A Comparative Study: Tarjan-Vishkin VS Slota-Madduri

Saiyam Jain & Shail Shah

2023101135 2023101060

Date: May 6, 2025

Contents

1 Introduction

In the field of graph theory, a **biconnected component** (BCC) is defined as a maximal subgraph in which the removal of any single vertex does not disconnect the subgraph. Identifying such components is critical for a variety of applications, including network reliability analysis, social network clustering, and circuit design, where ensuring resilience and fault tolerance in the network structure is a primary requirement.

This project focuses on the implementation and comparative evaluation of two prominent algorithms for detecting biconnected components in undirected graphs:

- **Tarjan-Vishkin-based Algorithm**: A classical sequential algorithm that employs a Depth-First Search (DFS) strategy to efficiently determine articulation points and partition the graph into its biconnected components.
- **Slota-Madduri Algorithm**: A sequential algorithm originally designed with parallelism in mind, aiming to enhance performance, particularly on large-scale graph datasets.

The objectives of this project are:

1. To implement both algorithms in C++ and validate their correctness.
2. To conduct an experimental study assessing their execution time, memory consumption, and ability to scale with increasingly large undirected graph inputs.
3. To visualize the comparative performance of these algorithms through appropriate plots and analyses.

Project Context and Guidelines

This work forms part of a broader experimental project-based initiative where students work in teams of two to investigate algorithmic solutions on large real-world datasets. Key guidelines for the project include:

- Graph datasets will be sourced from the University of Florida Sparse Matrix Collection, featuring large sparse graphs with up to millions of vertices and edges.
- Each project involves comparing two different algorithms addressing the same problem, with their performance and efficiency illustrated via comparative plots.
- While certain algorithms in literature propose parallel implementations, this project will restrict itself to sequential implementations, maintaining fair grounds for comparison.

- Experimental runs will be conducted on server-class machines provided for this purpose.
- The project holds potential for future research extensions, including the exploration of parallel variants or applications in specialized domains, potentially leading to publishable results.

Problem Statement

The specific problem addressed in this project is the detection and listing of all biconnected components within a given undirected graph. Through implementing and evaluating the Tarjan-Vishkin and Slota-Madduri algorithms, the project seeks to investigate how each algorithm performs across varying graph sizes and densities, analyzing both computational and memory efficiency.

2 Tarjan-Vishkin Algorithm

2.1 Algorithm Overview

The Tarjan-Vishkin algorithm for detecting biconnected components (BCCs) in an undirected graph is based on a Depth-First Search (DFS) traversal that assigns timestamps and computes subtree relations. The algorithm identifies articulation points and groups edges into biconnected components by considering back edges and parent-child relationships within the DFS tree.

In this implementation, we closely follow the structure and methodology described in the paper *"An Efficient Parallel Biconnectivity Algorithm" by Robert E. Tarjan and Uzi Vishkin.* Although the original algorithm was designed for parallel execution, its core concepts are effectively applied in this sequential implementation for large sparse graphs.

The key ideas are:

- Each vertex is assigned a **discovery time (pre)** when first visited during DFS.
- The **low** and **high** values track the earliest and latest discovery times reachable from a given subtree.
- **Back edges** (edges connecting a vertex to an ancestor in the DFS tree) are identified during traversal.
- A **Disjoint Set Union (DSU)** structure is used to efficiently merge edges into biconnected components based on cases derived from DFS tree and back edge relationships.

2.2 Data Structures Used

The implementation relies on the following data structures, corresponding directly to the algorithmic framework laid out by Tarjan and Vishkin:

Data Structure	Purpose
`vector<vector<int>> g`	Adjacency list representing the undirected input graph.
`vector<vector<int>> tree`	Adjacency list representing the DFS tree constructed during traversal.
`vector<NodeInfo>`	For each vertex: discovery time (`pre`), low/high values, parent, and subtree size (`nd`).
`vector<pair<int, int>> edges, backEdges`	List of graph edges and identified back edges during DFS traversal.
`DSU (Disjoint Set Union)`	Implements union-find with path compression to efficiently merge edges into biconnected components.
`vector<int> edgeComp, mark`	Tracks the component each edge belongs to and temporary marking for ensuring unique listings of components.

