k-edge Connected Components in Graphs

by

Azadehsadat Mousavi
B.Sc.(Software Engineering), Amirkabir University of Technology, 2013
M.Sc.(Software Engineering), Sharif University of Technology, 2018

Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science
of The University of Victoria

k-edge Connected Components in Graphs

by

Azadehsadat Mousavi
B.Sc.(Software Engineering), Amirkabir University of Technology, 2013
M.Sc.(Software Engineering), Sharif University of Technology, 2018

Supervisory Committee

Dr. Venkatesh Srinivasan, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Co-supervisor
(Department of Computer Science)

ABSTRACT

A k-edge-connected component (KECC) is a group of vertices in a graph $G(V, E)$ where every pair of vertices has at least k edge-disjoint paths between them. This research delves into the detection and analysis of KECCs within graphs, presenting algorithms and thoroughly evaluating their performance.

The study introduces algorithms tailored to find KECCs for a given value of k. One such algorithm, known as the fix-k algorithm, exhibits a time complexity of $O(V^2.E^2)$. By leveraging various maximum flow algorithms and exploiting properties of k-edge connectivity, we optimize this algorithm to perform E times faster, resulting in a new approach with time complexity of $O(V^2.E)$. Further enhancements are achieved through randomization, leading to an algorithm with an improved time complexity of $O(V^3.ln^{O(1)}(V))$. Additionally, we identify another algorithm with a time complexity of $O(V.E^{1+o(1)})$.

Moreover, algorithms are proposed to find all KECCs in a graph $G(V, E)$ for all values of $1 \leq k \leq V$. It is demonstrated that for any graph $G(V, E)$, there exists a threshold value h, where for all $k \geq h$, KECCs are singletons. An efficient algorithm is devised to determine this h in $O(log(V))$ time, which significantly impacts the all-k problem, allowing us to focus solely on finding KECCs for $1 \leq k < h$. This approach yields an algorithm with a time complexity of $O(h * O(Fixkalgorithm))$ to find all KECCs. Additionally, another algorithm with a time complexity of $O(V^2.E^2)$ is presented, which can be further optimized to achieve time complexities of $O(V^2.E)$, $O(V^3.ln^{O(1)}(V))$, and $O(V.E^{1+o(1)})$.

The insights gained from meticulous implementation and experimentation shed light on the behavior of these algorithms across various graph structures and scenarios.

Keywords: k-edge-connected components, KECC, graph algorithms, edge connectivity, maximum flow, minimum cut.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I am deeply grateful to the following individuals and institutions:

My family Their unwavering love, support, and encouragement have been the cornerstone of my academic journey. Without their belief in my potential and their steadfast encouragement, this research would not have been possible.

My supervisor I extend my sincere appreciation to Dr.Venkatesh Srinivasan for their invaluable guidance, support, and patience throughout this research endeavor. Their mentorship has not only shaped my academic trajectory but also broadened my intellectual horizons.

My co-supervisor I am grateful to Dr.Alex Thomo for their insightful feedback, support, continuous encouragement, and patience.

University of Victoria and my supervisors I express my gratitude to the University of Victoria and my supervisors for their collaborative efforts, funding, and support.

Chapter 1

Introduction

1.1 Background and Context

Graph theory, a fundamental branch of computer science and mathematics, focuses on the study of relationships and connections between entities. In graphs, entities are represented as vertices, and relationships are depicted by edges, providing a versatile framework for modeling complex systems across various domains. From computer networks and social interactions to transportation systems and biological networks, graphs serve as powerful tools for analysis and understanding.

Connectivity, a central concept in graph theory, plays a crucial role in deciphering the resilience, efficiency, and structure of these systems. It is a pivotal problem in theoretical computer science with numerous real-world applications spanning communication, transportation, production, scheduling, network security, computational biology, and power engineering.

For instance, in social networks, understanding connectivity among individuals is essential for assessing closeness and influence. In many practical scenarios, such as social networks, the focus is often on analyzing connectivity within smaller groups rather than the entire network.

A k-edge-connected component, a subgraph where every pair of vertices is connected by at least k edge-disjoint paths, is particularly significant. Edge connectivity contributes to fault tolerance, network reliability, and communication pathways in various applications. For example, in communication networks, k-edge-connected components ensure robust data transmission, even in the event of link failures or disruptions.

In the following section, we will delve into the main contributions of this study.

1.2 Contributions

The research makes significant contributions in the following areas:

1. Explored, developed and evaluated a range of algorithms aimed at identifying k-edge-connected components (KECCs) within graphs.

2. Proposed and implemented an algorithm (algorithm 3) to identify KECCs in graphs for a given k. The approach involved pre-calculating edge connectivity using a maximum flow algorithm for each vertex pair and subsequently detecting KECCs using depth-first search (DFS). The algorithm achieves a time complexity of $O(V^2 \times O(\text{Max flow algorithm}))$, leveraging the EdmondKarp algorithm [7](algorithm 2), resulting in a total running time of $O(V^3 \times E^2)$.

3. Implemented the fix-k algorithm (algorithm 5) to compute KECCs for a specific k in $O(V \times O(\text{Max flow algorithm}))$. By constructing a forest over the graph, this algorithm identifies rooted trees for each KECC. Leveraging the EdmondKarp algorithm [7], the fix-k algorithm achieves a time complexity of $O(V^2 \times E^2)$, providing a significant improvement in speed compared to previous approaches.

