Dynamic Parallel Maintaince of Strongly Connected Components in MPI

A Report submitted in partial fulfillment for the Degree of

Bachelor of Technology
in
Computer Science and Engineering

by

Soham Tripathy*

Supervised by Professor. Rupesh Nasre

Submitted May 16, 2024

*Student ID: CS20B073, Email: cs20b073@smail.iitm.ac.in

Abstract

This report explores the criticality of maintaining strongly connected components (SCCs) amidst dynamic changes in graph structures, emphasizing the efficiency of preserving SCCs during edge deletions and additions. Understanding the significance of SCCs in various applications, it becomes very challenging to recompute SCCs in response to changes in the graph structure. This requires efficient algorithms and optimization techniques to minimize the computational overhead and ensure real-time updates. Building upon prior research, we worked on extending an optimal algorithm tailored for this purpose. The algorithm maintains the structure of strongly connnected components over sequence of edge updates in $O(mn)$ total time, with constant query time. This is an advancement over the best currently known deterministic algorithms, which run in $O(m^2)$ or $O(n^3)$ total time. Each facet of the algorithm deals with various cases of graph updates, which is elucidated with illustrative examples and accompanying pseudo code.

Leveraging MPI for distributed parallel computation, we implement our algorithm, addressing encountered challenges and intricacies of the implementation process to facilitate comprehension of the code. Detailed explanations of the design implementation, including the SCC tree construction, decremental and incremental maintenance, are provided. Rigorous testing on standard and specialized graphs, augmented with dynamic updates, is conducted, juxtaposing the performance against static SCC algorithms. Results that are depicted through graphs and tables, are analyzed in the conclusion, offering insights and proposing avenues for future enhancements.

Table of Contents

Acronyms

MPI: Message Passing Interface

SCC: Strongly Connected Components

STN: SCC-Tree Node

DFS: Depth First Search

BFS: Breadth First Search

List of Figures

List of Tables

Acknowledgements

I would like to express my heartfelt gratitude to *Professor Rupesh Nasre*, Department of Computer Science and Engineering, Indian Institute of Technology, Madras for his invaluable guidance, unwavering support, and insightful feedback throughout the duration of this project. His expertise and mentorship have been instrumental in shaping the direction of this work.

Additionally, I extend my sincere appreciation to the M.Tech students, *Barenya Kumar Nandy* and *Anurag Sao*, whose valuable insights, references, and constant assistance have greatly enriched this endeavor.

I would also like to thank the Indian Institute of Technology, Madras, for providing the necessary resources and infrastructure for the successful completion of this project.

Soham Tripathy

Main Content

1 Introduction

Strongly connected components (SCCs) play a crucial role in graph theory, offering profound insights into the structure and connectivity of directed graphs. They are essential for understanding various real-world phenomena, ranging from social networks to computer algorithms.

- **Social Networks**: In social networks like Facebook or Twitter, individuals form groups based on common interests, affiliations, or interactions. SCCs can help identify these tightly-knit communities within the network. For example, in a political analysis, identifying SCCs can reveal clusters of like-minded individuals or factions within a larger social network.
- **Transportation Networks**: In transportation networks, such as road or railway systems, SCCs can represent regions where travel between any pair of locations is possible without leaving the component. This is crucial for optimizing routes, identifying traffic bottlenecks, and designing efficient public transportation systems.
- **Internet and Web Graphs:** The internet can be represented as a directed graph, where web pages are nodes and hyperlinks are edges. Identifying SCCs in this graph can reveal clusters of interconnected websites that share similar content or themes. This information is valuable for search engines to improve the relevance of search results and for analyzing the structure of the web.
- **Compiler Design**: In compiler construction, analyzing the control flow graph of a program involves finding SCCs. This helps in optimizing code, identifying loops, and performing various program analyses, such as data-flow analysis and reaching definitions analysis.
- **Database Management**: In database systems, transactions between different parts of a database can be represented as a graph, where transactions are nodes and dependencies between them are edges. Detecting SCCs in these graphs helps in identifying sets of transactions that must be executed together or can be executed concurrently, improving database performance and transaction management.

In each of these cases, identifying strongly connected components provides valuable insights into the underlying structure and connectivity of the system, facilitating optimization, analysis, and decision-making processes. Thus, developing efficient algorithms to find SCCs in directed graphs is a fundamental problem in graph theory and computer science.

There are various static linear time alogithms to find SCCs in a directed graph, such as Kosaraju's algorithm [2], Tarjan's algorithm [3], that are partical and efficient for large-scale applications. However, these algorithms are designed for static graphs, where the graph structure does not change over time. The problem of finding SCCs in a dynamic graph, where edges can be inserted or deleted, is more challenging and has received significant attention in recent years. The static algorithms are re-run on the entire graph to find the SCCs after each set of updates, which is computationally expensive and inefficient for large-scale graphs.

The goal of this project is to understand, formulate, and implement a dynamic parallel algorithm to find and maintain SCCs in a directed graph, where edges can be inserted or deleted dynamically. We aim to develop an efficient parallel algorithm that can handle large-scale graphs and leverage the computational power of modern multi-core processors and parallel computing platforms.

2 Literature Review

In this section, we review various algorithms and techniques that have been proposed to find strongly connected componenets (SCCs) in a graph. We also discuss the work that has been done to indentify SCCs through parallel processing and maintaining SCCs in response to dynamic changes in the graph structure.

2.1 Static SCC Algorithms

The problem of finding SCCs in a graph is a well-studied problem in the field of graph theory. In this regards many static alogrithms have been proposed, like the Kosaraju's algorithm [4], Tarjan's algorithm [3], and the path-based algorithm [5]. These algorithms achieve a time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Kosaraju's algorithm is a two-pass algorithm that traverses the graph twice to find SCCs. In the first pass, it performs a depth-first search (DFS) to create a stack of vertices in order of their finishing times. In the second pass, it reverses the graph and performs DFS again, this time starting from vertices popped from the stack, thus identifying SCCs.

Tarjan's algorithm is a single-pass algorithm based on DFS and uses a depth-first search traversal to identify SCCs. It maintains a stack to keep track of vertices in the current SCC being explored and assigns a unique identifier (low-link value) to each vertex. By maintaining a stack and tracking low-link values, Tarjan's algorithm efficiently identifies SCCs.

Both algorithms have the same time complexity, but their implementations may have different constant factors, making one algorithm more suitable than the other depending on the specific characteristics of the graph being processed. They have linear time complexity in terms of the number of vertices and edges in the graph, making them practical even in large-scale applications.

With the advent of parallel and distributed computing, researchers have explored parallelizing these algorithms to improve their performance on large graphs. Parallel implementations of these algorithms have been proposed using shared-memory parallelism (e.g., OpenMP) and distributed-memory parallelism (e.g., MPI). These parallel implementations aim to exploit the inherent parallelism in the graph traversal process to achieve better performance.

2.2 Parallel SCC Algorithms

For large-scale graphs, parallel algorithms are essential to achieve efficient computation of SCCs. Standard static algorithms like Kosaraju's and Tarjan's algorithms are not inherently parallelizable, as they rely significantly on DFS. Unfortunately, DFS is inherently sequential, making it challenging to parallelize these algorithms effectively. This is observered in a restricted version of lex-BFS (Lexicographic Breadth-First Search), which is $\mathcal{P}$-Complete [6]. However, Aggarwal and Anderson, proposed random NC-alorithm for DFS in [7, 1988], which was later improved to form a parallel DFS in general directed graphs in [8, 1990].

The expected time of this latter algorithm is $O(\log^7 n)$, and it requires an impractical $n^{2.376}$ processors. Improvements were made to this algorithm in [9, 1993], which runs the algorithm in $O(\log^2 n)$ time with $O(n/\log n)$ processors for an n-vertex connected undirected planar graph. Since, deterministic parallelism of DFS was difficult, various parallel algorithms for finding SCCs were proposed, which avoid the use of dept-first search.

Gazit and Miller devised an NC algorithm for SCC in [10], which is based upon matrix-matrix multiplication. This algorithm was improved by Cole and Vishkin [11], but still requires $n^{2.376}$ processors and $O(\log^2 n)$ time. Lisa K. Fleischer, Bruce Hendrickson, and Ali Pinar in their paper [12], proposed a divide-and-conquer algorithm for strongly connected components based on dept-first search, which has significantly greater potential for parallelization. For a graph with n vertices in which degrees are bounded by a constant, they show the expected serial running time of our algorithm to be O(n log n).

The forward-backward (FW-BW) method [12] and Orzan's coloring method [13] are two SCC detection algorithms that are amenable to both shared-memory and distributed-memory implementation. These methods use very different subroutines and were proposed in different contexts, FW-BW for graphs arising in scientific computing and coloring in the context of formal verification tool design. Developments on this algorithms can be found in the paper [14], by George M. Slota, Sivasankaran Rajamanickam and Kamesh Madduri.

The paper by Sudharshan S. [15] provides a detailed implementation of an efficient parallel algorithm for finding SCCs in large-scale graphs on distributed-memory systems. It shows how to implement the FW-BW method using the MPI library and provides insights into the challenges and optimizations required to achieve good performance on large-scale graphs. The results show performance speedups up to 3.1x and memory reductions up to 2.6x with respect to the serial implementations.

