22.6/7 (2001), pp. 753–758. DOI: 10.1016/S0167-8655(01)00017-4. URL: https://doi.org/10.1016/S0167-8655(01)00017-4.

[21] Kaspar Riesen and Horst Bunke. "Approximate graph edit distance computation by means of bipartite graph matching". In: *Image Vis. Comput.* 27.7 (2009), pp. 950–959. DOI: 10.1016/j.imavis.2008.04.004. URL: https://doi.org/10.1016/j.imavis.2008.04.004.

[22] Pádraig Cunningham and Sarah Jane Delany. "K-Nearest Neighbour Classifiers - A Tutorial". In: *ACM Comput. Surv.* 54.6 (July 2021). ISSN: 0360-0300. DOI: 10.1145/3459665. URL: https://doi.org/10.1145/3459665.

[23] László Kozma Lkozma. "k Nearest Neighbors algorithm (kNN)". en. In: (), p. 33.

[24] Haneen Arafat Abu Alfeilat et al. “Effects of Distance Measure Choice on K-Nearest Neighbor Classifier Performance: A Review”. In: *Big Data* 7.4 (2019). PMID: 31411491, pp. 221–248. DOI: 10.1089/big.2018.0175. eprint: https://doi.org/10.1089/big.2018.0175. URL: https://doi.org/10.1089/big.2018.0175.

[25] Ida Schomburg et al. “BRENDA, the enzyme database: updates and major new developments”. In: *Nucleic Acids Res.* 32.Database-Issue (2004), pp. 431–433. DOI: 10.1093/nar/gkh081. URL: https://doi.org/10.1093/nar/gkh081.

[26] László Babai and Ludek Kucera. “Canonical Labelling of Graphs in Linear Average Time”. In: *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*. IEEE Computer Society, 1979, pp. 39–46. DOI: 10.1109/SFCS.1979.8. URL: https://doi.org/10.1109/SFCS.1979.8.

[27] *Informationen zum Windows-Subsystem für Linux — Microsoft Docs*. Apr. 2020. URL: https://web.archive.org/web/20200428093523/https://docs.microsoft.com/de-de/windows/wsl/about (visited on 08/30/2022).

[28] *Cython - an overview — Cython 0.29.32 documentation*. URL: https://cython.readthedocs.io/en/stable/src/quickstart/overview.html (visited on 08/30/2022).

[29] Karsten M. Borgwardt et al. “Protein function prediction via graph kernels”. In: *Proceedings Thirteenth International Conference on Intelligent Systems for Molecular Biology 2005, Detroit, MI, USA, 25-29 June 2005*. 2005, pp. 47–56. DOI: 10.1093/bioinformatics/bti1007. URL: https://doi.org/10.1093/bioinformatics/bti1007.