4. Introduced an optimization technique leveraging the Ford and Fulkerson algorithm [9](algorithm 1) to enhance the efficiency of the fix-k algorithm. This optimization results in a substantial speedup, reducing the time complexity to $O(V^2 \times E)$.

5. Utilized the Karger and Stein algorithm [14](algorithm 6) to further enhance the fix-k algorithm, achieving a time complexity of $O(V^3 \times \log^{O(1)}(V))$ through randomization.

6. Theoretically improved the time complexity of the fix-k algorithm to $O(V \times E^{1+o(1)})$ using recent advancements in maximum flow algorithms[5].

7. Introduced, designed, and implemented algorithms to detect k-edge-connected components (KECCs) for all $1 \leq k \leq V$ in a graph $G(V, E)$ with V vertices and E edges.

8. Demonstrated the existence of a critical value h for any graph $G(V, E)$, where $1 \leq h \leq V$, such that all k-edge-connected components become singletons for $k \geq h$. Developed an efficient algorithm (algorithm 8) to identify h in $O(\log(V))$ time. This discovery paved the way for devising an algorithm to identify all KECCs in graphs, achieving a time complexity of $O(h \times O(\text{fix-}k \text{ algorithm}))$.

9. Introduced and implemented the all-k algorithm, an approach to identify all k-edge-connected components (KECCs) in graphs. This algorithm [26] leverages an Auxiliary graph to store the connectivities of $V-1$ pairs of vertices, effectively representing a weighted spanning tree over $G(V, E)$. Algorithm 9 provides the pseudocode for constructing the auxiliary graph, with a time complexity of $O(V \times \text{Maxflow Algorithm})$ or $O(V^2 \times E^2)$, utilizing the EdmondKarp algorithm for maximum flow implementation. Once the auxiliary graph is constructed in $O(V^2 \times E^2)$ time, the all-k algorithm efficiently identifies KECCs for any query k in linear time (as shown in Algorithm 10).

10. Demonstrated that optimization techniques used in chapter 2 to enhance the fix-k algorithm can be applied to the all-k algorithm. Resulting in new algorithms with time complexities of $O(V^2 \times E)$, $O(V^3 \times \log^{O(1)}(V))$, and $O(V \times E^{1+o(1)})$.

11. Conducted comprehensive experiments to analyze the features and performance of the fix-k algorithm across various graph datasets. The experiments revealed patterns in running time and KECC counts as k increases in graph $G(V, E)$, shedding light on algorithm behavior and efficiency.

12. Investigated the impact of increasing vertex count on the fix-k algorithm's performance, observing corresponding increases in both h and running time for $k = h$.

13. Conducted comprehensive experiments to analyze the features and performance of the all-k algorithm across various graph datasets.

14. Conducted comparative experiments between the all-k and fix-k algorithms for solving all-k and fix-k problems. Results confirmed expectations, with the all-k algorithm demonstrating superiority for the all-k problem, while both algorithms exhibited similar running times for the fix-k problem.

In the subsequent section, key terminologies utilized throughout the report will be introduced.

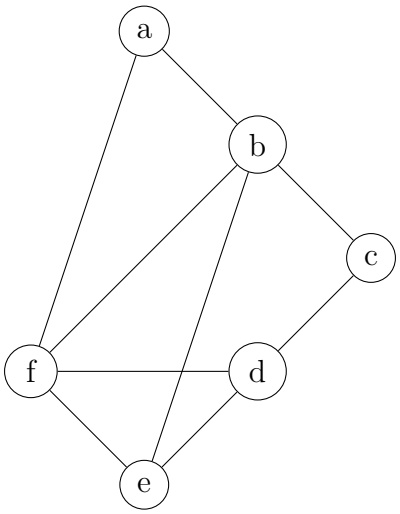

Figure 1.1: Undirected graph with 6 vertices: a, b, c, d, e, f.

1.3 Key Concepts and Definitions

This section contains important definitions and notations used throughout various parts of this report.

Definition 1.3.1 (Directed Graph)**.** A directed graph, or digraph, is a type of graph in which edges have direction. Each edge in a directed graph, denoted as (u, v), represents a directed connection from vertex u to vertex v, where the direction matters. This means that the edge (u, v) is distinct from the edge (v, u). In contrast, undirected graphs do not specify directionality for edges.

Figure 1.2 illustrates a directed graph with 3 vertices, while Figure 1.1 illustrates an undirected graph with six vertices.

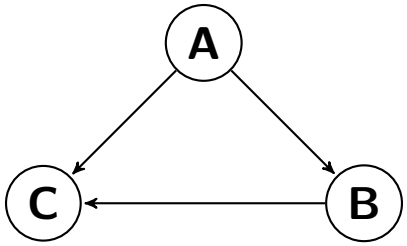

Figure 1.2: Example of a directed graph.

Definition 1.3.2 (Edge-cut (X, Y))**.** In an undirected graph $G = (V, E)$, an edge-cut between two sets of vertices X and Y is defined as the set of edges $E_c \subseteq E$ such

that deleting them disconnects X from Y in G. Consequently, removing all edges in an edge-cut results in the absence of any path between X and Y, whether direct or indirect.