Table 1: Primary Data Structures in Tarjan-Vishkin Implementation

2.3 Algorithm Implementation Details

The implementation closely follows the framework proposed by Tarjan and Vishkin, structured into the following stages:

1. **Input Parsing:**
    - The undirected graph is read from a Matrix Market (`.mtx`) file format and stored as an adjacency list for efficient traversal.
2. **Depth-First Search (DFS) Traversal:**
    - A DFS is performed on the input graph, assigning each vertex a discovery time (`pre`) as well as computing its `low` and `high` values.
    - During traversal, a DFS tree is constructed by recording tree edges, while back edges are separately identified as edges connecting a vertex to an ancestor with a lower discovery time.
3. **Component Identification via Merging Rules:**
    - Based on Tarjan and Vishkin's merging criteria, edges are grouped into biconnected components using the following cases:
        - **Case (i) — Back Edges:** Each back edge immediately causes the two connected vertices to be merged within the disjoint set, as back edges inherently form cycles.
        - **Case (ii) — Tree Edge Conditions:** For each tree edge (v, w) where v is the parent of w in the DFS tree, if `low[w]` $\leq$ `pre[v]`, the edge is merged into the same component, signifying that a cycle exists through this subtree back to an ancestor of v.

- **Case (iii) — Unrelated Subtrees:** If two subtrees rooted at vertices w and w' under the same parent v have `low[w]` and `low[w']` both less than or equal to `pre[v]`, their respective edges are merged, indicating indirect connection via other paths in the graph.

4. **Disjoint Set Union (DSU) Operations:**
   - A union-find data structure with path compression is used to efficiently merge edges into biconnected components according to the cases above. This allows near-constant time merging and component lookups during traversal.
5. **Component Enumeration:**
   - After all merges are complete, the final biconnected components are enumerated by iterating through edges and grouping them based on their DSU parent. Temporary marking is used to ensure that each vertex appears only once within each component listing.

2.4 Time and Space Complexity Analysis

This section presents an analysis of the time and space complexities of the Tarjan-Vishkin algorithm implementation, focusing on key components, loops, and data structures.

2.4.1 Time Complexity Analysis

The algorithm consists of two main phases: the DFS traversal and the component identification phase. Let n be the number of vertices and m be the number of edges.

1. **DFS Traversal:** The DFS traversal explores each vertex and edge once. For each vertex, we perform a constant amount of work (updating discovery and low values, processing edges). Thus, the time complexity of the DFS traversal is:

$$O(n + m)$$

2. **Component Identification:** The component identification phase involves processing each edge to check the conditions for merging edges into biconnected components. This step includes:
   - **Case (i) — Back Edges:** Each back edge is processed in constant time.

- **Case (ii) — Tree Edge Conditions:** For each tree edge, the algorithm checks the low values and possibly merges components in constant time, so each edge is processed once.
- **Case (iii) — Unrelated Subtrees:** This step involves checking the relationships between vertices in different subtrees. Each edge is processed once for this case as well.

Thus, the time complexity for the component identification phase is:

$$O(m)$$

3. **Disjoint Set Union (DSU):** The union-find operations (union and find) are optimized with path compression and union by rank, yielding an amortized time complexity of $O(\alpha(n))$ per operation, where α is the inverse Ackermann function, which grows extremely slowly and is practically constant for all reasonable values of n.

   The total time for union-find operations is therefore:

$$O(\alpha(n) \cdot m)$$

   Combining the complexities, the overall time complexity of the algorithm is:

$$O(n + m)$$

   since the union-find operations are effectively constant time for large inputs.

2.4.2 Space Complexity Analysis

The space complexity of the algorithm is determined by the storage of various data structures used during execution.

1. **Graph Representation:** The graph is represented as an adjacency list, which requires $O(n + m)$ space. Each vertex stores a list of its adjacent vertices, leading to a total space complexity of $O(n + m)$.

2. **DFS Tree:** The DFS tree is represented by a vector of adjacency lists for each vertex, which also requires $O(n + m)$ space, since each vertex and edge is included.

3. **Node Information:** The `NodeInfo` array stores discovery times, low/high values, subtree sizes, and parent pointers for each vertex. Since each vertex requires constant space, the total space complexity for this array is:

$$O(n)$$

4. **Edge and Back Edge Lists:** The edge and back edge lists store all edges in the graph. Each edge is stored once in these lists, so their combined space complexity is:

$$O(m)$$

5. **Disjoint Set Union (DSU):** The DSU structure uses two arrays (parent and size) to manage component merging. Each array has size n, so the space complexity for the DSU is:

$$O(n)$$

6. **Auxiliary Arrays:** The arrays `edgeComp` and `mark` are used for marking components and ensuring unique component listings. Both arrays have size n, contributing:

$$O(n)$$

7. **Final Component Enumeration:** The final biconnected components are stored in a vector of vectors. Each component contains a subset of the vertices and edges, leading to a total space complexity of $O(n + m)$ for the biconnected components list.

