An Incremental Partitioning Graph Similarity Search Based on Tree Structure Index

Yuhang Li[1], Yan Yang[1,2(✉)], and Yingli Zhong[1]

[1] School of Computer Science Technology, Heilongjiang University, Harbin, China
yangyan@hlju.edu.cn
[2] Key Laboratory of Database and Parallel Computing of Heilongjiang Province, Harbin, China

Abstract. Graph similarity search is a common operation of graph database, and graph editing distance constraint is the most common similarity measure to solve graph similarity search problem. However, accurate calculation of graph editing distance is proved to be NP hard, and the filter and verification framework are adopted in current method. In this paper, a dictionary tree based clustering index structure is proposed to reduce the cost of candidate graph, and is verified in the filtering stage. An efficient incremental partition algorithm was designed. By calculating the distance between query graph and candidate graph partition, the filtering effect was further enhanced. Experiments on real large graph datasets show that the performance of this algorithm is significantly better than that of the existing algorithms.

Keywords: Graph similarity search · Similarity search · Graph partition

1 Introduction

In recent years, a large number of complex and interrelated data has grown. In order to ensure the integrity of data structure, it is modeled as a graph, and graph search operations are frequently used in database retrieval, so it is widely concerned. Due to the inevitable natural noise and human error input in real life, it is very necessary to search the similarity of graphs. It aims to search the set of graphs similar to the query graphs specified by users in large graph database. At present, there are many indicators to measure the similarity of graphs, such as the largest common subgraph [1, 2], edge missing [3]. Among them, the most commonly used similarity measure is the graph editing distance (GED), which can not only accurately capture the structural differences between graphs, but also can be applied in various fields, such as computer vision handwriting recognition [4], molecular analysis of compounds [5].

Therefore, this paper studies graph similarity search based on graph editing distance constraint: given a graph database and a query graph, the user gives GED threshold, the graph set in query graph database with graph editing distance within the threshold. The graph editing distance $GED(G_1, G_2)$ refers to two graphs G_1 and G_2. It is the minimum number of operations to convert G_1 to G_2 by inserting and deleting vertices/edges or relabel labels. But calculating the GED of two graphs has been proved to be NP hard [6].

J. Zeng et al. (Eds.): ICPCSEE 2020, CCIS 1257, pp. 16–32, 2020.
https://doi.org/10.1007/978-981-15-7981-3_2

At present, the most widely used method to calculate the GED is A* algorithm [7], and literature [2] shows that the A* algorithm cannot calculate the editing distance of graphs with more than 12 vertices. Therefore, the solution of search problem in large graph database adopts the frame of filter and verification. Generally, a set of candidate graphs is generated under some constraints, and then GED is calculated for verification. At present, many different GED lower bounds and pruning technologies have been proposed, such as k-AT [8], GSimSearch [9], DISJOINT-PARTITION BASED FILTER [10], c-star [11], but they all share a lot of common substructures, resulting in the loose GED lower bounds. Inves [15] proposes a method of incremental graph partition, and considers the distance between partitions, but the filtering conditions of this method are too loose. Based on Inves, this paper makes the following contributions:

- We study the idea of incremental partition from a new perspective, and use the idea of q-gram to enhance the partition effect, so as to reduce the number of candidate sets.
- We propose a clustering index framework, which optimizes the K-Means clustering, uses the dictionary tree structure as the index, uses the dynamic algorithm to divide the incremental graph, and groups the similar graph according to the graph division structure to enhance the filtering ability.
- We have carried out a wide range of experiments on datasets, and the results show that the algorithm in this paper is significantly better than the existing algorithm in performance.

2 Preliminary

2.1 Problem Definition

For the convenience of illustration, this paper focuses on simple undirected label graph, and the method can also be extended to other types of graphs. Undirected label graph G is represented by triple (V_g, E_g, L_g), where V_g is the vertex set, $E_g \subseteq V_g \times V_g$ is the edge set, L_g is a function that maps vertices and edges to labels. $V_g(u)$ is the label of vertex u, $V_g(u)$ is the label of edge (u, v). In practical applications, labels can be expressed as properties of vertices and edges, such as chemical composition of compounds, chemical bonds, etc.

The editing operation of a graph refers to the editing operation of converting from one graph to another, including:

– Insert an isolated labeled vertex in the graph
– Remove an isolated labeled vertex from the graph
– Insert a labeled edge in the graph
– Delete a labeled edge in the graph
– Relabel a vertex label
– Relabel an edge label

The graph editing distance between graph G_1 and G_2 refers to the minimum number of graph editing operations converted from G_1 to G_2, expressed as $GED(G_1, G_2)$.

Example 1. Figure 1 shows two graphs G_1 and G_2, in which the vertex label is a chemical molecule and the edge label (single line and double line) is a chemical bond. G_1 can be converted to G_2 by changing u_3 label to 'S', u_4 label to 'N', deleting double key edge (u_6, u_7), and insert single key edge (u_6, u_7). Therefore, $GED(G_1, G_2) = 4$.