For instance, in Figure 1.3, the red edges represent the edge-cut between the sets X and Y.

It's worth noting that an edge-cut can also be defined between two individual vertices. In other words, if removing a set of edges disconnects two vertices u and v, we denote this set as *edge-cut*(u, v). Consequently, removing all edges in the edge-cut prevents the existence of any directed path between u and v or vice versa.

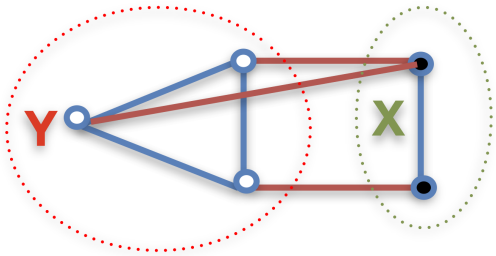

Figure 1.3: Red edges show the edge-cut between two sets of X and Y.

Definition 1.3.3 (Min-cut (X, Y))**.** A *min-cut* of sets X and Y is an edge-cut between X and Y with the minimum cardinality.

Similarly, the minimum cut can be defined for two individual vertices. *Min-cut*(u, v) represents an edge-cut between vertices u and v with the smallest number of edges compared to all possible cut edges for u and v.

Definition 1.3.4. The *edge connectivity* of a graph is the minimum number of edges that must be removed to disconnect the graph into two or more components.

Additionally, edge connectivity between two vertices u and v can be quantified by the number of disjoint paths between u and v.

Definition 1.3.5. The maximum number of edge-disjoint paths between two vertices u and v in a graph $G(V, E)$ can be denoted by $\lambda(u, v, G)$[20].

Definition 1.3.6 (k-edge-connected vertices)**.** Two vertices u and v in a graph G are said to be *k-edge-connected* if and only if there is no edge cut with a cardinality of less than k that disconnects them. Equivalently, vertices u and v are k-edge-connected if

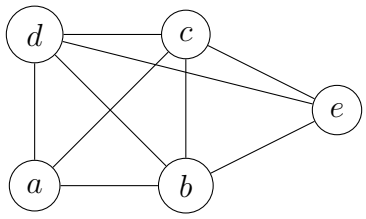

Figure 1.4: A 3-edge-connected graph.

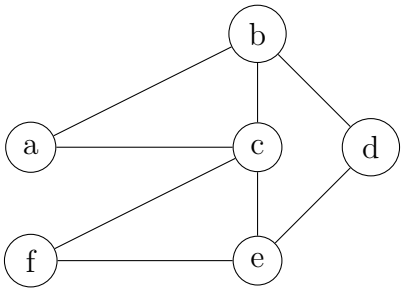

Figure 1.5: A 2-edge-connected graph.

$\lambda(u, v, G) = k$, indicating that they remain connected even after deleting fewer than k edges. Alternatively, if there are at least k edge-disjoint paths between vertices u and v, they are considered k-edge-connected. It is important to note that vertices u and v are also i-edge-connected for any $i \leq k$.

Edge connectivity can also be defined in a directed graph. If there are k_1 edge-disjoint paths from u to v and k_2 edge-disjoint paths from v to u, u and v are k-edge connected with $k = \min(k_1, k_2)$.

Definition 1.3.7 (k-edge-connected graph)**.** A graph is considered *k-edge-connected* if every pair of vertices in the graph are connected by at least k edge-disjoint paths, or in other words, if any two vertices are k-edge-connected.

Example 1.3.1. As illustrated in Figure 1.4, the depicted graph is 3-edge connected; evidently, it is also 2-edge and 1-edge connected. The set $\{ad, ac, ab\}$ represents a cut-edge with a size of 3 for this graph. Moreover, $\lambda(a, b, G) = 3$ since there exist three edge-disjoint paths ab, adb, acb between vertices a and b.

Example 1.3.2. Figure 1.5 illustrates a graph that is 2-edge connected, consequently implying that it is also 1-edge connected. $\lambda(a, d, G) = 2$ since there are two edge-disjoint paths, "abd" and "$acfed$", between vertices a and d.

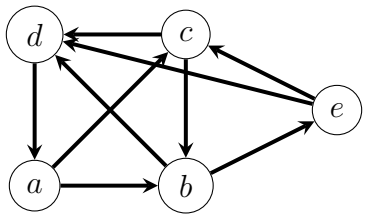

Figure 1.6: A directed connected graph.

Definition 1.3.8 (k-edge-connected component)**.** In a graph $G = (V, E)$, a vertex set $V_0 \subseteq V$ is a k-edge-connected component if it is a maximal subset of V such that any two vertices $u, v \in V_0$ are k-edge-connected in G. In other words, for any pair of vertices $(u, v) \in V_0$, we have $\lambda(u, v, G) \geq k$.

Example 1.3.3. In Figure 1.5, $\{b, c, e\}$ forms a 3-edge-connected component because for any pair of vertices $(u, v) \in \{b, c, e\}$, there exist three edge-disjoint paths between them in G, and we cannot expand the set without breaking this property. If we add any other vertex to the set, it becomes 2-edge-connected. $\{a, b, c, d, e, f\}$ is a 2-edge-connected component, which is the union of $\{a\}, \{b, c, e\}, \{d\}, \{f\}$, each of which is a 3-edge-connected component within G. Overall, any k-edge-connected component is the union of some $(k + 1)$-edge-connected components. Additionally, any vertex, as a singleton, can be considered a k-edge-connected component for any $1 \leq k \leq V$.