2.3 Dynamic SCC Algorithms

The identification of SCCs in a dynamic graph is a challenging problem due to the changing nature of the graph structure. The graph can undergo edge insertions and deletions, which can affect the SCCs in the graph. The above mentioned algorithms are designed for static graphs and are not directly applicable to dynamic graphs, thereby necessitating the development of new algorithms for dynamic SCC detection.

The problem of dynamically maintaining SCCs has been given considerable attention only in the recent years, and hence they are addressed less frequently than the static problem. One approach to decremental maintenance is to adapt existing algorithms for computing SCCs to handle edge deletions efficiently. Frigioni et al. proposed an algorithm in [16, 2001], which, while providing a solution, had a worst-case time complexity of $O(m^2)$, where m is the number of edges in the graph. This complexity is comparable to recomputing SCCs from scratch after each update, rendering it inefficient for large graphs.

However, there have been advancements in this area. Roditty and Zwick introduced a Las Vegas algorithm in [17] for decremental maintenance of SCCs. A Las Vegas algorithm is a probabilistic algorithm that always produces the correct result but may have varying running times. In this case, the algorithm maintains strongly connected components with an expected time complexity of $O(mn)$ for any sequence of edge removals.

Further improvements in deterministic maintainance of SCCs were made in [18, 2013], by Jakub Lacki, who proposed a data structure called the SCC-Tree that allows for efficient decremental maintenance of SCCs. The SCC-Tree is a compact representation of the SCCs in the graph that can be updated efficiently in response to edge deletions. The SCC-Tree data structure allows for efficient queries about the SCCs in the graph and can be used to maintain SCCs under dynamic updates.

The preprocessing time of the graph to build the SCC-tree takes $O(nm)$ time and this data structure maintains the SCCs of the graph in $O(m\delta)$ total time, where δ is the depth of the SCC-tree (worst-case $\delta = O(n)$). This preprocessing time of the graph is improved in [19, 2013], by Liam Roditty, from $O(nm)$ to $O(m \log n)$. We take inspiration from [18], and extend it to maintain SCCs under both decremental and incremental updates. The implementation and evaluation of this extended algorithm are discussed in the following sections.

3 Theoretical Methodology

In this part, we elaborate on and enhance the algorithm proposed in [18] for maintaining strongly connected components (SCCs) under a sequence of updates. The algorithm is designed to accommodate n vertices and m edges as input, proficiently manages the following operations:

- QUERY(u, v): Verifies whether both u and v belong to the same SCC.
- DELETE(u, v): Removes the edge from u to v.
- ADD(u, v): Introduces the edge from u to v.

This algorithm achieves its objectives through the creation and continuous maintenance of a specialized data structure known as the SCC Tree, which is further explained in the subsequent section. Additionally, it employs an SCC mapping array to facilitate queries in constant time *O(1)*.

The algorithm initiates with an initialization stage, wherein it constructs the proposed SCC tree and populates the SCC mapping array based on the identified strongly connected components within the graph.

Subsequently, the process of constructing and maintaining these pivotal data structures under the delete and add updates is elaborated in the following sections, accompanied by their respective pseudo codes.

3.1 Data Structures

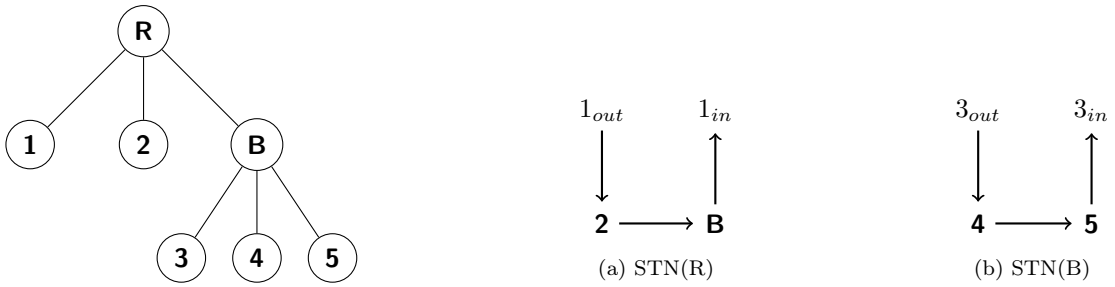
In this section, we delve into a comprehensive exploration of the specialized data structures crucial to the functionality and efficiency of the algorithm. Through this detailed examination, we aim to provide clarity and insight into the design principles, operational mechanisms, and computational complexities of these structures.

3.1.1 SCC Tree

The SCC tree serves as a vital component in maintaining the internal connectivity of vertices within the strongly connected components of graph G. For each SCC identified in the graph, a corresponding SCC tree is constructed.

The node in the SCC tree corresponding to the label R encapsulates a tuple that can be represented as $\text{STN}(R) = (V, E)$, where V represents set of vertices and E represents set of edges connecting them. We can thus informally infer that STN(R) holds some graph-like structure G. We will refer to the vertex set of the SCC tree node labeled R as $\text{STN}(R).V$ and the edge set as $\text{STN}(R).E$. This encapsulation enables the SCC tree to maintain the connectivity of the vertices that are a part of the SCC labeled R, also facilitating efficient traversal and update operations.

Each vertex $v \in V$ in the STN of R is a label uniquely associated with an SCC tree. This tree is a subtree of the SCC tree represented by the label R. The SCC tree for a graph containing one vertex v has only a single STN represented as $\text{STN}(v) = (\{v\}, \emptyset)$.



(a) STN(R) (b) STN(B)

Figure 1: Example SCC Tree and its corresponding STNs

The SCC tree shown in Fig.1, can be represented by a collection of STNs as follows:

- $\text{STN}(R) = (\{1, 2, B\}, \{(1, 2), (2, B), (B, 1)\})$
- $\text{STN}(B) = (\{3, 4, 5\}, \{(3, 4), (4, 5), (5, 3)\})$
- $\text{STN}(v) = (\{v\}, \emptyset)\, \forall v \in \{1, 2, 3, 4, 5\}$

The SCC tree represented by label B contains the STNs of $\{B, 3, 4, 5\}$, which form a subtree of the SCC tree represented by label R, similarly, the SCC tree represented by labels $\{1, 2\}$ also form R's subtree.

3.1.2 SCC Mapping Array

In graph G, each vertex v is inherently associated with a strongly connected component (SCC), denoted by its corresponding SCC label. The SCC mapping array effectively captures this relationship between vertices and their respective SCC labels.

Suppose vertex v in graph G belongs to the strongly connected component labeled as R. In that case, we express this association using the SCC mapping array notation as $\text{SM}_G(v) = R$. This signifies that vertex v is mapped to the SCC labeled as R within the context of the graph G.

3.2 Definitions

In this section, we establish key definitions that form the foundation of the algorithms presented subsequently. These definitions are instrumental in understanding the intricacies of the algorithms and their associated data structures.

3.2.1 FindScc(G)

For a given graph G, let $V(G)$ represent the set of vertices of graph G. We consider k set of vertices as $U_i \forall i \in [1, k]$ such that it satisfies that following conditions:

- $\bigcup_{i=1}^{i=k} U_i = V(G)$
- $U_i \cap U_j = \emptyset \ \forall i, j \in [1, k] | i \neq j$
- For each U_i, we say $\forall v, u \in U_i | v \neq u$, $\text{Query}(u, v) = true$.
- For any U_i and U_j such that $i \neq j$, $\forall v \in U_i$ and $\forall u \in U_j$, $\text{Query}(u, v) = false$.

We define the function FindScc(G) as the process of identifying the set of vertices U_i that satisfies the above conditions. The function FindScc(G) is instrumental in identifying the strongly connected components within the graph G. We can do this by using any standard linear time algorithm, some of which are mentioned in [4], [2], and [3].

3.2.2 Condense(G)

We define Condense(G) as condensing the graph G into a new graph G', where each vertex in G' represents a strongly connected component in G, *ie.* $V(G') = \{\text{SM}_G(v) | v \in V(G)\}$. The edges in G' are such that if there is an edge from u to v in G, and u and v belong to different strongly connected components in G, then there is an edge from the strongly connected component containing u to the strongly connected component containing v in G'. Therefore, a edge $(u, v) \in E(G) | \text{Query}(u, v) = false$ correponds to an edge $(\text{SM}_G(u), \text{SM}_G(v)) \in E(G')$.

Algorithm 1: Condense(G)

```
Data: G
Result: G'
U = FindScc(G);
for U_i ∈ U do
    L = new label;
    for v ∈ U_i do
        SM_G(v) = L;
    end
    V(G') = V(G') ∪ L
end
for (u, v) ∈ E(G) do
    if Query(u, v) = false then
        E(G') = E(G') ∪ (SM_G(u), SM_G(v))
    end
end
```

3.2.3 Split(G,d)

Consider any graph G with a set of vertices $V(G)$ and a set of edges $E(G)$ such that $|\text{FindScc}(G)| = 1$. Let $\exists d \in V(G)$ that can be split into two vertices d_{in} and d_{out} producing a new graph G' such that any edge incident on d is now incident on d_{in} and any edge originating from d is now originating from d_{out}.

