Department of Informatics, University of Zürich

MSc Basic Module

Understanding Faster Deterministic Fully-Dynamic Graph Connectivity

Xinyu Zhu

Matrikelnummer: 20-743-324

Email: `xinyu.zhu@uzh.ch`

June 14, 2021

supervised by Prof. Dr. Sven Helmer and Qing Chen

University of Zurich UZH

Department of Informatics

1 Introduction

The dynamic graph connectivity problem is derived from a case, where we are asked to determine whether two vertices are connected in a graph whose edge would be inserted or deleted, in other words, changed dynamically. The connectivity means the two vertices can access each other by a path. For frequent changes of edges and queries of connectivity, we should store information we acquired by search or traverse in a delicate method and structure, to avoid repetitive and trivial work. Hence, many researchers are working towards the optimization of the algorithm to reduce update or query time complexity, such as [Tho00] and [WN13]. They use the cluster tree to store connectivity information of edges and vertices. In their approach, there are three important components: 1) basic cluster tree, including concept of level and rules of maintaining tree with dynamic graph. 2) introduction of local tree and bitmap and 3) lazy local tree. The goal of this basic module is to implement the first component from scratch and figure out how last two mechanisms work, then apply them to real datasets. Having these implementations, we will also compare our results with some other algorithms. This report will be organized into the following sections:

- **Preliminaries.** In this section, we will introduce the preliminary knowledge that we need. More specifically, we will introduce the Graph, Cluster Tree, Level, Depth First Search, and Balanced Binary Search Tree. Among them, the concept of graph and tree is the most important and fundamental knowledge of the whole report.

- **Approach.** In this section, we will thoroughly explain the mechanisms of essential components in our system: creation of cluster tree, insertion of edge, deletion of edge, maintain and update of tree. Besides, we will also induce local tree and bitmap , which are used to make algorithm more efficient for future optimization.

- **Running Example.** In this section, we will present a simple running example of using our approach to perform maintenance of cluster tree with several insertions and deletions. Also, we will show visualization of the procedure including graph, cluster tree, subsearch and local tree with bitmap.

- **Implementation.** In this section, we will explain and demonstrate how we implement the data structure and algorithm. More specifically, we will present the detailed implementation of creation, insertion and deletion. And some tricks of index matching to simplify query, list reuse to reduce space complexity will also be shown.

- **Experiment.** In this section, we will execute our algorithm on a small sample and a real dataset Enron with 87273 vertices and 1148072 edges, and try to test correctness and robustness. Besides these, we will also try to discover the significance of the algorithm in practical applications.

2 Preliminaries

Graph: In this fully-dynamic graph problem, we focus on a graph G over a dynamic vertex set V and a dynamic edge set E. The graph G may be updated by insertions and deletions of edge. Accordingly, the vertex set V and edge set E would be updated. There are six update cases: 1) insertion of an edge with both incident vertices already in vertex set V, no update in vertex set V, one edge added into edge set E. 2) insertion of an edge with only one incident vertex already in vertex set V, one vertex added into vertex set V, one edge added into edge set E. 3) insertion of an edge without incident vertex already in vertex set V, two vertices added into vertex set V, one edge added into edge set E. 4) deletion of an edge with both incident vertices who have other incident edge, no update in vertex set V, one edge deleted from edge set E. 5) deletion of an edge with only one incident vertex who has other incident edge, one vertex deleted from vertex set V, one edge deleted from edge set E. 6) deletion of an edge without incident vertex who has other incident edge, two vertices deleted from vertex set V, one edge deleted from edge set E.

Cluster Tree: Cluster tree is a rooted tree with nodes, arcs and leaves. A node in a Cluster Tree represents a sub-graph of a connected component in graph. Especially, we distinguish leaves with other inner nodes. A leaf in a Cluster Tree represents a vertex in graph. Arcs link node to node or node to leaf. There are also some trivial concepts in Cluster Tree: 1) Leaf Descendants, belonging to nodes, represents all leaves rooted as it. 2) Parent Node, representing a node one level upper. 3) Child Node, representing a node one level down. 4) Grandparent Node, representing two level upper node. 5) Bidirected Pointer, linking leaf in tree and vertex in graph, in other words, we can easily find corresponding vertex in graph from a leaf in tree or corresponding leaf in tree from a vertex in graph. Child node represents subgraph of parent node, and each sibling node is a partition consisting a full graph of their parent node.

Level of Edge: Each edge has a level. When an edge is inserted into graph, we assign it an initial level 0. Level of edges changes in case of deletion, only increase and not decrease. Level of edge indicates the frequency of edge be validly explored in replacement search.

Level of Node: Each node has a level. Level of node is corresponding to its height in tree. In other words, root node has level 0, and one layer deeper, one level higher. Level of nodes changes in case of deletion, only increase and not decrease. Level of node indicates the subdivision of node. Level-i node has higher-level edge.

Multigraph: Multigraph is an undirected graph allowed to include self loops and parallel edges. In our algorithm, a vertex in multigraph can be a real single vertex of graph or a subgraph of graph, in other words, a leaf or node of tree. Replacement subsearches of deleted

Level-i edge will be executed on mutigraph spanned by Level-$i-1$ ancestor nodes of two endpoints.

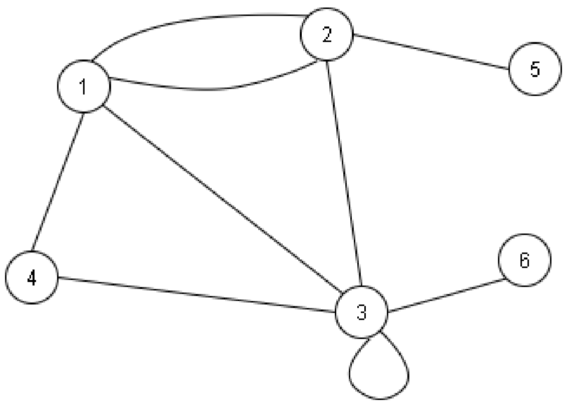

Figure 2.1: Example of a multigraph

Depth First Search: Depth first search is a classic graph search algorithm. Search starts from a vertex of multigraph, find an incident edge leading to an unexplored vertex, jump to the vertex and do search again. If no vertex to jump, go back previous one and do search again. Search terminates when there is nothing to do. Replacement subsearches can be depth first search, breadth first search or any other type of search. But in our work, we choose depth first search because it is easy to be implemented and understood.

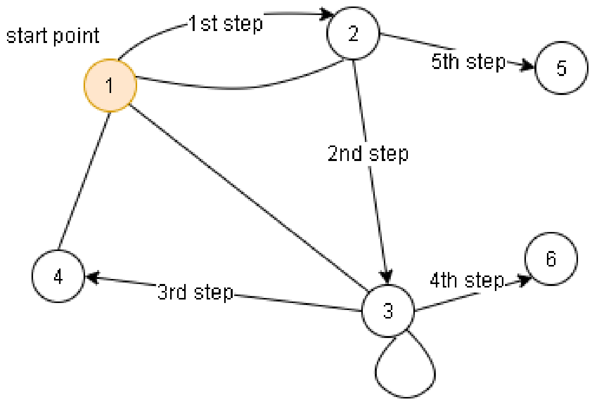

Figure 2.2: Example of a depth first search

Balanced Binary Search Tree: In some search problems, the time complexity of worst case is O(n). If we can use some prior Knowledge to make search grouped almost evenly, in other words, make a balanced search tree, we can reduce the time complexity of worst case to O($logn$).

3 Approach

In this work, we will build a data structure to store the connectivity information, and some maintenance functions that performs the creation, insertion and deletion operation of the graph and tree to update connectivity information by search. For the graph, we use a class Graph with two attributes, set of vertices and edges, two methods, insertion and deletion of edge while for the tree, a class Cluster with two attributes, corresponding Graph and set of tree nodes, two methods, insertion and deletion of edge. In this section, we will thoroughly introduce insertion and deletion of graph and corresponding update of tree.

3.1 Class: Graph

There are two essential methods to store graph in computer, adjacent matrix or adjacent list. In this work, we focus on undirected graph, so we would store each vertex once and each incident edge twice. As introduced in Section 2 Preliminaries, we will try to convert the raw data into our data structure to adapt the algorithm, and reuse the data structure that we have created when update the cluster tree.

Class of graph represents a traditional basic graph with some extra qualities. It means 1) edge link a vertex to itself will not show up in graph. 2) parallel edges link a vertex to another vertex will be regarded as one edge. 3) a single vertex without any incident edge will be deleted from graph to save storage space. 4) edges should be stored twice with its level, because of two endpoint and convenience for search. 5) bidirected pointers between vertices in graph and leaves in cluster tree should be kept, so that subsearches in cluster tree can start quickly when graph is changed, and level of edge can be obtained fast when execute search in cluster tree.

