Dynamic Graph Algorithms on Strongly Connected Components in CUDA

SUBMITTED TO IITM

A Project Report

submitted by

SAI SIDDARTH PETA

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2024

THESIS CERTIFICATE

This is to certify that the thesis titled **Dynamic Graph Algorithms on Strongly Connected Components in CUDA**, submitted by **Sai Siddarth Peta**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre
Professor
Dept. of CSE
IIT Madras, 600036

Place: Chennai

Date: 14th May 2024

ACKNOWLEDGEMENTS

This work was completed under the guidance of **Professor Rupesh Nasre**. His advice at various stages of the project helped me stay on the right track. I also gained significant insights into research and academia through his mentorship. I am grateful for his constant support and invaluable guidance.

Special thanks to Mathew, Barenya, Adarsh Gupta, and my friend Soham Tripathy for their insights, and guidance, and for always being there when I needed assistance. Their support has made a significant difference in the completion of this project.

ABSTRACT

KEYWORDS: **Internal Structure**; **Induced Vertex**; **SCC Tree**; **Node**; **Reachability**; **Primary Ancestor**; **Master Node**.

Dynamic graph algorithms are essential for efficiently managing and analyzing graphs that undergo frequent changes, such as the addition or removal of edges. A key challenge in this field is computing Strongly Connected Components (SCCs) in a graph $G = (V, E)$, where each node within a component is reachable from every other node. The classic algorithm by Tarjan computes SCCs in $O(m + n)$ time, where $m = |E|$ and $n = |V|$, but recalculating SCCs can be computationally expensive, especially with large, frequently updated graphs.

CUDA-based parallel algorithms leverage the processing power of GPUs to boost performance in SCC computation. This approach distributes large-scale graph processing across multiple threads, leading to faster execution times. The efficiency is enhanced by limiting computations to only those parts of the graph affected by edge additions or deletions, avoiding recalculations across the entire graph. This targeted strategy is particularly beneficial in dynamic graph scenarios where changes are frequent.

The proposed approach uses an SCC tree structure to represent the hierarchical organization of strongly connected components. By dividing the algorithm into incremental and decremental components, it allows efficient updates to the SCC structure. The incremental component handles edge insertions, potentially expanding SCCs, while the decremental component manages edge deletions, which might cause SCCs to split or shrink. By focusing on specific sections of the SCC tree, this approach significantly reduces computation time. The results indicate that this methodology, centered on the SCC tree and utilizing CUDA-based parallel processing, provides a scalable and efficient solution for managing dynamic graphs in various applications, such as social networks and transportation systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

SCC	Strongly Connected Components
SAC	Split and Condense
LCA	Lowest Common Ancestor
IITM	Indian Institute of Technology, Madras

CHAPTER 1

INTRODUCTION

Many real-world problems can be represented as graphs, with elements as vertices and their relationships as edges. These graphs evolve over time—vertices and edges are added or removed as social networks grow, road maps are updated, or new connections are established. In a social media network, for instance, users can follow or unfollow others, create or deactivate accounts, and engage in various other interactions. Given the dynamic nature of these networks, re-evaluating the entire graph every time a change occurs can be computationally expensive, particularly with large-scale networks. This challenge has prompted the development of dynamic algorithms and data structures designed to efficiently handle dynamic graph updates.

Dynamic graph algorithms are typically categorized based on the operations they support. A fully dynamic algorithm allows for both insertions and deletions of edges. In contrast, an incremental algorithm permits only edge insertions, while a decremental algorithm is focused solely on edge deletions. A fundamental problem in these dynamic graphs is identifying Strongly Connected Components (SCCs), which are subgraphs where each node is reachable from every other node within the same component.

In the context of social media networks, SCCs can represent tightly connected groups or clusters of users. For example, a popular influencer with a large number of followers has many edges leading to them. As people follow or unfollow, these connections change frequently, affecting the overall graph structure. Traditional algorithms for computing SCCs require a full recomputation each time an edge is added or removed, making them inefficient for dynamic environments.

The proposed approach addresses these limitations by focusing on parts of the graph impacted by edge insertions and deletions, without recalculating the entire SCC structure. This selective approach ensures quick updates, making it ideal for dynamic environments where relationships change frequently. By reducing the

computational overhead, this methodology facilitates efficient graph management and opens new possibilities for real-time applications in rapidly evolving networks.

1.1 Problem Statement

This problem Statement addresses the issue of efficiently managing dynamic graphs, particularly focusing on the creation and maintenance of Strongly Connected Components (SCC) trees when edges are added or removed. With graphs becoming larger and more complex, traditional methods of recomputing SCCs from scratch are no longer feasible. This challenge is made even more difficult by the need to maintain the consistency of the graph while also quickly reacting to updates. The goal is to devise algorithms capable of selectively updating SCC trees, thus reducing the time and computational resources required for managing dynamic graph structures.

A significant hurdle lies in determining how to handle edge insertions and deletions without disturbing the overall structure of the graph. This involves both adding new edges and removing existing ones while carefully considering the effects these changes will have on SCC trees. By formulating an optimal strategy for dynamic graph algorithms, we aim to streamline the management of evolving graph structures while ensuring efficient and effective responses to changes.

1.2 Background and Related Work

Strongly Connected Components are fundamental components in graph theory, representing subsets of nodes in a directed graph where every node is reachable from every other node within the same subset. SCC algorithms play a crucial role in various graph-related applications, including network analysis, social network modeling, and compiler optimizations.

1.2.1 Classic SCC Algorithms:

Tarjan's Algorithm Introduced in 1972 by Robert Tarjan, this algorithm is known for its efficiency and simplicity. It operates in linear time complexity $O(|V| + |E|)$, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges in the graph. Tarjan's algorithm utilizes depth-first search (DFS) to discover strongly connected components (SCCs) by keeping track of the vertices in a stack and ensuring that each SCC is identified once all its vertices have been processed. **Kosaraju's Algorithm** Proposed in 1978 by S. Rao Kosaraju, this algorithm also achieves linear time complexity. It is particularly efficient for dense graphs. Kosaraju's algorithm works by performing two passes of depth-first search (DFS). The first pass is on the original graph to determine the finishing times of the vertices, while the second pass is on the transposed graph (where all edges are reversed) to discover the SCCs. **Gabow's Algorithm** Another significant contribution to SCC computation, Gabow's algorithm, developed by Harold Gabow, also operates in linear time. It combines features of both Tarjan's and Kosaraju's approaches, using DFS and a stack-based method to improve performance on certain types of graphs.

1.2.2 Parallel SCC in CUDA:

With the increasing size and complexity of graphs, parallel and GPU-based approaches have gained prominence for SCC computation. These approaches leverage the computational power of GPUs to accelerate graph processing tasks. CUDA-based parallel algorithms offer significant improvements in performance. Many methods have been developed to expedite CUDA-based parallelism for finding SCCs in directed graphs, which is a fundamental problem in graph theory.

Static algorithms typically use depth-first search to find SCCs. However, since DFS is generally difficult to parallelize, researchers have explored alternative approaches such as Forward-Backward Trim and Coloring. In the methods presented in Amilkanthwar *et al.* (2016), the process starts by identifying nodes with the highest in-degree and out-degree and implementing an enhanced pivot selection scheme. This approach improves load balance across warp threads and reduces

thread serialization, leading to significant performance gains. These methods achieve a speedup of up to 8x compared to Tarjan's sequential algorithm and up to 2x compared to previous CUDA implementations.

1.2.3 Dynamic SCC Algorithm:

Dynamic graphs are constantly changing due to edge additions and deletions, which complicate the computation of Strongly Connected Components (SCCs). These changes require efficient algorithms to maintain the SCCs without recomputing them from scratch, as this can be computationally expensive. Edge additions can lead to the merging of SCCs, while edge deletions can cause SCCs to split, necessitating updates to the graph's structure.

In this paper, we enhance the algorithm proposed by Lacki (2013), Introducing the SCC-Tree data structure that enables efficient maintenance of SCCs during edge deletions. The SCC-Tree provides a streamlined representation of the graph, allowing for quick updates and efficient querying as edges are removed. We extended this approach to handle both incremental and decremental updates, achieving a time complexity of $O(m\delta)$ for the tree creation, where δ is the height of the SCC tree. This paper examines updates to the SCC tree during edge insertions and deletions, highlighting differences in major cases such as master nodes and non-leaf nodes. It incorporates the decremental reachability concept from Monika Henzinger and Schulz (2020) and presents the final results of our dynamic SCC algorithm implemented in CUDA, noting the setbacks encountered.

CHAPTER 2

Preliminaries

To help understand algorithms like SCC tree creation, incremental, and decremental algorithms, we introduce several terms and concepts:

2.1 Basic Info:

Let $G = (V, E)$ represent a directed graph, where $V(G)$ denotes the set of vertices and $E(G)$ the set of edges. For vertices $u, v \in V(G)$, the expression $uv \in E(G)$ indicates that $u \to v$ is a directed edge in G. If there is a directed path in G from u to v, we denote it as $u \xrightarrow{G} v$. An inclusion maximal strongly connected set is called a strongly connected component, or SCC. Every vertex in G belongs to exactly one SCC or a unique SCC tree.

2.2 Condense Form:

A condensation of a graph G is a directed graph G' that is derived by contracting all its strongly connected components (SCCs) into single vertices. Each vertex in G' represents a set of vertices that form an SCC in G.

We refer to the condensation of G as CONDENSE(G). The condensed node encapsulates the internal structure of their respective SCCs. Each edge in CONDENSE(G) directly corresponds to an edge in G, and we often associate these corresponding edges with each other in our discussions.

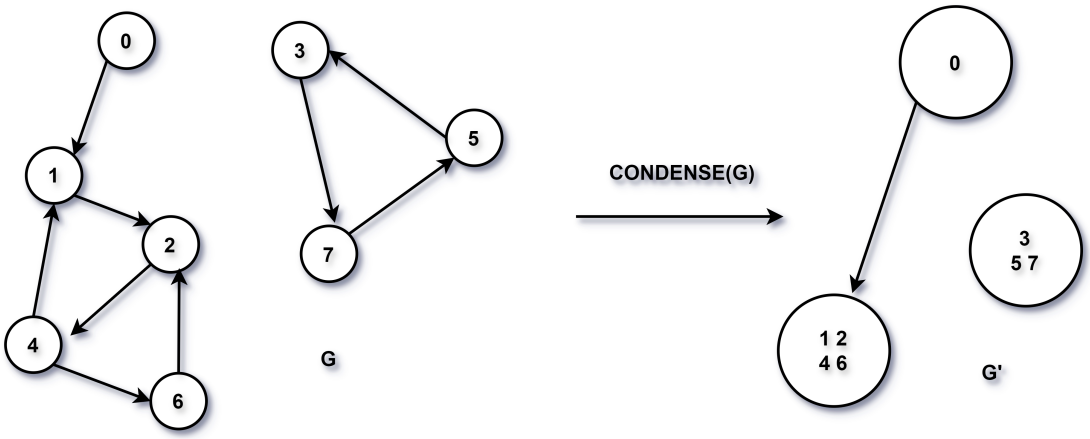


Figure 2.1: Graph G and its condense form G':= CONDENSE(G).

2.3 SPLIT(G,s):

Let $s \in V(G)$. We define SPLIT(G, s) as the graph resulting from dividing vertex s into two vertices: s_{in} and s_{out}. In this split, s_{in} receives all incoming edges of s, and s_{out} all outgoing edges. We also refer to this process as inducing the graph using vertex s, denoted by SPLITANDCONDENSE(G, s) := CONDENSE(SPLIT(G, s)). This denotes the operation where we split the graph at vertex s and subsequently condense it, based on the SCCs that are formed.

It is noteworthy that SPLIT(G, s) inherently lacks strong connectivity, featuring a singular source $\{s_{\text{out}}\}$ and a singular sink $\{s_{\text{in}}\}$. Consequently, SPLITANDCONDENSE(G, s) comprises multiple vertices. These vertices serve as the distinguished source and sink, fundamental in assessing the reachability within the SCCs, a critical component of the algorithm.

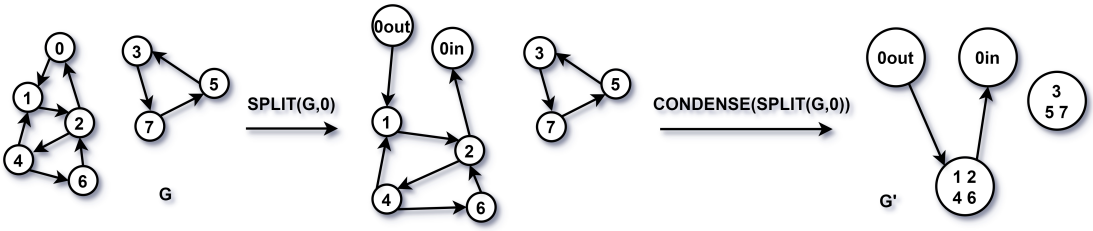


Figure 2.2: Graph G, SPLIT(G,0) and CONDENSE(SPLIT(G,0)).

2.4 Unreachability:

First, let's focus on reachability to understand unreachability better. Reachability is a key concept in the context of edge deletions or the decremental part of the algorithm. Reachability refers to the list of nodes that can be reached from a specific node. In other words, vertex V is considered reachable from vertex U if there is a path from U to V. In this paper, we define reachability somewhat differently from the conventional approach, as we are working with a graph structure that has a single source and sink. Given this structure, we define reachability as the nodes that lie along any path from the source to the sink. Thus, a node is considered reachable if it exists on any of the paths between the source and the sink. Vertices that are not reachable are classified as unreachable nodes. When working on edge deletions, the list of these unreachable nodes plays a crucial role.

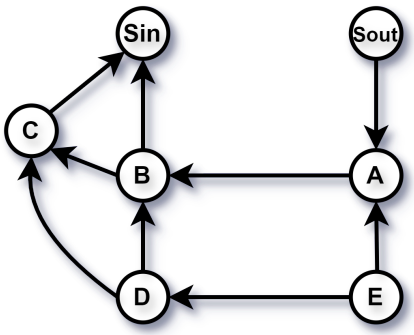


Figure 2.3: Graph G and CONDENSE(G,s).

For a more detailed understanding of this concept, consider the example shown in Fig 2.3, where the structure and the paths from source to sink are illustrated. This provides a clearer picture of reachability within the context of the split graph structure.

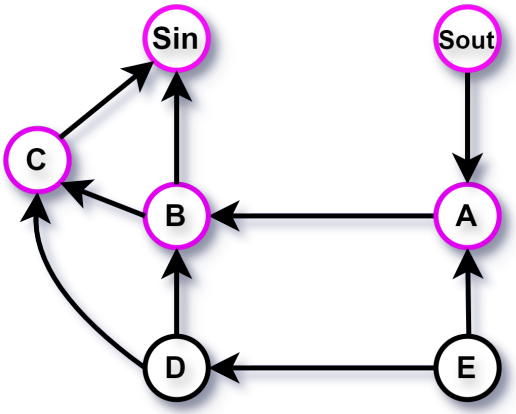


Figure 2.4: Reachable Nodes from Sout.

Working with CUDA, we can simplify the process by applying parallelized breadth-first search (BFS) or depth-first search (DFS) from the Source or Sout. This approach identifies vertices that are reachable from the source, including the source in the list. As depicted in Fig 2.4, the colored vertices represent those reachable from the source, which include {Sout, A, B, C, and Sin}.

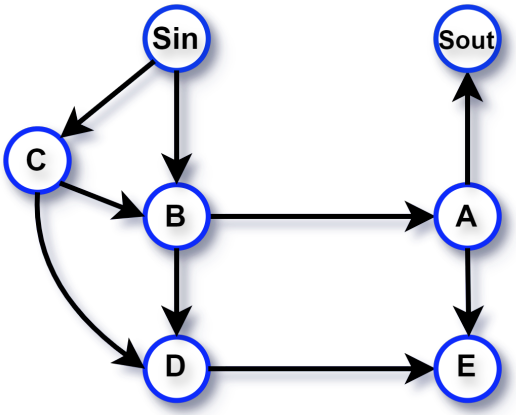


Figure 2.5: Reachable Nodes from Sin after reversing Graph.

