Chapter 3

Graph Matching using Extensions to Graph Edit Distance

3.1 Introduction

Graph edit distance is one of the important and most widely used technique for graph matching. The basic concept of graph edit distance is to transform one graph to the other one using the minimum number of deformations. These distortions are defined as edit operations consisting of insertion, deletion, and substitution of nodes and edges. Due to the flexibility of this distortion model and its associated cost, graph edit distance applies to a wide range of applications. The main advantage of the graph edit distance is that it can be applied to any arbitrary graph labeled or unlabeled and its ability to handle any type of structural distortion. On the other hand, the major disadvantage of graph edit distance is that it is computationally expensive, and it takes exponential time with respect to the number of nodes in input graphs. In this chapter, we describe the extensions of graph edit distance, which can be used as a trade-off between computation time and accuracy of various graph matching applications.

We describe an approach to error-tolerant graph matching using graph homeomorphism to find the structural similarity between two graphs [81]. In this technique, we use path contraction to reduce the number of nodes in the input graphs, while preserving the topological equivalence of these graphs. We also present an approach to error-tolerant

graph matching using node contraction, in which a given graph is transformed into another graph by contracting smaller degree nodes [82]. The above approaches lead to a reduction in the search space needed to compute graph edit distances between transformed graphs.

This chapter is organized as follows. Section 3.2 describes preliminaries and section 3.3 presents homeomorphic graph edit distance. Error-tolerant graph matching using extended graph edit distance is introduced in section 3.4. Experimental evaluation of the above techniques is shown in section 3.5. Finally, section 3.6 summarizes this chapter.

3.2 Preliminaries

In this section, we review the basic definitions and concepts related to graph matching and the proposed work.

Let G_1 and G_2 be two graphs. A bijective mapping $f : V_1' \rightarrow V_2'$ from G_1 to G_2 is called an *error-tolerant graph matching*, if $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$ [3].

A common set of edit operations involves insertion of vertices or edges, deletion of vertices or edges, and substitution of vertices or edges. Insertion of a vertex u is denoted by $\varepsilon \rightarrow u$, deletion of vertex u by $u \rightarrow \varepsilon$, and substitution of a vertex u by vertex v is denoted by $u \rightarrow v$. We use an additional edit operation for path contraction of a simple path in which every vertex except first and last have degree two. We denote this path contraction by $(u_1,...,u_n) \rightarrow (u_1,u_n)$. We use path contraction to remove each node with degree two, while maintaining the topological equivalence of the graphs.

Let $G = (V,E)$ be a graph having an edge $e = (u,v) \in E$. Subdivision of the edge e produces another graph $G' = (V',E')$ such that $V' = V \cup \{w\}$, and $E' = E \setminus \{(u,v)\} \cup \{(u,w),(w,v)\}$, where w is an additional vertex and (u,w), (w,v) are additional edges. A subdivision of a graph G is another graph G' obtained by performing the subdivision operation on one or more edges in G. Two graph G_1 and G_2 are homeomorphic or topologically equivalent, if both graphs G_1 and G_2 are a subdivision of some graph G.

Since subdivision of G, changes the count of vertices of degree two only, hence for two graphs to be homeomorphic they must have the same number of vertices of each degree except two.

3.3 Homeomorphic Graph Edit Distance

Two graphs are said to be homeomorphic, when one of the graph can be transformed to the other one after performing subdivision operations on the edges by inserting the additional nodes along its edges.

Let $G = (V,E)$ be a graph with V and E be vertex set and edge respectively and let $e = (u,v) \in E$. Subdivision operation on the edge E generates another graph $G' = (V',E')$ so that $V' = V \cup \{w\}$, and $E' = E \setminus \{(u,v)\} \cup \{(u,w),(w,v)\}$, here w is a new vertex and (u,w), (w,v) are new edges.

Homeomorphic graph edit distance is defined as the minimum count of edit operations required to convert a graph G_1 to make it homeomorphic to graph G_2. The elementary idea behind the homeomorphic graph edit distance is that if two graphs are homeomorphic or topologically equivalent then they must have the same number of vertices of each degree, except degree two vertices. Therefore we can remove each vertex of degree 2, and again get a homeomorphic graph.

Definition 3.3.1. Let $G_i = (V_i, E_i, \mu_i, \nu_i)$ for $i = 1,2$ be two graphs, the homeomorphic graph edit distance (HGED) between G_1 and G_2 is defined by

$$HGED(G_1,G_2) = GED(G_1',G_2')$$

$$= min_{(e_1,\ldots,e_k)\in\varphi(G_1',G_2')} \sum_{i=1}^{k} c(e_i) \tag{3.1}$$

Where G_1' and G_2' are graphs obtained from G_1 and G_2 respectively by performing path contraction from each vertex, $\varphi(G_1',G_2')$ denotes the set of edit distance paths converting G_1' into G_2', and $c(e_i)$ is the cost function associated with every edit operation e_i.

Specifically, we perform a path contraction of all simple paths $(u_1,\ldots,u_n)$ in a graph, such that all vertices along this path have degree two (for $i = 2$ to $n-1$, $Deg(u_i) = 2$) except

first (u_1), and last vertex (u_n). These path contractions will save the huge processing of several vertices and edges, which are required in the computation of graph edit distance.

Proposition 3.3.1. *Let $G = (V, E, \mu, \nu)$ be a graph. If $G' = (V', E', \mu', \nu')$ be the graph obtained by performing path contraction $(u_1, ..., u_n) \to (u_1, u_n)$ by removing or smoothing the vertices $u_2, ..., u_{(n-1)}$ where $Deg(u_i) = 2$ for $i = 2$ to $n-1$, then G and G' are homeomorphic.*

The process of contraction removes the 2-degree nodes only, and other nodes remain unaffected. Since smoothing reverses the operation of the subdivision, therefore G and G' are homeomorphic.