Algorithm 2: Split(G,d)

```
Data: G, d
Result: G'
V(G') = (V(G) \ {d}) ∪ {d_in, d_out};
for (u, v) ∈ E(G) do
    if v = d then
        E(G') = E(G') ∪ (u, d_in)
    end
    else if u = d then
        E(G') = E(G') ∪ (d_out, v)
    end
    else
        E(G') = E(G') ∪ (u, v)
    end
end
```

3.2.4 Merge(G, s, t, d)

Let there be a graph G such that it has a source vertex s and a sink vertex t. We define Merge(G,s,t, d) as merging the vertices s and t into vertex d in the graph G to produce a new graph G'. This operation is the inverse of the Split(G, d) operation.

Algorithm 3: Merge(G,s,t,d)

```
Data: G,s,t,d
Result: G'
V(G') = V(G) \ {t, s} ∪ {d};
for (u, v) ∈ E(G) do
    if v = t then
        E(G') = E(G') ∪ (u, d)
    end
    else if u = s then
        E(G') = E(G') ∪ (d, v)
    end
    else
        E(G') = E(G') ∪ (u, v)
    end
end
```

3.2.5 Unreachable(G,s,t)

Let there be a graph G such that it has a source vertex s and a sink vertex t. A vertex v is said to reachable if there exists a path from s to v and a path from v to t. We define the function Unreachable(G,s,t) as the process of identifying the set of vertices that are unreachable in G.

3.3 Constructing SCC Tree

We will look at the construction of the SCC tree for the graph shown in Fig.2a. We start by finding all the SCCs of the graph and then construct the SCC-Tree for each SCC in it. In the process of finding all the

Algorithm 4: UNREACHABLE(G,s,t)

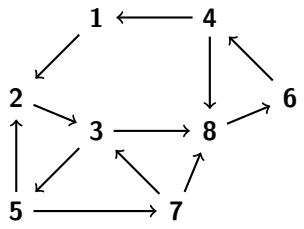
Data: G, s, t
Result: U
$U = \emptyset, R = \emptyset$;
$G' =$ MERGE(G, s, t, d);
$S =$ FINDSCC(G');
for $U_i \in S$ **do**
  if $s \in U_i$ **then**
    $R = U_i$
  end
end
$U = V(G) \setminus R$;

SCCs, we would also fill the SCC mapping array. A special tree node STN(M), called the master node would preserve the connectivity of the SCCs (condesed form of the original graph). Suppose we have a strongly connected graph G, the SCC Tree for the graph is constructed as follows:

- If $|V(G)| = 1$ and $v \in V(G)$, then the SCC tree is $SCC(v) = STN(v) = (\{v\}, \emptyset)$.
- If $|V(G)| > 1$ and $d \in V(G)$, then SCC-tree node would contain the graph CONDENSE(SPLIT(G, d)), and for each SCC in the graph CONDENSE(SPLIT(G, d)), we add its SCC-tree as a subtree of R, with exception that we would add only one tree for vertex d instead of d_{in} and d_{out}.

Algorithm 5: CONSTRUCTDS(G)

Data: G
Result: SCC mapping, SCCTREES, STN(M)
$SM_G = \emptyset, \text{SCCTREE} = \emptyset$;
$S =$ FINDSCCS$(G), V_l = \emptyset, E_l = E(G)$;
for *each* $s \in S$ **do**
  $L_s =$ LABEL(s);
  $G_s = G \cap s$;
  SCCTREE$(L_s) =$ MAKETREE(G_s, L_s);
  for *each* $v \in s$ **do**
    $SM_G(v) = L_s$;
  end
  $V_l = V \cup \{L_s\}$;
  $E_l = E_l - \{e \in E(G_s)\}$;
end
STN$(M) = (V_l, E_l)$;
return SM_G, SCCTREEs, STN$(M) = (V_l, E_l)$;



(a) Graph 1

(b) condensed graph 1

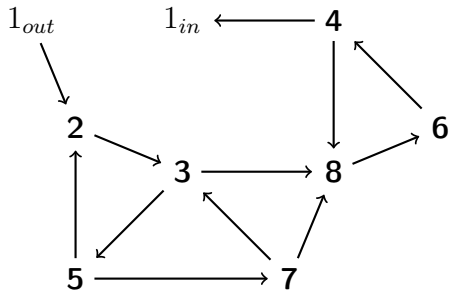
Figure 2: Graph 1 and its condensed graph

Algorithm 6: MakeTree(G, L)

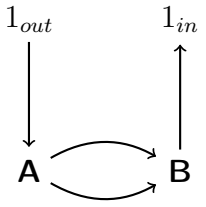
Data: G | G is strongly connected, L | label of the root node
Result: SccTree(L)
$v = random(V(G))$;
$\text{SccTree}(L) = \emptyset$;
if $|V(G)| = 1$ **then**
  $\text{STN}(L) = (\{v\}, \emptyset)$;
  $\text{SccTree}(L) = \text{STN}(L)$;
  return SccTree(L);
end
$G' = \text{Split}(G, v)$;
$S = \text{FindSccs}(G')$;
for *each* $s \in S$ *and* $v_{in} \notin s$ **do**
  $L_s = \text{Label}(s)$;
  $G_s = G' \cap s$;
  $\text{SccTree}(L_s) = \text{MakeTree}(G_s, L_s)$;
  $\text{SccTree}(L) = \text{SccTree}(L) \cup \text{SccTree}(L_s)$;
end
$\text{STN}(L) = \text{Condense}(G')$;
$\text{SccTree}(L) = \text{SccTree}(L) \cup \text{STN}(L)$;
return SccTree(L);

We can understand the algorithm by looking at the following figures. In Fig.2, we have the graph G, which is strongly connected. Its condensed form would contain a single node R. The SCC mapping array would map all the nodes to R and the master node would contain the graph in Fig.2b.

After the strongly connected components of the graph are indentified, they are individually processed by the MakeTree algorithm. The algorithm starts by selecting a random vertex from the SCC and then splits the graph on that vertex. The split graph is then condensed and the strongly connected components of the condensed graph are found. This is illustrated in Fig.3, where we can see the condensed components A and B. The condesed graph in Fig.3b is stored in the STN of the root node R.



(a) SCC(R) split on 1



(b) condensed graph

Figure 3: SCC(R) split on 1 and its condensed graph

The MakeTree algorithm then processes all the condensed components of the split graph in a recursive manner. The condensed component A is split on 2, as in Fig.4, and the condensed component B is split on 4, as shown in Fig.5. The component A contains C which is split on 3, as shown in Fig.6. The condesed graphs in Fig.4b, Fig.5b, and Fig.6b are stored in the STN of A, B, and C respectively.

(a) SCC(A) split on 2 (b) condensed graph

Figure 4: SCC(A) split on 2 and its condensed graph

(a) SCC(B) split on 4 (b) condensed graph

Figure 5: SCC(B) split on 4 and its condensed graph

(a) SCC(C) split on 3 (b) condensed graph

Figure 6: SCC(C) split on 3 and its condensed graph

The figure Fig.7 shows the SCC tree for the graph in Fig.2a. The SCC tree is constructed by the MakeTree algorithm, which recursively processes the condensed components of the split graph. The child of each parent node in the SCC tree is a node present in the STN of the parent node.
SccTree(R) = STN's of $\{R, 1, A, 2, C, 3, 5, 7, B, 4, 6, 8\}$

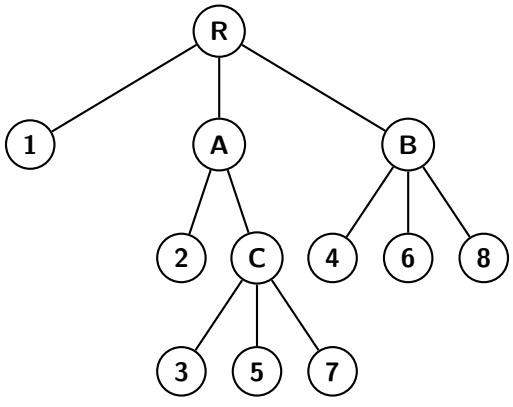


Figure 7: SCC Tree of Graph 2a

3.4 Decremental Maintainance of SCC Tree

In this section, we delve into the decremental maintenance of the SCC tree under the delete operation. As mentioned earlier, the DELETE(u, v) operation removes the edge from vertex u to vertex v in the graph G. This deletion operation necessitates the corresponding update in the SCC tree to maintain the internal connectivity of the vertices within the strongly connected components. The deletion may also introduce new strongly connected components that may be formed due to the decomposition of the earlier SCCs.

Given, a graph G and an edge (u, v) is to be deleted from the graph G. Suppose if the vertex u and v belong to different strongly connected component, then the deletion of the edge (u, v) will not affect any SCC tree. It would also not introduce any new SCCs, since the edge (u, v) was not a part of any SCC. In this case, we can simply remove the edge (u, v) from the edge set of the master node.

However, if the vertices u and v belong to the same strongly connected component, then the deletion of the edge (u, v) may introduce new SCCs or change the internal connectivity of the SCC tree. Therefore, while we delete an edge (u, v), we first must identify the SCC tree node that contains the edge (u, v). This can be done by finding the lowest commen ancestor of the vertices u and v in the SCC tree. We then proceed to delete the edge from the identified node.