To find the reachability from the Sink (Sin), we reverse the graph and apply parallelized breadth-first search (BFS) or depth-first search (DFS) on the Sink. This identifies the vertices that are reachable from the sink, including the Sink (Sin) in the list. As shown in Fig 2.5, the colored vertices represent those reachable from the Sink, which is {Sin, C, B, D, A, E, and Sout}.

To obtain the list of reachable and unreachable nodes, we use the union and intersection operations on the reachability lists derived from the source and the

sink. According to the given example, the reachable nodes are {Sin, A, B, C, Sout}, while the unreachable nodes are {D, E}. This is how unreachability is handled within the induced graph structure inside a node, which is beneficial for the decremental part of the algorithm.

This method provides an efficient way to identify which nodes are reachable and which are not in the context of edge deletions. By focusing on the unreachability from both source and sink, you can determine which nodes require additional processing during the decremental operations.

CHAPTER 3

Algorithms

In this section, we will explore the algorithms developed to mitigate the need for complete recomputation of the strongly connected components (SCC) algorithm on dynamically changing graphs. Instead, these algorithms focus on updates localized to the regions of change within the graph, leveraging CUDA for efficient computation.

Our approach incorporates four key subroutines: **edgeDeletion(u, v)**, **edgeInsertion(u, v)**, **query(u, v)**, and **sccCount()**. Each subroutine plays a vital role in maintaining the integrity and accuracy of the SCC information as the graph evolves.

- **edgeDeletion(u, v):** This subroutine manages the deletion of an edge between vertices u and v.
- **edgeInsertion(u, v):** This subroutine is responsible for inserting an edge between vertices u and v.
- **query(u, v):** This function determines whether the vertices u and v are part of the same Strongly Connected Component (SCC).
- **sccCount():** This function returns the number of SCCs currently identified in the graph.

The foundational data structure underpinning these operations is the SCC Tree. This structure encapsulates the entire graph by transforming it into a tree format that retains detailed SCC information about each vertex and edge. The SCC Tree is dynamic; it adapts to changes in the graph's topology triggered by edge insertions and deletions. This adaptive capability is critical for efficiently updating SCC information without necessitating a full recomputation, thus enhancing the performance of SCC algorithms on dynamic graphs. Let's examine in detail how SCC tree creation works with a planar graph. We'll explore how the SCC tree is dynamically updated through edge insertions and deletions, and also discuss how queries are answered.

3.1 SCC-Tree Creation

The structure for each node in the SCC tree accommodates the results from the SPLITANDCONDENSE(G, s) algorithm. This arrangement allows tracking of the strong connectedness of graph G as long as only edges that belong to SPLITANDCONDENSE(G, s) are removed or inserted. Additionally, the attributes within the node retain information about the strongly connected components (SCCs). This recursive approach results in a dynamic and robust graph representation, allowing efficient monitoring of connectivity between the SCCs.

Given a strongly connected graph $G = (V, E)$, the construction of an SCC tree is as follows:

- A master node is a unique and essential node for every graph G, initially containing all the information about the graph. But the Graph in the master node doesn't induce on any vertex. As the SCC tree is constructed, this information is gradually dispersed across the tree.
- If the graph $G = (\{s\}, \emptyset)$ contains only a single vertex s, the SCC tree consists of a single node representing this vertex and a master node without any edges.
- If G has more than one vertex, the root node of the SCC tree is initialized with the result of SPLITANDCONDENSE(G, s), (where s is an arbitrary induced vertex selected from the nodes of the current node). The child nodes of the root represent the strongly connected components (SCCs) produced by SPLITANDCONDENSE(G, s). For each SCC generated from a vertex s, its corresponding SCC tree is added as a subtree to the root node. However, rather than adding two separate children s_{in} and s_{out}, only one child node is added for each induced vertex s.

The SCC tree can be visualized as a hierarchical structure where nodes represent vertices or clusters of vertices from the original graph. Leaves in the SCC tree correspond exclusively to individual vertices from graph G. Inner nodes, identified by negative numbers, represent non-leaf nodes in the tree, while non-negative numbers denote vertices. We denote the parent node of any node N by $p(N)$, the graph structure within a node N is referenced as $D(N)$ and the induced subgraph rooted at node N, is referred to as $I(N)$ which is formed by implementing split and condense operations on node N.

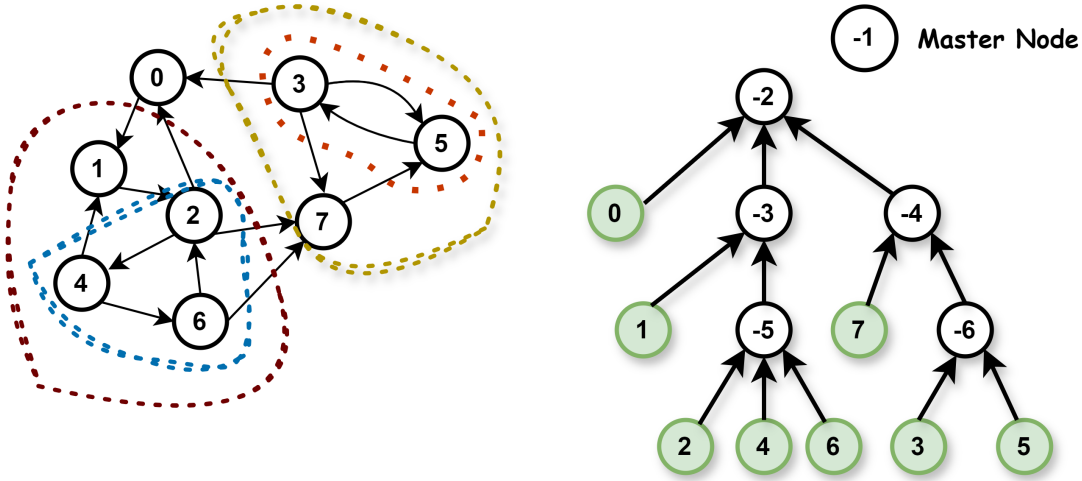


Figure 3.1: Graph G and SCC Tree T(G).

The procedure for creating an SCC tree involves several key steps. Consider that the current node under analysis is N. To start, retrieve all the edges associated with node N, and the corresponding graph structure of N is SPLITANDCONDENSE$(D(N), s)$:= I(N), where $D(N)$ represents the graph structure within node N, and $s \in V(D(N))$. By applying the SCC algorithm to this graph, you identify the strongly connected components (SCCs) within node N.

For each SCC discovered, create a new node and distribute the edges from node N among its newly created children, based on the labeling strategy used to form these nodes. The same process is then recursively applied to each of the new nodes until the algorithm reaches the leaf nodes, where no further division is possible. A comprehensive flowchart illustrates this process in Fig 3.2.

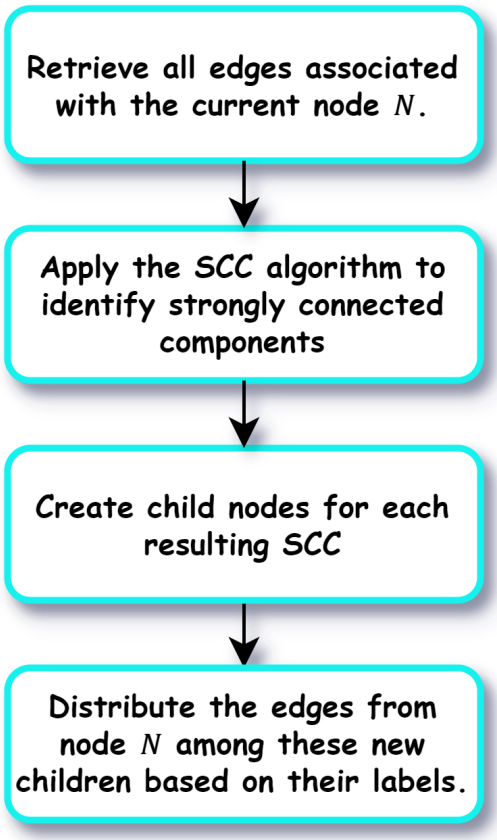


Figure 3.2: Flow Chart for Tree Creation.

A detailed example is presented below to demonstrate the approach in practice. This example explains the sequential steps and outcomes during the construction of the SCC tree.

In the context of the graph $G = (V, E)$ depicted in Fig 3.1, we apply the SCC Tree algorithm to identify the strongly connected components (SCCs). The initial setup involves a master node with label -1 containing all information about the graph, including all edges E, with a fixed label of -1. This master node encompasses all vertices in the range $(0, V)$.

When the SCC algorithm is applied to the master node, it identifies the primary SCCs within the graph G. These SCCs form the basis for individual SCC trees. Each SCC tree represents a unique strongly connected component and develops its internal structure as it grows. The edges from the master node are distributed among these SCC trees, following the structure defined by the SCC algorithm.

The number of nodes within the master node indicates the total number of

SCCs in graph G. As each SCC tree develops, it functions independently, maintaining its structure while contributing to the overall representation of the strongly connected graph G.

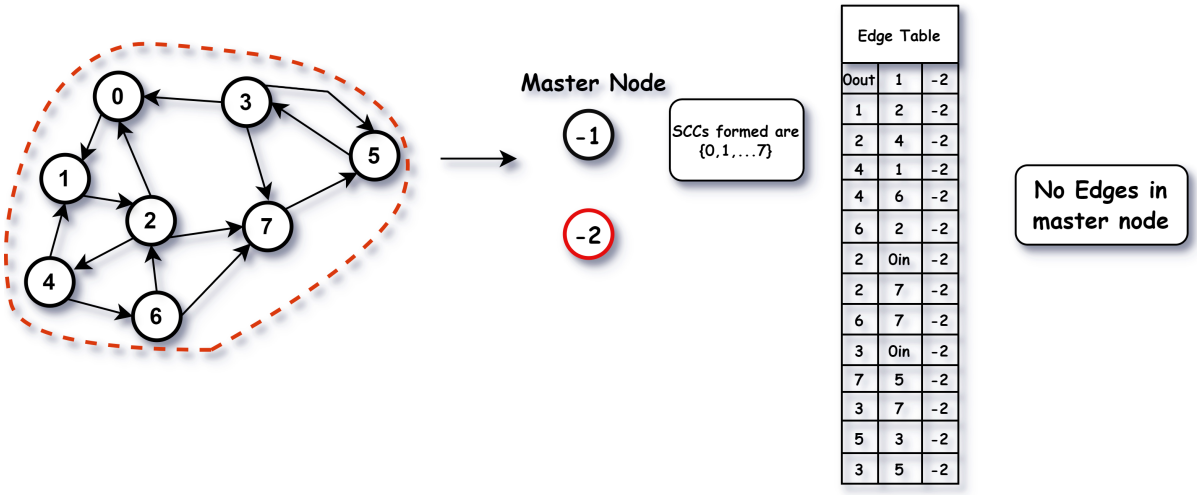


Figure 3.3: **Node -1**: Graph G $\xrightarrow{Algo}$ Tree,Edge Table,Final Structure

Applying the SCC algorithm to the master node results in a single strongly connected component (SCC) containing the nodes {0, 1, 2, . . . , 7}. This leads to the creation of a new child node, labeled -2, with the induced vertex chosen randomly—assumed here to be 0. The next step involves distributing the edges based on the labels of these newly created children.

Since vertex 0 is the induced vertex in node -2, the graph structure contained within node -2 is considered SPLIT(D(-2), 0) initially. The edge distribution is adjusted accordingly. The specific arrangement of edges is depicted in Fig 3.3, with additional details provided in the diagram adjacent to the edge table. This layout demonstrates the edges or graph remaining in node -1 after the algorithm has completed its process. Given that the entire graph forms a single SCC, there are no remaining edges in the master node.

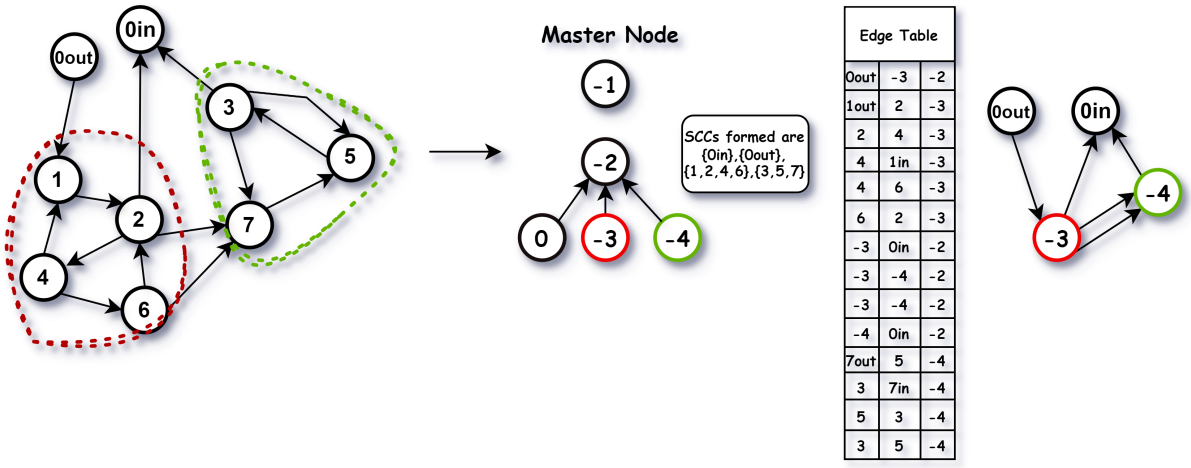


Figure 3.4: **Node -2:** SPLIT$(D(-2), 0) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-2), 0)$

Applying the SCC algorithm to node -2 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{0_{\text{in}}\}$, $\{0_{\text{out}}\}$, {1, 2, 4, 6}, and {3, 5, 7}. This results in the creation of three new child nodes, labeled 0, -3, and -4. The induced vertex for node -3 is randomly chosen and assumed to be 1, while the induced vertex for node -4 is 7.

As specified in the third step of the SCC tree creation process, instead of creating two separate children for $\{0_{\text{in}}\}$ and $\{0_{\text{out}}\}$, only a single child node is added for the induced vertex 0. The next step involves distributing the edges based on the labels of these newly created children.

Given that the induced vertex for node -3 is 1, the initial graph within this node is derived from SPLIT(D(-3), 1). Similarly, the induced vertex for node -4 is 7, which leads to SPLIT(D(-4), 7). The edges from the current node -2 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in {1, 2, 4, 6} is replaced by -3, and those with vertices in {3, 5, 7} are replaced by -4. The edge labels are updated accordingly to reflect the new structure.

The exact distribution of the edges is shown in Fig 3.4, with further details in the diagram beside the edge table. This diagram illustrates the state of node -2 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-2), 0).