3.3.1 Homeomorphic Edit Cost

Homeomorphic edit cost functions are based on Euclidean distance measure, which allocates constant cost to insertion, deletion of nodes and edges. Substitution cost of nodes, edges and paths are assigned proportional to their Euclidean distance.

For two graphs G_1 and G_2, we define homeomorphic graph edit cost function for all nodes $u \in V_1$, $v \in V_2$ and for all edges $e \in E_1$, $e' \in E_2$ by
$c(u \to \varepsilon) = x_{node}$
$c(\varepsilon \to v) = x_{node}$
$c(u \to v) = y_{node}.||\mu_1(u) - \mu_2(v)||$
$c(e \to \varepsilon) = x_{edge}$
$c(\varepsilon \to e') = x_{edge}$
$c(e \to e') = y_{edge}.||\nu_1(e) - \nu_2(e')||$
$c((u_1, ..., u_n) \to (u_1, u_n)) = z_{path}.||\mu_1(u_1) - \mu_1(u_n)||$
Where x_{node}, x_{edge}, y_{node}, y_{edge}, and z_{path} are non-negative parameters, and $c(u \to \varepsilon)$ is the cost of deletion of node u, $c(\varepsilon \to v)$ is the cost of insertion of node v, $c(u \to v)$ is the cost of substitution of node u by node v, $c(e \to \varepsilon)$ is the cost of deletion of edge e, $c(\varepsilon \to e')$ is the cost of insertion of edge e', and $c((u_1, ..., u_n) \to (u_1, u_n))$ is the cost of path contraction from $(u_1, ..., u_n)$ to (u_1, u_n). We observe that the substitution cost defined above is proportionate to Euclidean metric of the corresponding vertex and edge labels.

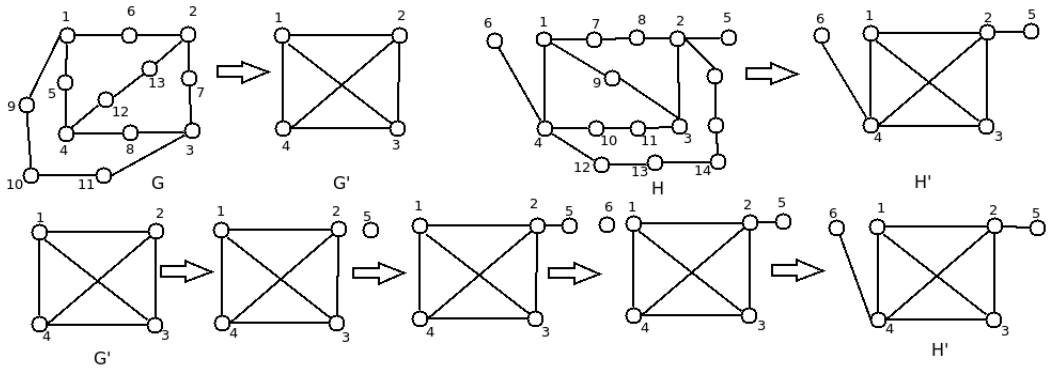

FIGURE 3.1: Homeomorphic edit path from G to H

The above graph edit function is similar to the edit cost of graph edit distance with the introduction of an additional operation $(u_1,...,u_n) \rightarrow (u_1,u_n)$ and a parameter z_{path}.

Proposition 3.3.2. *Path substitution* $c((u_1,...,u_n) \rightarrow (u_1,u_n))$ *can save up to* $(2n-2)$ *edit operations.*

Here, we can observe that $z_{path} = (n-1)x_{edge} + (n-2)x_{node} + x_{edge} = n.x_{edge} + (n-2)x_{node}$, and hence we save up to $(2n-2)$ operations for every path contraction.

An example for homeomorphic edit path computation from graph G to graph H is shown in Figure 3.1. First, G is converted to G' and H is converted to H' using path contraction, and finally G' is transformed to H' using four edit operations, these are node insertion followed by edge insertion and again node insertion followed by edge insertion.

3.3.2 Algorithm

The computation of error-tolerant graph matching using graph homeomorphism is described in Algorithm 1. The input to the Homeomorphic-Graph-Edit-Distance algorithm is two graphs G_1 and G_2, where $G_i = (V_i, E_i, \mu_i, \nu_i)$ for $i = 1,2$. The graphs G_1 and G_2 have n and m vertices respectively, and output of the algorithm is a minimum cost homeomorphic graph edit distance between G_1 and G_2. A brief description of this algorithm is as follows. The algorithm proceeds by performing path contraction on both graph G_1 and G_2. For each vertex of graph G_1, it searches for simple paths in which all intermediate vertices have degree two, and then it updates this path by removing all such intermediate vertices

Algorithm 1 : Homeomorphic-Graph-Edit-Distance (G_1, G_2)

Input: Two Graphs G_1, G_2, where $G_i = (V_i, E_i, \mu_i, \nu_i)$ for $i = 1,2$ where $V_1 = \{u_1, ..., u_n\}$ and $V_2 = \{v_1, ..., v_m\}$