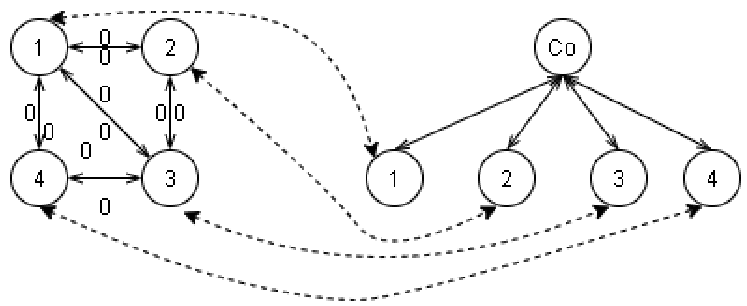

Figure 3.1: Example of graph and cluster tree

Algorithm 1: Insertion of Edge in Graph

```
Input: A list $V$ to store vertices, a corresponding same-length list $E$ to store incident
       edge and an edge $(A, B)$ to be inserted
Result: The output of updated $V$ and $E$
if A in V and B in V then
    indexA = V.index(A)
    indexB = V.index(B)
    E[indexA].append(B)
    E[indexB].append(A)
else
    if A in V then
        indexA = V.index(A)
        V.append(B)
        E[indexA].append(B)
        E[-1].append(A)
    else
        if B in V then
            indexB = V.index(B)
            V.append(A)
            E[indexB].append(A)
            E[-1].append(B)
        else
            V.append(A)
            V.append(B)
            E[-2].append(B)
            E[-1].append(A)
        end
    end
end
```

We begin with insertion of an edge into graph. We don't consider just inserting a single vertex into graph because the case has nothing with connectivity. We call a vertex which never shows up before in list of graph's vertices new vertex. When an edge is inserted into graph, zero, one or two new vertices is inserted into list of graph's vertices with initial level of their incident edge into corresponding edge list. We call the three types type 0,1 and 2 insertion, which leads to three type of insertion in cluster tree update.

The insertion algorithm 1 is the simplest and most fundamental data-graph conversion as it stores all the vertices in a graph to all their incident edges in two different lists with same index. Therefore, we can easily find all incident edges given a vertex or just its index.

Algorithm 2: Deletion of Edge in Graph

```
Input: A list $V$ to store vertices, a corresponding same-length list $E$ to store incident
       edge and an edge $(A, B)$ to be deleted
Result: The output of updated $V$ and $E$
indexA = V.index(A)
indexB = V.index(B)
if len(E[indexA]) != 1 and len(E[indexB]) != 1 then
    indexEA = E[B].index(A)
    indexEB = E[A].index(B)
    del E[B][indexEA]
    del E[A][indexEB]
else
    if len(E[indexA]) != 1 then
        indexEB = E[A].index(B)
        del E[A][indexEB]
        del E[indexB]
        del v[indexB]
    else
        if len(E[indexB]) != 1 then
            indexEA = E[B].index(A)
            del E[B][indexEA]
            del E[indexA]
            del v[indexA]
        else
            del E[indexA]
            del v[indexA]
            del E[indexB]
            del v[indexB]
        end
    end
end
```

When we delete an edge from graph, we should check if the edge exists in graph first. If a single vertex without any incident is made after deletion, we should delete the single vertex as well. We distinguish three types of deletion, making zero, one or two single vertices. After deletion in graph, we will transfer related information to later steps so that we do deletion in cluster tree. We call the three types type 0,1 and 2 deletion, which leads to three type of deletion in cluster tree update. By insertion and deletion, we converted the dynamic data into a structured stored graph. As cluster tree should be updated correspondingly, we can directly go next step to find how the cluster tree created and updated. Besides, we need induce the space free mechanism in deletion operation. In other words, an item of list to be deleted should be set to "Null" and reuse in later insertion. It's what "Del" function mentioned in algorithm 2 actually means. We don't just extend the list endlessly to overflow or simply delete the item to make index interrupted.

3.2 Class: Cluster

Each update operation of graph is followed by an update of cluster tree. As we mentioned in Preliminaries, there are six update cases in graph. Accordingly, we have even more than six update cases in cluster tree: 1) insertion of an edge without incident vertex already in vertex set V. We should create a new level-0 node with two leaves (as is newly added vertices) representing a subgraph spanned by the two vertices and a level-0 edge. 2) insertion of an edge with only one incident vertex already in vertex set V, We should add a new leaf (as is newly added vertex) as child of already existing leaf's root, representing connectivity between the newly vertex and existing subgraph with a level-0 edge. 3) insertion of an edge with both incident vertices already in vertex set V, if they go back to same root in cluster tree, no more update because the connectivity between the two vertices is already implied. 4) insertion of an edge with both incident vertices already in vertex set V, if they go back to different root in cluster tree, one root will adopt the other root's children, representing the connectivity of two node identical. 5) deletion of an edge with both incident vertices who have other incident edge, if we can find a replaceable path with endpoints as the two vertices, we assert the connectivity remained. Then we select a smaller searched partition to be promoted for one level plus, which makes our search paid. 6) deletion of an edge with both incident vertices who have other incident edge, no update in vertex set V,if we can not find a replaceable path with endpoints as the two vertices, we assert the connectivity vanished. Then we select a smaller searched partition to be detached, which also makes our search paid. 7) deletion of an edge with only one incident vertex who has other incident edge, the other vertex should be removed from cluster tree. 8) deletion of an edge without incident vertex who has other incident edge, the two vertices should be removed from cluster tree. Actually, as node in cluster is corresponding to a sub-graph in graph, level-i node must have at least one level-i edge. That means the sub-graph has at least one level-i edge, in other words, leaf descendants have at least one level-i incident edge. When we see sub-graph as a big "vertex" in graph, a multigraph comes up, where there might be edges linking a vertex with itself.

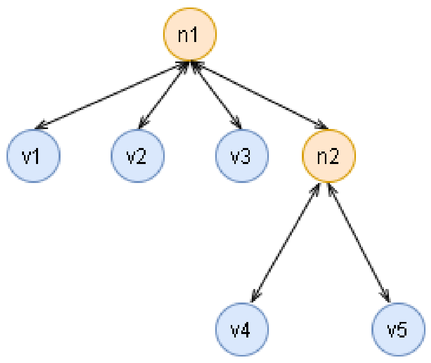

Figure 3.2: Example of cluster tree with node and leaf

Algorithm 3: Insertion of Edge in Tree

```
Input: A list $N$ to store nodes, $V$ and $E$ of graph
Result: The output of updated $N$
if A in V and B in V then
    indexA = V.index(A)
    indexB = V.index(B)
    rootA = traverse2root(V[indexA].parent)
    rootB = traverse2root(V[indexB].parent)
    if rootA == rootB then
        pass
    else
        while not all rootB child be traversed do
            N[rootA].child.append(N[rootB].child)
            N[rootB].child.parent = rootA
        end
    end
else
    if A in V then
        rootA = traverse2root(V[indexA].parent)
        V[indexB].parent = rootA
        N[rootA].child.append(B)
    else
        if B in V then
            rootB = traverse2root(V[indexB].parent)
            V[indexA].parent = rootB
            N[rootB].child.append(A)
        else
            root.child.append(A)
            root.child.append(B)
            V[indexA].parent = root
            V[indexB].parent = root
            N.append(root)
        end
    end
end
```

Once an edge is inserted into graph, a correspondingly insertion should be executed in cluster tree. As we mentioned previously, three types of graph's edge insertion lead to three types of cluster tree's insertion. Type 0 insertion leads to two cases, two existing vertices has same tree root or not. If it's former, nothing should be done because connectivity holds. Otherwise, a arbitrary root should adopt all children of another root and the latter one will vanish. Type 1 insertion makes the new vertex become a leaf child of existing vertex's root in cluster tree. Type 2 insertion creates a new root node with two new vertices as leaf child in cluster tree. The procedure is reflected in the algorithm 3.