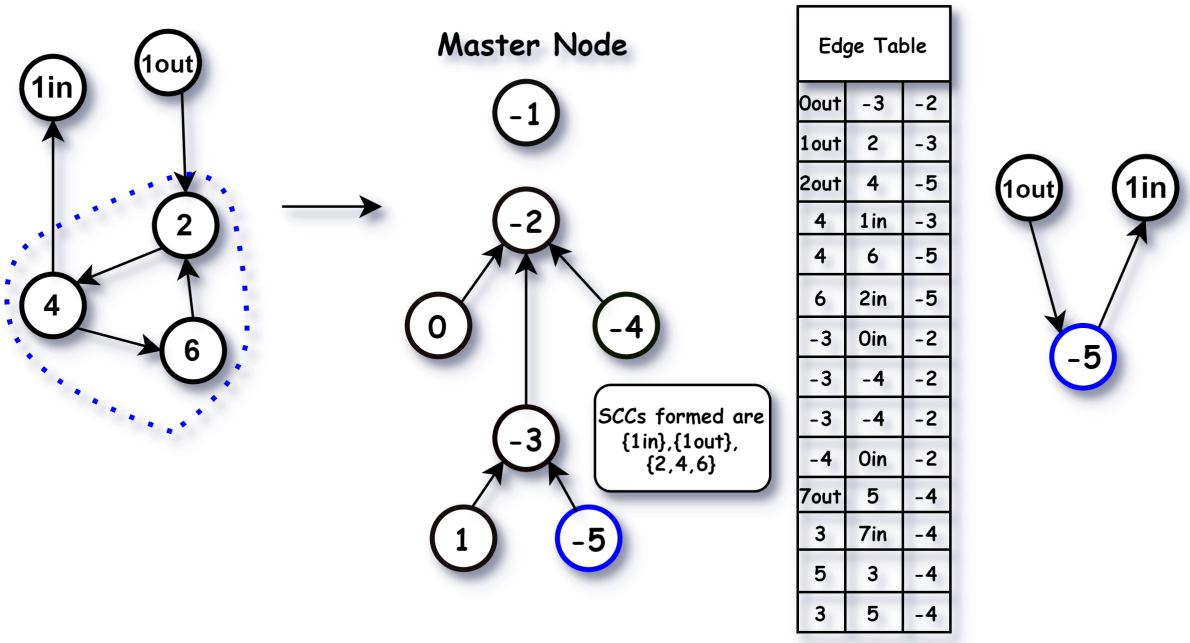


Figure 3.5: **Node -3:** SPLIT$(D(-3), 1) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-3), 1)$

Applying the SCC algorithm to node -3 leads to the formation of three strongly connected components (SCCs) with the following nodes: $\{1_{\text{in}}\}$, $\{1_{\text{out}}\}$, and {2, 4, 6}. This results in the creation of two new child nodes, labeled 1, and -5. The induced vertex for node -5 is randomly chosen and assumed to be 2.

As specified in the third step of the SCC tree creation process, instead of creating two separate children for $\{1_{\text{in}}\}$ and $\{1_{\text{out}}\}$, only a single child node is added for the induced vertex 1. The next step involves distributing the edges based on the labels of these newly created children.

Given that the induced vertex for node -5 is 2, the initial graph within this node is derived from SPLIT(D(-5), 2). The edges from the current node -3 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in {2, 4, 6} is replaced by -5. The edge labels are updated accordingly to reflect the new structure.

The exact distribution of the edges is shown in Fig 3.5, with further details in the diagram beside the edge table. This diagram illustrates the state of node -3 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-3), 1).

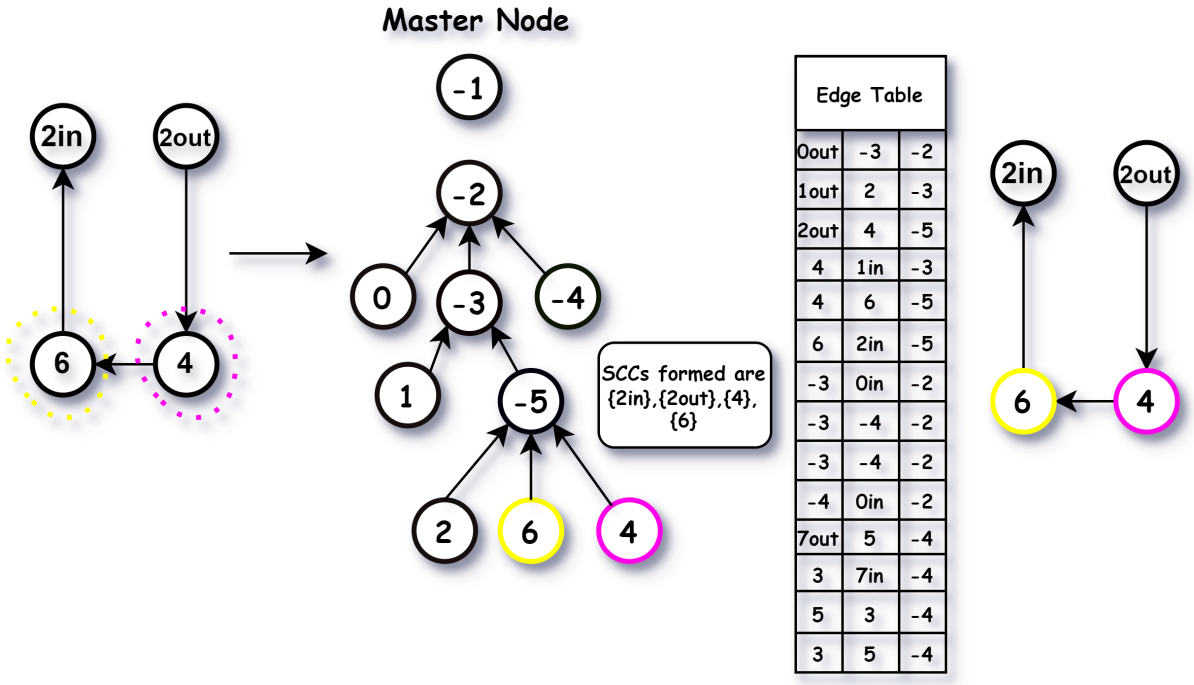


Figure 3.6: **Node -5:** SPLIT$(D(-5), 2) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-5), 2)$

Applying the SCC algorithm to node -5 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{2_{\text{in}}\}$, $\{2_{\text{out}}\}$, $\{4\}$, and $\{6\}$. This results in the creation of three new child nodes, labeled 2, 4, and 6. As they are leaf nodes no induce vertex is chosen. Two separate children $\{2_{\text{in}}\}$ and $\{2_{\text{out}}\}$, are considered as single node 2.

Given that the induced vertex for node -5 is 2, the initial graph within this node is derived from SPLIT(D(-5), 2). The edges from the current node -5 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in $\{4\}$ is replaced by the same 4, and those with vertices in $\{6\}$ are replaced by 6 itself. The edge labels are updated accordingly to reflect the new structure.

Fig 3.6 illustrates the state of node -5 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-5), 2).

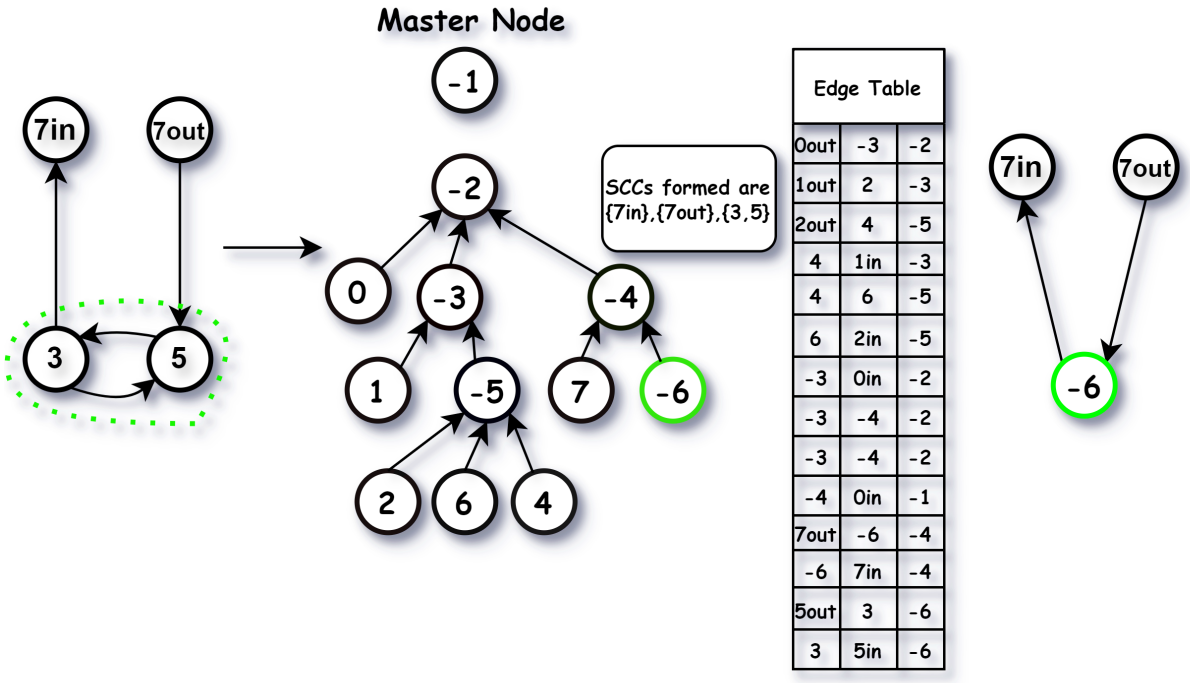


Figure 3.7: **Node -4**: SPLIT$(D(-4), 7) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-4), 7)$

Applying the SCC algorithm to node -4 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{7_{\text{in}}\}$, $\{7_{\text{out}}\}$, and {3,5}. This results in the creation of two new child nodes, labeled 7, and -6. The induced vertex for node -6 is randomly chosen and assumed to be 5. Two separate children $\{7_{\text{in}}\}$ and $\{7_{\text{out}}\}$, are considered as single node 7.

Given that the induced vertex for node -4 is 7, the initial graph within this node is derived from SPLIT(D(-4), 7). The edges from the current node -4 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in {3,5} is replaced by -6. The edge labels are updated accordingly to reflect the new structure.

Fig 3.7 illustrates the state of node -4 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-4), 7).

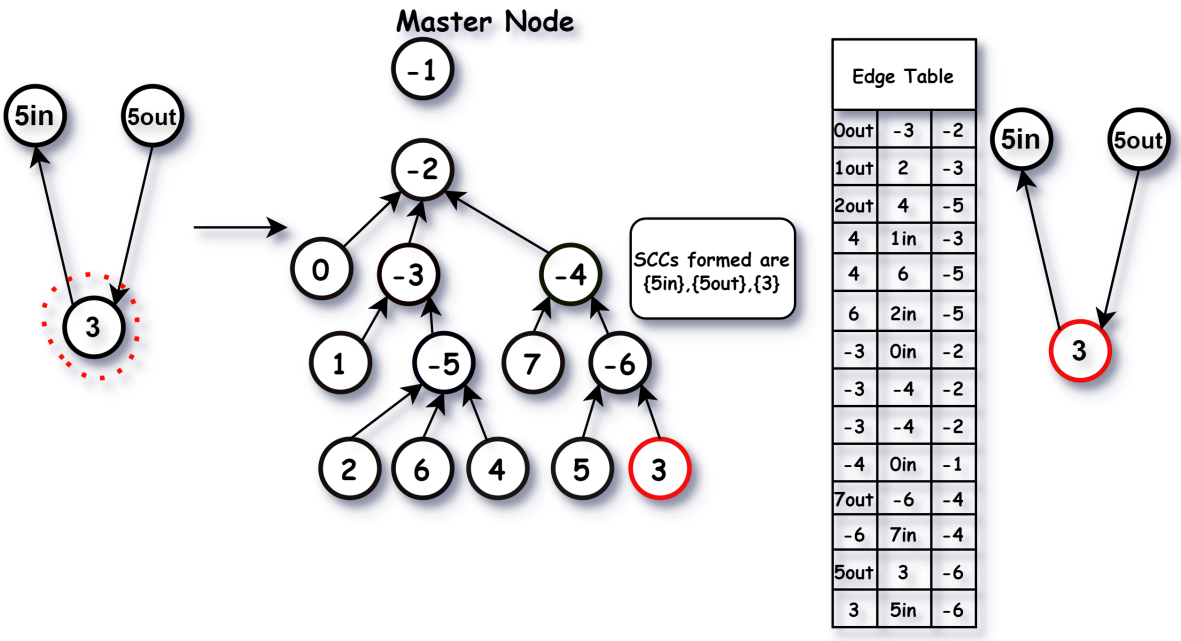


Figure 3.8: **Node -6:** SPLIT$(D(-6), 5) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-6), 5)$

Similarly, if we apply the algorithm to node -6, we can follow the same steps outlined above to construct an SCC tree. This structure proves to be useful for efficiently handling the deletion and insertion of edges. In the next section, we will explore the algorithms for inserting and deleting edges within this SCC tree.

3.2 Updating SCC Tree after Edge Insertion

This section explains the process of handling edge insertions in the SCC tree structure. The algorithm maintains one SCC tree for each strongly connected component (SCC) of the graph. When an edge within an SCC is inserted, the corresponding SCC tree is updated to reflect the modified graph structure. If the insertion causes SCCs to merge, a new SCC tree is generated by merging the respective SCC trees.

The incremental algorithm for updating the SCC tree is similar to the SCC tree creation algorithm, with additional conditions and adjustments.

To facilitate query responses, a map is maintained to store the labels of nodes and their corresponding addresses in GPU memory. This map allows the algorithm

to quickly locate the address of a given node. When a query about two nodes u and v, is made a kernel function is launched to find their addresses. The attribute pa_u retrieves the primary ancestor label for node u, and pa_v does the same for node v.

To determine if u and v are part of the same strongly connected component, the algorithm checks whether the primary ancestors for both nodes are the same. If $\text{pa}_u == \text{pa}_v$, then u and v belong to the same SCC. This allows for constant-time query resolution. Since the initial assumption is that the graph is strongly connected, for each node u, pa_u points to the root of a unique SCC tree.

3.2.1 Procedure:

The procedure for updating an SCC tree during edge insertion involves several steps, similar to the creation of an SCC tree. While the general process remains consistent across cases, there are some differences in the metadata on which the algorithm operates.

Suppose we are inserting an edge uv into a graph G. The first objective is to determine where this edge should be added. This is accomplished by finding the Lowest Common Ancestor (LCA) of the nodes u and v. To do this, we identify the addresses of u and v and then apply the LCA algorithm. Once the LCA node is identified, the new edge is inserted according to the induced structure defined in the LCA node.

For example, consider the current node under analysis N. The updated graph structure within node N after the edge insertion is denoted as X'. Induced vertex in node N is d. Consequently, the induced structure in node N is given by SPLITANDCONDENSE(X', d) where $uv \in E(\text{SPLITANDCONDENSE}(X', d))$. When inserting the edge uv into the LCA, the first vertex u should be mapped to a node labeled A, and the second vertex v should be mapped to a node labeled B. In this context, $P(A)$ and $P(B)$ should be LCA. Additionally, $u \in V(D(A))$ and $v \in V(D(B))$. This implies that the LCA must have children labeled as A and B, with $V(D(A))$ containing vertex u, and $V(D(B))$ containing vertex v.

After retrieving all the edges associated with node N along with the new edge

inserted, apply the SCC algorithm to SPLITANDCONDENSE(X', d) to identify the strongly connected components (SCCs). During this process, SCCs containing a single node can be ignored, as they are part of the original node N. For SCCs with more than one node, check if their SCC size is equal to the number of nodes in node N; if so, they are already represented by node N. If not, create a new node, appending the nodes from these SCCs as children to the new node created. Distribute the edges from the current node N among the newly created children based on their labels. If no new children are created, we can skip this.

The same process can be recursively applied to the newly created children down the SCC tree, leading to further SCC tree development. This update has a significant impact if the LCA is the master node, as it might cause a merge of SCC trees when the new edge is added which can lead to a decrement in the number of SCCs in the graph G. In cases where the LCA is not the master node, the structure of the SCC tree changes but does not affect the number of SCCs in a graph G. It updates the internal restructuring which can be useful for future updates or changes to the SCC tree. A detailed explanation of the process is illustrated in the flow chart in Fig 3.9, providing a comprehensive view of the steps involved in applying the incremental SCC algorithm.