Output: A min. cost homeomorphic GED between G_1 and G_2

for each $(u_i \in V_1)$ **do**
  if (there is a path $(u_i, u_{i+1}, ..., u_{i+k})$ such that $deg(u_{i+1}) = deg(u_{i+2}) = ... = deg(u_{i+k-1}) = 2$) **then**
    $(u_i, u_{i+1}, ..., u_{i+k}) \rightarrow (u_i, u_{i+k})$
    $V_1 \leftarrow V_1 \setminus \{u_{i+1}, ..., u_{i+k-1}\}$
  end if
end for
for each $(v_j \in V_2)$ **do**
  if (there is a path $(v_j, v_{j+1}, ..., v_{j+k})$ such that $deg(v_{j+1}) = deg(v_{j+2}) = ... = deg(v_{j+k-1}) = 2$) **then**
    $(v_j, v_{j+1}, ..., v_{j+k}) \rightarrow (v_j, v_{j+k})$
    $V_2 \leftarrow V_2 \setminus \{v_{j+1}, ..., v_{j+k-1}\}$
  end if
end for
Update $G_1, G_2, n \leftarrow n', m \leftarrow m'$
$A \leftarrow \emptyset$
for each $(v_j \in V_2)$ **do**
  $A \leftarrow A \cup \{u_1 \rightarrow v_j\}$
end for
$A \leftarrow A \cup \{u_1 \rightarrow \varepsilon\}$
while (1) **do**
  Prune A using optimizing techniques
  Compute min. cost graph edit path C_{min} from A
  if (C_{min} is a complete graph edit path) **then return** C_{min}
  else
    if (all nodes $(u_i \in V_1)$ are processed) **then**
      for each unprocessed $(v_j \in V_2)$ **do**
        $C_{min} \leftarrow C_{min} \cup \{\varepsilon \rightarrow v_j\}$
      end for
      $A \leftarrow A \cup \{C_{min}\}$
    else
      for (each unprocessed node $(u_i \in V_1)$) **do**
        for (each $(v_j \in V_2)$) **do**
          $C_{min} \leftarrow C_{min} \cup \{u_i \rightarrow v_j\} \cup \{u_i \rightarrow \varepsilon\}$
        end for
      end for
      $A \leftarrow A \cup \{C_{min}\}$
    end if
  end if
end while

and insert an edge between first and last vertices. After performing same path contraction operations on G_2, both graphs and their parameters are updated. The algorithm then initializes an empty set A. Simultaneous substitution of first vertex u_1 of G_1 with every other vertex of G_2 is inserted in A along with the deletion of u_1. After that, *while* loop is executed until we get a minimum cost edit distance C_{min}, which is also a complete graph edit path. In *while* loop, pruning on A is performed to reduce the search space using various optimizing techniques and heuristic methods. If C_{min} is the complete edit path, i.e., the set of graph edit path is identical to input graph, then algorithm returns this value. Otherwise, if all nodes of G_1 are processed and some nodes of G_2 are not processed then these unprocessed nodes are inserted in C_{min}, and A is updated in the inner loop of *if* part. Finally, in the inner loop of the *else* part, every unprocessed node of G_1 is substituted by all nodes of second graph G_2 and these substitutions are inserted in C_{min} along with the deletion of unprocessed nodes of G_1 and A is updated.

The correctness of Homeomorphic-Graph-Edit-Distance algorithm can be established using Proposition 3.3.1. The algorithm reduces the number of vertices in input graph, which lowers the search space required for further processing of edit distance computation. Various pruning strategies can be used to decrease the total execution time. Like, we can keep a fixed number of vertices in the set A at any given time and whenever there is an update in A only that much minimum cost fix entries are retained out of all available edit path entries.

3.4 Extended Graph Edit Distance

The importance of error-tolerant graph matching is based on the premise that it can be used to accommodate errors that may have been incurred in the input graph due to the presence of noise or distortions during the processing or retrieval steps. For a dense graph, if the distortion starts with the addition (or removal) of a node or an edge followed by successive additions (or removal) of nodes or edges, then to perform the graph matching it may be reasonable to ignore the smallest degree nodes, if we want to do the matching using as few number of nodes of graph as possible. As an example, suppose we have a graph G as shown in Figure 3.2, and due to distortion or presence of noise, additional nodes

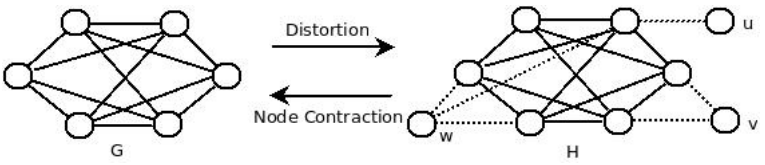

FIGURE 3.2: Node contraction on the graph H

(u,v,w) got added, shown to be connected with dotted edges in graph H. After removing or contracting the 1-degree, 2-degree and 3-degree nodes, we get the same graph as G.

The idea behind the node contraction is to ignore the less important nodes in the graph before proceeding to the process of graph matching. One of the measures of importance or centrality of nodes in a graph is degree centrality. Considering the degree centrality as a node importance indicator, we ignore or delete the nodes starting from least degree nodes. By simply removing nodes from the graph will change the topology of the graph drastically. To use a middle path, where the topology of the graph does not get modified abruptly, we delete the nodes and its associated edges, only when this does not disconnects the graph. In other words, we remove the smaller degree nodes provided that the total number of connected components of the graph remains the same. Keeping in mind the above point of view, we can extend the graph edit distance to ignore the smaller degree nodes.

Definition 3.4.1. *Node contraction* of a node u in a graph G is simply the deletion of the node u and its associated links provided that the node u is not a cut vertex.

Cut vertex is a vertex whose removal disconnects the graph. It is also called as cut point or articulation point.

Definition 3.4.2. *k-degree node contraction* is the operation of performing the node contraction of all the nodes of degree k of a graph G. We use kNC(G) to denote k-degree node contraction on a graph G.

Definition 3.4.3. *k^*-degree node contraction* is the process of applying k-degree node contraction consecutively from one degree to k degree on a graph G.

If we denote k^*-degree node contraction on a graph G by k^*-$NC(G)$ then k^*-$NC(G)$=$kNC((k-1)NC((k-2)NC(...1NC(G)...)))$.

Definition 3.4.4. *k-GED* is the the computation of graph edit distance between two graphs with k-degree node contraction performed on both graphs.

For two graphs G_1 and G_2, we have

$$k\text{-}GED(G_1, G_2) = GED(kNC(G_1), kNC(G_2))$$

For $k = 2$, k-GED generalizes to HGED [81], when path contraction is performed so that all simple paths $(u_1, ..., u_n)$ are replaced by (u_1, u_n) where $Deg(u_i) = 2$ for $i = 2$ to $n-1$.

Definition 3.4.5. *k^*-GED* is the computation of graph edit distance between two graphs with k-degree node contraction performed on both graphs starting from one-degree nodes to k degree nodes.