Algorithm 7: DELETE(G, u, v)

```
Data: G, u, v
Result: G'
if QUERY(u, v) = false then
    STN(M).E = STN(M).E \ {(u, v)};
    return;
end
L = LCA(u, v);
STN(L).E = STN(L).E \ {(u, v)};
UPDATESCCTREE(L);
```

After the deletion of the edge (u, v), we must ensure that the SCC tree is updated to reflect the changes in the internal connectivity of the vertices. The updates ensures that the every vertex in each SCC tree node is reachable from the vertex d_{out} and d_{in}, where d is the vertex that was split to construct the SCC tree node, we refer §3.2.5 for the definition.

The following steps are performed to update the SCC tree after the deletion of the edge (u, v) which belongs to the SCC tree node L, and d is the vertex that was split to construct the SCC tree node L:

- The unreachable vertices are removed form the SCC tree node L, and are added in the vertex set of the SCC tree node $p(L)$, where $p(L)$ denotes parent node of L. If L is the root node, then the unreachable vertices form new SCC-tree's, and thus updating the master node and the SCC mapping array.
- If vertices were removed from the SCC tree node L, then repeat the above steps for the parent node of L. This process stops when L is the root node of the SCC tree, we then update the master node and the SCC mapping array.
- If unreachable vertices are present, then we expose the connectivity of these vertices to the parent node of L. This can be achieved by transferring the edges that were a part of the unreachable vertices in L to the parent node and also updating the edges in the edge set of the parent node that involved the unreachable vertices, which were a part of L. This can be seen in Fig.12b, further details can be found in §3.4.1 and §3.4.2.

3.4.1 Deleting Edge from the Root Node

We first consider the case when the deleted edge (u, v) belongs to the root node of the SCC tree. The edge is deleted from the graph held in STN(R) as show in Fig.8. As we can see in Fig.8b, the unreachable

Algorithm 8: UpdateSCCTree(L)

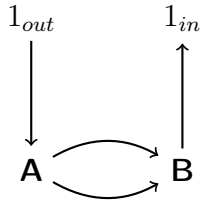
Data: L | SCC Tree Node
Result: Updated SCC Tree
$U = \text{Unreachable}(L, d_{in}, d_{out})$;
if $U == \emptyset$ **then return**;
$E\_to\_expose = \emptyset$;

{Step 1: Removing vertices and edges}
for $v \in U$ **do**
  $\text{STN}(L).V = \text{STN}(L).V \setminus \{v\}$;
end
for $(u, v) \in \text{STN}(L).E$ **do**
  if $u \in U$ *or* $v \in U$ **then**
    $\text{STN}(L).E = \text{STN}(L).E \setminus \{(u, v)\}$;
    $E\_to\_expose = E\_to\_expose \cup \{(u, v)\}$;
  end
end

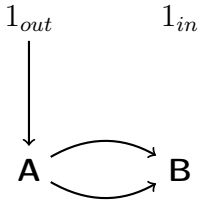
{Step 2: Exposing the connectivity}
if $L ==$ *Root* **then**
  $p(L) = \text{STN}(M)$;
end
for $(u, v) \in \text{STN}(p(L)).E$ **do**
  for $k \in U$ **do**
    $P = \text{SccTree}(k).Label$;
    if $u \in P$ **then**
      $\text{STN}(p(L)).E = \text{STN}(p(L)).E \setminus \{(u, v)\}$;
      $\text{STN}(p(L)).E = \text{STN}(p(L)).E \cup \{(k, v)\}$;
    end
    if $v \in P$ **then**
      $\text{STN}(p(L)).E = \text{STN}(p(L)).E \setminus \{(u, v)\}$;
      $\text{STN}(p(L)).E = \text{STN}(p(L)).E \cup \{(u, k)\}$;
    end
  end
end
$\text{STN}(p(L)).E = \text{STN}(p(L)).E \cup E\_to\_expose$;
$\text{STN}(p(L)).V = \text{STN}(p(L)).V \cup U$;

{ Step 3: Recursion (or) Updating the SCC Mapping Array}
if $L \neq$ *Root* **then**
  UpdateSCCTree(p(L));
end
else
  for $v \in U$ **do**
    for $u \in \text{SccTree}(v).Label$ **do**
      if $\text{STN}(u).E == \emptyset$ **then**
        $\text{SM}_G(u) = v$;
      end
    end
  end
end

nodes A and B are removed from STN(R) and form their own tree nodes in the SCC tree. The SCC-tree corresponding to label A and B gets separated from R and form new SCC-trees with root node A and B respectively. The SCC tree is updated as shown in Fig.9b.

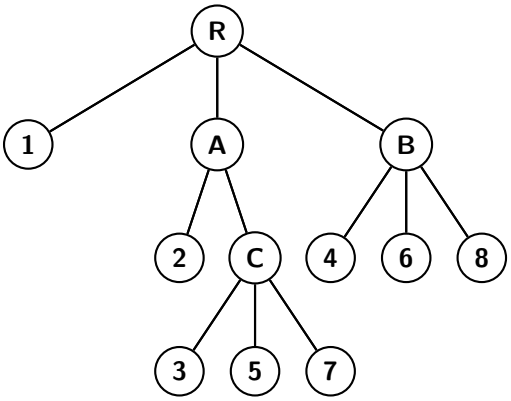


(a) Graph in SCC Tree Node R

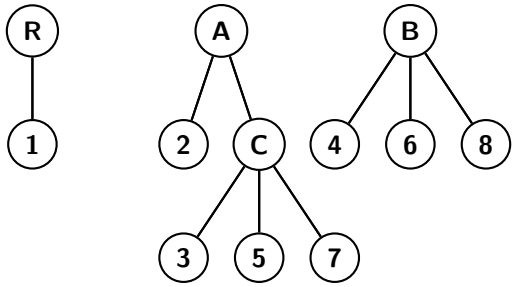


(b) Graph after deleting edge 4 to 1

Figure 8: Graph in SCC Tree Node R after deleting edge 4 to 1



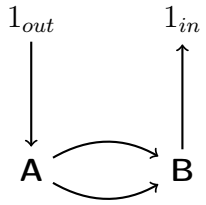
(a) SCC Tree of Graph 2a



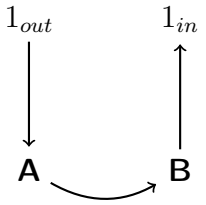
(b) SCC Tree of Graph 2a after updates

Figure 9: SCC Tree updates are propogated, deleting edge 4 to 1

The example in Fig.10 shows the graph in STN(R) after deleting edge 3 to 8. In this case, the nodes A and B are still reachable and hence remain in the SCC-tree node R.



(a) Graph in SCC Tree Node R



(b) Graph after deleting edge 3 to 8

Figure 10: Graph in SCC Tree Node R after deleting edge 3 to 8

3.4.2 Deleting Edge from an Internal Node

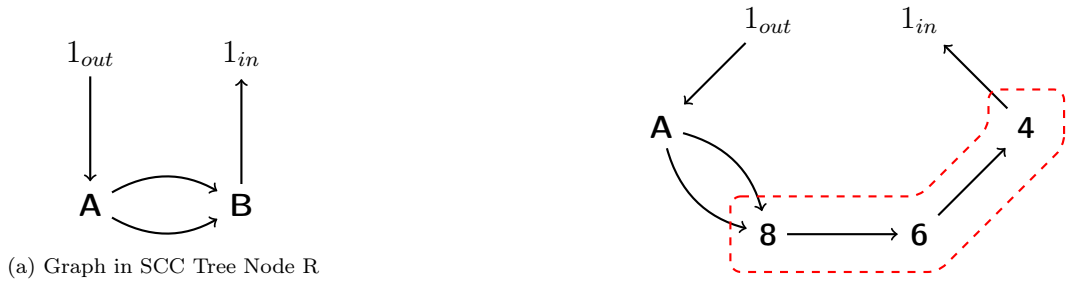
In this example, we consider the case when the deleted edge (u, v) belongs to an internal node of the SCC tree. The edge (4, 8) is deleted from the graph held in STN(B), since the lowest common ancestor of 4 and 8 is B from the SCC tree in Fig.7. We can see in Fig.11 that after removing the edge (4, 8) from the graph in STN(B), vertices 6 and 8 become unreachable from 4.

(a) Graph in Tree Node B

(b) Graph after deleting edge 4 to 8

Figure 11: Graph in Tree Node B after deleting edge 4 to 8

Since, vertices 6 and 8 are unreachable from 4, we expose these vertices and the corresponding edges in STN(B) to its parent node R. As shown in Fig.12b, the unreachable nodes 6 and 8 are exposed to the parent node R by adding edges (8,6), (6,4) from B and changing the edges (A,B), (B, 1_{in}) in R to (A,8), (4, 1_{in}) respectively. The red line in Fig.12b shows the exposed graph which was initially a part of B.