Algorithm 4: Deletion of Edge in Tree

```
Input: A list $V$ to store vertices, a corresponding same-length list $E$ to store incident
       edge and an edge $(A, B)$ to be deleted
Result: The output of updated $V$ and $E$
indexA = V.index(A)
indexB = V.index(B)
indexEA = E[B].index(A)
l = E[B][indexEA].level
if len(E[indexA]) != 1 and len(E[indexB]) != 1 then
    seqEA, seqNA = DFS(A,l)
    seqEB, seqNB = DFS(B,l)
    seqEP, seqNP = min(sumofleaves(seqNA),sumofleaves(seqNB)) if $\exists$ seqVA[i].root
     = seqVB[j].root then
        promote(seqEP, seqNP)
    else
        detach(seqEP, seqNP)
    end
else
    if len(E[indexA]) != 1 then
        indexN = V[indexA].parent
        del N[indexN].child[B]
    else
        if len(E[indexB]) != 1 then
            indexN = V[indexB].parent
            del N[indexN].child[A]
        else
            indexN = V[indexA].parent
            del N[indexN]
        end
    end
end
```

Once an edge is deleted from graph, a correspondingly deletion should be executed in cluster tree. As we mentioned previously, three types of graph's edge deletion lead to three types of cluster tree's deletion. Type 0 deletion leads to two cases, a replacement path found or not after two subsearches though same level edge has finished. If it's former, explored edge should be promoted for one level, so should two endpoints of edge in cluster tree. Otherwise, we do same thing as well as a detachment because of connectivity changed. Then we reconnect detached node to its grandparent node. Then the procedure will be recursively executed in zoomed in graph, in other words, smaller level. Type 1 and 2 deletion makes detachment of new single vertex's as the leaf child in cluster tree. After each detachment, we should check if there is a changed node has only a leaf child. If so, we should make parent of the node adopt the leaf child until there is no such to be suppressed node. The procedure is reflected in the algorithm 4.

4 Running Example

4.1 Create Cluster Tree

First of all, assume given a random unweighted undirected graph as below stored by adjacent matrix or linked list. We are going to build a tree structure named Cluster storing edges and vertices (only leaf) by some rules. Especially, we identify the "vertex" "edge" in graph and "node" "arc" in Cluster.

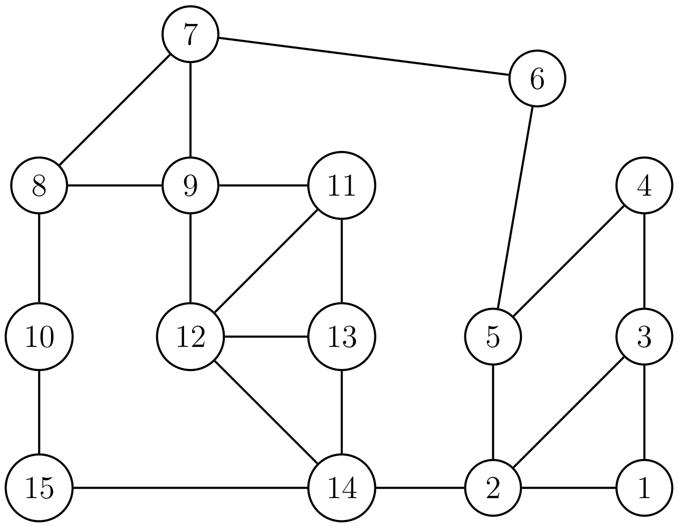

Figure 4.1: A simple undirected graph

We assign each edge (1,2),(1,3)...(14,15) a 0-level,and put related vertices into each level-1 node of Cluster. Then we get a Cluster consisting of a level-0 node and 15 level-1 leaf nodes, each of which has a double direction pointer with vertex so that we can get edge information from each leaf nodes.

As what mentioned in [WN13] 3.2, merge the root if we find the new added edge have existing endpoint in cluster. Because the graph is connected, we got a cluster like below.
And we keep three laws: 1.Edge has level no larger than $\lfloor logn \rfloor$. 2.Node has level no larger than $\lfloor logn \rfloor$ and equal to its depth in cluster. 4.An i-level node has most $n/2^i$ leaf descendants. 3.Each i-level non-leaf node has least one i-level incident edge to its leaf descendant, or the node should be suppressed.

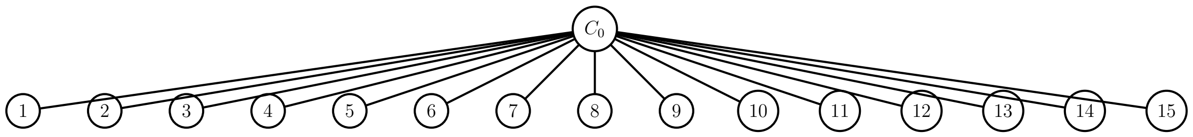

Figure 4.2: A cluster tree from a graph

4.2 Deletion

4.2.1 Step1

First, we want to delete an i-level edge. Then we should check connectivity of two i+1-level nodes (in multigraph) who own the two endpoints of the edge through i-level edges.

We proceed two sub-DFS starting from two nodes and excute by turn. DFS explores an unexplored node, then notes it as explored. Sub-DFS terminates when one of them get to the node which the other sub-DFS explored (case1) or no more node should be explored (case2). In case1, we promote nodes and corresponding edges from one of the two sub-DFS whose nodes have fewer leaf descendants (to keep the tree balanced). In case2, we promote nodes and corresponding edges from one of the two sub-DFS which doesn't have edge to be explored and leaf-descendants are fewer than $\lfloor n/2^i \rfloor$ (otherwise choose the other sub-DFS).

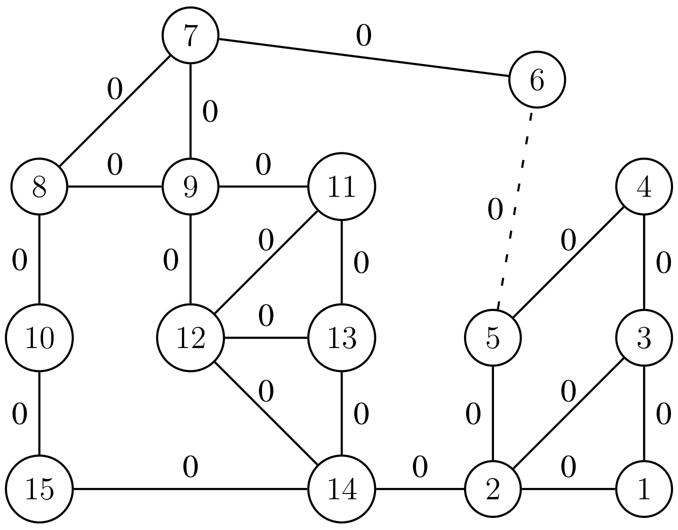

Figure 4.3: An edge to be deleted

For example, we want to delete (5,6). Two sub-DFS are like below. We note sub-DFS with last explored node as C_v without last explored node and the other one as $C9_u$. So, C_v:5,2,1,3,4,14 C_u:6,7,8,9,11,12 (without last explored node). Because a node shouldn't show up in both subsearches, it will be excluded from sequence belonging to last explored meet point. We promote C_v with edges (5,2)(2,1)(1,3)(3,4)(2,14) or C_u with edges (6,7)(7,8)(8,9)(9,11)(11,12) (because they have equal number of leaf descendants).

Besides, if two subsearches meet right away after one jump, no promotion should be done, because we won't promote just a single node without any promotion on edge. After promotion, level of incident edge and cluster node including their endpoints match again. Also, the quality of balanced search tree holds because we always promote nodes less than half of all siblings.