Example 1.3.4. Figure 1.6 illustrates a connected directed graph G, or in other words, a 1-edge-connected graph. The set $\{be\}$ forms an edge cut of G because its removal disconnects the graph. The set $\{b, c\}$ shows a 2-edge-connected component in G because there are two edge-disjoint paths from b to c ("$bdac$","bec") and two edge-disjoint paths from c to b ("cb","$cdab$"). The set $\{\{a\}, \{b, c\}, \{d\}, \{e\}\}$ represents the 2-edge-connected components, while $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$ represents the 3-edge-connected components in G. Thus, G can be seen as the union of its 2-edge-connected components, each of which is in turn the union of 3-edge-connected components.

The upcoming section will delve into the problem statement, providing a focused exploration of the challenges at hand.

1.4 Problem Statement

As previously stated, a k-edge-connected component within the graph is a subgraph where a minimum of k edge-disjoint paths exists between any pair of vertices.

There are four different problems considered in this project:

1) Problem 1: Given a graph $G = (V, E)$, the goal is to compute its edge connectivity. Chapter 2 delves into an algorithm aimed at addressing this specific problem.

2) The fix-k problem: Given a graph $G = (V, E)$ and a number k, the task is to identify the k-edge-connected components of G. Chapter 2 presents algorithms that addresses this problem.

3) The All-k problem: Given a graph $G = (V, E)$, the goal is to identify k-edge-connected components of G for all $k \geq 1$. Chapter 3 introduces algorithms to address this challenge.

4) Problem 4: Given a graph $G = (V, E)$, the objective is to determine the smallest value of k for which calling the fix-k algorithm for all $i \leq k$ is necessary to find all k-edge-connected components of G. This concept is elaborated upon in sections within chapters 3 and 4.

The subsequent section will explore the motivation behind these problems.

1.5 Motivation

The problem of finding k-edge-connected components in a graph holds several significant motivations in various domains. It contributes to the development of graph algorithms and adds to the understanding of graph structures, enriching the field of graph theory. There are many problems in the real world that can be modeled using graphs and networks. Understanding k-edge-connected components provides insights into the robustness and connectivity of networks. It helps identify substructures that are resilient against edge failures, ensuring reliable communication, transportation, and other network-related processes. In critical infrastructure systems like power grids and transportation networks, identifying k-edge-connected components aids in designing systems that can withstand multiple edge failures, minimizing the impact of disruptions and ensuring continuous operation. In modeling disease spread or information diffusion, finding KECCs helps identify regions of interaction that are robust against transmission failures, aiding in predicting and controlling the spread of diseases or information.

1.6 Related Works

A k-edge-connected component (KECC) is a maximal subset of vertices V_0 in a graph $G(V, E)$ such that for any pair of vertices $(u, v) \in V_0$, there exist at least k edge-disjoint paths between them in G. Several studies have been conducted to detect KECCs in graphs, with the concept of edge-connectivity and k-edge-connected graphs first introduced by Camelia Jordan in 1869 [13].

For $k = 1$, determining k-edge-connected components is straightforward. It involves finding connected components in the graph, which can be achieved in linear time using simple Depth First Search (DFS) or Breadth First Search (BFS) algorithms.

For $k = 2$, Tarjan presented a linear time solution based on Depth First Search (DFS) in 1974 [23].

For $k = 3$, Hopcroft and Tarjan introduced an algorithm to divide a graph into triconnected components [12], while Galil and Italiano proposed the first linear time algorithm to compute 3-edge-connected components [10]. Nagamochi and Ibaraki also presented another linear time algorithm for computing 3-edge-connected components in graphs [19].

For $k = 4$, Dinitz and Westbrook presented a new algorithm that processes a sequence of q queries (to determine whether the graph is 4-edge-connected), n Insert-Vertex, and m Insert-Edge operations (starting with an empty graph) in $O(q + m + n \log n)$ total time [6]. Georgiadis, Italiano, and Kosinas later introduced the first linear time algorithm to compute 4-edge-connected components [11].

For $k = 5$, Kosinas recently presented the first algorithm for detecting 5-edge-connected components in linear time, utilizing an oracle with size $O(V)$ for answering connectivity queries in linear time [16].

There are some other works [22, 24, 15, 25] focused on finding k-edge-connected components for small k's($k \leq 5$) in linear time.

Even and Tarjan used Minimum cut algorithms in 1975 to compute edge connectivity in graphs [8]. Minimum cut algorithms have also been employed in this project for computing edge-connectivity. Matula gave an algorithm in 1987 to find edge-connectivity of a graph in $O(V.E)$ [18] and showed that there is an algorithm with a time complexity of $O(k.V^2)$ for checking if the graph $G(V, E)$ is k-edge-connected.

In 1993, Nagamochi and Watanabe proposed an algorithm with time complexity of $O(|V|.|E|.min(k, |V|, \sqrt{E}))$ for finding k-edge-connected components in a graph

$G(V, E)$ [20].