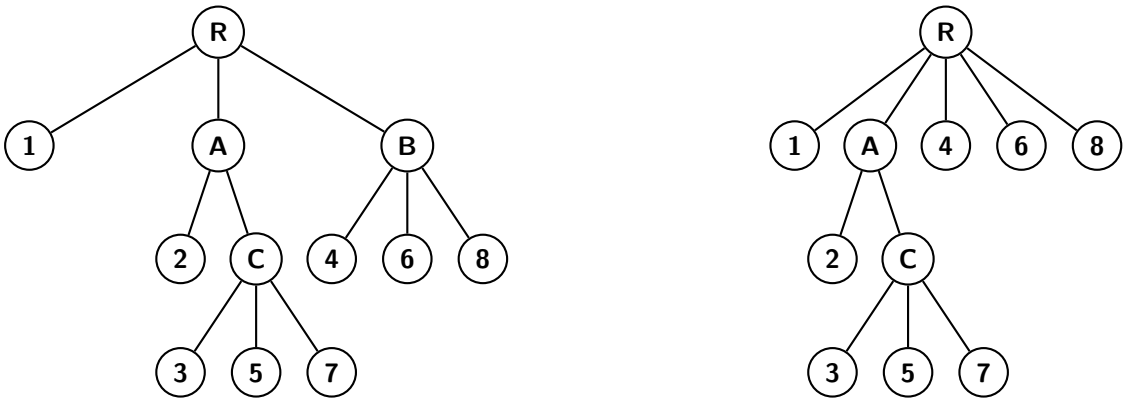


(a) Graph in SCC Tree Node R

(b) After Exposure

Figure 12: Graph in SCC Tree Node R after unreachable nodes and its corresponding edges are exposed

We observe that every vertex in Fig.12b is reachable, thereby concluding the algorithm. The above edge and vertex exposure process has a change in the SCC-tree, which is shown in Fig.13.



(a) SCC Tree of Graph 2a

(b) SCC Tree of Graph 2a after updates

Figure 13: Tree updates are propogated, deleting edge 4 to 8

3.5 Incremental Maintaince of SCC Tree

In this section, we understand the process of maintaining the SCC tree under the add operation. The ADD(u, v) operation introduces the edge from vertex u to vertex v in the graph G. This operation when performed may join strongly connected components or change the internal connectivity of the SCC tree.

Algorithm 9: ADD(G, u, v)

Data: G, u, v
Result: G'
$L = \text{LCA}(u, v)$;
$\text{STN}(L).E = \text{STN}(L).E \cup \{(u, v)\}$;
UPDATESCCTREEI(L);

For graph G and an edge (u, v) is to be added to the graph G. Suppose if the vertex u and v belong to different strongly connected component, then the addition of the edge (u, v) might combine multiple SCCs to form a sigle larger SCC. In this case, we add the edge (u, v) to the master node and update the SCC mapping array to reflect the newly formed SCC. A new SCC-tree node is introduced by adding the vertices and their corresponding edges which were a part of the SCCs that were combined in the master node. These combined SCCs become the children of the newly formed SCC-tree node. This new SCC-tree node has to propogate its updates to maintain the internal connectivity of the vertices.

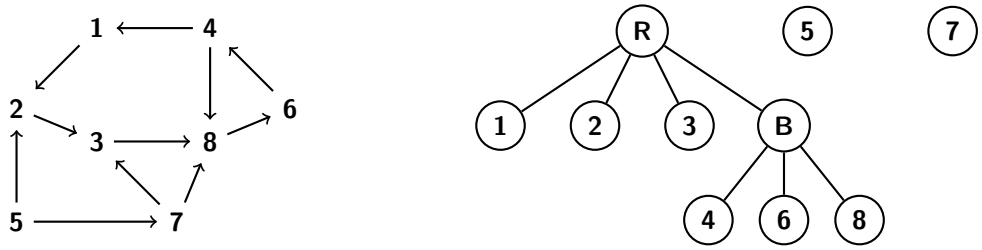
However, if the vertices u and v belong to the same strongly connected component, then the addition of the edge (u, v) will never introduce any new SCCs, but would significantly change the internal structuring of the SCC tree. These changes are to be propogated to maintain the SCC-tree structure and property. The update propogation is nothing but re-running the SCC-tree creation algorithm on the SCC-tree node that has changes to propogate. The difference is that we do not have to re-run the algorithm on the entire graph, but only on the graphs stored in the SCC-tree nodes.

Algorithm 10: UPDATESCCTREEI(L)

Data: L | SCC Tree Node
Result: Updated SCC Tree
$G = \text{STN}(L)$, $U = \text{FINDSCC}(G)$;
for $U_i \in U$ ***and*** $|U_i| \neq 1$ **do**
    $L_i = \text{LABEL}(U_i)$;
    $\text{STN}(L_i) = \text{STN}(L) \cap U_i$;
    $\text{STN}(L_i) = \text{SPLIT}(\text{STN}(L_i), d)$;
    UPDATESCCTREEI(L_i);
end
$\text{STN}(L) = \text{CONDENSE}(G)$;
for $U_i \in U$ ***and*** $|U_i| = 1$ **do**
    $T = U_i$;
    while $T \neq \emptyset$ **do**
        $V = T.pop()$;
        if $\text{STN}(V).E == \emptyset$ **then**
            $\text{SM}_G(V) = L_i$;
            continue;
        end
        $T = T \cup \text{STN}(V).V$;
    end
end

3.5.1 Updating SCC-tree under the Add operation

We would now understand the process of updating the SCC-tree, when the edge (u, v) is added to the graph G. The graph in Fig.14a is considered for this example, we can observe the SCC-tree of the graph in Fig.14b. The graph has 3 stongly connected components which are identified by labels R, 5 and 7.

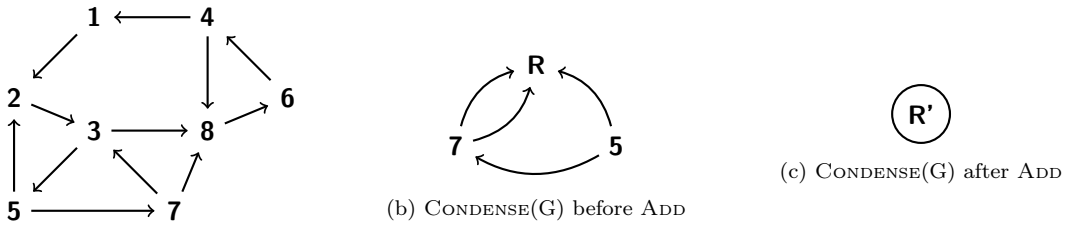


(a) Graph 2

(b) SCC Tree of Graph 2

Figure 14: Graph 2 and its SCC-trees

The edge $(3, 5)$ is added to the graph, it combines the strongly connected components R, 5 and 7 to form a single SCC R'. This change can be seen in the figure Fig.15, we notice that the edge (3,5) correponds to the edge (R',5) in the master node, which upon addtion combines the SCCs R, 5 and 7.



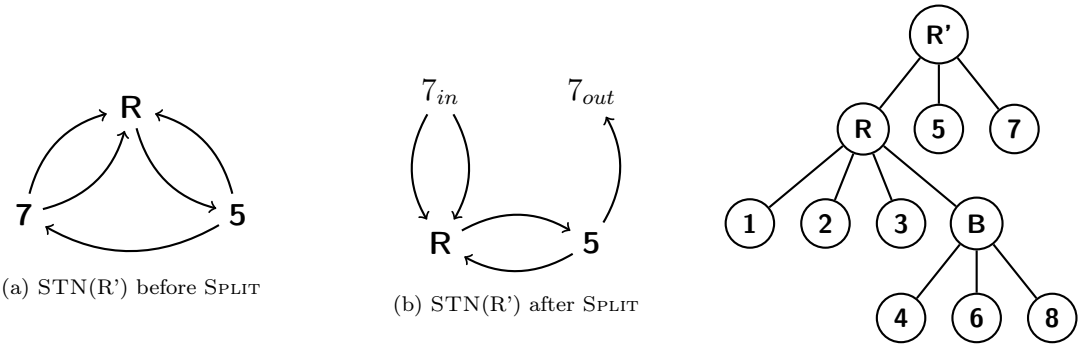
(a) Graph 2

(b) Condense(G) before Add

(c) Condense(G) after Add

Figure 15: Graph 2, Condense(G) before and after Add

A new SCC-tree node R' is introduced to represent the new SCC formed by the addition of the edge $(3, 5)$. It captures the subgraph that is formed by the vertices and edges that were a part of the SCCs R, 5 and 7. The SCC-tree node R' graph is split on the vertex 7, as shown in Fig.16. When we condense the graph in STN(R'), we find that the vertices R and 5 combine to form a single vertex A, as shown in Fig.17.



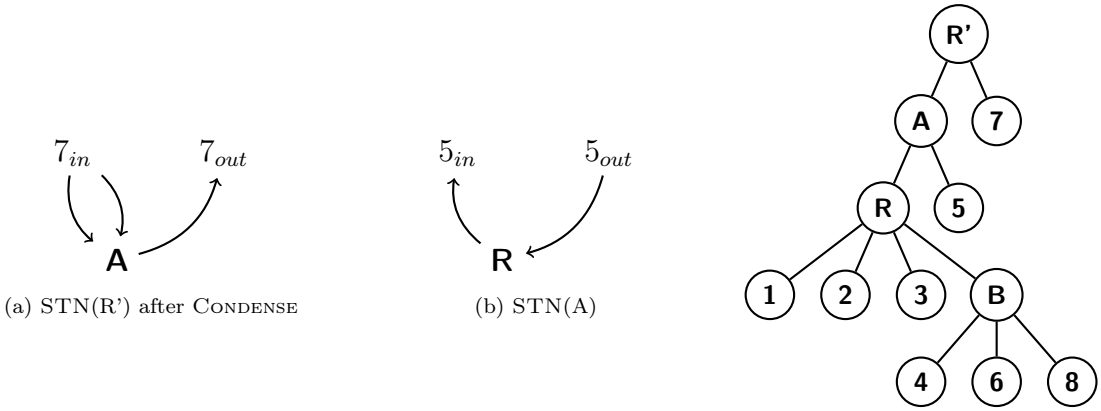
(a) STN(R') before Split

(b) STN(R') after Split

(c) updated SCC-tree

Figure 16: STN(R') before and after Split and updated SCC-tree

The new SCC-tree node A is introduced that captures the subgraph formed by the vertices and edges that were a part of R and 5, in the SCC-tree node R'. This graph can be seen in Fig.17, when we condense the graph in STN(A), we find that no combination of vertices is possible, and thus the SCC-tree is updated. The SCC-tree is updated to reflect changes brought by the addition of the edge $(3, 5)$ is shown in Fig.17.