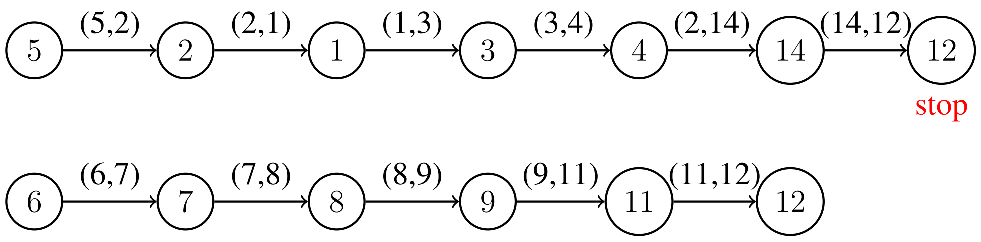

Figure 4.4: Two searches with same sum of leaf nodes

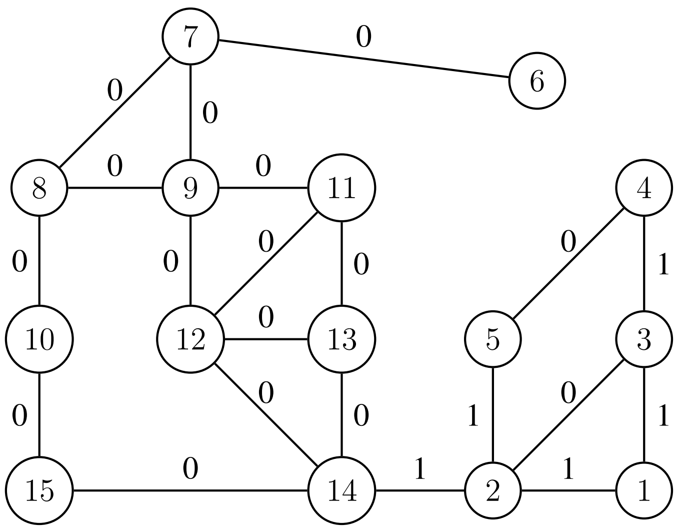

Figure 4.5: Update of edges' level in case 1

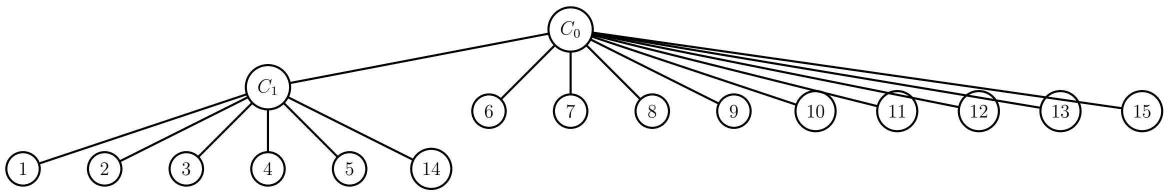

Figure 4.6: Update of cluster tree in case 1

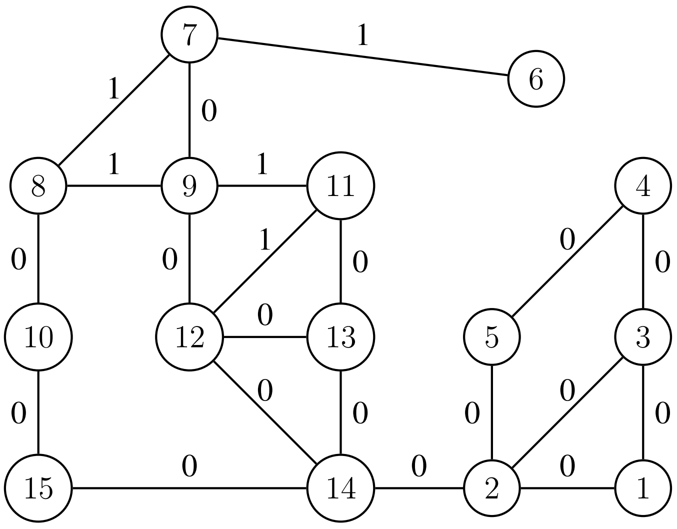
Figure 4.7: Update of edges' level in case 2

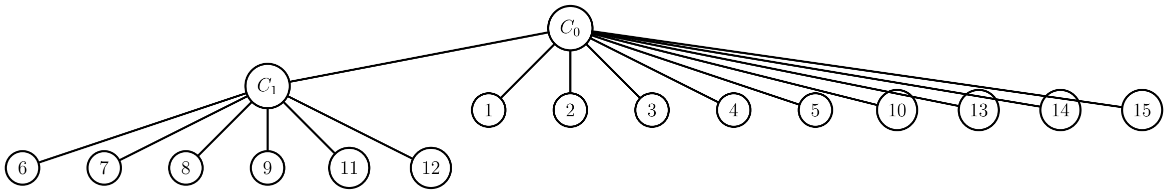
Figure 4.8: Update of cluster tree in case 2

4.2.2 Step2

Case1

We want to delete (2,14) which is a 1-level edge. So we should sub-DFS in 2-level nodes through 1-level edges. If we start from 14, we will find there is no 1-level incident edge to go through in 2-level nodes. So, We don't promote anything but detach 14 from its parent and build a new parent then link to its original grandparent (if there is no grandparent, detach to build a new tree). Obviously, 14 is no longer incident by a 1-level edge, so we suppress it for a level.

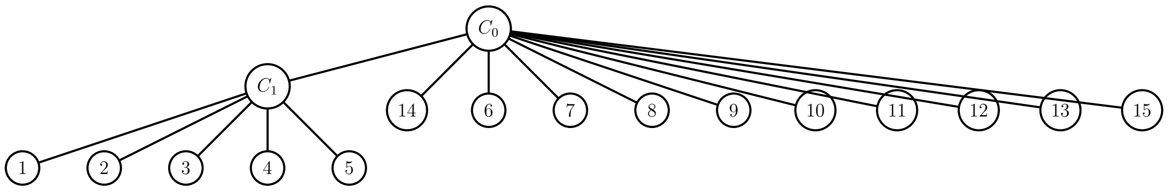
Figure 4.9: Update of cluster tree

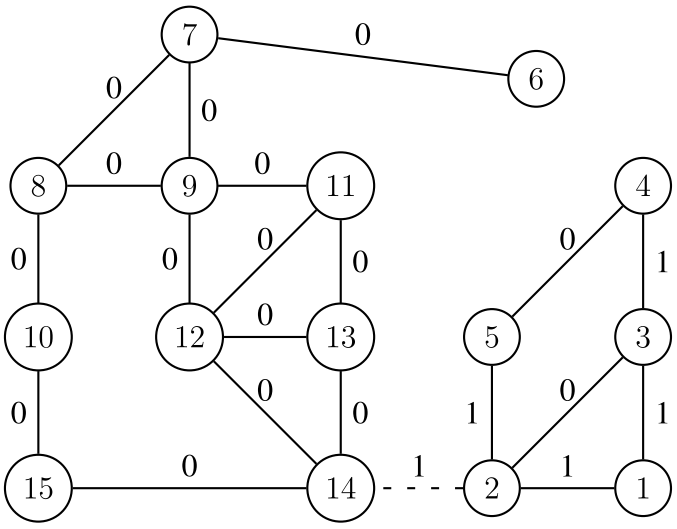

Figure 4.10: An edge to be deleted

Then we recursively search 0-level edge in 1-level nodes (i-1 until 0-level). It's like we zoom in from a graph of vertices to a multigraph of a certain leveled cluster tree nodes. It embodies the idea of divide and conquer as well as distributed operation. Again, there are no 0-level incident edge from C_1. So, We don't promote anything but detach C_1 from its parent and build a new parent then link to its original grandparent (if there is no grandparent, detach to build a new tree). As we can see, C_2 becomes root of a new tree with a detached node child C_2. Then we check initial parent of detached node whether C_0 has only one leaf child to be suppressed. Because search on level 0 edge excuted, update of cluster tree is done.

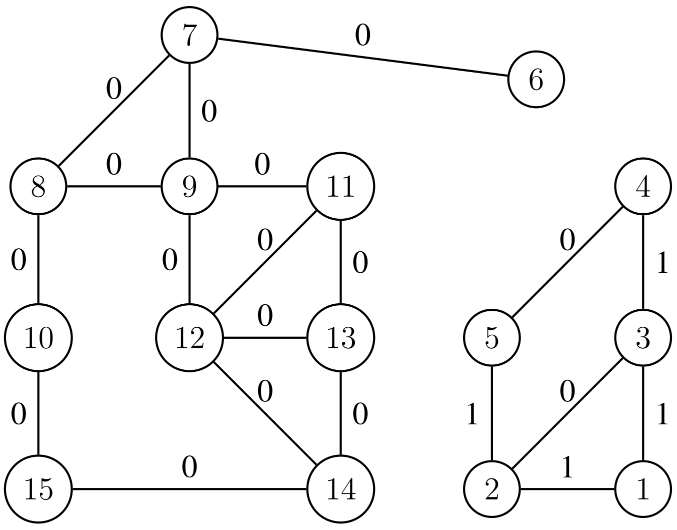

Figure 4.11: Update of edges' level

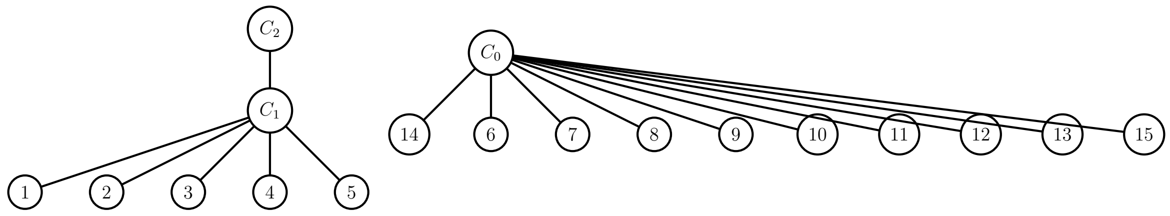

Figure 4.12: Update of cluster tree

Case2

We want to delete (2,14) which is a 0-level edge. So we should sub-DFS in 1-level nodes through 0-level edges. We promote C_v with edges (14,12)(12,13)(14,15)(15,10) or C_u with edges (2,1)(1,3)(3,4)(4,5) (depend on starting order). Then we put a new parent on newly promoted cluster, then link it to originally grandparent (if no grandparent, detach to build a new tree).