Combining the space requirements, the total space complexity of the algorithm is:

$$O(n + m)$$

2.5 Summary

This implementation effectively adapts the principles from Tarjan and Vishkin's parallel biconnected component (BCC) algorithm into a sequential framework, optimized for handling large undirected sparse graphs. Through the use of depth-first search (DFS) for discovery times and low values, along with subtree interval analysis and back edge detection, the algorithm systematically identifies biconnected components. By leveraging the union-find (disjoint set union) structure, it efficiently merges edges into components, ensuring both correctness and performance. This approach provides a scalable and robust solution for real-world graph datasets, facilitating accurate component identification even in large, sparse networks.

3 Slota-Madduri Algorithm

3.1 Algorithm Overview

The Slota-Madduri algorithm for detecting articulation points in an undirected graph is based on a series of breadth-first search (BFS) traversals. It

identifies articulation points by analyzing the connectivity of nodes in the graph and the relationships between their neighbors. Specifically, it checks if removing a node disconnects any of its children in the DFS tree.

This implementation closely follows the principles outlined in the work of Slota and Madduri, with adaptations to handle sequential execution for large, sparse graphs. The algorithm consists of several key stages, each performing critical checks and operations to efficiently identify articulation points.

The main concepts of the algorithm are:

- **BFS Traversals**: The algorithm uses BFS to explore nodes and check connectivity by excluding a specific node and verifying if the remaining graph is still connected.
- **Exclusion Condition**: The algorithm checks if excluding a vertex and traversing its subtree can still allow the traversal of other children within the same component.
- **Root Articulation Points**: The algorithm identifies whether the root of a DFS tree is an articulation point by analyzing if its removal disconnects its children from each other.
- **Stack Implementation**: For managing and merging components, the algorithm relies on finding the nodes which are part of current component using the edges in the stack.

3.2 Data Structures Used

The implementation of the Slota-Madduri algorithm relies on several data structures to manage the graph's properties and facilitate the efficient identification of articulation points. These data structures are defined as follows:

Data Structure	Purpose
`vector<vector<int>> adj`	Adjacency list representing the undirected input graph, storing the neighbors for each vertex.
`vector<int> Parent`	Parent array used during BFS to track the parent of each vertex in the DFS tree.
`vector<int> Level`	Level array storing the level (distance from the root) of each vertex during BFS traversal.
`vector<int> stamp`	Array that marks the traversal timestamps during BFS, ensuring proper isolation of BFS executions.
`vector<bool> visited`	Boolean array to keep track of whether a vertex has been visited during the traversal.
`vector<bool> done`	Boolean array to keep track of whether a vertex can reach higher level without its parent.
`vector<bool> isArt_slota`	Array to mark whether a vertex is an articulation point according to Slota-Madduri's conditions.
`queue<int>`	Used for performing BFS to explore the graph and identify articulation points.
`stack<int>)`	For finding the edges which are part of the current bcc set.

Table 2: Primary Data Structures in Slota-Madduri Implementation

These data structures facilitate the detection of articulation points by ensuring efficient graph traversal, connectivity checks, and component management. The BFS operations and the management of subgraph components are central to the algorithm's ability to scale for large graphs.

3.3 Algorithm Implementation Details

The implementation of the Slota-Madduri algorithm closely follows the structure and methodology proposed in the original paper, and is divided into the following stages:

1. **Input Parsing:**
    - The undirected graph is read from a Matrix Market (`.mtx`) file format. The graph is then stored as an adjacency list (`adj`) for efficient traversal during the BFS and DFS phases.
2. **Breadth-First Search (BFS) Traversal:**
    - The algorithm starts by performing a BFS traversal from each vertex in the graph. During BFS, the discovery times for each vertex are tracked using a `stamp` array. This is used to ensure that vertices are processed efficiently during the subsequent checks for articulation points.
    - The BFS function (`exclusion_bfs`) checks whether any vertex, when excluded, prevents the traversal from reaching other vertices. It returns `true` if the vertex can be excluded without disconnecting its neighbors, and `false` if excluding the vertex disconnects the graph.
3. **Root Articulation Point Identification:**
    - Special handling is done for the root vertex. The algorithm checks whether the root itself is an articulation point by looking at its children in the DFS tree. If any child cannot reach another child when the root is removed, the root is marked as an articulation point.
    - This is achieved through in a similar way as above just that instead of seeing if reaches higher level its sees whether it can reach its neighbour.
4. **Articulation Point Check for Non-root Vertices:**
    - For each non-root vertex, the algorithm checks if it is an articulation point by examining its children in the DFS tree. If any child cannot reach any other child (when the vertex is excluded), the vertex is marked as an articulation point.