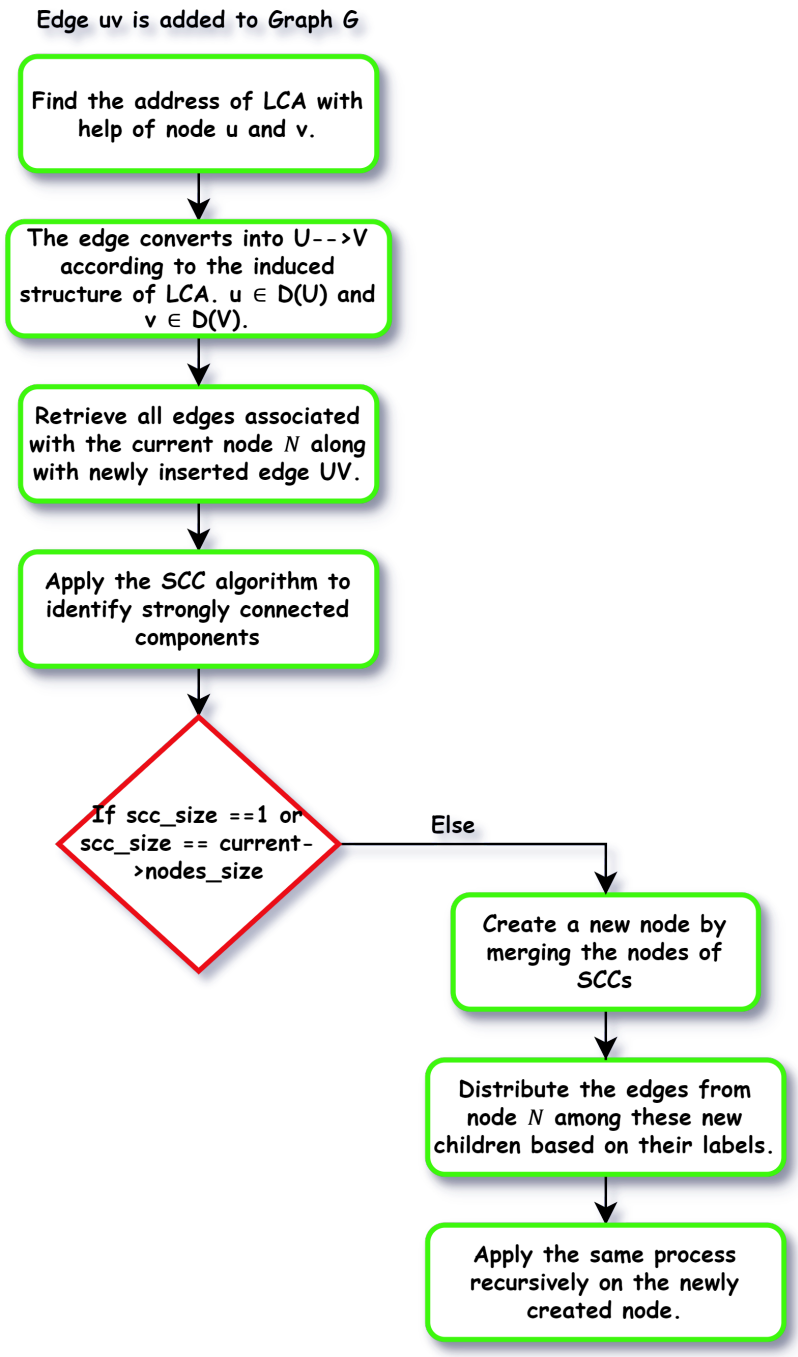


Figure 3.9: Flow Chart for Inserting an Edge into Graph G

3.2.2 Inserting an Edge in Master Node:

Inserting an edge into a master node follows the same process as described earlier, but the metadata used is slightly different from other cases where we apply the incremental algorithm. Specifically, there is no induced vertex in the master node. As a result, the SCC algorithm is applied directly to the graph contained in the

master node, rather than inducing it on a specific vertex. We will explore this process further with an example in Fig 3.10.

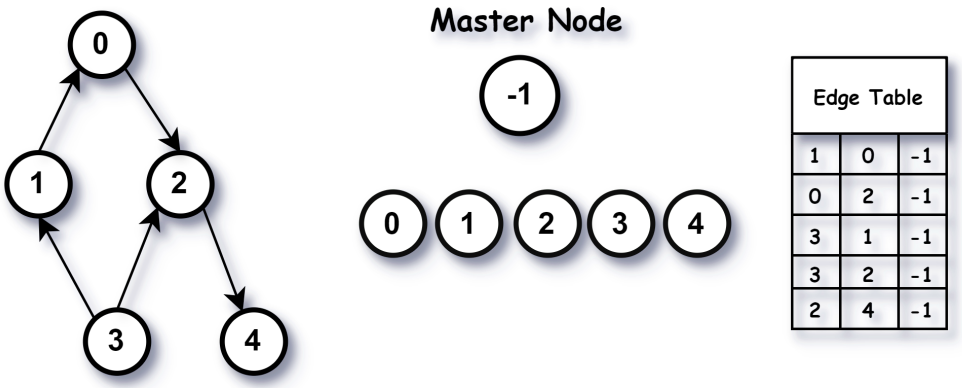


Figure 3.10: Graph G, SCC Tree T(G) and Edge Table

The graph $G = (V, E)$ depicted in Fig 3.10 leads to the formation of 5 singly strongly connected components (SCCs), resulting in five individual SCC trees, each with a single node. And the edge table detailing all the edges in graph G.

Suppose the edge to be inserted is $4 \rightarrow 3$. LCA is the master node. Since there's no induced structure in the master node, the edge remains unchanged as $4 \rightarrow 3$.

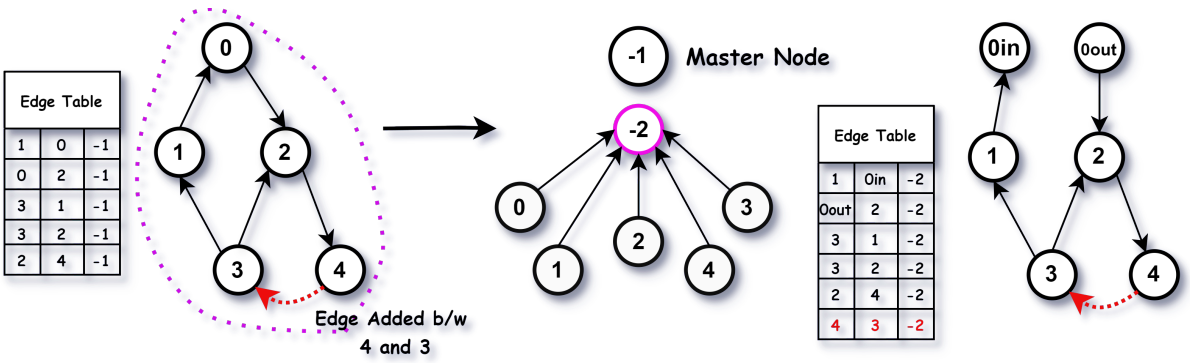


Figure 3.11: **Master Node**: Edge Insertion $\xrightarrow{Algo}$ Tree,Edge Table,SPLIT(-2,0)

To update the master node, we gather all the edges and apply the SCC algorithm. The outcome is a single strongly connected component containing the nodes {0, 1, 2, 3, 4}. As the size of this SCC is five, it satisfies the else condition specified in the flowchart, leading to the creation of a new child node, labeled -2. The induced vertex for this new child is randomly selected; in this example, it's

assumed to be 0. Thus, the graph structure in node -2 becomes SPLIT(G, 0). The next step is to distribute the edges. Since there are no nodes in the master node, every edge in graph G is transferred to node -2, with 0 as its induced vertex. This results in a graph structure that becomes SPLIT(G, 0), with all edges now having the label -2. Applying the same recursive approach to the newly created node -2 leads to further development of the SCC tree.

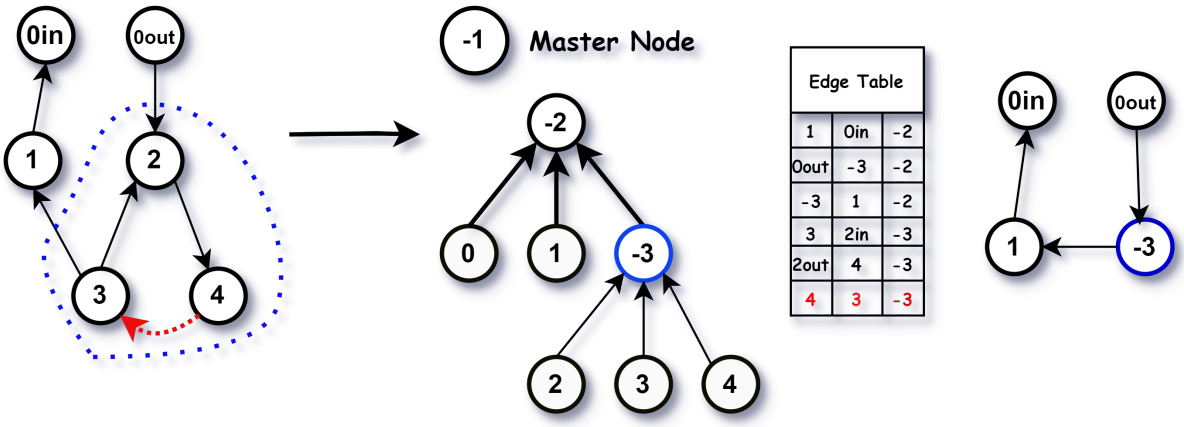


Figure 3.12: **Node -2**: SPLIT($D(-2), 0$) $\xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC($D(-2), 0$)

Applying the SCC algorithm to node -2 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{0_{\text{in}}\}$, $\{0_{\text{out}}\}$, $\{1\}$, and $\{2, 3, 4\}$. This results in the creation of a new child node, labeled -3, with the induced vertex chosen randomly—assumed here to be 2.

According to the **if condition**, we will ignore the single nodes that are $\{0_{\text{in}}\}$, $\{0_{\text{out}}\}$, and $\{1\}$. A new node is created for the SCC $\{2, 3, 4\}$. The next step involves distributing the edges based on the labels of these newly created children.

Given that the induced vertex for node -3 is 2, the initial graph within this node is derived from SPLIT(D(-3), 2). The edges from the current node -2 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in $\{2, 3, 4\}$ is replaced by -3. The edge labels are updated accordingly to reflect the new structure.

The exact distribution of the edges is shown in Fig 3.12, with further details in the diagram beside the edge table. This diagram illustrates the state of node -2 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-2),

0). Applying the same recursive approach to the newly created node -3 down the tree.

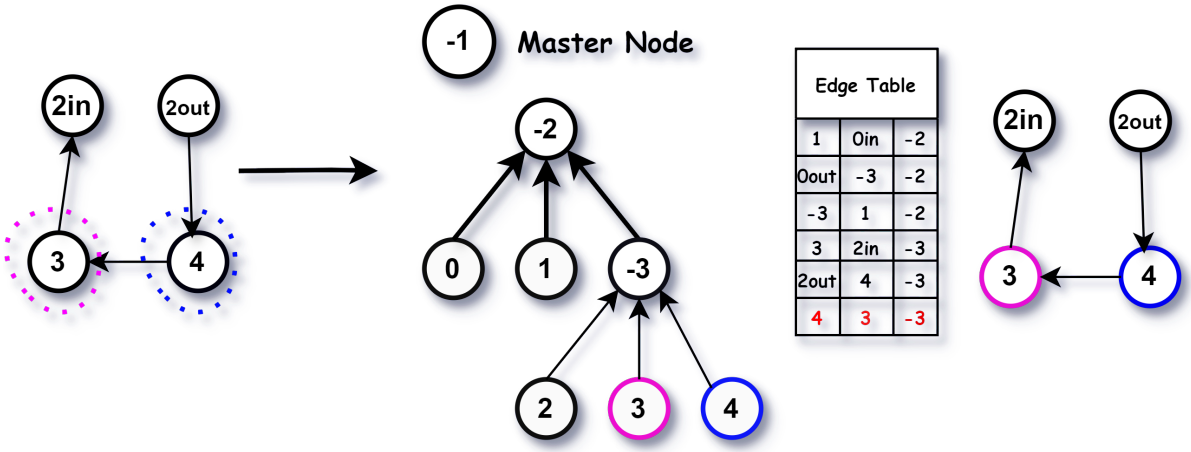


Figure 3.13: **Node -3**: SPLIT($D(-3), 2$) $\xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC($D(-3), 2$)

Applying the SCC algorithm to node -3 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{2_{\text{in}}\}$, $\{2_{\text{out}}\}$, $\{3\}$, and $\{4\}$. This does not lead to the creation of any new nodes because, according to the **if condition**, single-node SCCs are ignored. No edge distribution occurs because no new nodes are created. Thus, the structure within node -3 remains unchanged at the end of the algorithm, i.e., SPLITANDCONDENSE(D(-3), 2).

As you can see, edge insertion in the master node can lead to the merging of various SCCs, reducing the total number of SCCs in the graph.

3.2.3 Inserting an Edge in Non-Leaf Node:

Edge insertion doesn't involve induce vertex of node:

In contrast to edge insertion in the master node, inserting an edge in a non-leaf node within a tree does not change the number of SCCs but alters the internal structure of the tree. Let's examine the impact of edge insertion in a non-leaf node within a tree.

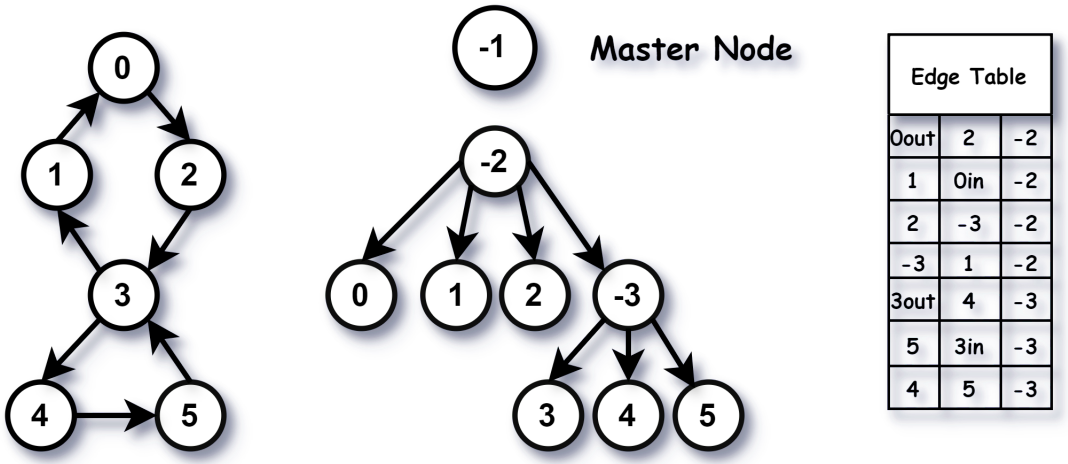


Figure 3.14: Graph G, SCC Tree T(G) and Edge Table

Consider a graph $G = (V, E)$ depicted in Fig 3.14, which forms a single strongly connected component (SCC), resulting in a unique individual SCC tree. The graph structure in node -2 is SPLITANDCONDENSE(D(-2), 0), with 0 as the induced vertex, while the graph structure in node -3 is SPLITANDCONDENSE(D(-3), 2), with 2 as the induced vertex. The edge table details all the edges in graph G.

Suppose the edge to be inserted is $1 \rightarrow 2$. The first step is to find the Lowest Common Ancestor (LCA) in the tree to determine where to insert the edge. The LCA points to node -2. Since neither the first nor the second vertex in the edge is connected to the induced vertex, which is 0, the edge remains unchanged as $1 \rightarrow 2$.