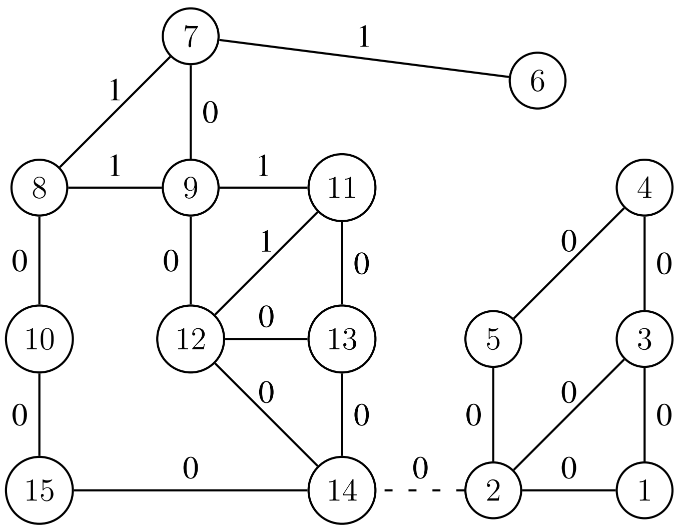

Figure 4.13: An edge to be deleted

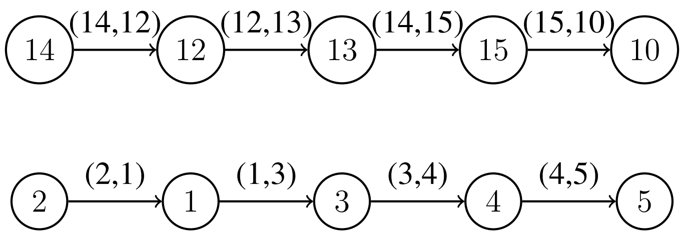

Figure 4.14: Two subsearches

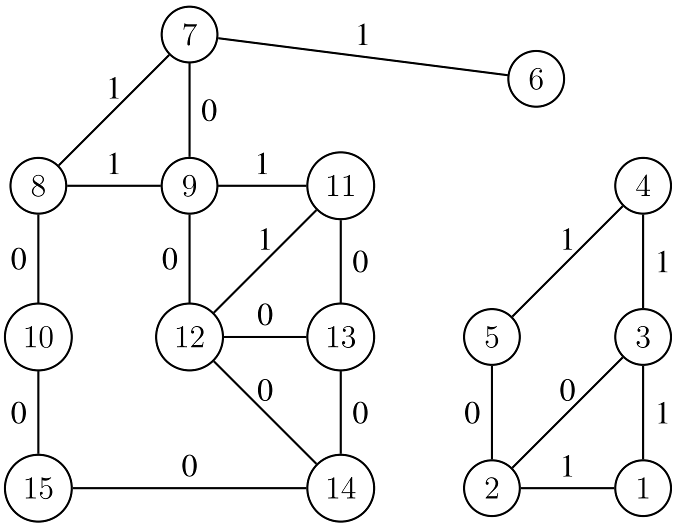

Figure 4.15: Update of edges' level

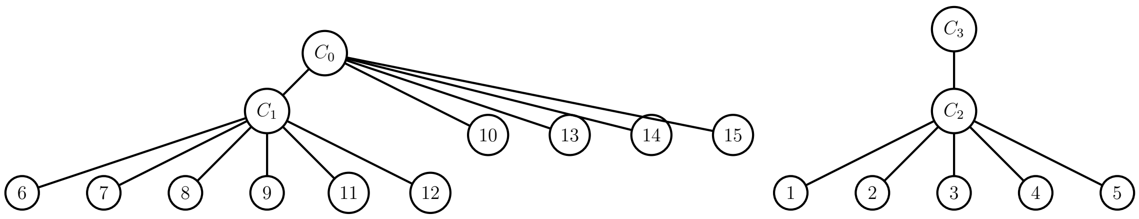

Figure 4.16: Update of cluster tree

4.2.3 Step3

Case1

We want to delete edge(11,12), which is a 0-level edge, so we should search in 1-level nodes from 11 and 12. Assume we will immediately meet at node 9. There is only 12 left without any edge. So we do nothing.

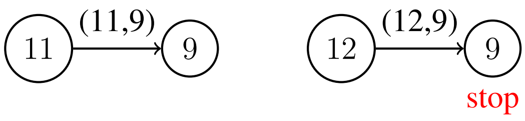

Figure 4.17: Two subsearches

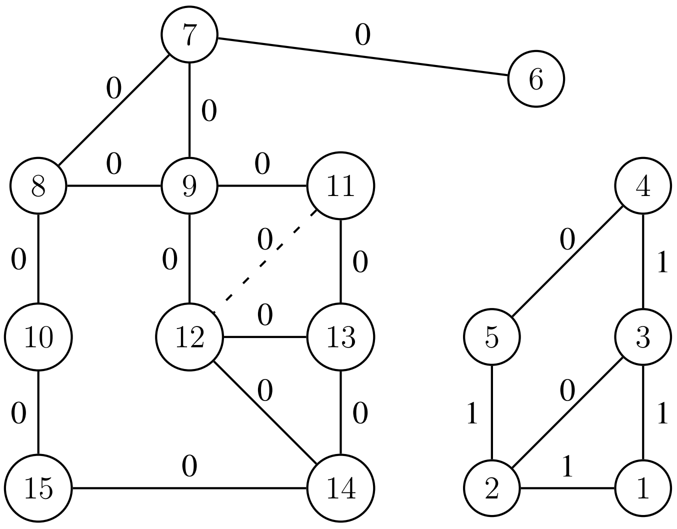

Figure 4.18: An edge to be deleted

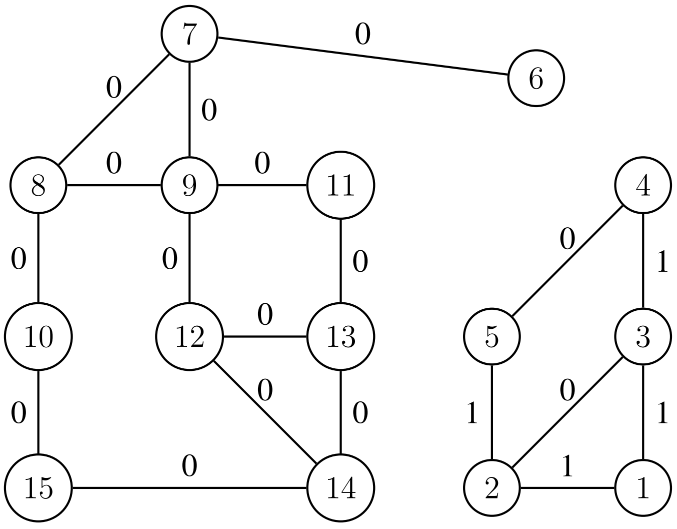

Figure 4.19: Update of edges' level

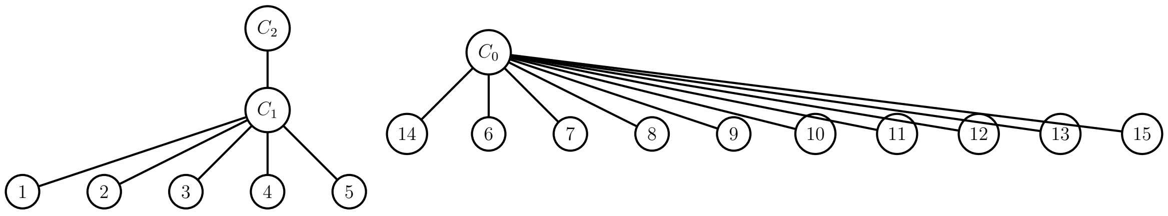

Figure 4.20: Update of cluster tree

Case2

We want to delete edge(11,12), which is a 1-level edge, so we should search in 1-level nodes from 11 and 12. There is no 1-level edge incident to 12, so we detach it and suppress it as above. Then we recursively search in 0-level nodes from C_1 and 12. As shown below, right sub-DFS is shorter but with a larger number of leaf descendants. So we promote left side one.