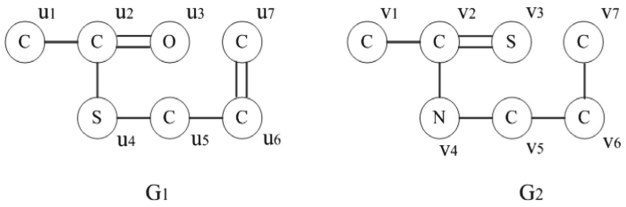

Fig. 1. Data graph G_1 and G_2

Definition 1. (graph similarity search problem) Given graph database $D = \{G_1, \ldots, G_n\}$, a query graph Q, and GED threshold τ, the problem of graph similarity search is to find the set of graphs satisfying $GED(G_i, Q) \leq \tau, G_i \in D$.

2.2 Related Work

Recently, much attention has been paid to graph similarity search. The previous work is to use the overlapping substructures of different graphs to filter, and then carry out expensive GED calculation.

k-AT [8] is inspired by the idea of q-gram of approximate string matching. By decomposing the graph into several subgraphs, the lower limit of editing distance is estimated by using the number of common subgraphs, and the inverted index is established by using the extracted features. However, k-AT algorithm is only suitable for sparse graphs. GSimSearch [9] proposes path based q-grams, which uses matching and mismatching features, and proposes a local label filtering method in its verification phase. Although it solves the limitation of k-AT, the paths overlap each other, and the partition size is fixed, resulting in poor pruning function.

c-star [10] uses 1-gram to construct star structure based on k-AT, and uses binary matching between star structures to filter. On the basis of traditional c-star, DISJOINT-PARTITION BASED FILTER [11] will tighten the filter lower bound of c-star by removing the leaf nodes, but this method still shares many common substructures, making the editing distance lower limit too loose. CSI_GED [2] based on the combination of edge mapping and backtracking algorithm, in the verification phase faster calculation of the graph GED. GBDA [12] proposes the graph branch distance and estimates the graph edit distance using probability method, but when calculating the prior distribution, the sampling of data is not accurate.

Pars [13] adopts random graph partition strategy, and proposes non overlapping and variable size graph partition. ML_index [14] uses vertex and edge tag frequency to define partition, and proposes selective graph partition algorithm, so as to improve filtering performance. Both Pars and ML_index use graph as index features, but large graph creation index costs a lot. In this paper, the dynamic algorithm is used to divide the graph incrementally, and a clustering index based on dictionary tree is proposed to enhance the filtering performance, so as to ensure its search performance in large-scale graph database.

3 IP-Tree Algorithm

3.1 Partition Based Filtering Scheme

Because it is very expensive to calculate the GED of each graph and query graph in a large graph database, this paper uses the filter and verification framework to calculate the GED lower bound between the candidate graph and query graph by using the partition method before calculating the GED accurately, so as to filter the candidate set and reduce the cost of calculating the GED accurately. Before we formally introduce the filter framework, we start with defining the induced subgraphs of graph partitions.

Definition 2. (induced subgraph isomorphism) Given a graph G_1 and a graph G_2, if there is a mapping $f : V_{G_1} \rightarrow V_{G_2}$ satisfying (1) $\forall u \in V_{G_1}, f(u) \in V_{G_2} \wedge L_{G_1}(u) = L_{G_2}(f(u))$ (2) $\forall u \in V_{G_1}, \forall v \in V_{G_2}, L_{V_{G_1}}(u, v) = L_{V_{G_2}}(f(u), f(v))$, it is said that the graph G_1 is the induced subgraph isomorphism of the graph G_2, expressed as $G_1 \subseteq G_2$.

Example 2. Figure 2 shows three graphs P_1, P_2 and G_1. It can be found that P_1 is not the induced subgraph isomorphism of G_1, because $L_{P_1}(u_2, u_3) \neq L_{G_1}(u_2, u_3)$, P_2 is the induced subgraph isomorphism of G_1.

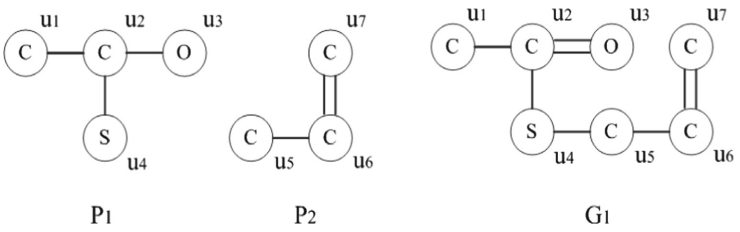

Fig. 2. Induced subgraph isomorphism

Definition 3. (graph division) The division of graph G is $P(G) = \{p_1, \ldots, p_k\}$, where p_k satisfies the condition: (1) $\forall i, p_i \subseteq G$ (2) $\forall i, j, V_{p_i} \cap, V_{p_j} = \varnothing$ *s.t.* $i \neq j$ (3) $V_G = \cup_{i=1}^{k} V_{p_i}$.