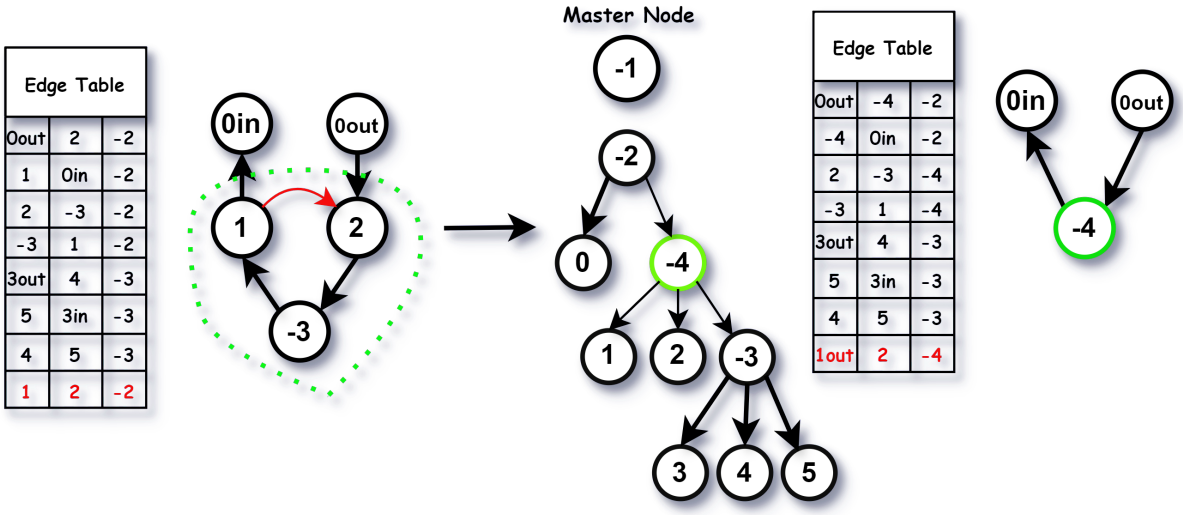


Figure 3.15: **Node -2**: SPLIT$(D(-2), 0) \xrightarrow{\text{Algo}}$ Tree, Edge Table, SAC$(D(-2), 0)$

Since the edge needs to be inserted in node -2, the graph structure in node -2 is SPLITANDCONDENSE(D(-2), 0), as shown in Fig 3.15, with the inserted edge highlighted in red. Applying the SCC algorithm to node -2 results in three strongly connected components (SCCs) with the following nodes: $\{0_{\text{in}}\}$, $\{0_{\text{out}}\}$, and {1, 2, -3}. This leads to the creation of a new child node, labeled -4, with the induced vertex chosen randomly—assumed here to be 1.

According to the **if condition**, single nodes like $\{0_{\text{in}}\}$ and $\{0_{\text{out}}\}$ are ignored. A new node is created for the SCC {1, 2, -3}. The next step involves distributing the edges based on the labels of these newly created children.

Given that the induced vertex for node -3 is 3, the initial graph within this node is derived from SPLIT(D(-3), 3). The edges from the current node -2 are then distributed according to the new labels of the children. Any edge with either the first or second vertex in {1, 2, -3} is assigned to -4. The edge labels are updated accordingly to reflect the new structure.

The exact distribution of the edges is shown in Fig 3.15, with further details in the diagram beside the edge table. This diagram illustrates the state of node -2 after the algorithm has been completed, i.e., SPLITANDCONDENSE(D(-2), 0). Applying the same recursive approach to the newly created node -4.

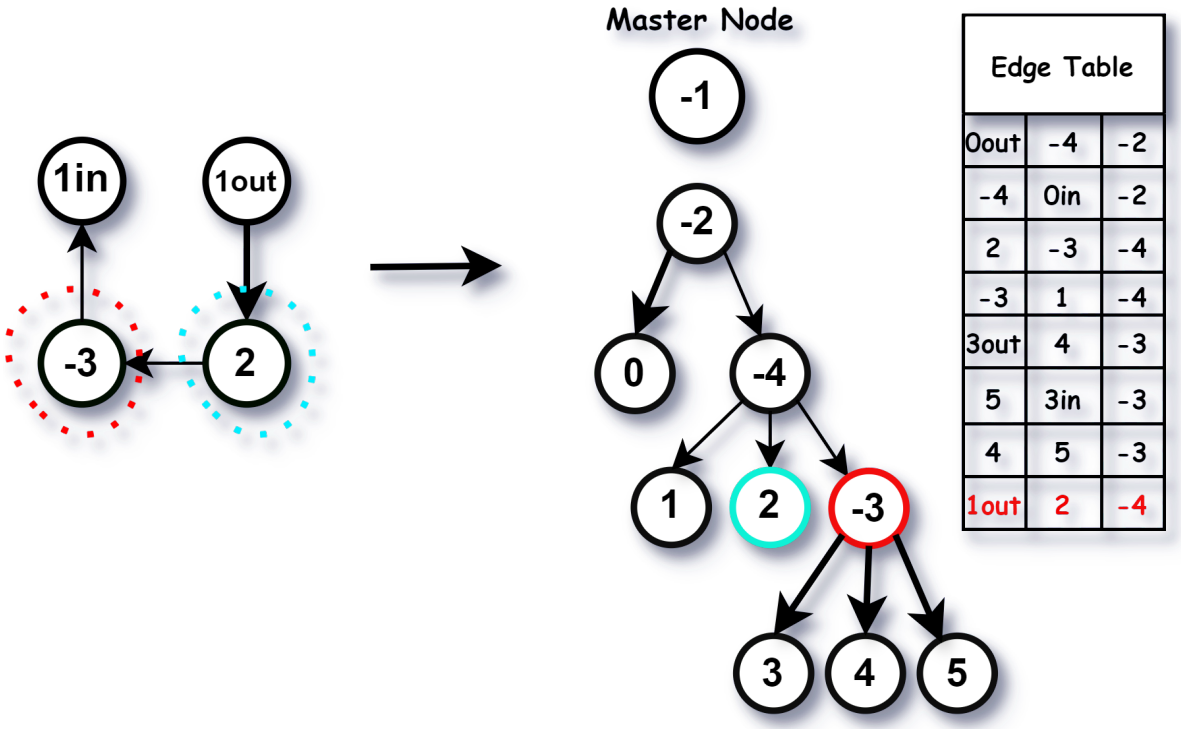


Figure 3.16: **Node -4**: SPLIT($D(-4), 1$) $\xrightarrow{\text{Algo}}$ Tree, Edge Table

Applying the SCC algorithm to node -4 leads to the formation of four strongly connected components (SCCs) with the following nodes: $\{1_{\text{in}}\}$, $\{1_{\text{out}}\}$, {-3}, and {2}. This does not lead to the creation of any new nodes because, according to the **if condition**, single-node SCCs are ignored. No edge distribution occurs because no new nodes are created. Thus, the structure within node -4 remains unchanged at the end of the algorithm, i.e., SPLITANDCONDENSE(D(-4), 1).

Edge insertion in a non-leaf node doesn't alter the number of strongly connected components (SCCs), but it can change the internal structure of the tree. This modification will be beneficial for future operations on the SCC tree.

Edge Insertion involve induce vertex of node:

This scenario involves a non-leaf node, which, although lacking theoretical significance, can offer practical optimization when inserting an edge into a non-leaf node. The unique aspect of this case is that edge insertion involves the induced vertex of the node where the edge is being inserted. This means that no new nodes are

created, allowing for straightforward edge insertion without the need to apply the incremental algorithm.

As the resulting structure does not require further processing, all nodes in this case converge into a single strongly connected component (SCC). This streamlined process can be particularly useful in optimization, where edge insertion in a non-leaf node doesn't necessitate additional restructuring or creation of new nodes. Let's examine this through an example provided below.

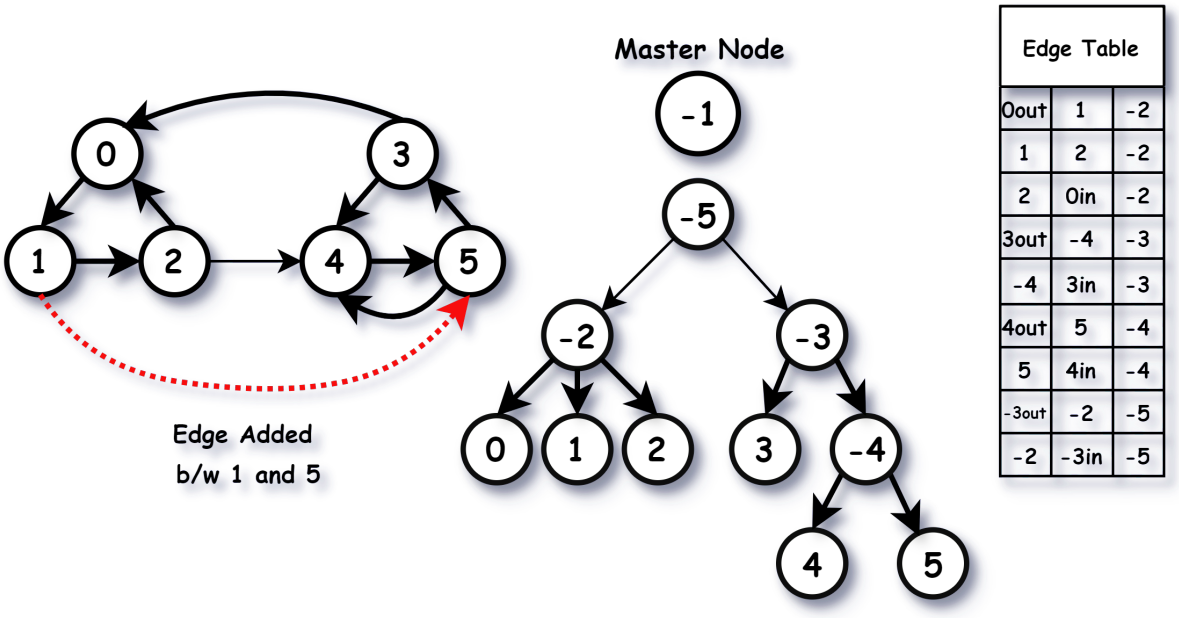


Figure 3.17: Graph G, Edge Insertion in Non-leaf Node, SCC Tree T(G) and Edge Table

Consider a graph $G = (V, E)$ depicted in Fig 3.17, which contains two strongly connected components (SCC), resulting in a single SCC tree. The graph structure in node -2 is SPLITANDCONDENSE(D(-2), 0), with 0 as the induced vertex. The graph structure in node -3 is SPLITANDCONDENSE(D(-3), 3), with 2 as the induced vertex, and it also leads to the formation of a new SCC labeled as -4, with 4 as the induced vertex. The edge table provides details about all the edges in graph G.

Suppose the edge to be inserted is $1 \rightarrow 5$. The first step is to find the Lowest Common Ancestor (LCA) in the tree to determine where to insert the edge. The LCA points to node -5. As described in the second step of the incremental algorithm, when inserting the edge uv into the LCA, the first vertex u should be

mapped to a node labeled A, and the second vertex v should be mapped to a node labeled B such that $P(A)$ and $P(B)$ are both the LCA and $u \in V(D(A))$ and $v \in V(D(B))$.

In this context, the first vertex 1 is mapped to node -2, while the second vertex 5 is mapped to node -3. Since $P(-2)$ and $P(-3)$ both point to node -5, and given that vertex $1 \in V(D(-2))$ and vertex $5 \in V(D(-3))$, the second vertex in node -3 is transformed into -3in, as -3 is the induced vertex for node -5. Consequently, the edge $1 \to 5$ is converted into $(-2 \to -3in)$.

Algorithm 1 Pseudo Code for Edge Insertion (u, v)

```
1: procedure INSERT_EDGE(int u, int v, SCC_Tree*T)
2:     node *LCA = findLCA(u, v);
3:     uv → AB where P(A), P(B) = LCA and u ∈ V(D(A)), v ∈ V(D(B))
4:     E(D(LCA)) ← E(D(LCA)) + (A, B)
5:     Update_SCC_Tree(LCA, T)
6: end procedure
```

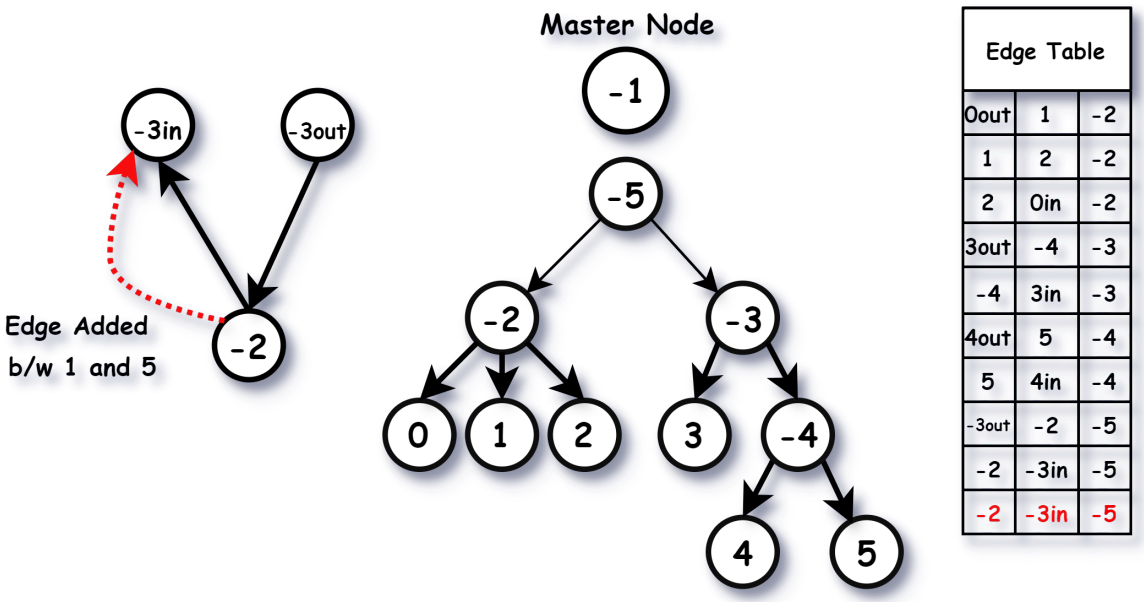


Figure 3.18: **Node -2**: SPLIT$(D(-5), -3)$,Tree, Edge Table, $\xrightarrow{\text{Algo}}$ Remains Same

Since the edge needs to be inserted into node -5, the graph structure within node -5 is SPLITANDCONDENSE(D(-5), -3), as shown in Fig 3.18, with the inserted edge highlighted in red. Applying the SCC algorithm to node -5 results in three strongly connected components (SCCs) consisting of the following nodes:

$\{-3_{\text{in}}\}$, $\{-3_{\text{out}}\}$, and {-2}. This does not create any new nodes because, according to the **if condition**, single-node SCCs like $\{-3_{\text{in}}\}$, $\{-3_{\text{out}}\}$, and {-2} are ignored.

This is typical for all edges added to a non-leaf node with an involved induced vertex. The graph structure has a single source and sink, with all other nodes strongly connected. Adding an edge that involves $\{s_{\text{in}}\}$ and $\{s_{\text{out}}\}$ doesn't create new nodes because the SCC algorithm produces individual SCCs that are already present in the current node. Consequently, there is no need to apply the incremental algorithm in this scenario, as the edge involving the induced vertex will not affect the existing SCC structure and may lead to optimization for practical purposes.

3.3 Updating SCC Tree after Edge Deletion:

This section outlines the process for handling edge deletions in the SCC tree structure. The tree creation algorithm constructs one SCC tree for each strongly connected component (SCC) in the graph. When an edge within an SCC is deleted, the corresponding SCC tree is updated to reflect changes in the graph structure. If the deletion causes the SCC to break, the tree may split into separate parts, potentially increasing the number of SCC trees.

The algorithm for decremental updates to the SCC tree is distinct and more complex compared to the incremental and creation algorithms. This complexity arises from the need to manage the split and potential reorganization of the SCC trees as a result of edge deletions.

We can facilitate query responses for query(u, v) similarly to the incremental approach mentioned in the Edge insertion part. Additionally, the sccCount() query returns the number of nodes in the master node. Since the master node contains all individual SCC trees, and its label is always -1, we can easily retrieve its address and determine the number of nodes it contains. This count provides the response to the sccCount() query. Let's delve into the edge deletion or decremental part of the algorithm.

3.3.1 Procedure:

The procedure for updating an SCC tree during edge deletion involves several key steps. Suppose we are deleting an edge uv from a graph G. The first task is to determine which node contains the edge to be deleted. To do this, we need to find the Lowest Common Ancestor (LCA) of nodes u and v. This is accomplished by identifying the addresses of u and v, and then applying the LCA algorithm to locate the appropriate node.