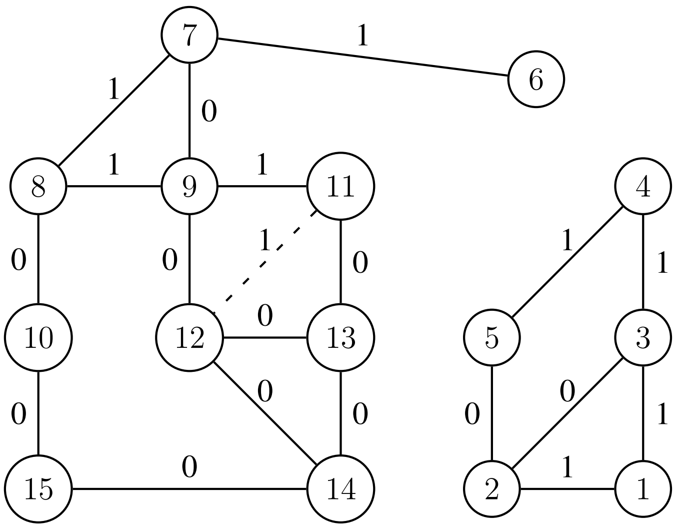

Figure 4.21: An edge to be deleted

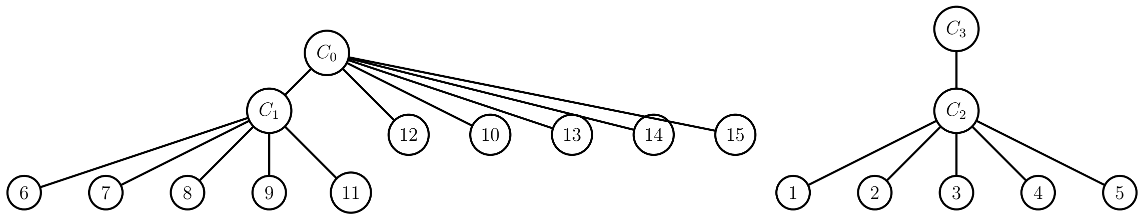

Figure 4.22: Update of cluster tree

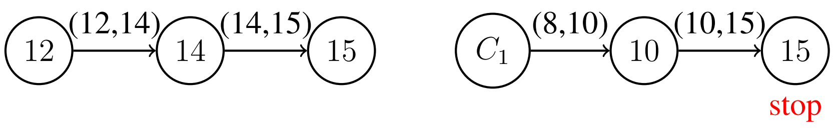

Figure 4.23: Two subsearches

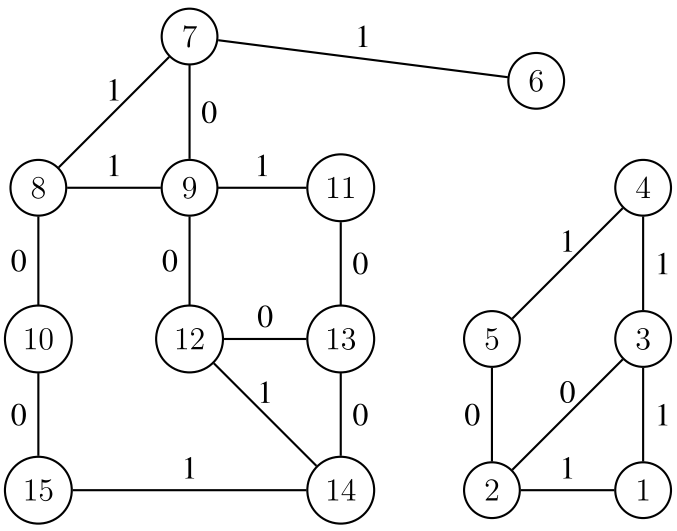

Figure 4.24: Update of edges' level

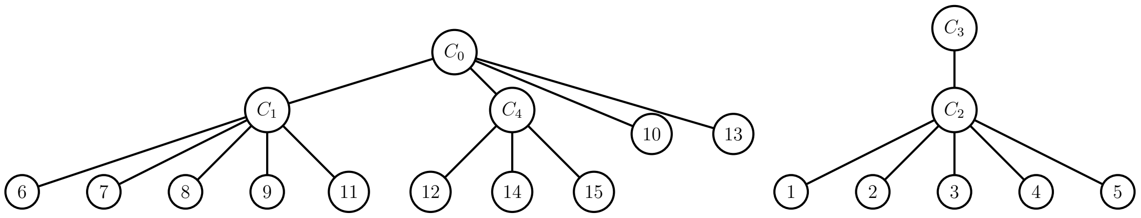

Figure 4.25: Update of cluster tree

4.3 Local tree with bitmap

Now, we want to delete a 0-level edge (9,12) from Step3-Case2. We will figure out how the search procedure executed exactly. In general, we should execute 2 sub-DFS in 1-level node through 0-level edge. On the one sub-DFS, we start from 9, which is a 2-level node, so we will go back to its 1-level ancestor C_1 and find each leaf descendants if there is an incident 0-level edge. Firstly, We find there is no 0-level incident edge to 6. Secondly, We find there is a 0-level incident edge (7,9) to 7, but 7 and 9 are from the same 1-level ancestor C_1, which means for the aspect of multigraph consisting of 1-level nodes, this edge links C_1 to itself, unacceptably. Thirdly, we find a 0-level incident edge (8,10) to 8, which helps jump out from C_1 into another 1-level node 10. While the worst case is we find every node with every edge then get the right one (O(n)), we can apply BBST (balanced binary search tree) to this search procedure to reduce amortized time complexity to O(logn). Then sub-DFS goes on so that cluster is maintained as above. As shown below, right sub-DFS owns a smaller number of leaf descendants. So we promote right side one.

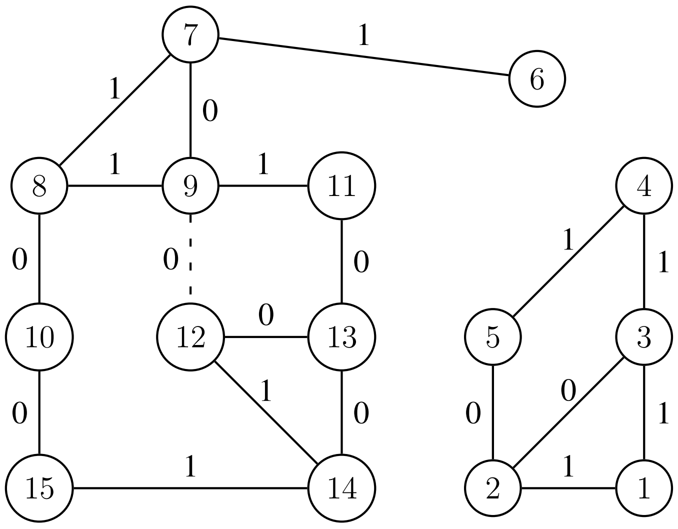
Figure 4.26: An edge to be deleted

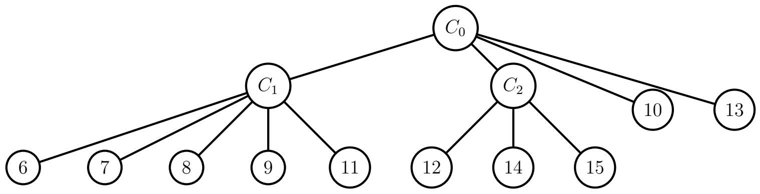
Figure 4.27: Update of cluster tree

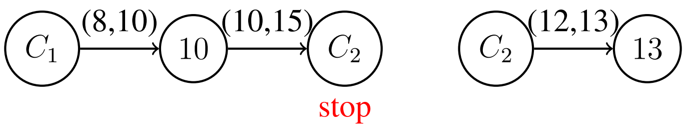

Figure 4.28: Two subsearches

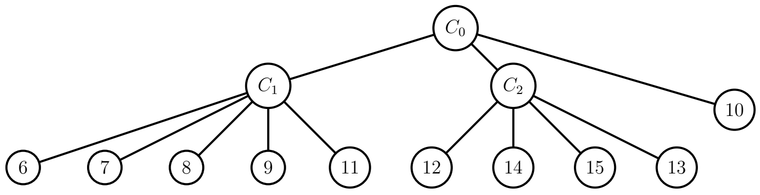
Figure 4.29: Update of cluster tree

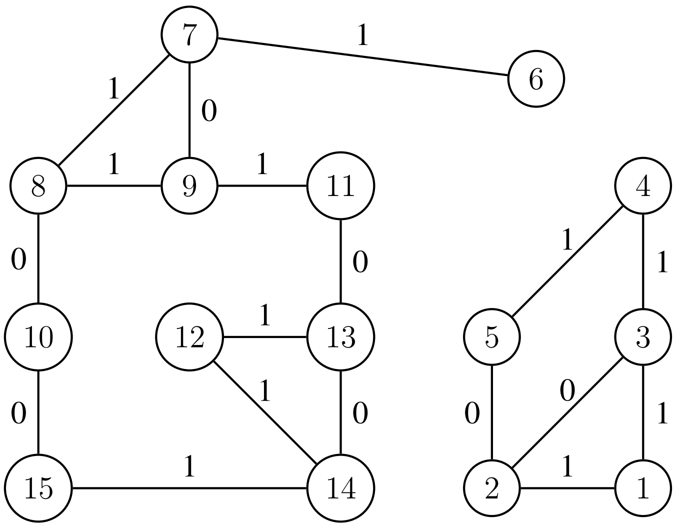

Figure 4.30: Update of edges' level

We would like search procedure be more efficient when go through cluster nodes. For example, we transform the C1 cluster into local tree. First, we rank the children of C1 by rank = $\lfloor logn \rfloor$, n is number of leaf descendants of each child. Then we combine nodes with same rank into a subtree, and add them into a path tree with descending order of rank, in other words, a BBST. At the same time, maintain a bitmap with i-th bit 0/1 indicating if there is at least an i-level edge incident to each leaf descendant. And execute bit-or from leaf to root. As example above, like what we do in a basic way, we will go back to 1-level ancestor C1 and find each leaf descendants if there is anincident 0-level edge. We go to subtree with bitmap[0]=1 recursively, left subtree first, then right subtree. Obviously, amortized time complexityof search procedure will be reduce to O(logn) from O(n). There are leaf nodes and corresponding incident edge. Additionally, for each leaf node, we can group their incident edges by level, then apply a BBST to each group, which will further promote worst case from O(l_{max}) to o($logl_{max}$) = O(loglogn). So far, we transform each search procedure into a purposeful one, not aimlessly searching one by one.