Let G_1 and G_2 be two graphs, then

$$k^*\text{-}GED(G_1, G_2) = \begin{cases} GED(G_1, G_2), \text{if } k = 0 \\ GED(k^*\text{-}NC(G_1), k^*\text{-}NC(G_2)), \text{if } k \geq 1 \end{cases}$$

To make the effect of node contraction on graph topology more explicit, we define *node deletion*, as a standard removal of a node and its associated edges, regardless of whether or not it is a cut vertex. Similar to k-degree node contraction, k^*-degree node contraction (k^*-NC), k-GED and k^*-GED, we define k-degree node deletion, k^*-degree node deletion (k^*-ND), k-ND-GED and k^*-ND-GED respectively, where in each definition node contraction operation is replaced by node deletion. Here, we observe that $|k^*\text{-ND}(G)| \leq |k^*\text{-NC}(G)|$, where $|G|$ denotes size or number of nodes in graph G. Since the removal of nodes in node deletion operation is unrestricted; therefore the number of nodes in a graph G after node deletion is less than or equal to that of the number of nodes after node contraction.

3.4.1 Extended Edit Cost

To define the edit cost of k-GED, we can extend the edit cost of graph edit distance by adding $c(u \to \varepsilon) = 0$, whenever $deg(u) = k$ and u is not a cut vertex of the graph.

k-GED uses Euclidean distance measure and assigns the constant cost to insertion, deletion and substitution of nodes and edges. Let G_1 and G_2 be two graphs, for all nodes $u \in V_1$,

$v \in V_2$ and for all edges $e \in E_1$, $f \in E_2$, we define the extended edit cost function as follows.
$c(u \to \varepsilon) = x_{node}$
$c(\varepsilon \to v) = x_{node}$
$c(u \to v) = y_{node}.||\mu_1(u) - \mu_2(v)||$
$c(e \to \varepsilon) = x_{edge}$
$c(\varepsilon \to f) = x_{edge}$
$c(e \to f) = y_{edge}.||\nu_1(e) - \nu_2(f)||$
$c(u \to \varepsilon) = 0$, if $deg(u) = k$ and u is not a cut point
Here, $c(u \to \varepsilon)$ denotes the cost of deletion of node u, $c(\varepsilon \to v)$ stands for the cost of insertion of node v, $c(u \to v)$ is the cost of substitution of node u by node v, $c(e \to \varepsilon)$ denotes the cost of deletion of edge e, $c(\varepsilon \to f)$ is the cost of insertion of edge f, and x_{node}, y_{node}, x_{edge}, y_{edge} are non-negative constants.

We observe that the above cost function satisfy the non-negativity property, i.e., $c(e_i) \geq 0$, for every node and edge edit operations e_i along with the triangle inequality property of insertion, deletion and substitution of nodes and edges.

3.4.2 Algorithm

In this section, we describe the algorithm to perform error-tolerant graph matching using node contraction. The computation of k^*-degree node contraction of an input graph is described in Algorithm 2. The input to the k^*-Node-Contraction algorithm is a graph G and a parameter k. The outer *for* loop of the algorithm in lines 1–16, iteratively perform k-degree node contraction from 1 to k. The *for* loop of the algorithm in lines 2–8, uses a boolean flag *visit*, which is set to 1 when the degree of a vertex of G is k else it is reset to 0. If the visit flag of a node is set to 1 and it is not a cut vertex, then the node and its associated edges are removed from G and visit flag is reset to 0 in the *for* loop of lines 9–15. Finally, the transformed graph G is returned in line 17. This algorithm can be considered as a preprocessing phase of the proposed error-tolerant graph matching framework.

Proposition 3.4.1. *k^*-Node-Contraction algorithm computes k^*-degree node contraction of G_1 and G_2.*

We can observe that the *for* loop in lines 1–16 of the Algorithm 2, ensures that k-degree node contraction starts from i=1 to k. *If* loop in line 10, allow only those nodes to be

Algorithm 2 : k^***-Node-Contraction** (G,k)

Input: A graph $G = (V,E,\mu,\nu)$ and a parameter k
Output: Transformed graph after applying k^*-degree node contraction on G

```
1:  for (i ← 1 to k) do
2:      for each (u ∈ G) do
3:          if ((deg(u)==k)) then
4:              u.visit ← 1
5:          else
6:              u.visit ← 0
7:          end if
8:      end for
9:      for each (u ∈ G) do
10:         if ((u.visit==1)&&(u is not cut vertex)) then
11:             V ← V \ {u}
12:             E ← E \ {(u,v)|(u,v) ∈ E, ∀v ∈ G}
13:             u.visit ← 0
14:         end if
15:     end for
16: end for
17: return G
```

removed, whose degree is k and it is not a cut vertex, whereas *visit* flag make sure that each vertex is considered only once for contraction.

Proposition 3.4.2. *k^*-Node-Contraction algorithm executes in $O(n)$ time.*

We can check whether a node is cut vertex in $O(n)$, and therefore *for* loop in lines 9–15 takes $O(n)$. *For* loop in lines 2–8 also take $O(n)$ time, while the outer *for* loop in lines 1–16 executes k times. So the Algorithm 2 takes overall $O(k.n)$, which is $O(n)$ time.

The k^*-GED computation of two graphs is described in Algorithm 3. A brief description of the k^*-Graph-Edit-Distance algorithm is as follows. The input to the algorithm is two graphs G_1, G_2, and a parameter k, and the output is the minimum cost k^*-GED between G_1 and G_2. It calls the Algorithm 2 to perform k^*-degree node contraction on G_1 and G_2 in lines 1–2. The transformed graphs are G'_1 and G'_2 with vertex set $V'_1 = \{u'_1,...,u'_{n'}\}$ and $V'_2 = \{v'_1,...,v'_{m'}\}$ respectively. The algorithm initializes an empty set S in line 3. In the *for* loop in lines 4–6, S is updated by substitution of vertex u'_1 of G'_1 with each vertex of G'_2, then deletion of u'_1 in line 7 is added to S. The *while* loop in lines 8–28, is used to compute the minimum cost edit path C_{min}, from S.