Once the LCA node is identified, the edge uv to be deleted can be found within that node, but it may not appear as $u \to v$ in its original form. Instead, it could be differently structured or arranged, as it might have been modified during prior operations. The edge may have undergone transformations, especially if it was inserted into a non-leaf node.

The LCA determines where the edge deletion process should focus. However the underlying edge in the LCA could point from u to v, but it might have a different representation due to earlier changes. The specific arrangement of the edge in the LCA may vary, so it's essential to carefully examine the edges in the node LCA to identify and remove the desired edge, ensuring the proper update of the SCC tree structure.

Consider a scenario where we are working on a node N in an SCC tree, and the edge uv to be deleted belongs to it. Assume that the first vertex chosen to split during the construction of the SCC tree is d, which is also the induced vertex of the node N. This means that $I(N) = \text{SPLITANDCONDENSE}(D(N), d)$, and the edge uv is part of $E(D(N))$. Although the edge in the SCC tree might have a different representation, it essentially maps to $u \to v$ in the original graph G.

After deleting the edge uv from $D(N)$, the graph structure within node N changes. Let X denote the graph derived from $D(N)$ by removing uv. The modified structure after the deletion is $SPLITANDCONDENSE(X, d)$. This transformation is achieved simply by deleting the edge uv from the initial graph structure within node N, which is $SPLITANDCONDENSE(D(N), d)$. This yields the final induced structure in node N after edge deletion.

Invoke UNREACHABLE(N) operation generates two lists: R_n and U_n. R_n

contains the list of nodes that are reachable from node N, while U_n contains the list of unreachable nodes from node N. The unreachable nodes are not part of the current SCC and thus are designated as siblings of the current node N. Reachable nodes remain part of the node N. However, in the case of the root node R of an SCC tree, where $P(R) = \emptyset$, any unreachable nodes are considered outside the current SCC tree and form a new, separate SCC tree.

After this, the edge distribution process takes place between the parent node $P(N)$ and node N. Edges that belong to node N are retained within it, while edges that don't are moved to $P(N)$, based on the reachable nodes in node N. Additionally, the edges within $P(N)$ are updated according to the new siblings added to $P(N)$. In this way, some edges are kept within node N, while others are reallocated to $P(N)$.

Algorithm 2 Pseudo Code for Edge Deletion (u, v)

```
1: procedure REMOVE_EDGE(int u, int v, SCC_Tree*T)
2:     node*LCA = findLCA(u, v);
3:     if LCA == Master Node then
4:         E(G) ← E(G) − (u, v)
5:     else
6:         E(D(LCA)) ← E(D(LCA)) − (u, v)
7:         Update_SCC_Tree(LCA, T)
8:     end if
9: end procedure
```

When adjusting for unreachable nodes, careful consideration is required. If an unreachable node of N becomes a sibling of node N, the original induced node of N may also need to be moved as a sibling. In such cases, a new induced node must be chosen for node N, and edges must be rearranged according to the new induced vertex before adjusting the edges of the node N.

Once the edge adjustments are complete, the focus shifts to $P(N)$. This process continues upward towards the root, modifying the structure of the SCC tree during the decremental phase until reaching the root of the SCC tree. This approach can potentially cause SCC trees to break down, leading to an increase in the number of SCCs within the graph G.

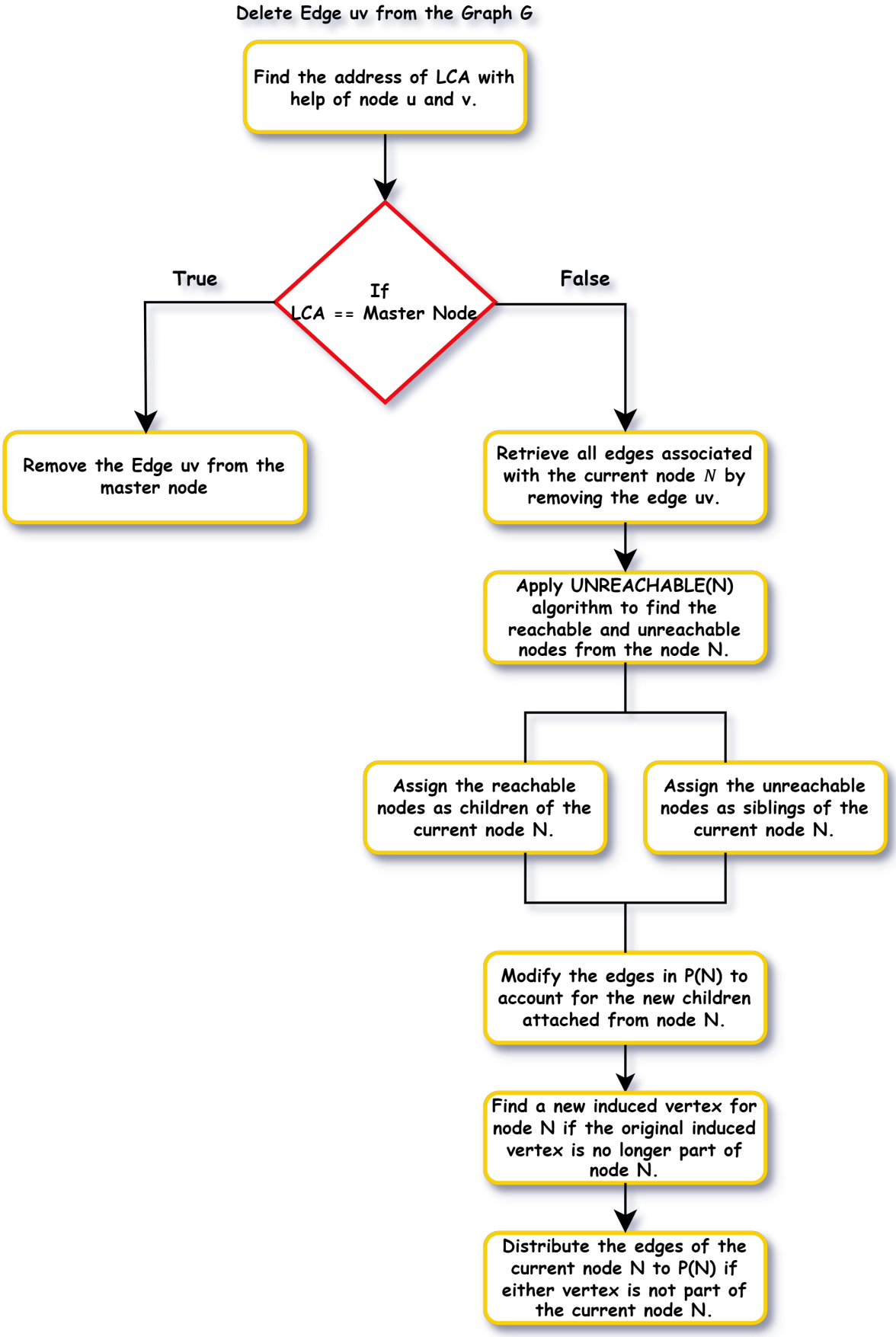


Figure 3.19: Flow Chart for Deletion of an Edge from Graph G

Let's consider an example to explore this topic, starting with a simple case of edge deletion.

3.3.2 Edge Deletion in master node:

Deleting an edge from a Master node is straightforward because it doesn't require recursive updations up the tree. Removing an edge between SCCs does not significantly alter the SCC structure. We can illustrate this with the example below.

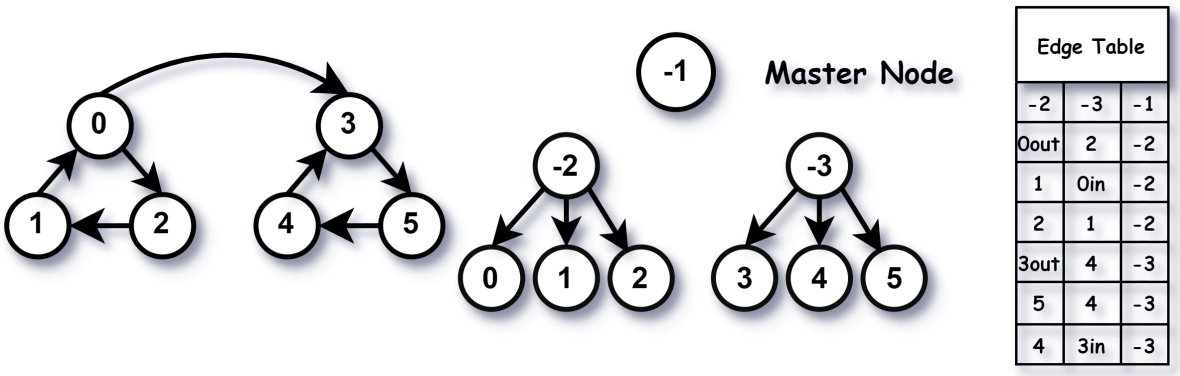


Figure 3.20: Graph G, SCC Tree T(G) and Edge Table

Consider a graph $G = (V, E)$ depicted in Fig 3.20, which consists of two strongly connected components, leading to two different SCC trees. The graph structure in node -2 is SPLITANDCONDENSE(D(-2), 0), with 0 as the induced vertex, while the graph structure in node -3 is SPLITANDCONDENSE(D(-3), 3) with 3 as its induced vertex.

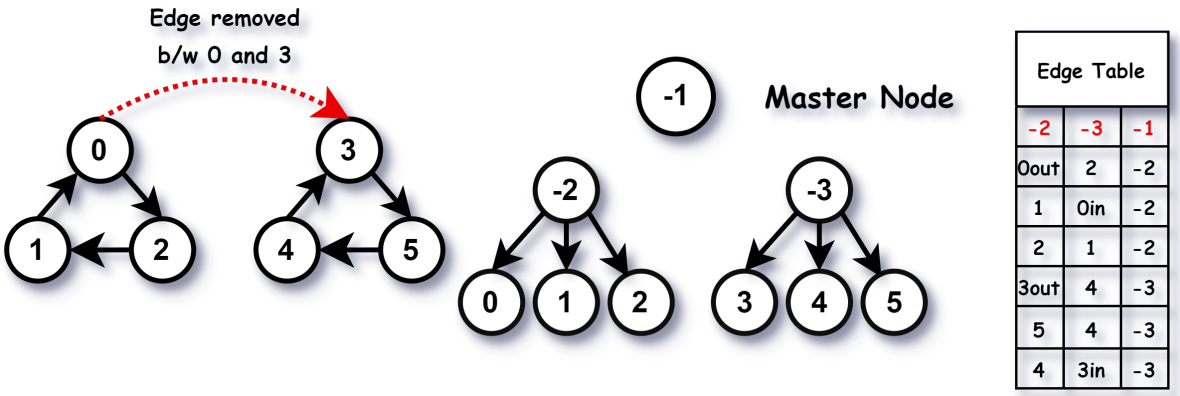


Figure 3.21: Edge Deletion between the SCCs

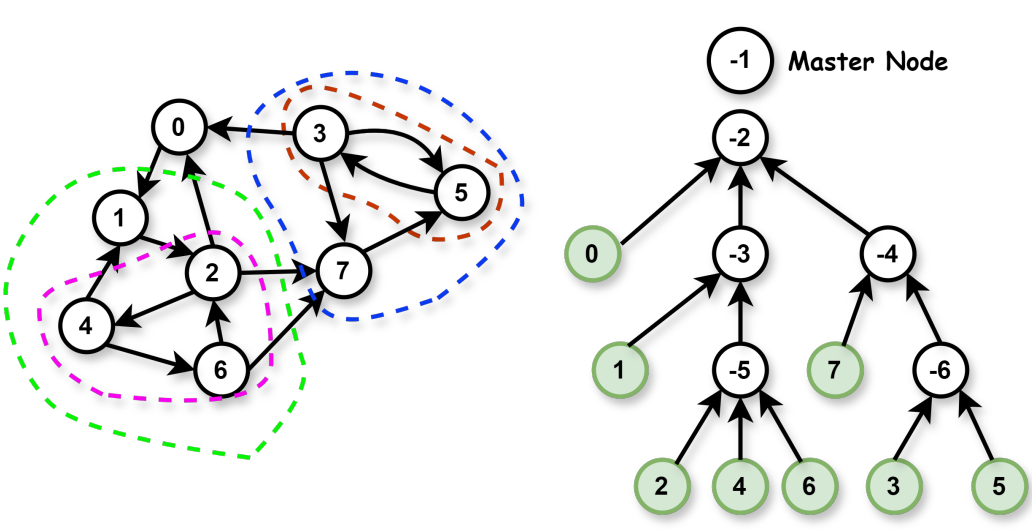
Suppose the edge to be deleted is $0 \rightarrow 3$. The first step is to find the node where this edge exists. By applying the LCA algorithm between nodes 0 and 3, the LCA is identified as node -1 (**Master node**). Since there are two distinct SCC trees, 0 is represented as -2 from the perspective of the Master node because it is part of node -2, while 3 is represented as -3 because it is part of node -3. Thus, the edge $0 \rightarrow 3$ becomes $-2 \rightarrow -3$.

By examining the edges in node -1, we find and delete this edge. However, this deletion does not significantly impact the structure of the SCC tree, since the edge was between two separate SCCs. Removing such an edge does not cause any SCC to split, as they are already distinct. This deletion is illustrated in Fig 3.21, where the removed edge is highlighted in red in the edge table.

This demonstrates that edge removal between SCCs or in the Master node does not affect the SCC tree's structure. Therefore, we can simply remove the edge without any further steps in cases of edge removal from the Master node.

3.3.3 Edge Deletion in non-leaf node:

Deleting an edge from a non-leaf node is significantly more complex than from a master node because it requires recursively updating the SCC tree, starting from the LCA node and moving up to the root of the tree. Let's explore this with an example below.



Edge Table		
0out	-3	-2
-3	0in	-2
-4	0in	-2
-3	-4	-2
-3	-4	-2
1out	-5	-3
-5	1in	-3
2out	4	-5
4	6	-5
6	2in	-5
7out	-6	-4
-6	7in	-4
5out	3	-6
3	5in	-6

Figure 3.22: Graph G,SCC Tree T(G) and Edge Table

Consider a graph $G = (V, E)$ depicted in Fig 3.22, which consists of a single strongly connected component (SCC), leading to an individual SCC tree. The graph structure in node -2 is SPLITANDCONDENSE(D(-2), 0), with 0 as the induced vertex, while the graph structure in node -3 is SPLITANDCONDENSE(D(-3), 1), and in node -4, it's SPLITANDCONDENSE(D(-4), 7).

Nodes -3 and -4 lead to the further formation of new nodes. The structure in node -5 is SPLITANDCONDENSE(D(-5), 2), with 2 as the induced vertex, and the structure in node -6 is SPLITANDCONDENSE(D(-6), 5), with 5 as the induced vertex. The edge table provides details on all the edges in graph G. Assuming the edge being deleted is from $2 \rightarrow 4$. Upon finding the LCA for nodes 2 and 4, it results in node -5, as depicted in Figure 3.23.