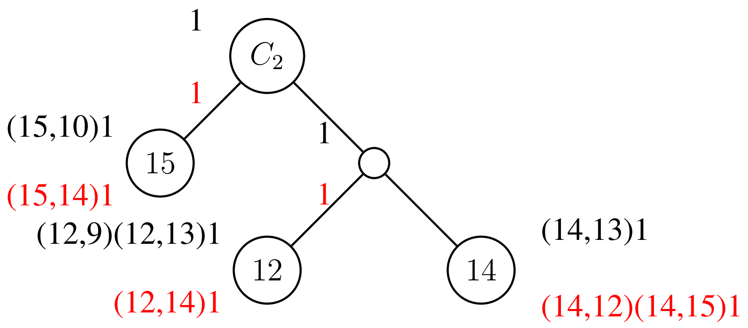

Figure 4.31: A local tree

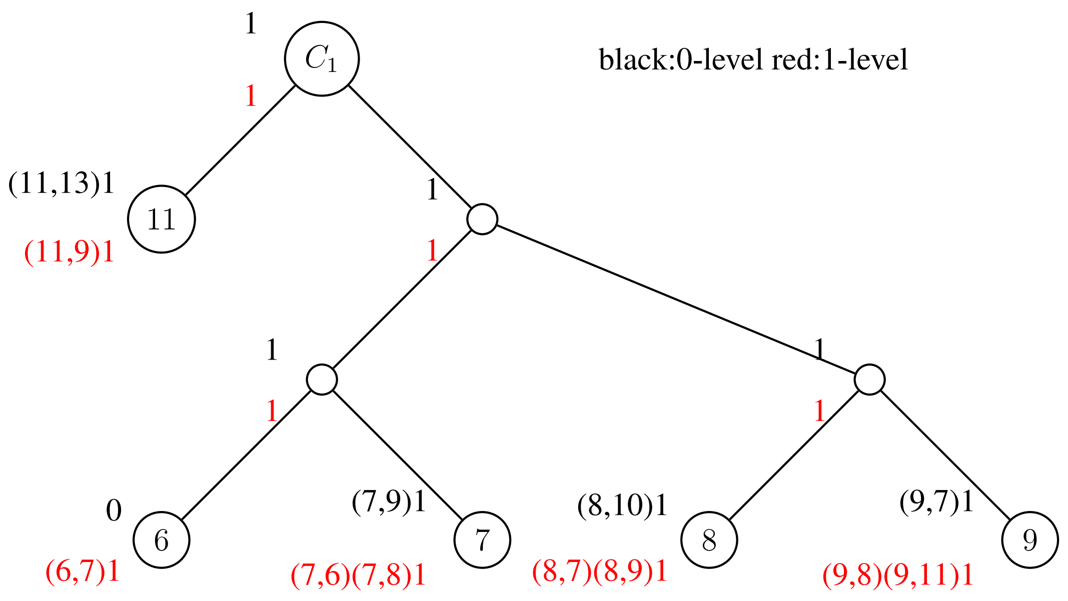

Figure 4.32: The other local tree

After sub-DFS, we maintain cluster, meanwhile, we should maintain local tree as well as bitmaps. We rerank the children of C_2 and build a new local tree rooted at C_2. Because we delete 0-level edge (12,9) and promote 0-level edge (12,13), there is no 0-level edge incident to node 12. So, we should update bitmap[0] of node 12 from 1 to 0. Then we propagate bitmaps by bit-or through nodes from leaf to root.

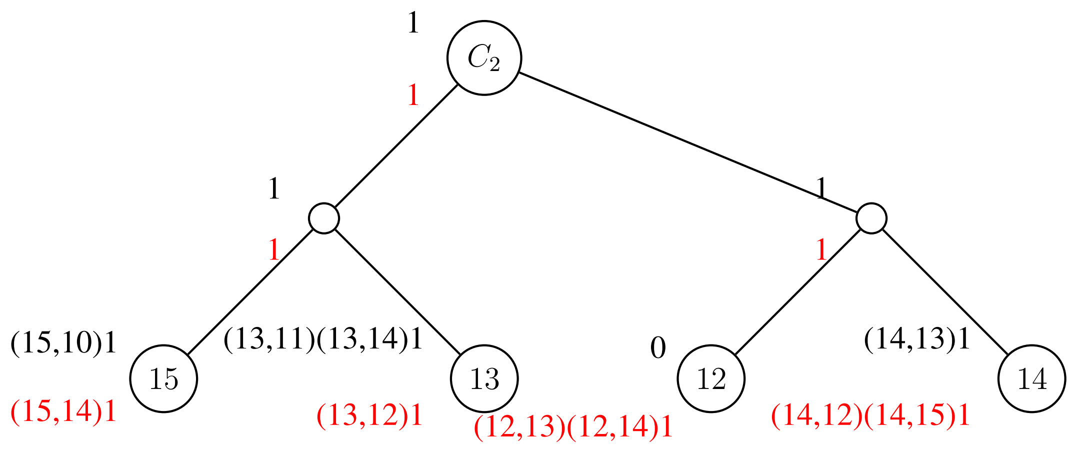

Figure 4.33: An updated local tree

5 Implementation

In our implementation, we focus on three main tasks: construct the data structure of graph and cluster tree, search and update related item when graph change, evaluating and visualizing processes and export the result to the persistent storage (i.e. the hard disk). We have made the following modules to achieve these goals.

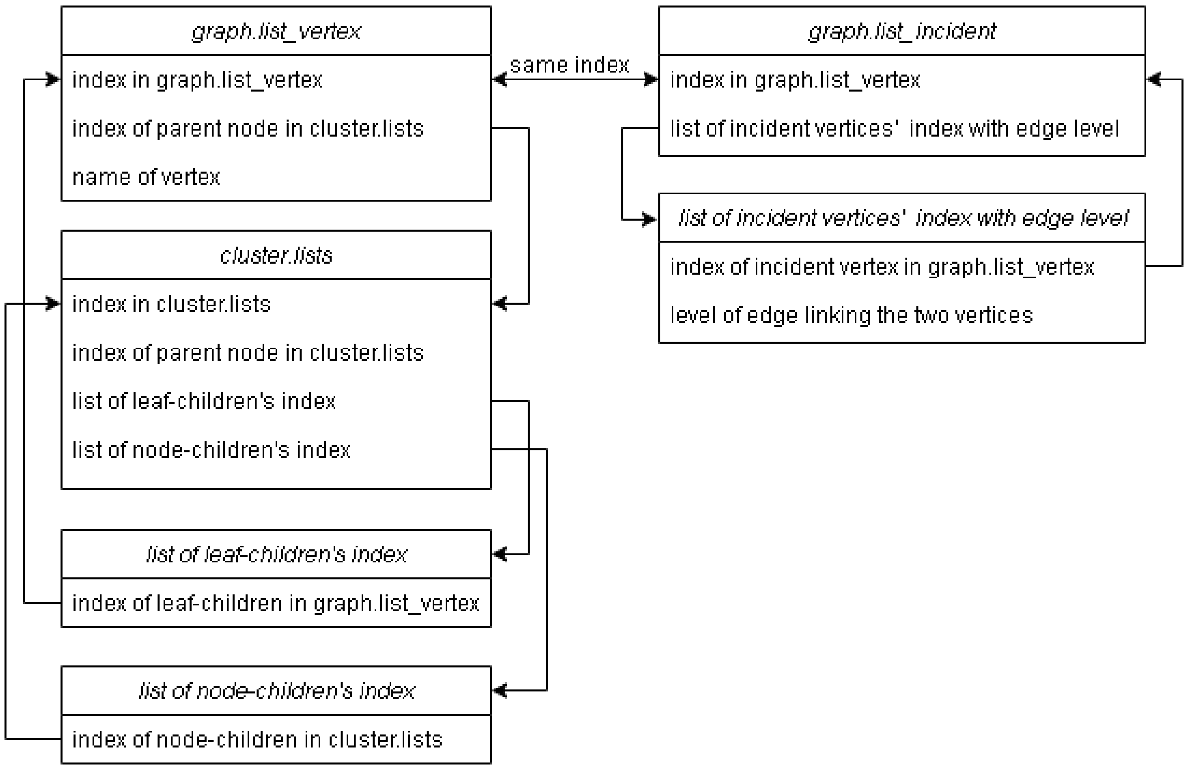
graph.list_vertex
index in graph.list_vertex
index of parent node in cluster.lists
name of vertex
same index
graph.list_incident
index in graph.list_vertex
list of incident vertices' index with edge level
list of incident vertices' index with edge level
index of incident vertex in graph.list_vertex
level of edge linking the two vertices
cluster.lists
index in cluster.lists
index of parent node in cluster.lists
list of leaf-children's index
list of node-children's index
list of leaf-children's index
index of leaf-children in graph.list_vertex
list of node-children's index
index of node-children in cluster.lists