One of the algorithms proposed for finding all KECCs in graphs is by Wang et al. [26]. They constructed an auxiliary graph in $O(F.V)$ time complexity, where F is the time complexity of a maximum flow algorithm and V is the number of vertices in the graph. They then find KECCs for any query k in linear time. In chapter 3, we implement this algorithm using the EdmondKarp algorithm, resulting in a time complexity of $O(V^2.E^2)$. Furthermore, we propose techniques and new algorithms to improve its time complexity.

Another definition of a k-edge-connected component considers it as a maximal induced subgraph of G wherein there exist at least k edge-disjoint paths between each pair of vertices within the induced subgraph. Papers that use this definition include [28, 1, 21, 3, 4]. Additionally, other studies have also applied this definition to identify all KECCs in graphs, as evidenced by works such as [27, 2]. In our definition of KECC, edge-disjoint paths should be inside G and can use edges outside the induced subgraph.

1.7 Outline

Chapter 2 introduces, algorithms to find k-edge connected components for any given k.

Chapter 3 introduces algorithms to find all k-edge connected components in a graph $G(V, E)$ for $1 \leq k \leq V$.

Chapter 4 presents experiments to evaluate and compare these two algorithms.

Chapter 5 explains future works and includes a conclusion.

Chapter 2

Computing k-edge-connected components in a graph

In this chapter, we delve into algorithms designed to identify k-edge-connected components within a graph for a specified k value. Throughout this project, we employ various algorithms, all of which leverage a maximum flow algorithm. Thus, we commence this chapter by outlining the pertinent definitions associated with the maximum flow-minimum cut problem.

Subsequently, we present a straightforward algorithm tailored for determining k-edge-connected-components corresponding to a given k value.

Moreover, we unveil a more intricate algorithm crafted to tackle this problem. The foundational principles of this algorithm draw from a methodology proposed by Hiroshi Nagamochi and Toshimasa Watanabe [20] in 1992. Our implemented algorithm, named the fix-k algorithm, utilizes the Edmond Karp maximum flow algorithm[7].

Lastly, we delve into techniques and novel solutions aimed at optimizing our implemented algorithm through three distinct approaches.

2.1 Maximum flow

Maximum flow is a fundamental problem in graph theory with widespread applications worldwide. Consider a network of water pipes: each pipe has a maximum capacity indicating the highest volume of water it can carry per second, alongside a current flow rate. The central question revolves around determining the maximum volume of water that can flow from a source to a sink within the network. This prob-

lem lends itself well to graph theory modeling: each pipeline corresponds to an edge in the graph, characterized by its capacity, with a flow rate assigned to each edge. Additionally, each vertex represents a junction. It's crucial that at each junction, the incoming water flow is distributed among the outgoing pipelines, with the exception of the source and sink nodes. The ultimate objective is to identify the maximum flow achievable from a source node to another designated node (the sink) in the graph.

Definition 2.1.1. In a network flow problem, a network graph refers to a directed graph denoted as $G(V, E)$, where each vertex in V represents a node, and each edge in E has a non-negative capacity denoted as c and a non-negative flow f. The flow along an edge cannot exceed its capacity, meaning: $0 \leq f(e) \leq c(e)$. Within the network, two vertices are designated as the source and the sink. The total incoming flow for any vertex equals the total outgoing flow, except for the source and sink vertices. The flow of the entire network equals the sum of all flows originating from the source and equals the sum of all flows consumed by the sink. A maximum flow is a flow that achieves the highest possible value within the network.

Example 2.1.1. The network flow depicted in Figure 2.1 comprises six vertices, with S designated as the source and T as the sink vertices. Initially, all flows are zero, and the capacities are displayed on each edge (with the left value indicating the flow and the right value indicating the capacity).

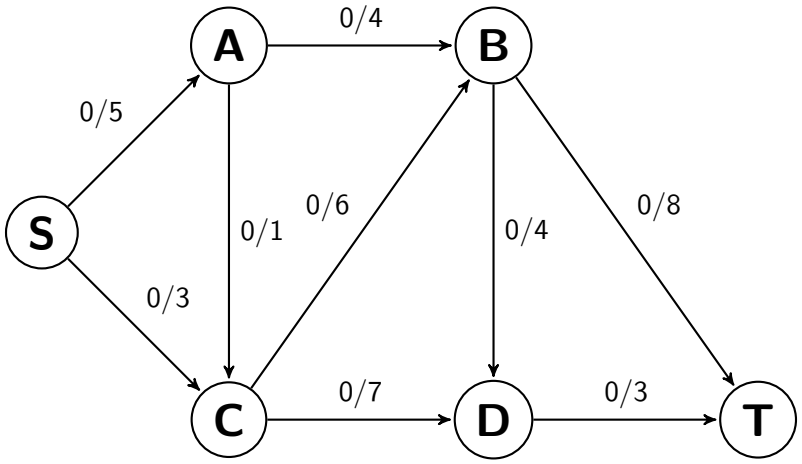

Figure 2.1: A network flow with 6 vertices. S serves as the source and T as the sink. Initially, all flows are zero, with capacities displayed on each arc.

In the maximum flow problem, the central question is: what is the maximum flow that can traverse from the source to the sink?

Example 2.1.2. Figure 2.2 illustrates the network with the maximum flow achievable, which is 8. This flow value of 8 corresponds to the sum of the flows emanating from the source vertex S and equates to the sum of flows entering the sink vertex T.