Algorithm 3 : k^***-Graph-Edit-Distance** (G_1, G_2)

Input: Two Graphs G_1, G_2, where $G_i = (V_i, E_i, \mu_i, \nu_i)$ for $i = 1,2$ where $V_1 = \{u_1, ..., u_n\}$ and $V_2 = \{v_1, ..., v_m\}$ and a parameter k

Output: A min. cost k^*-GED between G_1 and G_2

1: $G_1' \leftarrow k^*$-Node-Contraction(G_1, k) // $V_1' = \{u_1', ..., u_{n'}'\}$
2: $G_2' \leftarrow k^*$-Node-Contraction(G_2, k) // $V_2' = \{v_1', ..., v_{m'}'\}$
3: $S \leftarrow \emptyset$
4: **for** all $(v_i' \in V_2')$ **do**
5: $S \leftarrow S \cup \{u_1' \rightarrow v_i'\}$
6: **end for**
7: $S \leftarrow S \cup \{u_1' \rightarrow \varepsilon\}$
8: **while** $(True)$ **do**
9: Prune S using heuristic methods
10: Compute C_{min} the min. cost edit path from S
11: **if** (C_{min} is a complete edit distance path) **then**
12: **return** C_{min}
13: **else**
14: **if** (every node $(u_i' \in V_1')$ is processed) **then**
15: **for** every unprocessed $(v_j' \in V_2')$ **do**
16: $C_{min} \leftarrow C_{min} \cup \{\varepsilon \rightarrow v_j'\}$
17: **end for**
18: $S \leftarrow S \cup \{C_{min}\}$
19: **else**
20: **for** (each unprocessed node $(u_i' \in V_1')$) **do**
21: **for** (every $(v_j' \in V_2')$) **do**
22: $C_{min} \leftarrow C_{min} \cup \{u_i' \rightarrow v_j'\} \cup \{u_i' \rightarrow \varepsilon\}$
23: **end for**
24: **end for**
25: $S \leftarrow S \cup \{C_{min}\}$
26: **end if**
27: **end if**
28: **end while**

Computation of minimum cost edit path is usually performed using tree-based search algorithm like A^* search where the top node or root denotes the first edit operation and bottom or leaf node represents the last edit operation. Edit path from the root to leaf nodes constitutes a complete edit distance path, and it exhibits an edit path to transform a graph to another one which may or may not be optimal. If we want to return optimal edit path, then during each level of search we have to consider all the combinations of edit operations in the set S, which can be computationally much expensive. An alternative is to prune the set S (line 9) using various heuristic techniques to consider only the partial set of edit operations

which leads to a suboptimal but relatively efficient solution. For example, we can use beam search to limit the search space by considering only w best possibility at each level of search, where w is called as beam width. During the execution of the algorithm, if C_{min} is the complete edit distance path then the algorithm returns its value in line 12; otherwise all the unprocessed nodes of G'_2 are added in C_{min} and S is updated in lines 14–18. Finally, every unprocessed node of G'_1 are substituted by each node of G'_2 and these operations are added to C_{min}, together with the deletion of unprocessed nodes of G'_1 in lines 20–24, and S is updated in line 25.

Proposition 3.4.3. *k^*-Graph-Edit-Distance algorithm computes error-tolerant graph matching of G'_1 and G'_2.*

It follows from the properties of edit operations, i.e., addition, deletion and substitution of nodes and edges along with the definition of k^*-GED. A complete edit path returned by Algorithm 3 ensures that each node of G'_1 is uniquely mapped to G'_2, while Algorithm 2 makes sure that the modified vertex set V'_1 and V'_2 are subsets of V_1 and V_2 respectively.

The worst case complexity of k^*-Graph-Edit-Distance algorithm is exponential on the number of nodes in input graphs, but a suitable parameter k can be selected to reduce the overall execution time. Let us consider a tree-search based method implementation of k^*-Graph-Edit-Distance algorithm. At the first level of execution u'_1 is replaced by each of the v'_i for $i = 1$ to m', making it $O(m')$. Suppose substitution of u'_1 to v'_1 is selected at the first level, then during the next level u'_2 is again replaced by each of the v'_i for $i = 2$ to m', which is $O(m')$, making total time $O(m'^2)$ up to the second level. Similarly, at n'th level, the worst-case execution time would become $O(m'^{n'})$. As discussed in Algorithm 3, a heuristic method like beam search can be used to reduce the average case execution time of algorithm at the cost of getting a suboptimal solution.

To consider the effect of k on the size of the transformed graphs n' and m'. We observe that value of n' and m' is highly dependent on the topology and size of the input graphs. Let $G(n, p)$ be a random graph, where n is the number of nodes and p is the probability of edges between each pair of vertices. Then the vertex degree distribution, which is the number of vertices of degree k is given by $p_k = \binom{n-1}{k} p^k (1-p)^{n-k-1}$. Now, we have the following loose relationship between n and n'.