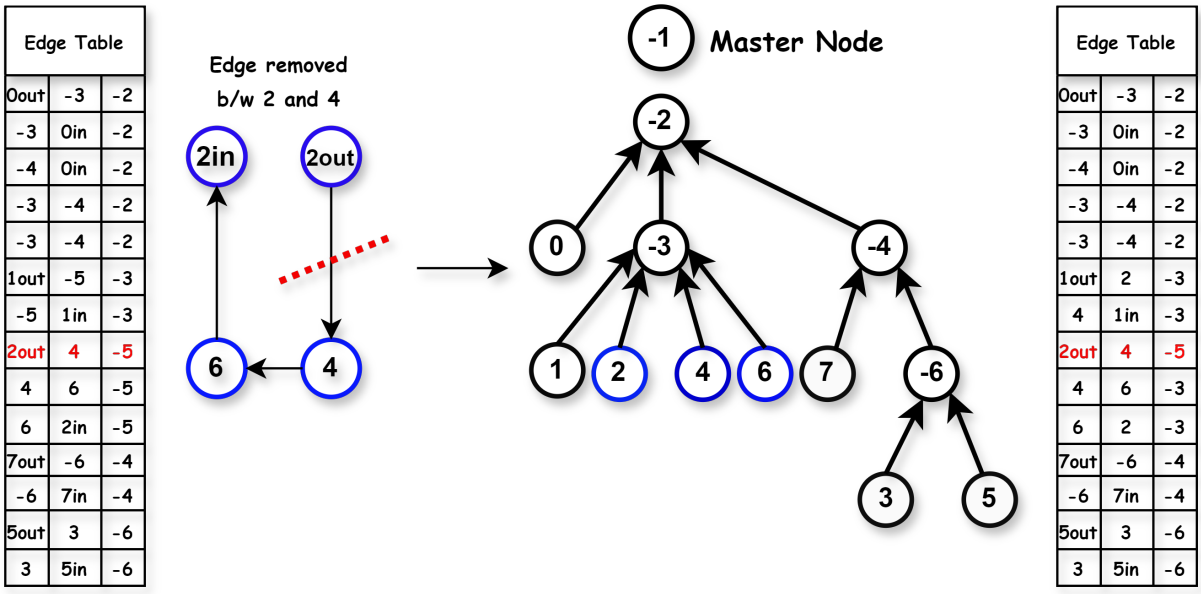


Figure 3.23: **Node -5:** SAC(D(-5),2) $\xrightarrow{\text{Unreachable(-5)}}$ Tree, Edge Table

To proceed further with node -5 after deleting the edge $2_{\text{out}} \rightarrow 4$, we invoke UNREACHABLE(-5), which yields the sets R_n and U_n from the current node. R_n is the empty set ($\emptyset$), while U_n contains $\{2_{\text{in}}, 2_{\text{out}}, 4, 6\}$. By considering 2_{in} and 2_{out} as one (i.e., 2), U_n becomes $\{2, 4, 6\}$.

Given that $R_n = \emptyset$, the nodes in -5 are all unreachable, allowing us to remove node -5 after adjusting its edges. The nodes in U_n ($\{2, 4, 6\}$) become siblings of node -5, with $P(2)$, $P(4)$, and $P(6)$ pointing to $P(-5) = -3$, as shown in Fig 3.23.

To modify the edges in $P(-5)$, consider the edges where the first or second vertex is in node -5. Determine the underlying vertex through which node -5 is connected in the original graph.

Edges in P(-5) = Node -3 are :

i.) Consider the first edge, $(1_{\text{out}} \rightarrow -5)$. The underlying vertex connecting -5 to 1_{out} is vertex 2 in the original graph. As 2 is a child of $P(-5)$, replace -5 with vertex 2.

ii.) Consider the second edge, $(-5 \rightarrow 1_{\text{in}})$. The underlying vertex connecting -5 to 1_{in} is vertex 4 in the original graph. As 4 is a child of $P(-5)$, replace -5 with vertex 4.

Since node -5 is deleted, there's no need to worry about the induced vertex. Let's focus on the edges currently in node -5.

Edges in Node -5 are :

i.) Consider the first edge, $4 \rightarrow 6$. Since nodes 4 and 6 are not part of the current node -5, move this edge to the parent of -5.

ii.) Consider the second edge, $6 \rightarrow 2_{\text{in}}$. As nodes 6 and 2 are no longer part of node -5, move this edge to the parent of -5. The edge becomes $6 \rightarrow 2$, as 1 is the induced vertex in node -3.

The changes in the tree structure and edge modifications are depicted on the right side of Fig 3.23.

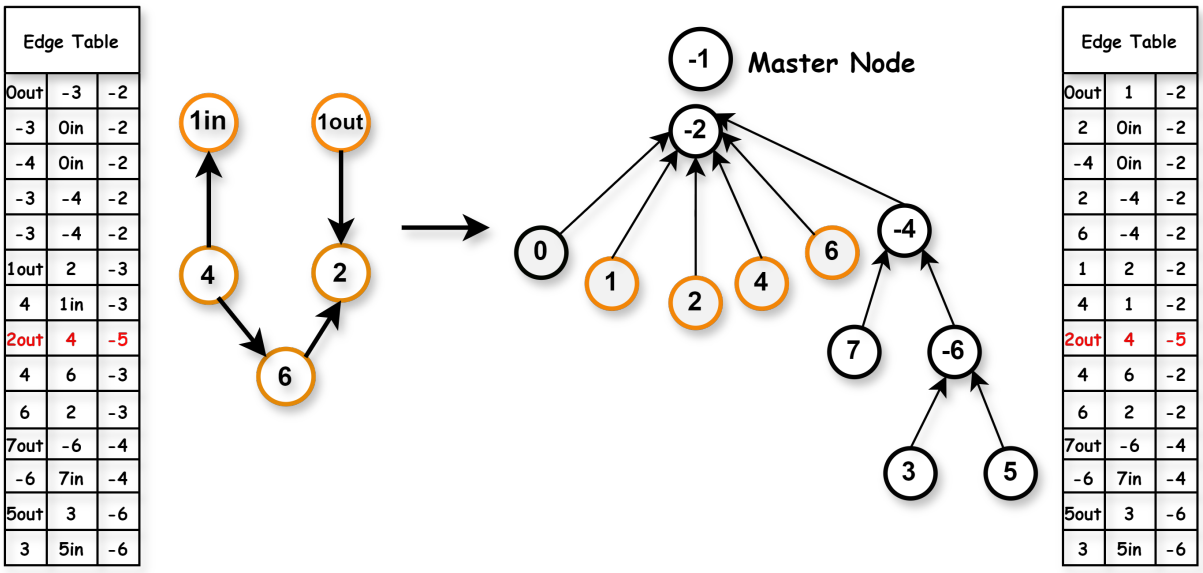


Figure 3.24: **Node -3:** SAC(D(-3),1) $\xrightarrow{\text{Unreachable(-3)}}$ Tree, Edge Table

Now let's proceed to node -3 and apply the same process recursively. Invoke UNREACHABLE(-3), which yields the sets R_n and U_n from the current node. R_n is the empty set ($\emptyset$), while U_n contains $\{1_{\text{in}}, 1_{\text{out}}, 2, 4, 6\}$. By considering 1_{in} and 1_{out} as one (i.e., 1), U_n becomes $\{1\ , 2, 4, 6\}$.

Given that $R_n = \emptyset$, the nodes in -3 are all unreachable, allowing us to remove node -3 after adjusting its edges. The nodes in U_n ($\{1, 2, 4, 6\}$) become siblings of node -3, with $P(1)$, $P(2)$, $P(4)$, and $P(6)$ pointing to $P(-3) = -2$, as shown in Fig 3.24.

To modify the edges in $P(-3)$, consider the edges where the first or second vertex is in node -3. Determine the underlying vertex through which node -3 is connected in the original graph.

Edges in P(-3) = Node -2 are :

i.) Consider the first edge, $(0_{\text{out}} \rightarrow -3)$. The underlying vertex connecting -3 to 0_{out} is vertex 1 in the original graph. As 1 is a child of $P(-3)$, replace -3 with vertex 1.

ii.) Consider the second edge, $(-3 \rightarrow 0_{\text{in}})$. The underlying vertex connecting -3 to 0_{in} is vertex 2 in the original graph. As 2 is a child of $P(-3)$, replace -3 with vertex 2.

iii.) Consider the third edge, $(-3 \to -4)$. The underlying vertex connecting -3 to -4 is the vertex 2 in the original graph. As 2 is a child of $P(-3)$, replace -3 with vertex 2.

iv.) Consider the next edge, $(-3 \to -4)$. The underlying vertex connecting -3 to -4 is the vertex 6 in the original graph. As 6 is a child of $P(-3)$, replace -3 with vertex 6.

Since node -3 is deleted, there's no need to worry about the induced vertex. Let's focus on the edges currently in node -3.

Edges in Node -3 are :

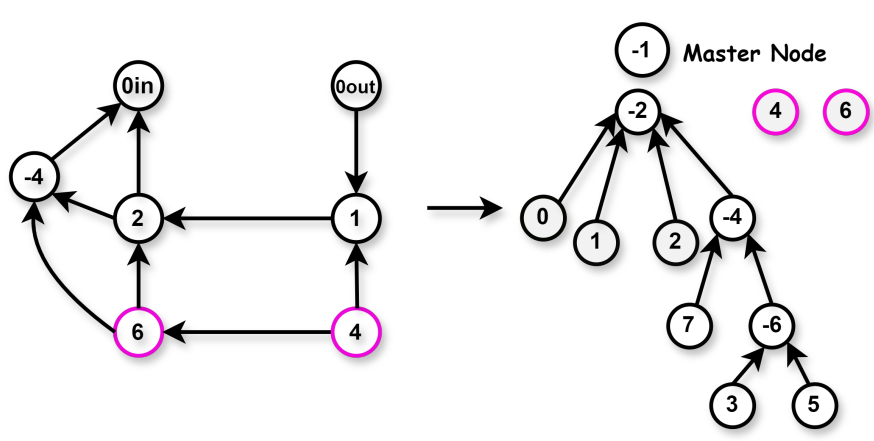
i.) Consider the first edge, $(1_{\text{out}} \to 2)$.. Since nodes 1 and 2 are not part of the current node -3, move this edge to the parent of -2. The edge becomes $1 \to 2$, as 0 is the induced vertex in node -2.

ii.) Consider the second edge, $4 \to 1_{\text{in}}$. As nodes 4 and 1 are no longer part of node -3, move this edge to the parent of -3. The edge becomes $4 \to 1$, as 0 is the induced vertex in node -2.

iii.) Similarly edges $4 \to 6$ and $6 \to 2$. As nodes 2, 4, and 6 are no longer part of node -3, move these edges to the parent of -3.

The changes in the tree structure and edge modifications are depicted on the right side of Fig 3.24.

Edge Table		
0out	1	-2
2	0in	-2
-4	0in	-2
2	-4	-2
6	-4	-2
1	2	-2
4	1	-2
2out	4	-5
4	6	-2
6	2	-2
7out	-6	-4
-6	7in	-4
5out	3	-6
3	5in	-6



Edge Table		
0out	1	-2
2	0in	-2
-4	0in	-2
2	-4	-2
6	-2	-1
1	2	-2
4	-2	-1
2out	4	-5
4	6	-1
6	-2	-1
7out	-6	-4
-6	7in	-4
5out	3	-6
3	5in	-6

Figure 3.25: **Node -2:** SAC(D(-2),0) $\xrightarrow{\text{Unreachable(-2)}}$ Tree, Edge Table

Now let's proceed to node -2 and apply the same process recursively. Invoke UNREACHABLE(-2), which yields the sets R_n and U_n from the current node. R_n contains {0_{in}, 0_{out}, 1 , 2, -4}, while U_n contains { 4, 6}. By considering 0_{in} and 0_{out} as one (i.e., 1), R_n becomes {0, 1 , 2, -4}.

The nodes in U_n ({4, 6}) become siblings of node -2, with both $P(4)$ and $P(6)$ becoming **nullptr**. This creates new SCC trees, each with a single node: one for 4 and one for 6.

Since -2 is the root of the SCC tree, we try to adjust the edges in the **Master node** based on the roots formed from the original graph. However, since there are no edges in the Master node, nothing needs to be modified.

Since the induced vertex for node -2 is 0, and it is among the reachable nodes for node -2, there's no need to find a new induced vertex for node -2. The induced vertex remains unchanged.

Edges in Node -2 are :

Consider the edges that involve the unreachable nodes, that are 4 and 6.

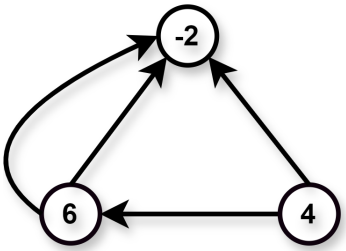
i.) Consider the first edge, $(6 \rightarrow -4)$. Since node 6 is not a part of the current node -2, move this edge to the master node. The edge becomes $6 \rightarrow -2$ because -4 is represented as -2 from the perspective of the Master node, given that -4 is within the SCC tree with -2 as the root.

ii.) Consider the second edge, $4 \rightarrow 1$. As node 4 is no longer part of node -2, move this edge to the master node. The edge becomes $4 \rightarrow -2$ because 1 is represented as -2 from the perspective of the Master node, given that 1 is within the SCC tree with -2 as the root.

ii.) Consider the thrid edge, $6 \rightarrow 2$. As node 6 is no longer part of node -2, move this edge to the master node. The edge becomes $6 \rightarrow -2$ because 2 is represented as -2 from the perspective of the Master node, given that 2 is within the SCC tree with -2 as the root.

iv.) The remaining edge $4 \rightarrow 6$. As nodes 4 and 6 are no longer part of node -2, move these edges to the master node. The edge remains the same as both are not part of node -2.

The changes in the tree structure and edge modifications are depicted on the right side of Fig 3.25. The final structure of the graph in the Master node is shown in Fig 3.26, which does not contain any induced vertex with an Edge table along.



Edge Table		
0out	1	-2
2	0in	-2
-4	0in	-2
2	-4	-2
6	-2	-1
1	2	-2
4	-2	-1
2out	4	-5
4	6	-1
6	-2	-1
7out	-6	-4
-6	7in	-4
5out	3	-6
3	5in	-6

Figure 3.26: **Master Node:** Final Graph and Edge Table after deletion

Decremental algorithms focus on removing edges from a graph. This process is straightforward if the edge removal is between separate SCCs but can be more complex within a non-leaf node. Generally, the decremental approach may maintain the same number or increase the number of SCCs in the graph. These algorithms are essential in situations where efficient updates through deletions are necessary, such as in dynamic graph connectivity.

CHAPTER 4

Performance Analysis

Here, we will explore the practical implementation of trees and the data structures used to support updates following changes in graph edges. Given the dynamic nature of graphs, with frequent edge insertions and deletions, efficient handling of these changes is critical. To simplify this process, I utilized the **Thrust library** Corporation (2021) in CUDA, which facilitates object insertion and deletion with ease.

4.1 Structures Used:

4.1.1 Edge:

```
struct edge
{
    int label; //Label to which Edge belongs to
    int vertices[2]; //Vertices of the Edge in its environment
    int connect[2]; //Vertices to which the edge is connected in
       the original graph
};
```

The structure **Edge** encapsulates information regarding edges, specifically pertaining to the node to which the edge belongs and the vertices comprising the edge within the node's environment. Additionally, the **Connect** array denotes the vertices to which the edge is initially connected in Graph G.

4.1.2 Node:

```
struct Node
{
    int label; //Label of the Node (Unique across the tree)
```

```
    int induce_vertex; //Induced Vertex of the Node
    int depth; //Depth of the Node from its root
    int primary_ancestor; //Primary Root Label
    node*parent; //Parent of the Current Node
    int nodes_size; //Nodes Size
    int*nodes; //Labels of the Nodes present
};
```

The structure **Node** encompasses essential information pertaining to a node within a tree. This information includes a unique label assigned to the node across the tree, the induced vertex, the depth of the node within the tree structure, the primary ancestor indicating the label of the SCC to which the node belongs, the address of its parent node, and the nodes associated with this particular node.

4.1.3 Tree:

```
class Tree
{
    int n_v; int n_e; //Number of Edges and Vertices
    thrust::device_vector<edge*>edges; //Edges in the Graph
    thrust::device_vector<node>map; //Nodes in the Graph
    node*master_node; // Master Node
    SCC_tree(int vertices,int edgesize)
    {
        this->n_v = vertices; this->n_e = edgesize;
        cudaMalloc(&master_node,sizeof(node));
        map = thrust::device_vector<node>(2*vertices);
        edges = thrust::device_vector<edge*>(edgesize);
    }
};
```

The Object **Tree** encapsulates comprehensive information regarding graph G. This includes details such as the count of edges and vertices present in the graph. It stores the address of the master node, which serves as a reference point, and the edges within the graph. These edges evolve as the tree creation progresses. **Nodes** array within the Tree is responsible for storing the nodes created along the progression of the tree structure.