- This is done by using BFS to check connectivity between the children and rest of the graph after excluding the current vertex from the graph.

5. **Articulation Point Storage:**
   - The articulation points identified during the BFS checks are stored in the `isArt_slota` array. Each element in this array corresponds to whether a vertex is an articulation point (true) or not (false).
6. **Memory Reporting:**
   - The algorithm reports the memory usage of key data structures. This includes the adjacency list (`adj`), the arrays for storing parent-child relationships (`Parent`, `Level`), the `stamp` array, `Done` and `visited` array for keeping track of the nodes and the boolean array for articulation points (`isArt_slota`).

3.4 Time and Space Complexity Analysis

This section presents an analysis of the time and space complexities of the Slota-Madduri algorithm implementation, focusing on key components, loops, and data structures.

3.4.1 Time Complexity Analysis

The algorithm consists of several stages, including BFS traversal, articulation point identification, and final processing. Let n be the number of vertices and m be the number of edges.

1. **BFS Traversal:** The BFS traversal explores each vertex and edge once. For each vertex, we perform a constant amount of work (updating discovery times, processing edges, and checking articulation points). The BFS operation is performed for each vertex in the graph. Thus, the time complexity of the BFS traversal is:

$$O(n + m)$$

2. **Root Articulation Point Identification:** The algorithm checks whether the root vertex is an articulation point by examining its children in the DFS tree. For each child, the algorithm performs a BFS to check connectivity. This involves iterating through the children and performing BFS for each child, which results in a time complexity of:

$$O(n + m)$$

for processing the root vertex's articulation points.

3. **Non-root Articulation Point Check:** For each non-root vertex, the algorithm checks if it is an articulation point by examining its children in the DFS tree. Similar to root articulation point identification, BFS is performed to check connectivity between children when the vertex is excluded. This step has a time complexity of:

$$O(n + m)$$

4. **Final Articulation Point Storage:** After identifying articulation points, the algorithm stores the results in the `isArt_slota` array. This operation is done in constant time for each vertex. Therefore, the time complexity for this step is:

$$O(n)$$

5. **Biconnected Components Computation:** The articulation points are used by the algo to find the bcc of the graph. This step involves each of the tree edge and each of the vertices of the graph, so its time complexity will be:

$$O(n + m)$$

Combining all the stages, the overall time complexity of the Slota-Madduri algorithm is:

$$O(n + m)$$

since the BFS and articulation point identification steps dominate the time complexity and each BFS is done with a constraint so each edge is visited a fixed number of times.

3.4.2 Space Complexity Analysis

The space complexity of the algorithm is determined by the storage of various data structures used during execution.

1. **Graph Representation:** The graph is represented as an adjacency list, which requires $O(n + m)$ space. Each vertex stores a list of its adjacent vertices, leading to a total space complexity of $O(n + m)$.

2. **BFS Traversal:** During the BFS traversal, the `stamp` array is used to store the discovery times of the vertices. This array has size n, so its space complexity is:

$$O(n)$$

3. **Articulation Point Information:** The `isArt_slota` array stores whether each vertex is an articulation point (true or false). This array also has size n, contributing a space complexity of:

$$O(n)$$

4. **Auxiliary Arrays:** The algorithm uses several other auxiliary arrays, including `Parent`, `Level`,`Done`,`visited`, and temporary marking arrays for BFS processing and articulation point identification. Each of these arrays has size n, resulting in an additional space complexity of:
$$O(n)$$

5. **Final Component Enumeration:** The final articulation points and results are stored in arrays, which require $O(n)$ space to store the final output.

Combining all space requirements, the total space complexity of the Slota-Madduri algorithm is:
$$O(n + m)$$
since the space for the graph and auxiliary arrays dominates the space complexity.

3.5 Summary

This implementation of the Slota algorithm efficiently detects biconnected components (BCCs) in undirected graphs using a combination of depth-first search (DFS), level-based traversal, and articulation point detection. The approach identifies articulation points by examining subtree structures and back edges, while systematically classifying vertices into biconnected components. During DFS traversal, the algorithm assigns discovery times and low values to each vertex, and checks for articulation points by exploring each vertex's children in the DFS tree. The exclusion BFS is used to determine if an ancestor can still be reached, ensuring accurate identification of articulation points. The algorithm operates efficiently without requiring a union-find data structure, focusing instead on depth-first traversal and direct component detection. With a time complexity of [O (n + m)], this solution is scalable for large, sparse graphs, making it an effective choice for graph analysis tasks such as network connectivity and component detection in large-scale real-world datasets.