Proposition 3.4.4. *For a random graph $G(n,p)$, the value of n and n' as used in Algorithm 3 satisfy the following inequality*

$$n - \sum_{n=1}^{k} \binom{n-1}{k} p^k (1-p)^{n-k-1} \leq n' \leq n - (n-1)p(1-p)^{n-2}.$$

Proof. Here, n is the number of nodes in G_1 and n' is the number of nodes in G_1' obtained after k^*-NC(G_1). So, the number of contracted nodes are $n-n'$. We have, number of nodes removed during k^*-NC$(G_1) \leq n-n' \leq$ number of nodes removed during k^*-ND(G_1), which implies $(n-1)p(1-p)^{n-2} \leq n-n' \leq \sum_{n=1}^{k} \binom{n-1}{k} p^k (1-p)^{n-k-1}$, where left most term is the number of nodes removed during 1^*-NC or 1^*-ND, since both are equivalent operations. Now multiplying the above expression by -1 and rearranging the terms $-\sum_{n=1}^{k} \binom{n-1}{k} p^k (1-p)^{n-k-1} \leq n'-n \leq -(n-1)p(1-p)^{n-2}$, by adding n in above expression we get $n - \sum_{n=1}^{k} \binom{n-1}{k} p^k (1-p)^{n-k-1} \leq n' \leq n-(n-1)p(1-p)^{n-2}$. □

3.5 Results and Discussion

3.5.1 Homeomorphic Graph Edit Distance

We executed Homeomorphic-Graph-Edit-Distance algorithm on random graphs with a given number of nodes. The comparison of execution times of homeomorphic graph edit distance computation versus graph edit distance computation without any optimization or pruning technique is shown in Figure 3.3. Here, each execution time is the mean value of several execution times taken by graph edit distance computation and homeomorphic graph edit distance computation for different random graphs with fixed nodes. Simple graph edit distance computation usually did not terminate beyond 10 number of nodes, even though homeomorphic graph edit distance computation was able to complete the execution well beyond this size.

The homeomorphism is basically a topological concept, which encapsulates the topological similarity of different objects. In fact, two homeomorphic objects or spaces are called topologically equivalent. This notion can be extended to measure the structural similarity of graphs. A set of homeomorphic graphs, exhibit some kind of similarity, which

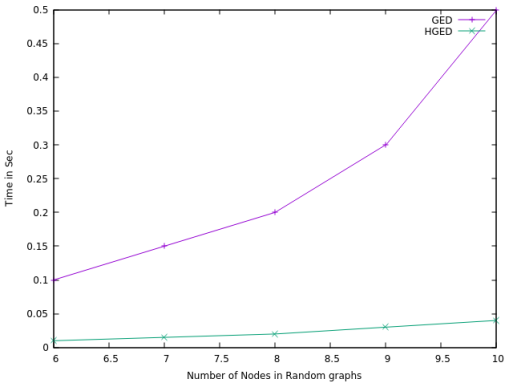

FIGURE 3.3: GED vs HGED computation

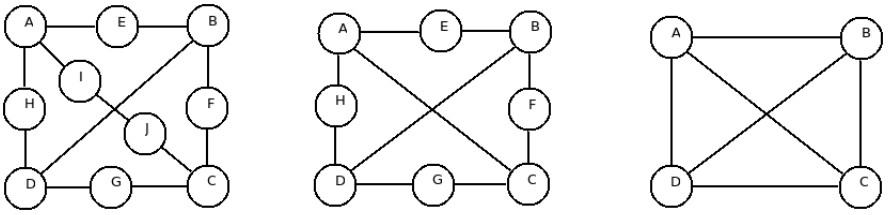

FIGURE 3.4: Homeomorphic Graphs

is denoted by homeomorphic graph edit distance. All graphs which are homeomorphic with respect to each other, will have same homeomorphic graph edit distance. For example, the homeomorphic graphs shown in Figure 3.4, have same homeomorphic graph edit distance.

This idea can be particularly helpful in real world applications, where data can be modified by the presence of noise and distortion. We can use Homeomorphic-Graph-Edit-Distance algorithm for error-tolerant graph matching, which has a wide range of applications ranging from pattern recognition, biometric identification, image processing to biological can chemical applications.

The proposed homeomorphic graph edit distance can be seen as 2-degree path contraction, since it is obtained by removing all the vertices along a path having degree 2. It can further be generalized to n-degree vertices path contraction, where a given graph will be transformed to another graph by removing all the vertices having up to degree n along any simple path of the input graph. The parameter n can be selected in an application specific manner.

3.5.2 Extended Graph Edit Distance

In this section, we compare the proposed graph matching framework with other important graph matching methods. To evaluate the execution time and accuracy of k^*-Graph-Edit-Distance, we use the IAM graph database [109] and graph matching toolkit [110]. We use the letter database and AIDS database of IAM graph repository for the results.

3.5.2.1 Dataset Description

Letter graphs consist of capital letters of alphabets, drawn using straight lines only. It includes 15 classes of capital letters namely A, E, F, H, J, K, L, M, N, T, V, W, X, Y, and Z. For each prototype graph, distortions of three distinct levels, i.e., high, medium and low are applied to generate various graph dataset. The graphs of letter dataset are uniformly distributed across the 15 letters. All nodes are labeled with an (x, y) coordinate representing its position in a reference plane. The average number of nodes and edges per graph in letter graph data of high distortion level is 4.7 nodes and 4.5 edges respectively. The maximum number of nodes and edges per graph for this graph dataset are both 9. AIDS dataset contains graph characterizing chemical compounds. It consists of two different class of molecular compounds, i.e., confirmed active and confirmed inactive. Active class molecules show activity against HIV, whereas inactive class represents inactivity against HIV. Labels on node denote chemical symbol whereas labels on edges exhibit valence. In AIDS dataset average number of nodes per graph is 15.7 nodes while the average number of edges per graph is 16.2 edges. The maximum number of nodes and edges per graph is 95 nodes and 103 edges respectively. In Figure 3.5 we can observe the fraction of graphs concerning their size for active and inactive molecules of the training set of AIDS dataset. We note that the number of nodes in active molecules is usually more than that of inactive molecules.