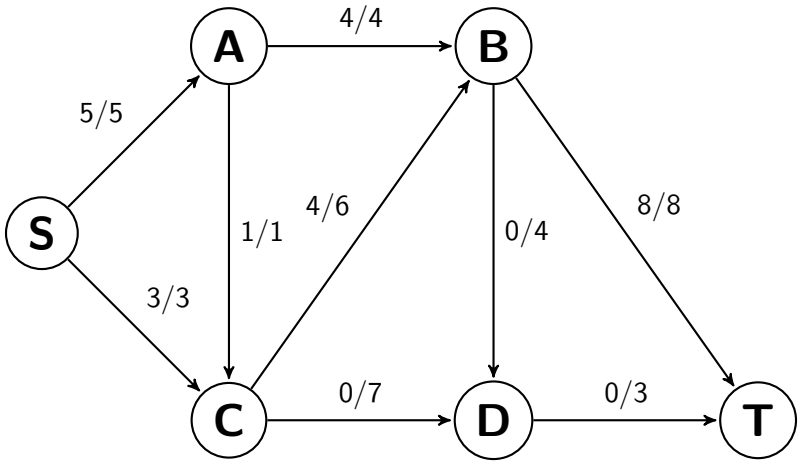

Figure 2.2: A network flow with a maximum flow of 8.

Given the significance of this problem and its applicability to various real-world scenarios, numerous methods have been devised to solve maximum flow in a graph. One of the primary methods was proposed by Ford and Fulkerson [17] in 1962. In the subsequent section, we will provide a concise illustration of this method.

2.1.1 Ford-Fulkerson method

Ford-Fulkerson, cited in [17], stands as one of the most renowned algorithms for tackling maximum flow problems. Algorithm 1 delineates the pseudocode for the Ford-Fulkerson algorithm. The algorithm's runtime is denoted as $O(E \cdot f')$, where E represents the number of edges in the graph, and f' signifies the maximum flow within the network.

Before delving into the mechanics of the algorithm, it's crucial to establish some key definitions:

Definition 2.1.2 (Residual capacity)**.** The residual capacity of an edge is determined by subtracting its current flow from its capacity. For instance, in Figure 2.2, the residual capacity of edge (C, B) is calculated as $6 - 4 = 2$. Residual capacity can also be defined for reversed edges. Here, the capacity is considered zero, and the flow is the negative value of the original flow. In other words, if (v, u) is the reversed edge of

Algorithm 1 Ford-Fulkerson Algorithm

1: Initialize flow $f(u, v) = 0$ for all edges (u, v)
2: **while** there exists an augmenting path p from s to t **do**
3: Find the bottleneck capacity $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
4: **for** each edge (u, v) in p **do**
5: Increase flow $f(u, v)$ by $c_f(p)$
6: Decrease flow $f(v, u)$ by $c_f(p)$ (if $f(v, u) > 0$)
7: **end for**
8: **end while**

(u, v), then $capacity(v, u) = 0$ and $flow(v, u) = -flow(u, v)$. For instance, for edge (B, C) (which is the reverse of edge (C, B) in Figure 2.2), the capacity is 0, and the flow is -4.

Definition 2.1.3 (Residual network)**.** A residual network for a given graph maintains the same vertices and edges as the original graph, but with a flow of 0 and residual capacities instead of the original capacities.

Definition 2.1.4 (Augmenting path)**.** An augmenting path in the residual network refers to a simple path along edges with positive residual capacities. Identifying augmenting paths within a network constitutes a fundamental step in solving the maximum flow problem, as the existence of such paths allows for flow augmentation.

The initial phase of solving the maximum flow problem using the Ford-Fulkerson method involves setting the flow of each edge to zero (as illustrated in Figure 2.1). Subsequently, the algorithm proceeds to find augmenting paths and augmenting the flow along these paths. This iterative process continues until no augmenting path can be found, indicating that the flow has reached its maximum capacity.

To elaborate further, during each iteration of augmenting the flow, an augmenting path needs to be identified. Let's denote C as the minimum capacity among all residual capacities along the path. The flow is then increased by C, which entails updating $f(u, v) += C$ and $f(v, u) -= C$. This incremental augmentation process ensures that the flow is progressively increased while adhering to the constraints imposed by the capacities of the edges.

As an illustration, any path from S to T in Figure 2.1 can serve as an augmenting path. Let's consider $S - A - B - T$ as the first augmenting path identified by the algorithm. The minimum residual capacity along this path, denoted as C, is determined as $C = \min(5, 4, 8) = 4$. Consequently, the current flow can be increased by 4. Figure 2.3 visualizes the network flow with the updated flow after augmenting the path $S - A - B - T$ with $C = 4$.

The subsequent augmenting path could be $S - A - C - B - T$, with the minimum residual capacity along this path calculated as $C = \min(5 - 4, 1 - 0, 6 - 0, 8 - 4) = 1$. Upon augmenting each edge in the path with $C = 1$, the resulting graph depicted in Figure 2.4 is attained.