4 Experimental Design

The experimental evaluation of the both algorithms for detecting biconnected components (BCCs) is designed to compare the performance in terms of memory usage and execution time. The following setup outlines the key components of the experimental framework:

- **Datasets:** The experiments are conducted using standard Matrix Market graph files (`.mtx` format), which contain real-world graph data

from the University of Florida Sparse Matrix Collection. The datasets vary in terms of the number of vertices (n) and edges (m), and are classified based on edge density. We define a graph as *sparse* if its edge density is below a threshold (0.005) and as *dense* if its edge density is above this threshold.

- **Metrics:** The following metrics are used to evaluate the performance of the algorithm:
  - **Total memory usage:** Measured in bytes to determine the memory consumption of the algorithm during execution. This metric is crucial for understanding the scalability of the algorithm with respect to graph size.
  - **Execution time:** Measured in seconds to assess the speed of the algorithm. This is particularly important for large graphs, where efficiency is critical.

The experiments will classify the datasets into sparse and dense graphs using the density threshold and compare the algorithm's performance for both types of graphs. The total memory usage and execution time will be recorded for each graph based on the number of edges.

5 Datasets Overview

The experimental evaluation was conducted on a diverse collection of real-world graph datasets from the University of Florida Sparse Matrix Collection. The table below summarizes the number of nodes and edges in each dataset used for benchmarking the algorithms.

Nodes (n)	Dataset	Edges (m)
1138	1138_bus.mtx	2596
5374	TSOPF_RS_b162_c1.mtx	205399
6001	exdata_1.mtx	1137751
7224	TSOPF_RS_b9_c6.mtx	54082
84414	TSOPF_FS_b300_c3.mtx	6578753
9506	nemeth01.mtx	367280
9506	nemeth02.mtx	202157
9506	nemeth03.mtx	202157
9506	nemeth04.mtx	202157
9506	nemeth05.mtx	202157
9506	nemeth06.mtx	202157
9506	nemeth07.mtx	202159
9506	nemeth08.mtx	202161
9506	nemeth09.mtx	202506
9506	nemeth10.mtx	205477
9506	nemeth11.mtx	208885
9506	nemeth12.mtx	228162
9506	nemeth13.mtx	241989
9506	nemeth14.mtx	252825
9506	nemeth16.mtx	298259
9506	nemeth17.mtx	319563
9506	nemeth18.mtx	352370
9506	nemeth19.mtx	413904
9506	nemeth20.mtx	490688
9506	nemeth21.mtx	591626
9506	nemeth22.mtx	684169
9506	nemeth23.mtx	758158
9506	nemeth24.mtx	758028
9506	nemeth25.mtx	760632
9506	nemeth26.mtx	760633
10000	G67.mtx	20000
11948	bcsstk18.mtx	80519
14098	TSOPF_RS_b39_c7.mtx	252446
14340	human_gene2.mtx	9041364

Table 3: Dataset sizes (part 1).

Nodes (n)	Dataset	Edges (m)
15374	TSOPF_RS_b162_c3.mtx	610299
18696	TSOPF_RS_b678_c1.mtx	4396289
20374	TSOPF_RS_b162_c4.mtx	812749
22283	human_gene1.mtx	12345963
25626	TSOPF_RS_b2052_c1.mtx	6761100
29214	TSOPF_FS_b300.mtx	2203949
35696	TSOPF_RS_b678_c2.mtx	8781949
37365	Chevron1.mtx	330633
38098	TSOPF_RS_b39_c19.mtx	684206
38120	TSOPF_RS_b2383.mtx	16171169
38120	TSOPF_RS_b2383_c1.mtx	16171169
41374	jan99jac120sc.mtx	260202
45101	mouse_gene.mtx	14506196
5374	TSOPF_RS_b162_c1.mtx	205399
56814	TSOPF_FS_b300_c2.mtx	4391071
60098	TSOPF_RS_b39_c30.mtx	1079986
76480	shyy161.mtx	329762
84414	TSOPF_FS_b300_c3.mtx	6578753
1465137	StocF-1465.mtx	11235263
921632	t2em.mtx	4590832

Table 4: Dataset sizes (part 2).