Definition 4. (partition matching) Given a graph G_1 and G_2, a partition $p \in P(G_1)$, if $p \subseteq G_2$, then partition p is the matching partition of graph G_2, otherwise partition p is the mismatching partition of graph G_2.

Lemma 1. Given graph G_1 and graph G_2 and partition $P(G_1)$ of graph G_1, $lb(G_1, G_2) = |p|p \subseteq P(G_1) \bigwedge p \nsubseteq G_2|$ is a lower bound of $GED(G_1, G_2)$.

Proof. Because G_1 partitions do not overlap and the editing operation of each partition does not affect other partitions, there is at least one editing operation from G_1 to G_2 for each mismatched partition.

Corollary 1. Given graph G_1, G_2, the division of graph G_1 and given GED threshold τ, if $lb(G_1, G_2) > \tau$, then graph G_1 is pruned and GED does not need to be calculated.

Example 3. Given two graphs G_1 and G_2 as shown in Fig. 1, the GED threshold τ. If we divide G_1 into $\{p_1, p_2, p_3\}$ according to the method shown in Fig. 3(a), we need to calculate GED of G_1 and G_2 according to corollary 1. If we divide G_1 into $\{p_1', p_2', p_3'\}$ according to the method shown in Fig. 3(b), according to corollary 1, we can prune it directly without calculating GED.

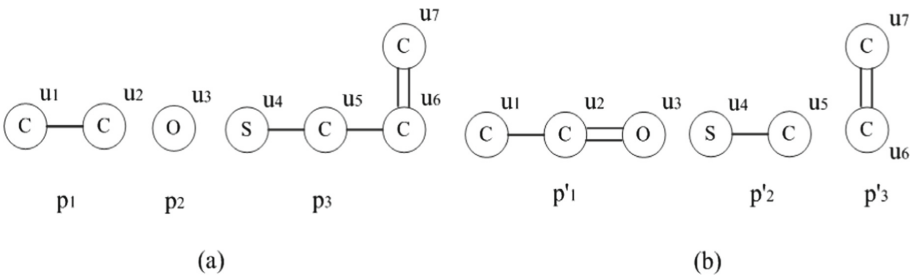

Fig. 3. Two divisions of figure G_1 in Fig. 1

It can be seen that the compactness of $lb(G_1, G_2)$ depends on the partition method of G_1, but the enumeration of all the partition methods is too time-consuming. Based on Inves [15], this paper studies the idea of incremental partition from a new perspective, and uses the idea of q-gram to enhance the partition effect and reduce the number of candidate sets.

3.2 Optimized Incremental Partition

In this section, we introduce an incremental partition strategy. The core idea is to minimize the editing of each mismatched partition p, that is, if some vertices of the partition p are deleted, the partition p will become a matching partition.

Definition 5. (incremental partition) Given two graphs G_1 and G_2, the incremental partition of G_1 is to extract the smallest mismatched partition from G_1. First, we obtain the vertex set $V_{G_1} = \{u_1, \ldots, u_n\}$ of graph G_1. We put the vertices in V_{G_1} into a

partition p in order until we put a vertex u_{i+1} to make $p \nsubseteq G_2$ stop partition. At this time, G_1 is divided into $P(G_1) = \{p, G_1/p\}$, where G_1/p refers to the induced subgraph composed of removing the vertices in the partition p. Repeat G_1/p division until $G_1/p \subseteq G_2$ or $G_1/p = \varnothing$.

The graph partition generated by Definition 5 satisfies the following properties, given two graphs G_1 and G_2, If G_2 uses incremental partition, the division is $P(g) = \{p_1, \ldots, p_k - 1, p_k\}$, then the $p_1, \ldots, p_k - 1$ does not match the G_2, p_k matches G_2. Therefore, $lb(G_1, G_2) = k - 1$.

Example 4. Given two graphs G_1 and G_2 as shown in Fig. 1, G_1 is divided incrementally. The process is shown in Fig. 4. First, select the vertex $\{u_1, u_2, u_3\}$ from G_1 to form the partition p_1 as shown in Fig. 4 (a). Because $(p_1 - \{u_3\}) \subseteq G_2 \bigwedge p_1 \nsubseteq G_2$, partition $p_2^{'}$ continues to be divided until it is shown in Fig. 4 (b). Because $p_4 = \varnothing$, the incremental division ends, $lb(G_1, G_2) = 3$.

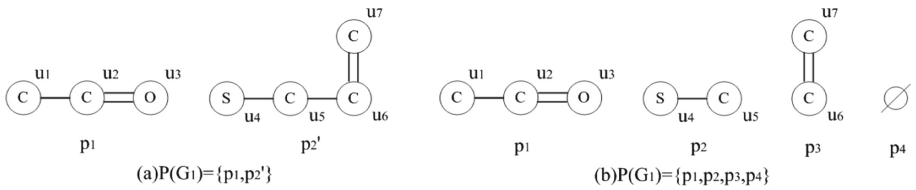

Fig. 4. Incremental division of G_1 in Fig. 1

After the incremental partition of the graph, it can be found that the last vertex put into the partition in each partition results in the mismatch between the partition and the target graph. As shown in the partition p_1 in Fig. 4 (a), the insertion of vertex u_3 makes $p_1 \nsubseteq G_2$, which indicates that the editing operation should be concentrated near the vertex u_3, not only that, the first selected vertex also has an impact on the subsequent entire incremental partition.