The final augmenting path that can be identified by the algorithm is $S - C - B - T$, with the minimum residual capacity along this path calculated as $C = \min(3 - 0, 6 - 1, 8 - 5) = 3$. Upon augmenting each edge in the path with $C = 3$, the resulting graph

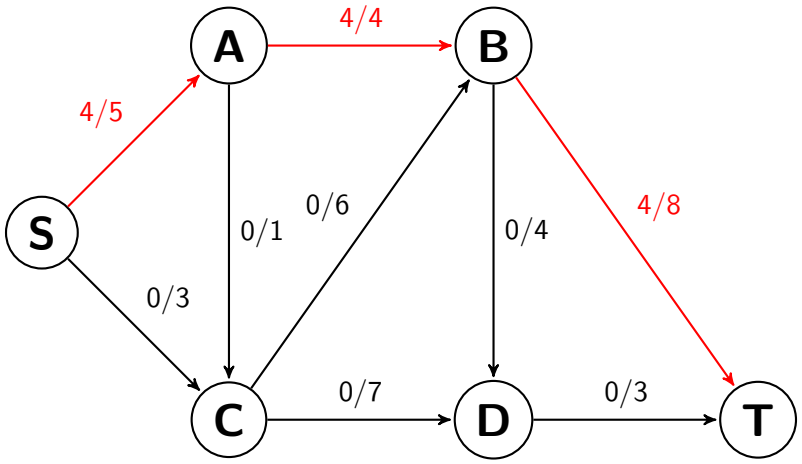

Figure 2.3: The augmenting path $S - A - B - T$ in the network flow of Figure 2.1 has a capacity $C = 4$. Consequently, 4 is added to the flow of each edge along the path.

depicted in Figure 2.5 is achieved, representing the final network flow with a maximal flow of 8. At this juncture, the algorithm concludes as no further augmenting paths can be found in the network. Additionally, the maximal flow of 8 corresponds to the sum of the maximal capacities of outgoing edges from S.

2.1.2 Edmonds-Karp algorithm

The Edmonds-Karp algorithm [7] serves as an implementation of the Ford-Fulkerson method, employing Breadth-First Search (BFS) to efficiently locate augmenting paths in a graph $G(V, E)$. By prioritizing the shortest augmenting path from source to sink, Edmonds-Karp significantly enhances the efficiency of Ford-Fulkerson, reducing its time complexity to $O(V \cdot E^2)$ compared to the worst-case complexity of $O(E \cdot f')$, where V denotes the number of vertices, E signifies the number of edges, and f' represents the maximum flow value achievable in the graph. Given its ability to handle larger graphs and potentially substantial flow values, the Edmonds-Karp algorithm stands as a valuable improvement over the original Ford-Fulkerson method, making it a preferred choice for various applications, including our current project.

The Edmonds-Karp algorithm, outlined in algorithm 2, demonstrates a method to compute the maximum flow (s, t) in a graph $G(V, E)$. Initially, the flow is set to zero (line 9). Within the while loop, the Breadth-First-Search (BFS) function identifies an augmenting path, determining the minimum capacity along the path

Algorithm 2 Max Flow(s,t) (Edmonds-Karp) Algorithm

Inputs:
$C[n \times n]$: Capacity Matrix
$G[n \times n]$: Graph
s: source
t: sink
Output:
f: maximum flow
Edmonds-Karp:
$f = 0$ {Flow is initially 0}
$F = [n \times n]$ {Residual capacity array}
while true **do**
  $minC, P =$ Breadth-First-Search(C, G, s, t, F)
  if $minC = 0$ **then**
    break
  end if
  $f = f + minC$
  $v = t$
  while $v \neq s$ **do**
    $u = P[v]$
    $F[u, v] = F[u, v] - minC$ {Reduce the residual capacity of the augmenting path}
    $F[v, u] = F[v, u] + minC$ {Increase the residual capacity of the reverse edges}
    $v = u$
  end while
end while
return f

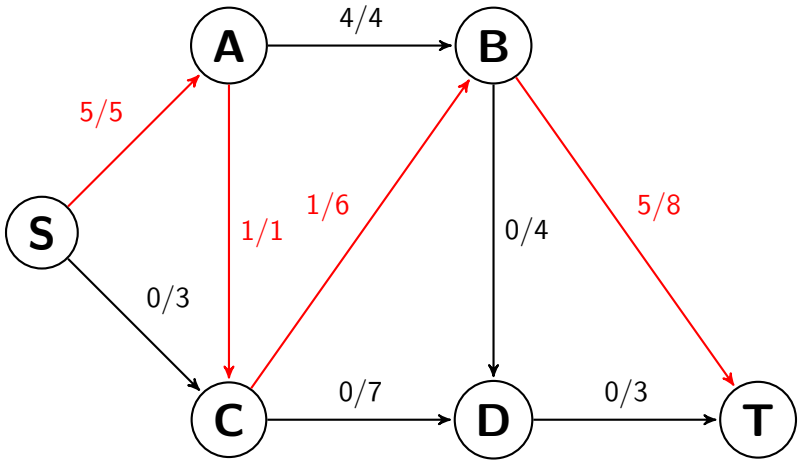

Figure 2.4: $S-A-C-B-T$ represents an augmenting path in the network flow depicted in Figure 2.3. The path has a capacity $C = \min(5-4, 1-0, 6-0, 8-4) = 1$. Consequently, 1 is added to the flow of each edge along this path.