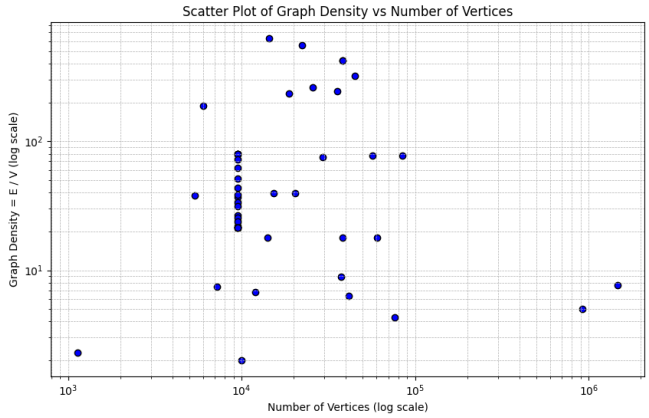

Figure 1: Density of graphs used

6 Performance Comparison

6.1 Time based comparision

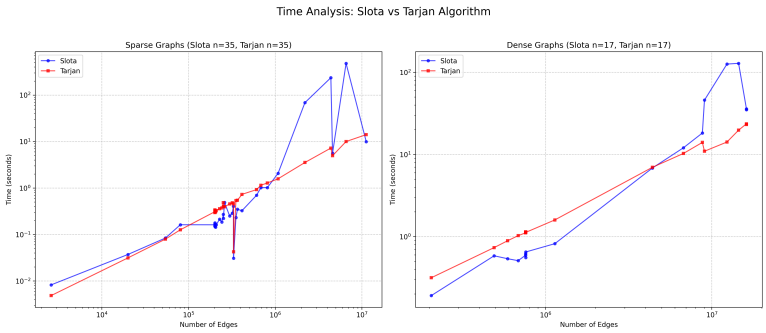

Figure 2: RunTime comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different graph sizes.

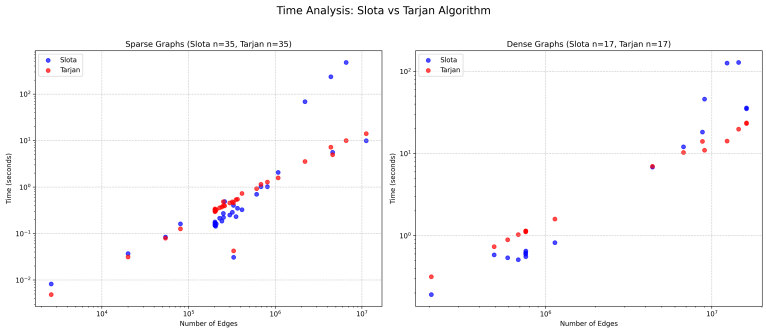

Figure 3: RunTime ScatterPlot between Tarjan-Vishkin and Slota-Madduri algorithms across different graph sizes.

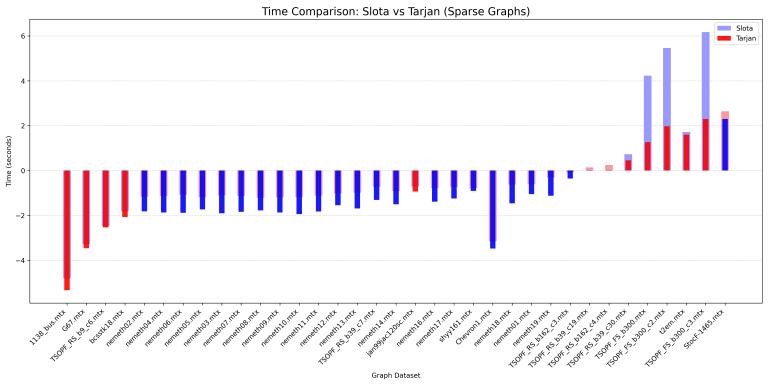

Figure 4: Logarithimic RunTime comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different sparse graph.

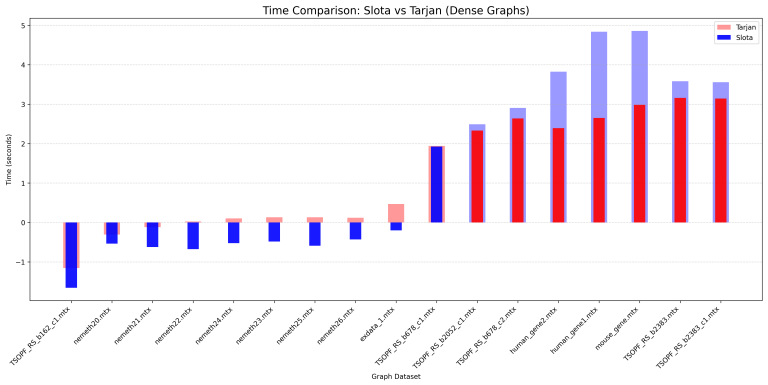

Figure 5: Logarithimic RunTime comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different dense graph.