(a) STN(R') after Condense (b) STN(A) (c) updated SCC-tree

Figure 17: STN(R'), STN(A) and updated SCC-tree

4 Practical Implementation

In this part, we delve into the practical implementation of our algorithm for maintaining strongly connected components (SCCs) within large-scale graphs using MPI in C++. By leveraging MPI's robust framework, we endeavor to unlock the full potential of parallelism, distributing the computational load across multiple nodes to expedite the identification and maintenance of SCCs.

Message Passing Interface (MPI), is a standardized and portable message-passing system designed to facilitate parallel computing across a distributed memory system. In the further sections, we will discuss the data structures used, the workload distribution in parallel, and the cache optimizations that were implemented to enhance the performance of our algorithm.

We will be using the Boost MPI library for the implementation of the algorithm. The Boost MPI library provides a high-level interface for the Message Passing Interface (MPI) standard, enabling the development of parallel applications in C++. We would also be using the Boost Serialization library for serializing the data structures used in the algorithm. This library provides a framework for serializing and deserializing C++ data structures to and from a sequence of bytes, facilitating the transmission of data across a network.

4.1 Data Structures

In distributed parallel computing, the efficacy of algorithms often hinges upon the efficiency of their underlying data structures. In this section, we discuss the design and characteristics of the data structures introduced in the theoretical underpinnings of our SCC maintenance algorithm. By elucidating their intricacies, time complexities, and other pertinent aspects, we aim to provide a comprehensive understanding of their role in facilitating efficient graph analysis within a distributed computing environment.

4.1.1 SCC Tree

The SCC-tree introduced in §3.1.1 is a hierarchical data structure that encapsulates the SCCs within a directed graph. It comprises of various components, including the SCC-tree node, the SCC-tree edge, and the SCC-tree itself.

The Edge class is a simple data structure that encapsulates the source and destination vertices of an edge within the graph stored in a SCC-tree node. The serialization and equality operator overloads are implemented to facilitate the serialization of the Edge object and comparison of two Edge objects. The Edge class is defined in Lst.1.

```
class Edge {
public:
    long int from;
    long int to;

    Edge() {}
    Edge(int from , int to) : from(from), to(to) {}

    friend boost::serialization::access;
    template < class Archive >
    void serialize(Archive & ar, const unsigned int version) {
        ar & from;
        ar & to;
    }

    bool operator==(const Edge& other) const {
        return from == other.from && to == other.to;
    }
};
```

Listing 1: Edge

The SCC-tree node class encapsulates vertices and edges connecting them, forming a graph, as discussed in §3.1.1. The class comprises various components, including the label, parent, corresponds-to, contains, and dept, which are described in detail in Lst.2. The SCC-tree node class also includes various functions, such as condenseFill, checkUnreachable, updateLabels, exposeToParent, and removeEdge, which are used to manipulate the SCC-tree node and its components.

```
class TreeNode {
public:
    /**
        * @brief this would represent the label of the node in the SCC tree
        * the label is unique for each node in the SCC tree
        *
        */
    long int                            label;
    /**
        * @brief this represents the parent of the current node in the SCC tree
        *  and is used to traverse the tree upwords.
        *
        */
    long int                            parent;
    /**
        * @brief stores the actual edge to labelled edge pair,
        * nodes forming a scc are condesed to one labelled node, and so are the edges,
        * the corresponds_to is used to store the actual edges to labelled edges
        * relationship.
        *
        */
    std::vector<std::pair<Edge, Edge>>  corresponds_to;
    /**
        * @brief contains the nodes which are part of the current node in the SCC tree.
        * They store the label of the child nodes and is used to travese the tree
        * downwards.
        *
        */
    std::unordered_set<long int>        contains;
    /**
        * @brief stores the depth of the current node in the SCC tree and is
```

```
32          * used to find the LCA of two nodes (Lowest Common Ancestor),
33          * important for update operations.
34          *
35          */
36      long int                            dept;
37      /**
38          * @brief The following function is used to serialize the TreeNode object,
39          * this is essential for the boost::serialization to work. The TreeNodes are
40          * transferred during update queries.
41          *
42          */
43      friend                              boost::serialization::access;
44      template < class Archive >
45      void serialize(Archive & ar, const unsigned int version) {
46          ar & label;
47          ar & parent;
48          ar & corresponds_to;
49          ar & contains;
50          ar & dept;
51      }
52
53      /**
54          * @brief the function is used to fill the corresponds_to edge pairs for the
55          * current node.The sccs is used to find the labelling of the nodes in the
56          * graph and create the corresponding edges in the SCC tree.
57          *
58          * @param edges are the actual edge present in the graph
59          * @param sccs is a unordered_map which stores the nodes and their
60          * corresponding sccs to which they belong.
61          */
62      void condenseFill(std::vector<Edge>& edges,
63                  std::unordered_map<long int, long int>& sccs);
64      /**
65          * @brief the function is used to find the unreachable nodes in the current node.
66          *
67          * @param unreachable is a set which stores the unreachable nodes after its
68          * execution.
69          */
70      void checkUnreachable(std::unordered_set<long int>& unreachable);
71      /**
72          * @brief the function is used to update the labels of the nodes in the
73          * current node.
74          *
75          * @param new_labels is a unordered_map which stores the new labels of the nodes.
76          */
77      void updateLabels(std::unordered_map<long int, long int>& new_labels);
78      /**
79          * @brief the function is used to indentify the edges which are unreachable
80          * in the current node, and transfer them to the parent by appropriately updating
81          * their labels and adding them to the parent node.
82          *
83          * @param parent The parent node to which the unreachable edges are to be
84          * transferred
85          * @param unreachable Set of unreachable nodes in the current node
86          */
87      void exposeToParent(TreeNode& parent, std::unordered_set<long int> unreachable);
88      /**
89          * @brief the function is used to remove the edge from the current node.
90          *
91          * @param edge The edge to be removed from the current node
92          */
93      void removeEdge(const Edge& edge);
94
95      //copy constructor
96      TreeNode(const TreeNode& other) {
```

```
        label = other.label;
        parent = other.parent;
        corresponds_to = other.corresponds_to;
        contains = other.contains;
        dept = other.dept;
    }
    //assignment operator
    TreeNode& operator=(const TreeNode& other) {
        label = other.label;
        parent = other.parent;
        corresponds_to = other.corresponds_to;
        contains = other.contains;
        dept = other.dept;
        return *this;
    }
    //default constructor
    TreeNode() {
        label = -1;
        parent = -1;
        dept = 0;
    }
};
```

Listing 2: SCC-Tree Node

The SCC-tree is collection of SCC-tree nodes, forming a tree structure that encapsulates the SCCs within a directed graph. This information is stored in the form of mappings between the labels of the nodes and the corresponding SCC-tree nodes. We hold the labels of the root nodes of the SCC-tree in a vector, which is used to traverse the tree and perform various operations.

```
    std::vector<long int> root_nodes;
    std::unordered_map<long int, TreeNode> scc_tree_nodes;
```

4.1.2 SCC Mapping Array

The SCC mapping array is a data structure that would store the mapping of the vertices to the SCCs label they belong to. We use an unordered map to store this information as it provides an average time complexity of O(1) for insertion, deletion, and lookup operations.

```
    std::unordered_map<long int, long int> scc_mapping;
```

4.2 Workload Distribution in Parallel

In this section, we discuss how the workload is distributed across multiple nodes in a parallel computing environment using MPI. The algorithm is designed to distribute the graph data across multiple nodes, with each node responsible for processing them separately. In future references, we will refer to these nodes as cores or processors interchangeably. Every core is assigned a unique rank, which is used to identify them in the MPI environment. These cores would each be reposible for storing an instance of class *MaintainSCC*, which would contain nessary data structures and functions to perform the SCC maintenance operations.

The *MaintainSCC* class contains the following:

- SCC_LABEL: A unique label that is assigned, if we form a new SCC-tree node. It is an integer corresponding to the next available (unused) label.
- world_size: The total number of cores in the MPI environment, used to assigned workload.
- roots: A vector that stores the labels of the root nodes of the SCC-tree.
- scc_tree_nodes: A map that stores the mapping between the labels of the nodes and the corresponding SCC-tree nodes.

- delete_cache: A set that stores the nodes that were affected by the deletion of edges.
- insert_cache: A set that stores the nodes that were affected by the insertion of edges.
- rank: A map that stores a mapping between each vertex in the graph and the rank of the core that is responsible for it. In other words, it stores the rank of the core that contains the vertex.
- scc_mapping: A map that stores the mapping between the vertices and the SCCs they belong to.