3.5.2.2 Execution Time Comparison

All results in this section are computed using the system having 9.8 GB of memory and running the processor at 3.40 GHz. Comparison of average execution time of graph matching in milliseconds for GED and k^*-GED (where k=1,2 and 3) for letter graphs, using

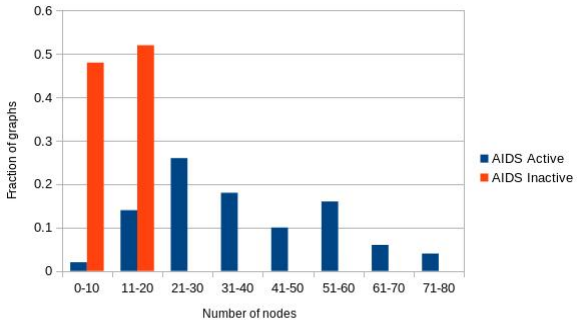

FIGURE 3.5: Fraction of graphs with their size for AIDS dataset

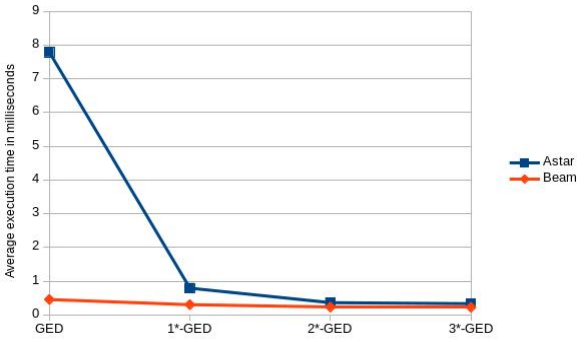

FIGURE 3.6: GED vs. k^*-GED computation on letter graphs

A^* and with beam search optimization having beam width $w = 10$ is shown in Figure 3.6. There is a sharp change in running time from GED to 1^*-GED using A^*. This decrease in execution time accounts for reduction in 1-degree nodes in the input graphs.

Comparison of running time for GED and k^*-GED for AIDS graphs using A^* is given in Figure 3.7, whereas Figure 3.8 shows the time in *ms* with beam search heuristic. Here also, there is a sudden decrease in execution time from GED to 1^*-GED using A^*, which becomes lesser from 1^*-GED to 2^*-GED and further from 2^*-GED to 3^*-GED. This is primarily due to reason that, we can not remove easily the 2-degree and 3-degree nodes from the graph as compared to 1-degree nodes without making the graph disconnected.

3.5.2.3 Accuracy Comparison

For accuracy evaluation, we consider the problem of classification of letter graphs using nearest neighbor classifier. We use 750 letter graphs of high distortion of the test dataset, which consists of 50 graphs for each of the 15 classes of letter dataset. We also use

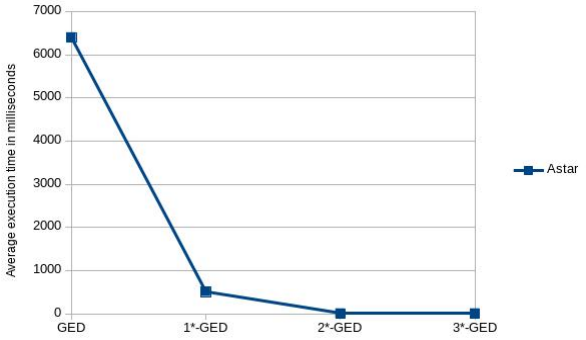

FIGURE 3.7: *GED* vs. *k*GED* computation on AIDS graphs

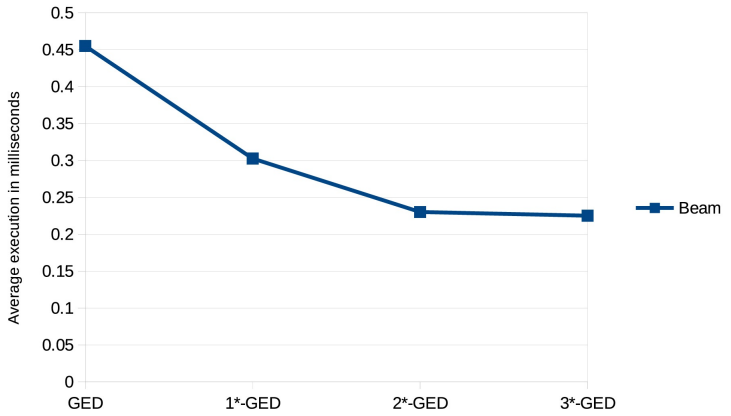

FIGURE 3.8: *GED* vs. *k*GED* computation on AIDS graphs using beam search

750 graphs of training dataset, which again includes 50 graphs for each of the 15 letters. Comparison of classification accuracy for each of the 15 letters using A^* based GED and k^*-GED for $k = 1$ to 3 is shown in Table 3.1, whereas classification accuracy of GED and k^*-GED based on beam search is shown in Table 3.2. Here, we can observe that letters having less number of straight lines have less accuracy while letter having more straight lines have usually higher accuracy using k^*-GED. For example letters I,L,T, and V have less accuracy whereas letters M,W,Y, and Z have higher accuracy. We also note that the accuracy of k^*-GED remains almost same with or without beam search, as the number of nodes in the letter graphs obtained after k^*-GED becomes so small such that the edit path considered by beam search becomes identical to that of A^*.

To evaluate accuracy on AIDS dataset, We use 1500 graphs of test dataset including 300 graphs from the active class and 1200 graphs from the inactive class of AIDS molecules. We also use 250 graphs of training dataset consisting of 50 graphs of active class and 200 graphs of inactive class. Comparison of classification accuracy of AIDS dataset for GED

TABLE 3.1: Accuracy on letter dataset using GED and k^*-GED

Class	GED	1*-GED	2*-GED	3*-GED
A	94	70	64	64
E	80	54	54	54
F	80	64	44	42
H	70	42	30	30
I	94	06	06	06
K	86	44	38	38
L	86	56	36	36
M	96	90	60	60
N	86	64	52	52
T	90	50	46	44
V	90	48	46	46
W	94	78	56	54
X	78	52	50	50
Y	92	76	52	52
Z	90	80	64	64

TABLE 3.2: Accuracy on letter dataset using GED and k^*-GED with beam

Class	GED	1*-GED	2*-GED	3*-GED
A	90	70	64	64
E	86	54	54	54
F	84	64	44	42
H	60	42	30	30
I	94	06	06	06
K	80	44	38	38
L	86	56	36	36
M	94	90	60	60
N	84	64	52	52
T	88	50	46	44
V	88	48	46	46
W	92	78	56	54
X	76	52	50	50
Y	92	76	52	52
Z	90	80	64	64