6.2 Time based comparision

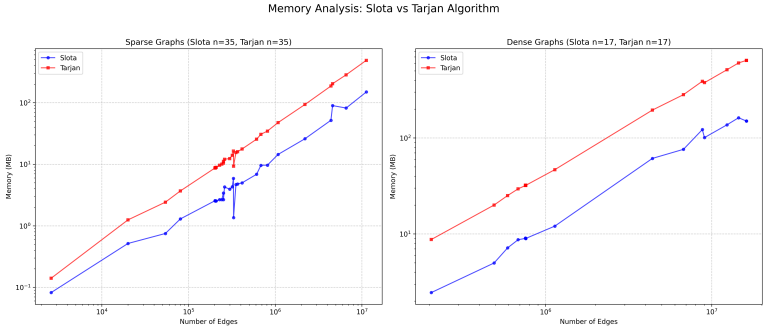

Figure 6: Memory comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different graph sizes.

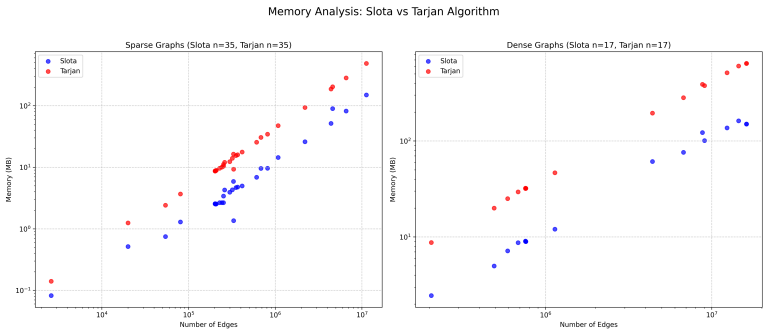

Figure 7: Memory ScatterPlot between Tarjan-Vishkin and Slota-Madduri algorithms across different graph sizes.

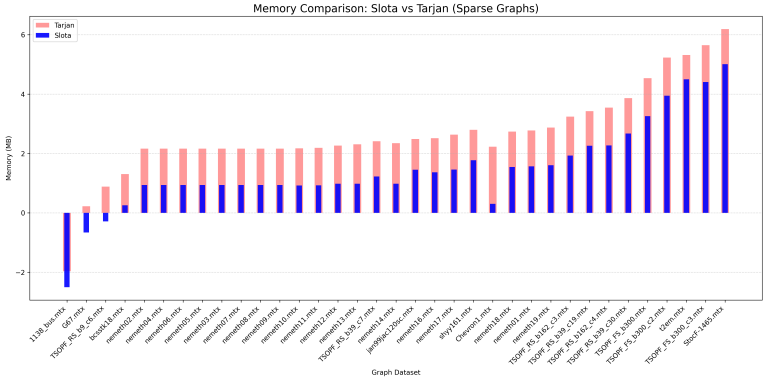

Figure 8: Logarithimic of Memory comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different sparse graph.

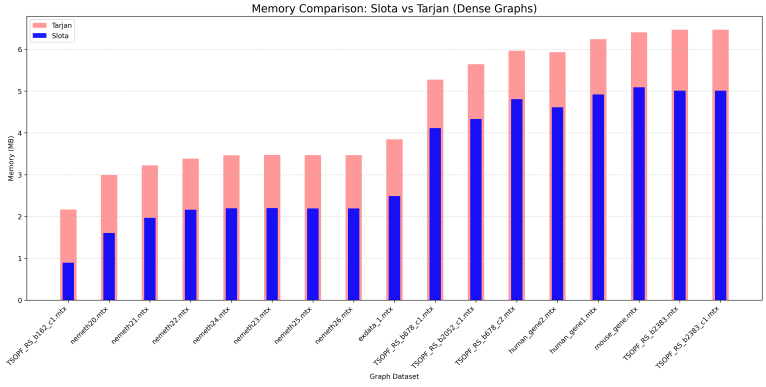

Figure 9: Logarithimic Memory comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different dense graph.

6.3 Equal number of Vertices(nemeth dataset)

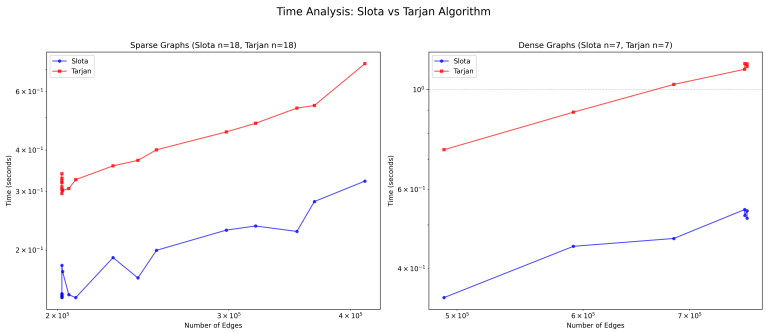

Figure 10: RunTime comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different graph with different numbers of edges with same number of vertices.