Therefore, the last vertex in the partition is taken as the starting vertex, and the partition is re divided in the same way. While maintaining connectivity, the points that may generate editing operations should be considered as early as possible, so as to reduce the size of mismatched partition.

Example 5. As shown in Fig. 5, suppose the vertex order of graph X is $\{u_1, u_2, u_3, u_4, u_5\}$. According to the incremental partition scheme, we choose the first partition p as $\{u_1, u_2, u_3\}$, $X/p \subseteq Y$, then $lb(X, Y) = 1$. We reorder the mismatched partition and get the new vertex order $\{u_3, u_2, u_1\}$ of the partition. Then we get the first partition p as $\{u_3, u_2\}$, $X/p \nsubseteq Y$. We continue to partition and get the second partition $\{u_1, u_4\}$, $X/p \subseteq Y$. We get a closer GED lower bound $lb(X, Y) = 2$.

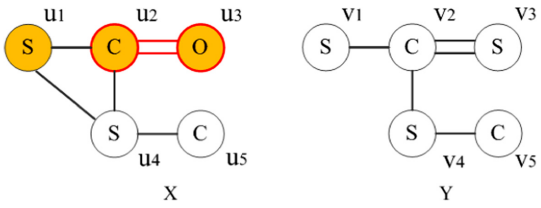

Fig. 5. Repartition mismatch partition

In this paper, the q-gram method is used to enhance the partition effect. Before incremental partition, the two graphs are decomposed into 1-gram to obtain the different structures of the two graphs. The vertices with the lowest frequency in the different structures are most likely to be relabel. The vertices are arranged in ascending order of frequency, and the incremental partition is carried out in this order.

Example 6. The graphs G_1 and G_2 in Fig. 1 are decomposed to get the 1-gram of G_1 and G_2 shown in Fig. 6. Comparing the different structures of G_1 and G_2, it can be found that the different structures are pink nodes, and the vertices with the lowest frequency in G_1 are yellow vertices. The vertices in G_1 are arranged in ascending order according to the frequency, and the vertex sequence $\{u_3, u_4, u_2, u_6, u_7, u_5, u_1\}$ is obtained, as shown in Fig. 7, incrementally partitions in this vertex order.

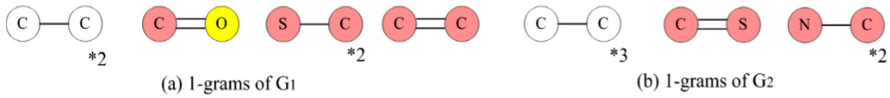

Fig. 6. 1-grams of graphs G_1 and G_2 in Fig. 1

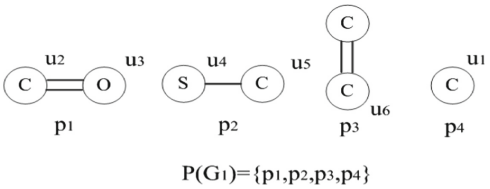

Fig. 7. Optimization increment division of G_1 in Fig. 1

The algorithm of incremental partitioning method is summarized in Algorithm 1.

Algorithm 1: IncremetalPartitioning(G, Q)

Data: Data graph G and Q
Result: A GED lower bound $lb(G, Q)$ based on incremental partition

```
1:  GetVertexOrder(G);
2:  i ← SubgraphIsomorphism(G, Q);
3:  if i > |V_G| then
4:      return 0;
5:  end
6:  p ← the first vertex of the mismatched partition of G;
7:  while |V_p| does not change do
8:      ReorderbByMismatchedVertices(p);
9:      i ← SubgraphIsomorphism(p, Q);
10:     p ← the first vertex of the mismatched partition of p;
11: end
12: G'\←G/p;
13: return 1+IncremetalPartitioning(G', Q);
```

Given a pair of graphs G and Q, the algorithm uses the incremental partition method to partition, so as to calculate the lower limit $lb(G, Q)$.

Firstly, the q-grams algorithm is used to count the frequency to determine the vertex order of G (line 1), Then, the isomorphic test of induced subgraph of G is performed based on this order, and the unmatched vertex position of G in Q (line 2) is returned. If the number of unmatched vertex positions returned exceeds the number of vertex sets in G, then this pair of matching, return 0 (lines 3–5). Otherwise, the mismatched vertices are placed in partition p (line 6). Using 7–11 lines to reduce the size of the partition, reorder the vertices in the partition P (line 8), continue to perform the induced subgraph isomorphism test (line 9) on Q in this order, and return the unmatched vertex position (line 10) of G in Q until the partition p does not change. Then the partition p is separated from G to get G' (line 12). Iterate over the number of partitions in G' and return $lb(G, Q)$ (line 13).