TABLE 3.3: Accuracy on AIDS dataset using GED and k^*-GED with beam

Class	Active	Inactive
GED with beam	98.6	99.9
1^*-GED with beam	98.6	99.3
2^*-GED with beam	97.3	99.2
3^*-GED with beam	96.6	99.1
4^*-GED with beam	96.6	99.1

TABLE 3.4: Accuracy on AIDS dataset using Bipartite GED and k^*-GED

Class	Active	Inactive
Bipartite GED	98.6	99.9
Bipartite 1^*-GED	98.6	99.3
Bipartite 2^*-GED	97.3	99.2
Bipartite 3^*-GED	97.0	99.2
Bipartite 4^*-GED	96.6	99.1

using beam search heuristic with k^*-GED for $k = 1$ to 4 using beam search heuristic having beam width $w = 10$ is shown in Table 3.3. Also, the comparison of classification accuracy of AIDS dataset using bipartite GED with bipartite k^*-GED for $k = 1$ to 4 is shown in Table 3.4. We can observe that classification accuracy for AIDS dataset is more than that of letter dataset. This increased accuracy shows that this graph matching model is more relevant for large graphs, as the contraction of nodes in small graphs tends to modify the graph more as compared to graphs with a large number of nodes.

3.5.2.4 Topological Considerations and Discussion

Node contraction operation changes the topology of the input graphs. We can observe the effect of k^*-NC on the graph topology of the training set of letter A in Figure 3.9. For comparison purpose, we have included the effect of k^*-ND on the structure of the same graphs. We can also examine the effect of k^*-NC and k^*-ND on the topology of graphs of the training set of active molecules of AIDS dataset in the Figure 3.10. Here, we can notice the abrupt change in the average number of nodes in k^*-ND from $k = 1$ to 3, as it

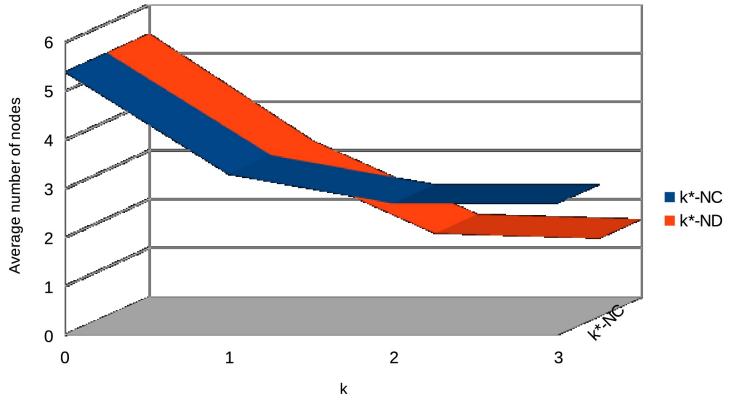

FIGURE 3.9: Effect of k^*-NC and k^*-ND on the topology of Letter A

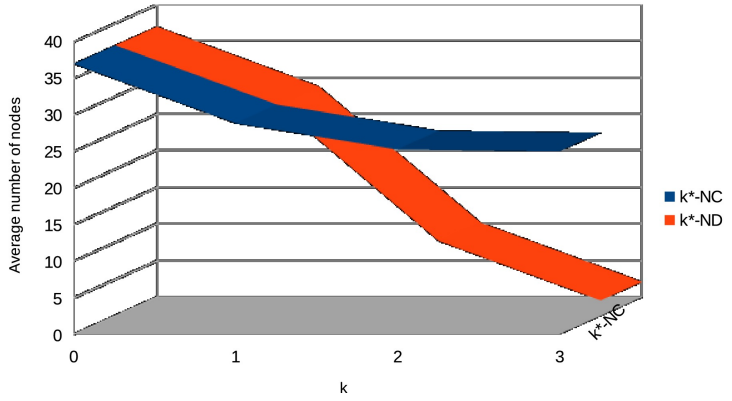

FIGURE 3.10: Effect of k^*-NC and k^*-ND on the topology of active molecules of AIDS dataset

deletes almost the entire structure of graphs. Therefore k^*-NC can be used as a compromise between change in topology and efficiency because it changes the structure of the graphs up to a certain extent only.

The proposed method can be useful in applications, where graph representation is relatively dense and have a large number of nodes. Depending on the requirement of time and exactness of similarity, we can proceed from $k = 0$ to $k = n$. If exact matching is must regardless of time spent, we have to use $k = 0$, which is nothing but standard graph edit distance. If the exact similarity is more important, then we can proceed from $k = 1$, otherwise if time is more crucial, then we can proceed from $k = n$. Where n is the minimum of the second highest degree of nodes between G_1 and G_2. It can also be used as an online or anytime graph matching scheme, where we need to find a matching under some time

constraint. As mentioned above, k^*-ND-GED which allows the deletion of nodes and its associated edges, even if it is a cut vertex can provide a further efficiency but may be at the cost of much less accuracy.

3.6 Summary

In this chapter, we described an approach to error-tolerant graph matching using the concept of graph homeomorphism. We presented Homeomorphic-Graph-Edit-Distance algorithm, which uses path contraction to decrease the number of vertices in the input graphs leading to the reduction in the computation of graph edit distance.

We also presented an approach to error-tolerant graph matching using node contraction. It works by removing the nodes based on their degree centrality as a measure of importance. We deleted the least degree nodes provided this does not increase the connected components. k^*-Node-Contraction can be used as a preprocessing step on the top of any graph matching algorithms. We implemented the proposed method on the letter and AIDS dataset, and the result shows efficiency in execution time at the cost of a decrease in accuracy. Results on AIDS dataset are more promising, which shows that the approach is more suitable for large graphs. We have also described the effect of node contraction on the topology of graphs. In general, the algorithm achieves efficiency without disturbing the topology of graphs too much.