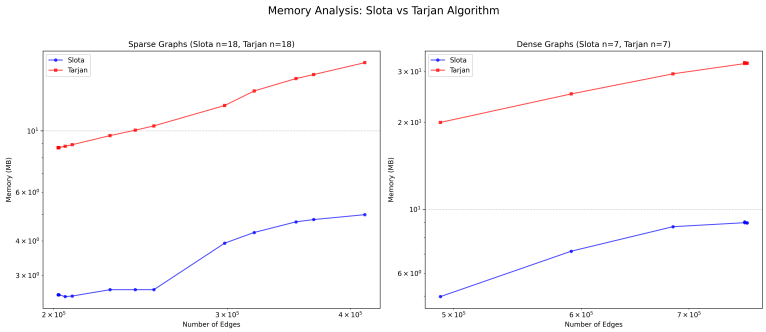

Figure 11: Memory comparison between Tarjan-Vishkin and Slota-Madduri algorithms across different graph with different numbers of edges with same number of vertices.

7 Inference

Based on the observed performance trends and the analysis of the underlying code structures for both the Tarjan-Vishkin and Slota-Madduri algorithms, several key conclusions can be drawn:

- **Time Performance:** The Tarjan-Vishkin implementation consistently outperforms the Slota-Madduri algorithm in terms of execution time across most graph sizes. This advantage stems from its efficient single-pass depth-first traversal, predictable control flow, and optimized data structure usage, particularly the use of union-find for component merging. In contrast, the Slota-Madduri algorithm's BFS-

based approach, with repeated exclusion traversals, introduces redundant operations that scale poorly on larger or denser graphs.

- **Memory Usage:** While the Tarjan-Vishkin algorithm generally consumes more memory due to its additional per-node tracking arrays and explicit back edge storage, this trade-off provides substantial benefits in runtime performance. The Slota-Madduri algorithm is relatively more memory-efficient since it recomputes information on demand and avoids maintaining extra data structures, albeit at the cost of time efficiency.

- **Scalability:** The Tarjan-Vishkin approach exhibits better scalability for large, sparse, or irregular graphs, with its runtime remaining more stable relative to graph density and topology. Slota's implementation, while viable for smaller or memory-constrained contexts, suffers from significant performance degradation in larger graphs due to its high number of BFS traversals in certain structures.

- **Algorithm Trade-offs:** The comparative study highlights a classic space-time trade-off scenario. Tarjan-Vishkin sacrifices additional memory for faster and predictable runtime performance, while Slota-Madduri conserves memory by recomputing intermediate states, which results in increased and variable execution time.

- **Recommendation:** For applications prioritizing runtime efficiency and predictable scaling behavior on large, sparse networks (e.g., infrastructure networks, biological systems, social graphs), the Tarjan-Vishkin implementation is preferable. However, in scenarios where memory availability is a critical constraint and runtime predictability is secondary, the Slota-Madduri approach remains a reasonable alternative.

8 Conclusion

This project presented a comparative study of two distinct algorithms for detecting biconnected components in undirected graphs: the Tarjan-Vishkin algorithm and the parallel Slota-Madduri approach. Through rigorous experimental evaluation across a diverse set of graph datasets varying in size and density, the study provided insights into the practical trade-offs between time efficiency and memory consumption inherent in these algorithms.

The Tarjan-Vishkin algorithm demonstrated superior performance in terms of execution time and scalability, especially on large, sparse, and irregular graphs. Its optimized depth-first search traversal and effective use of auxiliary data structures like disjoint set unions enabled predictable and

efficient processing. However, this came at the cost of increased memory usage due to additional tracking arrays and explicit edge categorization.

In contrast, the Slota-Madduri algorithm, with its BFS-based structure and minimal auxiliary storage, proved to be more memory-efficient. Nevertheless, its reliance on repeated exclusion traversals introduced significant runtime overheads, particularly as graph size and complexity increased, leading to less predictable scaling behavior.

The comparative analysis emphasized a classical algorithmic trade-off: sacrificing memory to gain speed or conserving memory at the cost of execution time. The study concludes that the Tarjan-Vishkin algorithm is better suited for applications demanding fast and reliable performance on large-scale graph data, while the Slota-Madduri implementation remains a viable option in memory-constrained environments or for moderately sized graphs where runtime is less critical.

Future work could explore further optimizations, parallelization strategies for Tarjan's approach, and hybrid techniques that adapt algorithmic behavior based on graph properties to achieve balanced performance across varied application contexts.