4.2 Results:

In this section, we will discuss the results of experiments conducted on specific graphs characterized by their number of vertices and edges, as detailed in Table 4.1. Timings are measured in seconds. This is a comparison between the standard static parallel SCC in CUDA, which includes the time needed for CSR format creation, and the time taken for updates happening in the SCC tree due to dynamic decremental updates in the edges. Below are the system specifications used for executing the programs:

- **Device**: Tesla V100-PCIE-32GB
- **Compute Capability**: 7.0
- **Global Memory**: 32510MB
- **Number of Multiprocessors**: 80
- **GCC version**: GCC 9.2.0
- **CUDA version**: CUDA 11.2

Graph	Number of Vertices	Number of Edges
email-Eu-core	1005	25571
p2p-Gnutella31	62586	147892
soc-Epinions1	75879	508837
Slashdot0811	77360	905468
Amazon0302	262111	1234877

Table 4.1: Graph Details

Updates	Timings (in s)	
(%)	Static	Dynamic
0.5	0.16115	0.15519
1	0.15893	0.31935
5	0.15263	1.57903
10	0.15095	3.05681
15	0.14173	4.49455

Table 4.2: Graph email-Eu-core

Updates	Timings (in s)	
(%)	Static	Dynamic
0.01	0.94642	0.14713
0.07	0.94923	0.98649
0.2	0.96521	2.54417
0.5	0.96782	5.90315
1	0.0.98342	13.27237

Table 4.3: Graph p2p-Gnutella31

Updates (%)	Timings (in s)	
	Static	Dynamic
0.01	3.36820	3.48786
0.03	3.33978	11.15053
0.05	3.30896	17.25843
0.07	3.28731	24.84426
0.1	3.24806	36.61064

Table 4.4: Graph soc-Epinions1

Updates (%)	Timings (in s)	
	Static	Dynamic
0.01	6.09226	20.99912
0.02	5.97089	41.24804
0.03	5.91531	64.68431
0.04	5.84316	86.34790
0.05	5.80441	109.04274

Table 4.5: Graph Slashdot0811

Updates (%)	Timings (in s)	
	Static	Dynamic
0.01	7.86258	136.60040
0.02	7.91265	264.14550
0.03	7.94855	412.97580
0.04	7.99789	534.95159
0.05	8.25054	674.25152

Table 4.6: Graph Amazon0302

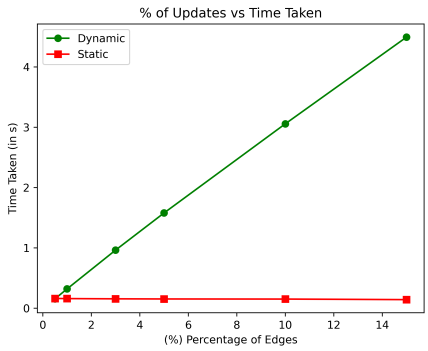


Figure 4.1: email-Eu-core

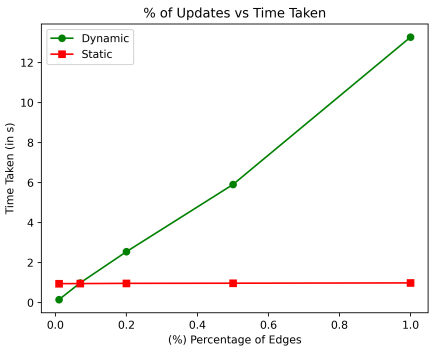


Figure 4.2: p2p-Gnutella31

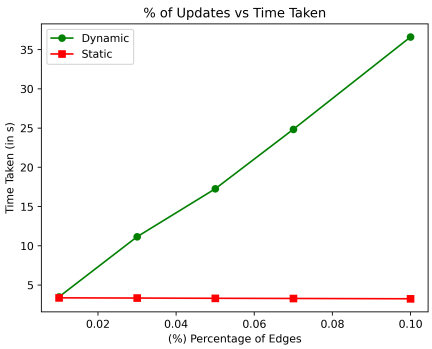


Figure 4.3: soc-Epinions1

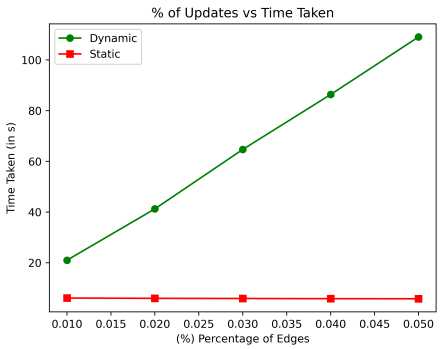


Figure 4.4: Slashdot0811

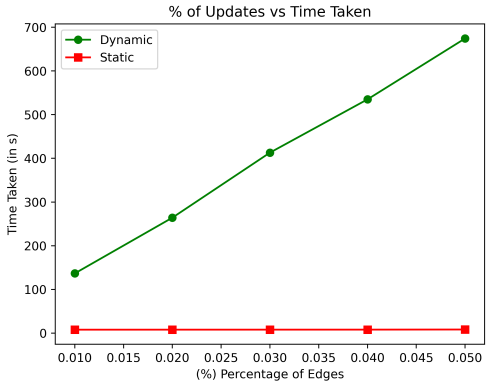


Figure 4.5: Amazon0302

Let's discuss why the dynamic algorithm in CUDA encounters optimization issues. As shown in Fig 4.1,4.2,4.3, during the initial percentage of updates, the time taken is less than or nearly equal to that of the parallel SCC algorithm. However, as we move to larger graphs Fig 4.4,4.5, this is no longer the case, and the parallel SCC algorithm is faster than the dynamic algorithm. Let's understand why this happens.

The primary issue lies in creating the CSR format for the edges associated with the node we are focusing on. In CUDA, to fully leverage parallelism, especially when dealing with graphs, we use a data storage format known as **Compressed Sparse Row (CSR)**, which involves sequential computation. On static graphs, CSR can provide up to three times better performance compared to implementations using Depth First Search **(DFS)** or Breadth First Search **(BFS)**. However, in our dynamic algorithm, which utilizes parallel SCC computation, it becomes necessary to generate the CSR format every time the SCC algorithm is invoked.

Storing edges directly in the CSR format is infeasible due to the algorithm's constraints, which include edges with negative indices. To create a CSR format involving negative vertices, we need to map the negative vertices to non-negative integers, apply the CSR format, and then dereference the vertices at the end. This added complexity makes it difficult to use the CSR format for edge storage and to keep track of all edge distributions and labels. Consequently, the CSR format must be created sequentially each time the SCC algorithm is invoked, adding significant overhead and complexity to the dynamic algorithm.

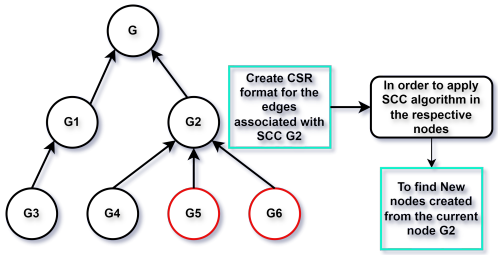


Figure 4.6: Insertion of an Edge

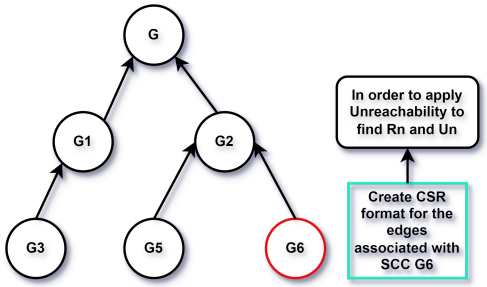


Figure 4.7: Deletion of an Edge

Let's discuss how the CSR format impacts the timings during **edge insertions**. As shown in Figure 4.6, to find the new SCCs created based on the incremental algorithm illustrated in Fig 3.9, we need to retrieve all edges associated with the LCA. To apply the SCC algorithm to these edges, we must first create the CSR format to fully parallelize the SCC computation. Consequently, for each incremental update, the algorithm performs sequential CSR format creation to proceed further down the tree at each level. As the percentage of updates increases, this sequential CSR creation leads to increased time compared to the parallel SCC algorithm.

The same scenario applies to the timings of **edge deletions**. As shown in Figure 3.19, applying the Unreachability algorithm on node N helps determine the reachable and unreachable nodes associated with it. To fully leverage parallelism in CUDA, we apply forward and backward propagation methods to identify common nodes from both processes. To accomplish this, we must create the CSR format for the edges associated with node N. This sequential creation of the CSR format for each update significantly impacts the performance of the dynamic algorithm, causing it to be less efficient than the parallel SCC algorithm in terms of timing.

This issue becomes particularly evident when we need to repeatedly create the CSR format for all nodes between the node LCA and the root of the particular SCC tree.

The CSR issue becomes particularly problematic with large graphs, where the number of edges can reach up to 10^8. The overhead of sequentially creating the CSR format for each update significantly impacts performance, making the dynamic algorithm less efficient than the parallel SCC algorithm. This overhead also affects the time taken to create the SCC tree when invoking the SCC algorithm during SCC tree creation.

4.3 Challenges Faced:

- **Accessing Struct Members**: Since structs are created in GPU memory, accessing their attributes can be challenging. To operate on specific data, you need to copy the relevant attributes to a separate device memory and then copy them back to the host memory for processing. This introduces additional overhead and complexity.

- **Memory Creation in Kernel Functions**: Creating memory on the GPU within kernel functions poses challenges. Initially, the idea was to use **cudaMalloc** directly in the kernel function for node creation and for initializing integer arrays in each node. However, accessing nodes immediately after creation led to issues. Using **malloc()** or **new** within kernel functions works, but can lead to increased execution time due to potential warp divergence. This is because memory allocation within kernel functions might result in threads running different amounts of code, causing performance issues. A better approach is to create memory on the host using **cudaMalloc** and then pass it to the kernel through a loop in the host code.

- **Graph Size and Kernel Launch Limitations**: Working with large graphs (with around 10 million edges) creates challenges in terms of kernel launch. Edge distribution among SCCs can cause warp divergence. Threads allocated to large SCCs can experience high loads when compared to smaller SCCs, leading to kernel functions running indefinitely. This issue is less pronounced in smaller graphs, which are less prone to warp divergence due to a smaller workload per thread.

- **Kernel Launch Errors**: Upon executing a kernel launch in CUDA, the status of the launch is not explicitly stated. This absence of explicit error reporting can result in kernel functionality issues, particularly when dealing with large graphs where data overflow may occur, leading to erroneous outcomes. To address this challenge, tools such as CUDA-GDB or CUDA methods like `cudaLastError()` can be employed to identify errors in kernel launches. These methods prove beneficial, especially in the context of larger

graphs, aiding in the identification and resolution of errors in the kernel launch process.

- **Height of SCC Tree:**

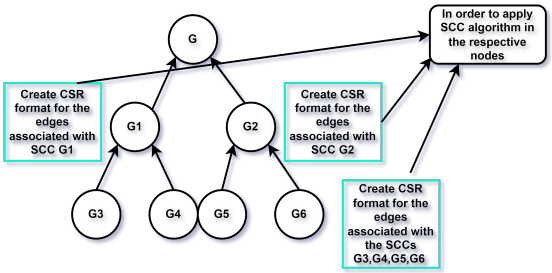


Figure 4.8: SCC Tree Creation

A significant challenge encountered with larger graphs is the height of the SCC tree. As mentioned above, the sequential implementation of the CSR format can drastically decrease the overall performance of dynamic algorithms for large graphs. To mitigate this issue, we impose a height limit on the SCC tree creation. Setting a limit stops the tree from growing beyond a certain level. Consequently, we need to establish this height limit during edge insertions to prevent further tree expansion and proceed with the same algorithm as usual.

4.4 Future Work:

To address the issues associated with the sequential creation of the CSR format, one potential solution is to store the edges of each node N in CSR format during the initial creation of the tree. This approach involves precomputing and storing the CSR format for the edges associated with each node at each level during tree creation. By doing so, we can avoid recreating the CSR format during subsequent operations, leading to improved performance. However, modifying and maintaining the CSR format during edge updates and distributions is complex, especially when dealing with negative vertices, which presents a significant challenge.

Another possible direction is to develop a parallel algorithm that does not rely on the CSR format. While this approach might simplify some aspects of the implementation and avoid the overhead associated with CSR creation, it would likely result in slower performance compared to using CSR in CUDA. This trade-off between complexity and speed needs careful consideration. Future research could focus on developing efficient data structures and algorithms that leverage

parallelism without CSR or finding optimized ways to handle CSR with minimal overhead.

Below are references for the graphs used.

- email-Eu-core - J. Leskovec and Faloutsos (2007)
- p2p-Gnutella31 - Ripeanu *et al.* (2002)
- soc-Epinions1 - Richardson *et al.* (2003)
- Slashdot0811 - K. Lang (2009)
- Amazon0302 - J. Leskovec and Adamic (2007)

CHAPTER 5

Conclusion

In conclusion, this thesis has explored the dynamic nature of graphs and the challenges associated with incremental and decremental updates in the context of SCC algorithms in CUDA. We discussed the inefficiencies arising from the sequential creation of the CSR format during edge insertions and deletions, particularly in handling negative vertices and maintaining edge distributions. The complexities involved in modifying the CSR format have been identified as a significant obstacle to achieving optimal performance.

Moreover, storing the CSR format during the initial tree creation promises to reduce the overhead of sequential CSR creation. However, its effectiveness is hindered by the complexity of updating and distributing edges, particularly with negative vertices or when converting labels to positive indices using a mapping data structure.

Looking ahead, future work should focus on developing alternative data structures that do not rely on the CSR format. By eliminating or optimizing the CSR format, we can potentially achieve faster computation times and reduce complexity. This would pave the way for more efficient SCC algorithms in CUDA and other GPU-based platforms, addressing the current limitations.

In summary, while the CSR format provides a structured way to manage edges in graph algorithms, overcoming its limitations could lead to significant improvements in performance and efficiency. Exploring alternative data structures and optimizations is crucial for advancing the dynamic SCC computation in large-scale graphs.

REFERENCES

1. **Amilkanthwar, M.**, **R. Nasre**, and **S. Devshatwar** (2016). Gpu-centric extensions for parallel strongly connected components computation. *GPGPU 16*, 2–11. URL `https://doi.org/10.1145/2884045.2884048`.

2. **Corporation, N.** (2021). Cuda c++ programming guide and thrust. URL `https://developer.nvidia.com/thrust`.

3. **J. Leskovec, J. K.** and **C. Faloutsos** (2007). Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*, **1**(1). URL `https://snap.stanford.edu/data/email-Eu-core.html`.

4. **J. Leskovec, L. A.** and **B. Adamic** (2007). The dynamics of viral marketing. *ACM Transactions on the Web (ACM TWEB)*, **1**(1). URL `https://snap.stanford.edu/data/amazon0302.html`.

5. **K. Lang, M. M., A. Dasgupta** (2009). Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, **6**(1), 29–123. URL `https://snap.stanford.edu/data/soc-Slashdot0811.html`.

6. **Lacki, J.** (2013). Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, **9**(3), 0–15. URL `https://doi.org/10.1145/2483699.2483707`.

7. **Monika Henzinger, K. H.** and **C. Schulz** (2020). Fully dynamic single-source reachability. *Proceedings of the Symposium on Algorithm Engineering and Experiments*, **24**, 0–7. URL `https://arxiv.org/abs/1905.01216`.

8. **Richardson, M.**, **R. Agrawal**, and **P. Domingos** (2003). Trust management for the semantic web. iswc. URL `https://snap.stanford.edu/data/soc-Epinions1.html`.

9. **Ripeanu, M.**, **I. Foster**, and **A. Iamnitchi** (2002). Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. ieee internet computing journal. URL `https://snap.stanford.edu/data/p2p-Gnutella31.html`.