3.3 Validation Algorithm

In the process of verification, A* algorithm is used to calculate GED. The performance of A* algorithm depends on the accuracy of the estimation of edit distance generated by unmapped vertices. In order to improve the accuracy of distance estimation, this paper uses the idea of bridge to predict the editing distance of unmapped vertices more accurately.

Definition 6. (bridge) Given a partition p, the bridge of vertex $u \in p$ is the edge of vertex u and v, where $v \notin p$.

Given a partition p in the graph partition of graph G_1, and a partition m matching the partition p in G_2, it is assumed that the vertex $u \in p$ of graph G_1 is mapped to the vertex $v \in q$ of graph G_2. Then, the edit distance between vertices u and v is $Be(u, v) = \Gamma(L_{br}(u), L_{br}(v))$, where $L_{br}(u)$ represents the set of labels of the bridge of vertices u, $\Gamma(X, Y) = max(|X - Y|, |Y - X|)$. The bridge editing distance from partition p to partition m is $B(p, m) = \sum_{u \to v \in M} Be(u, v)$, where m is the mapping of all vertices of partition p and partition M.

Example 7. Given the graphs G_1 and G_2 in Fig. 1, it is assumed that the matched partition is $p_1 = (u_1, u_2)$, and the partition p matches $m_1 = (v_5, v_6)$ in G_2, as shown in Fig. 8, $Be(u_1, v_5) = 1$, because there is no bridge in u_1, and there is a bridge in v_5, which is the same as $Be(u_2, v_6) = 1$, so $B(p_1, m_1) = 1 + 1 = 2$.

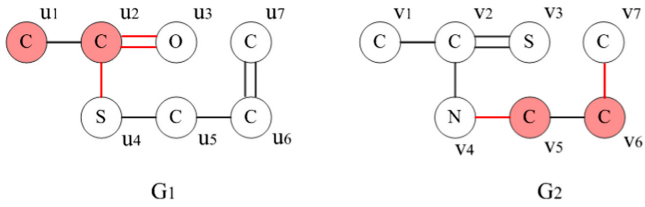

Fig. 8. Edit distance of matching partitioned bridges

When using the A* algorithm to calculate $GED(G_1, G_2)$, for the known vertex mapping set M, where $u \to v \in M, u \in V_{G_1}, v \in V_{G_2}$, we can get the predicted edit distance of unmapped vertices, as shown in Eq. 1.

$$h'(M) = B(M) + \Gamma(L_{G_1}(u'), L_{G_2}(v')) + \Gamma(L_{G_1}(u', v'), L_{G_2}(u', v')) \tag{1}$$

Where u', v' is the unmapped vertex, and the optimized A* algorithm is shown in Algorithm 2.

Algorithm 2: GED(G, Q, τ)

Data: Data graph G and Q, The vertex set of graph G is $V_G = u_1, \ldots, u_n$ and the vertex set of graph Q is $V_q = v_1, \ldots, v_n$, GED threshold τ

Result: If the edit distance of data graph G and Q is less than τ, then the edit distance is returned, otherwise $\tau + 1$ is returned

1: $Map = \emptyset$; $M_{min} = \emptyset$;
2: foreach *node* $w \in V_Q \bigcup \varepsilon$ do
3: $Map = Map \bigcup (u_1 \to w)$;
4: end
5: while *True* do
6: $M_{min} = arg_{min}\{h'(M_{min}) + g(M_{min})$;
7: $Map = Map/M_{min}$;
8: if $Complete(M_{min})$ *then*
9: if existingDistance(M_{min}) $> \tau + 1$ then
10: return $\tau + 1$;
11: end
12: else
13: return $existingDistance(M_{min})$;
14: end
15: end
16: if *All vertices in* V_G *are mapped* then
17: foreach *node* $w \in$ *Unmapped vertices* V_Q *do*
18: $Map = Map \bigcup (\varepsilon \to w)$;
19: end
20: end
21: else
22: foreach *node* $w \in$ *Unmapped vertices* $V_Q \bigcup \varepsilon$ *do*
23: $Map = Map \bigcup next\ vertex\ currently\ matched\ to\ V_G \to w$;
24: end
25: end
26: end

First, initialize the set M_{min} storing the minimum overhead mapping and the candidate mapping set *Map* (line 1), which maps the first vertex of graph G to all vertices in Q (line 2–4). Select the *Map* with the minimum accumulated known overhead and prediction overhead as M_{min}, and delete it from the map (line 6–7). If M_{min} is already a complete *Map*, if the editing distance of the map exceeds τ, return $\tau + 1$, otherwise returns the edit distance of the map. (line 8–13) if all vertices in V_G are mapped, and it is not a complete mapping, it means that the remaining vertices in V_Q need to be mapped to null, otherwise, the mapping will continue in order (line 14–22).