Since, MPI is a message passing interface, we need to define the message types that would be used to communicate between the cores. The following message types are defined:

- SCC_TREE: Used to transfer data that is used in the creation of SCC-tree.
- EDGE_DELETE: Used to transfer the edges that need to be deleted between the cores.
- EDGE_INSERT: Used to transfer the edges that need to be inserted between the cores.
- EDGE_QUERY: Used to transfer the edges that need to be queried between the cores.
- SCC_TRANSFER: Used to transfer the SCC-tree for load balancing.
- CLR_DEC_CACHE: Used to clear the delete cache.
- CLR_INC_CACHE: Used to clear the insert cache.
- EXIT: Used to signal the end of the program.

```
class MaintainSCC {

    long int                                SCC_LABEL = 0;
    const long int                          MOD = 1e10;
    int                                     world_size;
    std::vector<long int>                   roots;
    std::unordered_map<long int, TreeNode>  scc_tree_nodes;
    std::set<Cache, DecCache>               delete_cache;
    std::set<Cache, IncCache>               insert_cache;
    std::unordered_map<long int, int>       rank;
    std::unordered_map<long int, long int>  scc_mapping;

    enum MessageType
    {
        SCC_TREE,
        EDGE_DELETE,
        EDGE_INSERT,
        EDGE_QUERY,
        SCC_TRANSFER,
        CLR_DEC_CACHE,
        CLR_INC_CACHE,
        EXIT
    };

    /** some internal functions **/
    /**         .......         **/
public:
    bool query(long int v1, long int v2);
    void deleteEdges(std::vector<Edge> &decrement);
    void insertEdges(std::vector<Edge> &increament);
    void endAll();

    MaintainSCC(long int n, std::vector<Edge> &edges);
    ~MaintainSCC();
};
```

The 0th core is responsible for the initialization of the graph and the distribution of the workload across the cores. It takes in the entire graph and runs the SCC algorithm to find the SCCs in the graph, and then

distributes the vertices and edges across the cores based on the SCCs they belong to. The other cores are responsible for constructing the SCC-tree and maintaining the SCCs in the graph, this distribution is stored in the *rank* map. During this process, the 0th core also constructs and maintains the master node and the SCC mapping array.

The other cores work parallelly to construct the SCC-tree in the initialization stage. Once the SCC-tree is constructed, the cores are synchorized and ready for the update queries. When we call for an update, it is processed in the 0th core, which identifies to which core the update belongs to and sends the same. Upon recieving the update, the core processes the update and sends the result back to the 0th core. The following situation may arise:

- In case of an insert query, two SCCs might get combined to form a new SCC. Here, the master node is updated in the 0th core and upon identifying the SCCs that are combined, the SCC-tree corresponding to these components are sent from their respective cores to a common one. This ensures that the update propogation happens in a synchorized and easier manner.
- In case of a delete query, the SCCs might get divided into two or more SCCs. In this case, the labels of the newly created SCCs are sent to the 0th core, which then updates the master node, the SCC mapping array and sends the new label - core mapping back to the updated core. Upon recieving the new label - core mapping, the current core sends these SCCs to the respective cores, thereby balancing the load.

All other functionallity are implemented based on the above mentioned design and algorithm in §3. The public functions in the *MaintainSCC* class are used to interact with the cores and perform the necessary operations. These functions can only be called from the 0th core, which then analyzes and forwards the request to the respective core.

4.3 Update Cache Optimizations

In the section, we discuss the cache optimizations that were implemented to enhance the performance of the algorithm. When we delete or insert an edge in the graph, it affects the structuring of the SCC-tree and hence the changes have to be propogated. We notice that the process of propogation of changes is time consuming and can be optimized by caching the changes and propogating them in batches. For understanding the cache optimizations, we need to understand the following, referring to the article [18]:

- An SCC-tree can be constructed in $O(m\delta)$ time, where δ is the height of the resulting tree and m is the edges part of that tree. It requires $O(n+m)$ space.
- Given an SCC-tree of height δ, the algorithm processes any sequence of edge updates in $O(m\delta)$ total time and answers each query in $O(1)$ time, using $O(n+m)$ space.

Given a SCC-tree of height δ, we can delete or insert an edge in $O(\delta)$ time. This is because the deletion or insertion of an edge can affect the SCC-tree in the worst case by δ levels. The propogation of changes can be done in $O(m\delta)$ time, thus it is beneficial to cache the changes and propogate them in batches.

Suppose there are t updates to the graph, then the time complexity of the entire process would be $O(tm\delta)$, with the cache optimizations, the time complexity would be $O(m\delta+t\delta)$, where $t\delta$ is time taken to delete/insert edge only, and $m\delta$ is time taken to propogate changes. The time complexity is reduced from $O(tm\delta)$ to $O((m+t)\delta)$.

```
class Cache {
public:
    long int        dept;
    long int        label;

    bool operator<(const Cache& other) const {
        return dept < other.dept;
    }
```

```
    bool operator==(const Cache& other) const {
        return dept == other.dept && label == other.label;
    }
};

class DecCache : public Cache {
public:
    bool operator() (const Cache& a, const Cache& b) const {
        return a.dept > b.dept;
    }
};

class IncCache : public Cache {
public:
    bool operator() (const Cache& a, const Cache& b) const {
        return a.dept < b.dept;
    }
};

std::set<Cache, DecCache>               delete_cache;
std::set<Cache, IncCache>               insert_cache;
```

Listing 3: Cache Optimizations

We notice in §3.4 and §3.5, that for decremental changes we need to propogate the changes from the node to the parent, and for incremental changes we need to propogate the changes from the node to the child node. Thus, we use two sets, one for decremental changes and one for incremental changes with following properties:

- The class *Cache* is used to store the label of the node and its depth in the SCC-tree.
- The class *DecCache* is used to define sorting order for the decremental cache, where the nodes are sorted in decreasing order of their depth. Since, we need to propogate the changes from the node to the parent, we need to start from the deepest node and move upwards.
- The class *IncCache* is used to define sorting order for the incremental cache, where the nodes are sorted in increasing order of their depth. Since, we need to propogate the changes from the node to the child node, we need to start from the shallowest node and move downwards.

Table 1: Timings with/without Cache Optimizations (`Amazon0302`)

Updates(%)	With Caching	Without Caching
0	9.04671	10.0534
1.0	2.09399	15.9024
5.0	2.24362	23.2398
10.0	2.34654	40.28242
20.0	2.6431	50.2302
30.0	2.83777	56.28420
50.0	3.25233	88.23823

4.4 Challenges in Implementation with MPI

Utilizing the MPI library in C++ for distributed parallel computing introduces a unique set of challenges that demand careful consideration and adept problem-solving strategies. Below, we outline some common challenges encountered during MPI programming and the way we dealt with them:

- **Load Balancing**: One of the primary challenges in distributed computing is ensuring that the workload is evenly distributed across all nodes. We addressed this issue by distributing the vertices and edges

based on the SCCs they belong to, thereby ensuring that each core receives a balanced workload. Though this approach is effective, it may lead to an imbalance in the workload if the graph is not well-distributed. We plan to address this issue by implementing load balancing based upon the number of SCC-tree nodes each core is responsible for, in the future iterations of the algorithm. This would ensure that each core receives an equal number of SCC-tree nodes, thereby balancing the workload.

- **Communication Overhead**: In a distributed computing environment, communication overhead can significantly impact the performance of the algorithm. This is especially true in MPI programming, where inter-process communication is a crucial aspect. We tried to minimize the communication overhead by reducing the number of messages exchanged between the cores. This was achieved by the work distribution strategy, where the 0th core is responsible for processing the updates and distributing them to the respective cores.
- **Synchronization**: Synchronization is a critical aspect of parallel computing, especially in distributed environments where multiple nodes are involved. Since, the updates are processed parrallely in the cores, we had to ensure that the cores are synchronized, by using specialized Message Types to signal the end of the program and to synchronize the cores. MPI provides various synchronization mechanisms, such as barriers and point-to-point communication, which we leveraged to ensure that the cores are synchronized.
- **SCC-tree transfer**: When an SCC is formed or divided, the SCC-tree corresponding to these SCCs needs to be transferred between the cores. These transfers can lead to deadlocks if not handled properly, as the cores may be waiting for each other to send the SCC-tree. We addressed this issue by using a two-phase commit protocol, where the cores first signal the 0th core about the SCCs that are formed or divided, and then the 0th core sends the SCC-tree to the respective cores.
- **Memory Management and Resource Consumption**: Inefficient memory utilization and excessive resource consumption can lead to performance degradation and system instability. The testing server was very much capable of handling the memory requirements of the algorithm, but this might not be the case in a real-world scenario. The implementation must be optimized to minimize memory consumption and ensure efficient resource utilization, by using memory pooling and other memory management techniques.
- **Debugging and Testing**: Debugging and testing distributed parallel algorithms can be challenging due to the inherent complexity of the system. We used various debugging tools provided by MPI, such as MPI_Error_string and MPI_Abort, to identify and resolve issues in the code. We also employed unit testing and integration testing to ensure that the algorithm functions correctly in a distributed environment. The testing was done on a local cluster, and the results were compared with the sequential implementation to verify the correctness of the algorithm.
- **Recursion Limit**: The SCC-tree construction algorithm is recursive in nature, which can lead to stack overflow errors if the recursion depth is too high. We addressed this issue by implementing iterative versions of recursive algorithms where possible to avoid stack overflow issues. Alternatively, increase the stack size or switch to non-recursive approaches to mitigate segmentation faults.