Figure 5.1: Data structure of graph and cluster tree

In order to understand how our implementation works, we will show some important code snippets in this section. Given that we have talked about the update in last part, we will focus on search procedure this part. We will start by showing how traverse leaves (it is called in update of cluster tree) works in the Listing.

```
def trav_vdesc(node):
    for j in range(len(node[1])):
        sub_candi.append(node[1][j])
    for k in range(len(node[2])):
            trav_vdesc(self.lists[node[2][k]])
```

Listing 5.1: Traverse each leaf descendant

The function of traverse each leaf descendant return all of leaf descendants from the input node, which will be used in comparison of number of leaf descendants to determine which sequence to be promoted. It is the key that we keep cluster tree balanced and keep time complexity in O($logn$). Moreover, it's called in next function to check out which path to explored next in multigraph's cluster-like vertex.

```
def jump(i_start,candi_search,root_search,seq):
    if len(candi_search[i_start]) == 1:
        seq.append([1,root_search[1][i_start]])
        candi_search[i_start] = [-2]
        for i in range(len(self.Graph.list_incident\
[seq[-1][1]])):
            if self.Graph.list_incident[seq[-1][1]]\
[i][0] == level_edge:
                for j in range(len(candi_search)):
                    if self.Graph.list_incident[seq\
[-1][1]][i][1] in candi_search[j]:
                        seq.append([seq[-1][1],self.\
Graph.list_incident[seq[-1][1]][i][1]])
                        return j,candi_search,root_search\
,seq
```

Listing 5.2: Jump from a vertex to another in multigraph

The following function tries to find qualified vertex among candidates, all children of search root with the given level of deleted edge, and a lists of explored vertices. By recursively execution, we will get two sequence and determine which to promote later.

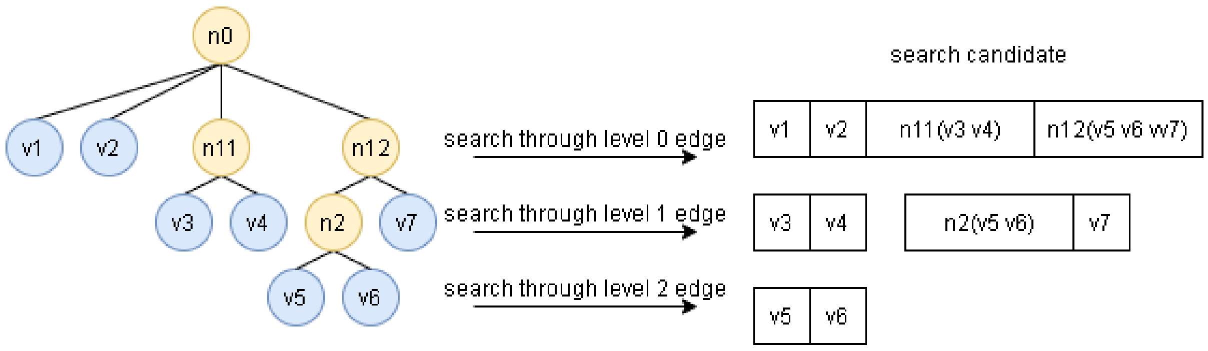

Figure 5.2: Data structure of search candidate

6 Experiment

In this work, we reproduced the results from a simple graph. It is stored as adjacent matrix, containing 22 edges and 15 vertices. We start from figure 6.1, execute update until figure 6.4 and export the visualization results. Since the experiment is significantly simplified, we decided to use larger dataset like Enron, a network of email communication in further application. Given tests of sampling points and running on big dataset without error, we can infer the logical correctness and robustness of the code with high confidence.

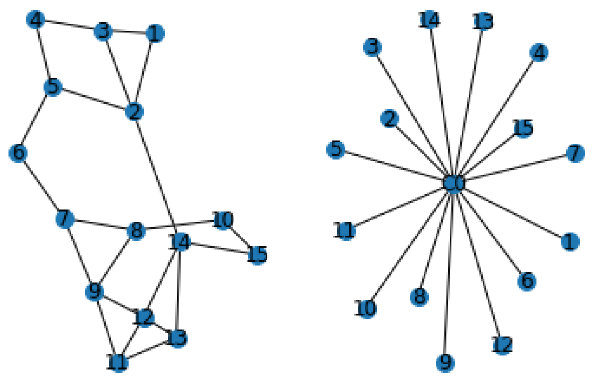

Figure 6.1: Initial graph and cluster

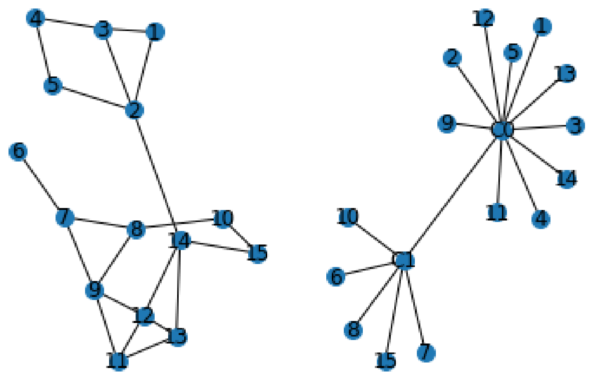

Figure 6.2: Updated graph and cluster after delete (5,6)

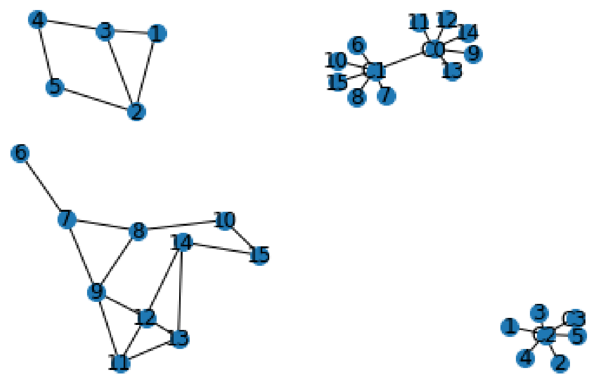

Figure 6.3: Updated graph and cluster after delete (2,14)

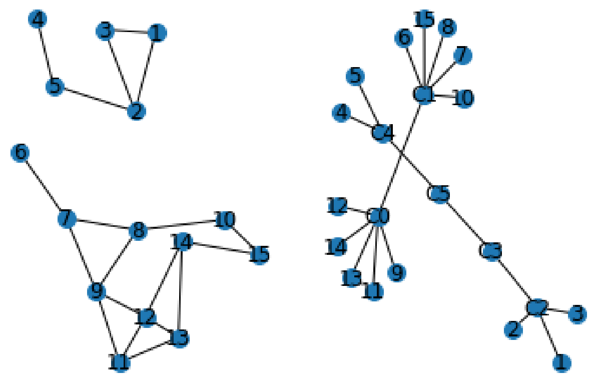

Figure 6.4: Updated graph and cluster after delete (3,4)

There are 1148072 records in Enron dataset, each of which consists of id of email sender and recipient as well as timestamp. First, we sorted the records in ascending order of timestamp. We regard all people as vertices of graph, then insert email communication as undirected and unweighted edge into graph one by one. Once we insert an edge, we check if there are edge with too early timestamp in graph and delete them. Since our implementation is significantly slower than others owing to no optimization with libraries, total running time on enron dataset is around 40 hours using ordinary local machine. In this run, we set check gap of time as one month, as is 2592000 transformed to timestamp. It means two people with no email contact for more than a month lose connectivity. We can tune up or down the gap to check people with looser or tighter connectivity. In summary, the advantages of the algorithm is that they have better performance when applied in the scenarios with low frequency update and high frequency query, and hence should work not only on homogeneous graph like within people, but also heterogeneous graph like between people and items. Like it can be applied to the recommendation algorithm in the future, according to the change of people's interest. However, even though our implement works, we also revealed that the algorithm still has a lot to be optimized and improved.

Bibliography

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 343–350, New York, NY, USA, 2000. Association for Computing Machinery.

[WN13] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1757–1769. SIAM, 2013.