3.4 Clustered Index Structure

In order to search in graph database faster, we propose a tree structure based on dictionary tree based on k-means. To enhance filtering, we try to assign more similar graphs to the same set. Then, a tree index is constructed for each set. In this paper, the i-th cluster is represented as C_i, and its corresponding tree index is expressed as T_i. When searching in the graph database, we search each index and merge the results of all searches. For a given query graph Q, if it shares a few graph partitions with C_i, most of the subtrees in T_i will be pruned to speed up the search time.

K-means algorithm is a common clustering algorithm. In this paper, the distance measure is defined as the GED lower bound of graph partition, and the center of clustering is expressed as C_i.

In order to solve the problem that k-means needs to specify the number of partitions, this paper proposes a method to keep the center of each cluster as far away from each other as possible. Firstly, the center of the first cluster is initialized randomly, and then the graph furthest from the current cluster center is selected as the new center iteratively until all graphs are close enough to the center of the cluster in which they are located. All partitions in the center of each cluster are counted, sorted according to the number of times that the graph in the cluster contains partitions, and a tree index is constructed based on the dictionary tree. The index has the following properties:

- The root node does not store data, and each path represents a partition of the cluster center map.
- Each node except the root node stores a set of graphs.
- From the root node to a node, the partition on the path belongs to every graph in the graph set.

Example 7. Given the $D = \{G_1, \ldots, G_n\}$ of graph database, first select G_1 as the center of the first cluster, calculate the distance between G_1 and other graphs, and obtain the division $P(G_1) = \{p_1, p_2, p_3, p_4\}$ of G_1, select the farthest graph G_k as the center of the second cluster, calculate the distance between G_k and other graphs, and update each cluster. Calculate the distance from the graph in each cluster to the cluster center, calculate the average distance, select the graph that is far away from both clusters as the new cluster center, iterate repeatedly until the distance between all vertices in the cluster and the cluster center is less than the average distance, or the cluster reaches the upper limit. Suppose that the cluster with G_1 as the center contains the graph G_2, G_4, G_5, G_8, where $p_1 \subseteq G_2 \wedge p_1 \subseteq G_4$, $p_2 \subseteq G_2 \wedge p_2 \subseteq G_5$, $p_3 \subseteq G_2 \wedge p_3 \subseteq G_5 \wedge p_3 \subseteq G_8, p_4 \subseteq G_5$ according to the number of times that the graph in the cluster contains the partition Row sorting gets the partition order $\{p_3, p_1, p_2, p_4\}$, and the tree index is constructed as shown in Fig. 9.

The algorithm for building the clustered index is shown in Algorithm 3.

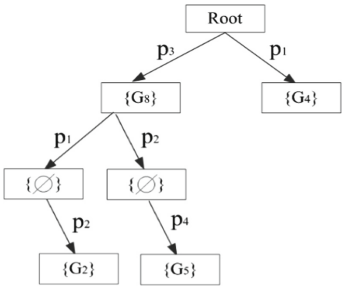

Fig. 9. Edit distance of matching partitioned bridges

Algorithm 3: Clustered_index(G, MAX)

Data: Graph database D and Maximum cluster number MAX
Result: Number of clusters n, set of tree index of clusters T

Randomly select a graph $G \in D$ *as the center* c_1 *of cluster* C_1;
IncremetalPartitioning(c_1, D);
Find C_2, *which is the farthest from* C_1 *editing distance, and initialize cluster* C_2;
The clusters C_1 and C_2 are stored in the cluster set C, $n = 2$;
while *True* do
    Calculate the distance from the graph in each cluster to the cluster center;
    Calculate the average distance L;
    if *Distance from* $C_i \in D$ *to the center of the current cluster longer than L and* $n < MAX$ then
        The new cluster C_{n+1} *is initialized, and the cluster center is* C_i;
        $n = n + 1$;
    end
    else
        break;
    end
end
foreach $C_i \in C$ do
    Calculate the graph partition $P_i = \{p_{i1}, \ldots, p_{ik}\}$ at C_i of each cluster;
end
$T = \emptyset$;
foreach $c_i \in C$ and graph $g \in C_i$ *do*
    foreach $partition\ p_{ij} \subseteq g, 1 \le j \le k$ do
        if $T.child$ *does not contain* p_{ij} then
            Insert an empty set node into the tree and assign the edge as p_{ij};
            when p_{ij} *is the last partition of* g *append* g *to the current node*;
        end
    $T = T.child$;
    end
end

Firstly, a graph is randomly selected as the center point C_1 (line 1) of the first cluster, and then the distance from the center point of the cluster to other graphs (line 2) is calculated. The graph C_2 with the farthest editing distance C_1 is found as the new cluster center vertex (line 3), calculate the distance from the graph in each cluster to the center of the cluster, find the average distance, select the graph that is far away from each cluster as the new cluster center, iterate repeatedly until the distance between all points in the cluster and the center of the cluster is less than the average distance, or the cluster reaches the upper limit (line 4–15). Get the partition in the center of each cluster (line 16–18).