5 Results

In this section, we present the results of the experiments conducted. The results are in the form of tables and graphs. The details of the graphs used in the experiments are presented in Table 2. The table Table.3 presents the timings for the experiments conducted on the graphs. The timings are presented in seconds. The timings are compared with the standard static algorithm and the dynamically maintained parallel algorithm. The timings are presented for different percentages of updates w.r.t. edges.

The machine specifications are retrieved using the `lscpu` command. The machine specifications are as follows:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 80
On-line CPU(s) list: 0-79
Thread(s) per core: 2
Core(s) per socket: 20
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 85
Model name: Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz
Stepping: 7
CPU MHz: 1000.061
CPU max MHz: 3900.0000
CPU min MHz: 1000.0000
BogoMIPS: 5000.00
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
L3 cache: 28160K
NUMA node0 CPU(s): 0-19,40-59
NUMA node1 CPU(s): 20-39,60-79

The software specifications are retrieved using the `gcc -version`, `mpirun -version`, `boost -version`, and `cmake -version` commands. The kernel and OS specifications, are retrieved using the `hostnamectl` command. The above specifications are as follows:

Operating System: Red Hat Enterprise Linux Server 7.6 (Maipo)
Kernel: 3.10.0-957.el7.x86_64
Compiler: gcc version 9.2.0
OpenMPI: 3.1
Boost: 1.84.0
CMake: 2.8.12.2

Table 2: Graph Details [1, SNAP Datasets]

Graph	Number of Vertices	Number of Edges
Slashdot0902	82168	948464
Wiki-Vote	7115	103689
web-BerkStan	685230	7600595
web-Google	875713	5105039
Amazon0302	262111	1234877
WikiTalk	2394385	5021410
soc-pokec-relationships	1,632,803	30,622,564

The each graph, the algorithm was tested by increasing the total number of updates w.r.t. the percentage of edges present in the graph. The 0 percent edge updates, indicates the time taken to initialization the SCC-tree for the maintaince process. The subsequent tests conducted, reveal the time taken to update the SCC-tree for the given percentage of updates. All the testing was conducted on the same machine, with `-np` parameter set to 64 for the `mpirun` command.

Table 3: Timings for Graphs in Table.2

Table 4: `Amazon0302`

Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	2.69061	9.04671
1.0	2.69061	2.09399
5.0	3.01042	2.24362
10.0	3.09495	2.34654
20.0	3.29579	2.6431
30.0	3.40107	2.83777
50.0	3.54919	3.25233

Table 5: `Slashdot0902`

Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	1.1383	5.30846
1.0	1.1383	0.509686
5.0	1.17131	0.64476
10.0	1.19611	0.761259
20.0	1.25575	1.04112
30.0	1.4406	1.24635
50.0	1.30949	1.67949

Table 6: `Wiki-Vote`

Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	0.102256	0.435191
1.0	0.102256	0.0353187
5.0	0.108545	0.042797
10.0	0.0966422	0.0540102
20.0	0.101367	0.0726061
30.0	0.0961294	0.0943561
50.0	0.118703	0.136274

Table 7: `WikiTalk`

Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	20.7767	37.7838
1.0	20.7767	15.4092
5.0	22.2939	17.7078
10.0	22.7917	18.5541
20.0	23.2082	19.1667
30.0	23.4775	20.6392
50.0	24.4343	23.8523

Table 8: `web-BerkStan`

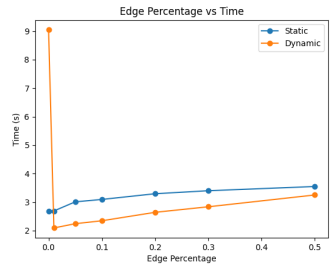
Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	8.28039	39.5969
1.0	8.28039	4.10159
5.0	9.12419	8.08547
10.0	9.88219	10.96283
20.0	11.0039	16.96717
30.0	11.4561	17.9223
50.0	13.0278	18.2617

Table 9: `web-Google`

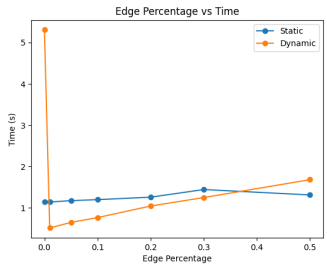
Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	13.6496	42.7283
1.0	13.6496	6.87915
5.0	14.4792	9.60964
10.0	14.9307	10.43506
20.0	15.4813	15.0343
30.0	15.8665	16.96263
50.0	16.8315	23.3846

Table 10: `soc-pokec-relationships`

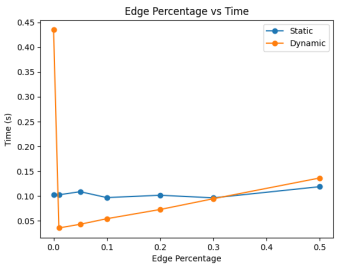
Update(%)	**Timings**	
w.r.t. edges	Static	Dynamic
0	134.453	257.21
1.0	138.789	102.434
5.0	139.223	120.928
10.0	140.712	139.932
20.0	139.883	141.283
30.0	141.291	170.221
50.0	142.843	174.898



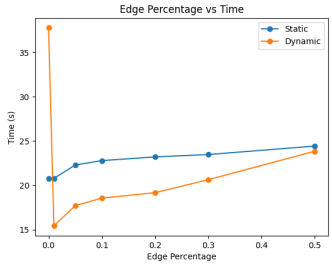
(a) `Amazon0302`



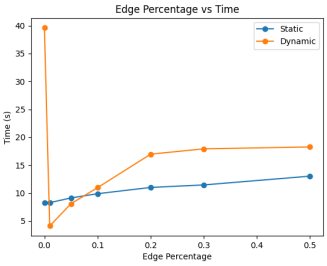
(b) `Slashdot0902`



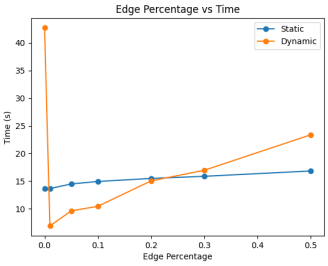
(c) `Wiki-Vote`



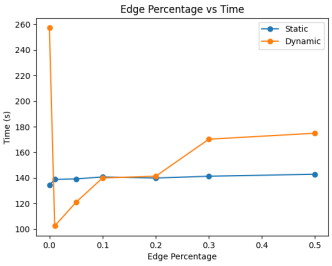
(d) `WikiTalk`



(e) `web-BerkStan`



(f) `web-Google`



(g) `soc-pokec-relationships`

Figure 18: Graphs Timings for Table.2

6 Future Work

The work presented in the paper is a preliminary study and implementation of the parallel dynamic graph algorithms. The SCC finding algorithm used in this paper is a linear sequential algorithm, which is not very efficient and fast on larger graphs. We can use any parallel SCC finding algorithms in [15] and extend it to fit the dynamic graph setting using the algorithm presented in this paper. The SCC-tree can further be exploited to make computations more parallel since the it is a directed acyclic graph and changes in the graph can be propagated in parallel.

The combination of distributed memory parallelism and shared memory parallelism can be used to make the algorithm more scalable. The underlying dynamic graph maintainance algorithm can be improved to make it more parallel and efficient. The future endeavors can also bring in the use of GPUs to make the algorithm more efficient and faster. Randomized algorithms can be used to make the process more faster, and efficient methods of synchronization can be used to make the algorithm more reliable and scalable.

References

[1] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[2] Laurent Théry. Formally-proven kosaraju's algorithm. 02 2015.

[3] Robert E. Tarjan and Uri Zwick. Finding strong components using depth-first search. *CoRR*, abs/2201.07197, 2022.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. *MIT Press and McGraw-Hill*, 2009.

[5] Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[6] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5), 1985.

[7] RJ Anderson and A Aggarwal. A random nc algorithm for depth first search. *Combinatorica*, 8:1–12, 1988.

[8] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM Journal on Computing*, 19(2):397–409, 1990.

[9] Ming-Yang Kao, Shang-Hua Teng, and Kentaro Toyama. Improved parallel depth-first search in undirected planar graphs. In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures*, pages 409–420, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[10] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, June 1988.

[11] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, June 1989.

[12] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In José Rolim, editor, *Parallel and Distributed Processing*, pages 505–511, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[13] Simona Orzan. On distributed verification and verified distribution. 01 2004.

[14] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, 2014.

[15] Sudharshan Srinivasan. Identifying strongly connected components on distributed networks. *DRP Technical Report, University of Oregon, Eugene, OR, USA*, 2021.

[16] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Dynamic algorithms for classes of constraint satisfaction problems. *Theoretical Computer Science*, 259:287–305, 01 2001.

[17] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37, 09 2002.

[18] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms, 9, 3, Article 27*, June 2013.

[19] Liam Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1143–1150, USA, 2013. Society for Industrial and Applied Mathematics.