For each cluster, the dictionary index is formed according to the graph containing partition in the cluster. The node in the tree is the set of graph, and the edge is the partition. It is constructed from the root node. If the partition P belongs to the current graph, when there is no partition p in the child of the root, an empty set node is inserted. If the partition P is the last structure belonging to the current graph, the current graph is inserted into the node (line 20–28).

4 Experiment

4.1 Data Set and Experimental Environment

In this paper, the real graph data set in graph similarity search field is selected.

1. AIDS: it is a data set of antiviral screening used in NCI/NIH development and treatment plan. It contains 42390 compounds, with an average of 25.4 vertices and 26.7 edges. It is a large graph database often used in the field of graph similarity search.
2. Protein: it comes from protein data bank and contains 600 protein structures, with an average of 32.63 vertices and 62.14 edges. It contains many dense graphs and has fewer vertex labels.

See Table 1 for data statistics, where LV and LE are vertex and edge labels.

Table 1. AIDS and Protein data set

Data set	$\|G\|$	$\|V\|_{arg}$	$\|E\|_{arg}$	$\|LV\|$	$\|LE\|$
AIDS	42390	25.40	26.70	62	3
Protein	600	32.63	62.14	3	5

For each data set, we use 100 random samples as query graphs. The index construction time, the size of the index, the query response time based on the GED threshold, and the size of the candidate set to be verified are compared in several aspects. All algorithms in this paper are implemented in Java. All experiments are run on MacBook Pro with inter Corei7 with 2.6 GHz and MacOS 10.14.6 (Mojave) system diagram with 16 GB main memory.

4.2 Comparison of Existing Technologies

In this paper, three commonly used algorithms are selected for comparison:

1. Pars: Pars [13] uses random partitioning strategy to propose a non overlapping, variable size graphics partitioning strategy and generate index. It is the most advanced partitioning method at present.
2. ML-index: ML-index [14] adopts a multi-layer graph index method. It contains L different layers, each layer represents a lower GED based on the partition, and a selection method is selected to generate the index. In this paper, the number of layers is defined as 3.
3. Inves: Inves [15] uses incremental partitioning method to divide the candidate graph accurately and gradually according to the query graph, so as to calculate the lower bound of their distance.

First of all, considering the time of index construction, as shown in Fig. 10. Notice that y-axis is log-scaled in all experiments. The Inves algorithm does not establish an index during the search process, and here it is modified using the tree index proposed in this article, so the index construction time is consistent with Ip-tree. Therefore, it is not shown in Fig. 10. Pars takes more time to build index, because it involves complex graph partition and sub graph isomorphism test in the index construction phase. In this paper, Ip-tree algorithm needs to cluster graph data, build index, and test some graph partition and subgraph isomorphism, so the time is slightly slower than ML-index.

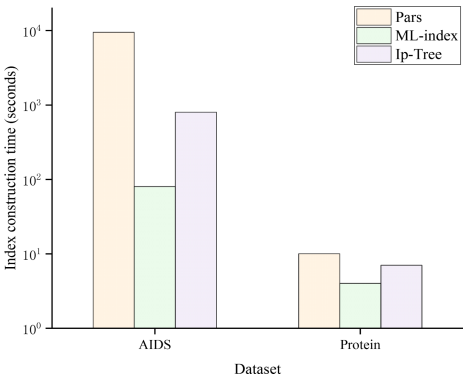

Fig. 10. Index construction time under AIDS and protein datasets

Figure 11 shows that under the AIDS data set, when the threshold changes, the processing time of different algorithms changes. Because the AIDS data set graph is large, we can clearly see that with the increase of threshold, the response time of all algorithms becomes longer. The algorithm in this paper is faster than other algorithms in processing time. Due to the small number of graphs in the protein data set, the GED threshold range is expanded to 1–8 in this paper. As shown in Fig. 12, the algorithm in this paper fluctuates slowly when the threshold rises.

Finally, this paper compares the number of candidate sets, as shown in Fig. 13 and Fig. 14. It can be found that the filtering effect of this algorithm is the best when the threshold value is in the range of 1–3.

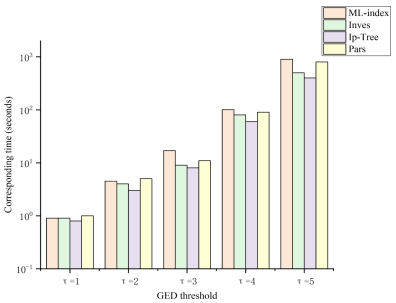

Fig. 11. Response time of AIDS

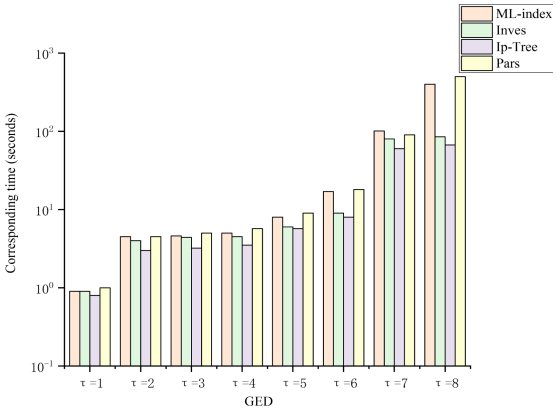

Fig. 12. Response time of Protein

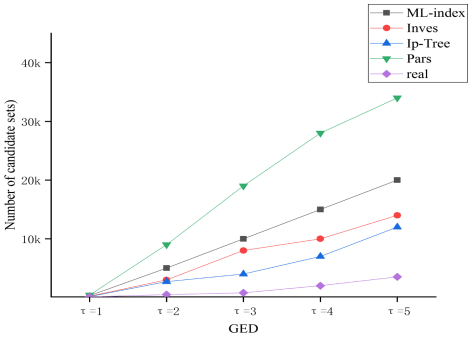

Fig. 13. AIDS candidate set size

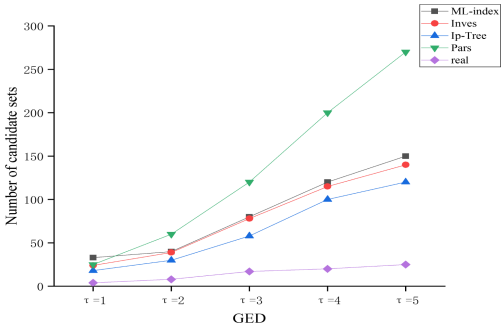

Fig. 14. Protein candidate set size

5 Conclusion

In this paper, we improve the existing algorithm Inves, and propose a dictionary tree based clustering index structure. The main idea is incrementally partition the candidate graph according to the query graph, and calculate its GED lower bound. Using clustering method, we cluster similar graphs and construct tree index, speed up the filtering of candidate graphs, and use the optimized A* algorithm to realize the accurate calculation of GED in the verification stage. Experiments on real large graph datasets show that the performance of algorithm proposed is significantly better than that of existing algorithms.

Acknowledgment. The Natural Science Foundation of Heilongjiang Province under Grant Nos. F2018028. Received 2000-00-00, Accepted 2000-00-00.

References

1. Shang, H., Lin, X., et al.: Connected substructure similarity search. In: SIGMOD 2010, pp. 903–914 (2010)
2. Gouda, K., Hassaan, M.: CSI_GED: an efficient approach for graph edit similarity computation. In: ICDE 2016, pp. 265–276 (2016)
3. Zhu, G., Lin, X., et al.: TreeSpan: efficiently computing similarity all-matching. In: SIGMOD 2012, pp. 529–540 (2012)
4. Maergner, P., Riesen, K., et al.: A structural approach to offline signature verification using graph edit distance. In: ICDAR 2017, pp. 1216–1222 (2017)
5. Geng, C., Jung, Y., et al.: iScore: a novel graph kernel-based function for scoring protein-protein docking models. Bioinformatics **36**(1), 112–121 (2020)
6. Zeng, Z., Tung, A.K.H., et al.: Comparing stars: on approximating graph edit distance. PVLDB **2**(1), 25–36 (2009)
7. Riesen, K., Emmenegger, S., Bunke, H.: A novel software toolkit for graph edit distance computation. In: Kropatsch, W.G., Artner, N.M., Haxhimusa, Y., Jiang, X. (eds.) GbRPR 2013. LNCS, vol. 7877, pp. 142–151. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38221-5_15

8. Wang, G., Wang, B., et al.: Efficiently indexing large sparse graphs for similarity search. IEEE Trans. Knowl. Data Eng. **24**(3), 440–451 (2012)
9. Zhao, X., Xiao, C., Lin, X., Wang, W., Ishikawa, Y.: Efficient processing of graph similarity queries with edit distance constraints. VLDB J. **22**(6), 727–752 (2013). https://doi.org/10.1007/s00778-013-0306-1
10. Zheng, W., Zou, L., et al.: Efficient graph similarity search over large graph databases. IEEE Trans. Knowl. Data Eng. **27**(4), 964–978 (2015)
11. Ullmann, J.R.: Degree reduction in labeled graph retrieval. ACM J. Exp. Algorithmics **20**, 1.3:1.1–1.3:1.54 (2015)
12. Li, Z., Jian, X., et al.: An efficient probabilistic approach for graph similarity search. In: ICDE 2018, pp. 533–544 (2018)
13. Zhao, X., Xiao, C., et al.: A partition-based approach to structure similarity search. PVLDB **7**(3), 169–180 (2013)
14. Liang, Y., Zhao, P.: Similarity search in graph databases: a multi-layered indexing approach. In: ICDE 2017, pp. 783–794 (2017)
15. Kim, J., Choi, D.-H., Li, C.: Inves: incremental partitioning-based verification for graph similarity search. In: EDBT 2019, pp. 229–240 (2019)