(stored in variable $minC$), and tracks the path using an array P. Each element $P[i]$ indicates the parent of vertex i in the augmenting path. If $minC$ equals zero, indicating that the flow cannot be increased and no augmenting path exists, the loop breaks. Subsequently (line 16), the current flow is augmented by $minC$. Lines 17 to 23 update the residual capacity array, reducing $minC$ from the residual capacity of edges along the path and adding $minC$ to the reverse edges. The algorithm concludes when BFS no longer finds any augmenting paths, leaving the maximum flow stored in variable f.

In the subsequent section, we elaborate on the methodology of computing edge connectivity utilizing maximum flow algorithms.

2.2 Computing edge connectivity

In Section 1.3, we discussed the definition of edge connectivity in a graph $G(V, E)$, where it can be defined between any pair of vertices. Specifically, two vertices u and v are considered connected, or 1-edge-connected, if there exists a path between them in the graph.

We formalize the notion of edge connectivity as follows:

Definition 2.2.1 (Edge connectivity (u, v, G))**.** Two vertices u and v are said to be k-edge connected in graph $G(V, E)$ if there exist at least k edge-disjoint paths between

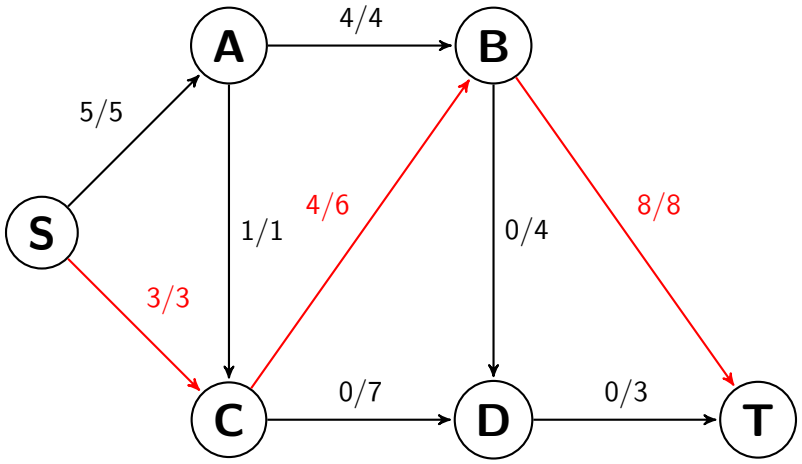

Figure 2.5: $S - C - B - T$ constitutes an augmenting path in the network flow illustrated in Figure 2.4, featuring a capacity $C = \min(3-0, 6-1, 8-5) = 3$. Consequently, 3 is added to the flow of each edge along this path. This configuration represents the final network flow, achieving a maximal flow of 8.

them.

Additionally, we introduce the concept of an $s-t$ cut in network flows:

Definition 2.2.2 ($s-t$ cut)**.** An $s-t$ cut between two vertices s and t in a network flow $G(V, E)$ is a partition of vertices into two sets S and T such that the source s is in S, the sink t is in T, and $S \cap T = \emptyset$ and $S \cup T = V$. The sum of the capacity of edges from set S to set T is referred to as the cut capacity.

These definitions provide fundamental concepts for understanding edge connectivity and $s-t$ cuts in graphs and network flows, which will be further explored in subsequent discussions.

Definition 2.2.3 (Minimum $s-t$ cut)**.** An $s-t$ cut in a graph that has the minimum capacity among all possible $s-t$ cuts is called the minimum $s-t$ cut.

Theorem 2.2.1 (Max-Flow Min-Cut Theorem)**.** The capacity of a minimum cut (s, t) in a graph is equal to the maximum flow (s, t).

It is evident that the maximum flow cannot surpass the capacity of any $s-t$ cut in a network flow. Hence, the maximum flow is at most equal to the capacity of the minimum cut, i.e., $maxflow(s, t) \leq mincut(s, t)$. However, the Max-Flow Min-Cut Theorem asserts a stronger relationship, stating that the maximum flow (s, t) is equal to the capacity of the minimum cut (s, t).

This equivalence implies that the value of the minimum cut is equal to the number of edge-disjoint paths between a pair of vertices (u, v). In numerous studies, including [20], edge-connectivity between vertices s and t is denoted by $\lambda(s, t, G)$, which equals the minimum cut (s, t) and maximum flow (s, t) when the capacities of all edges are considered as 1. Therefore, $\lambda(s, t, G)$ can be efficiently computed using any maximum flow algorithm.

In this research, we predominantly utilize the Edmonds-Karp algorithm for calculating edge-connectivity, or the minimum cut (s, t), between two vertices, as it offers a robust and efficient approach.

Example 2.2.1. In Figure 2.6, we observe an $s - t$ cut between vertices S and T, representing the minimum cut (S, T) in the graph. This minimum cut yields a maximum flow of 8 units. Notably, the source vertex S belongs to the set $\{S, A, C\}$, while the sink vertex T is in the set $\{B, D, T\}$.

The maximum flow between S and T is precisely equal to the sum of flows across the edges connecting the source set (depicted by blue vertices) and the sink set (depicted by green vertices). In this case, the maximum flow is achieved through the edges AB, CB, and CD, with flow values of 4, 4, and 0, respectively, resulting in a total flow of 8 units.

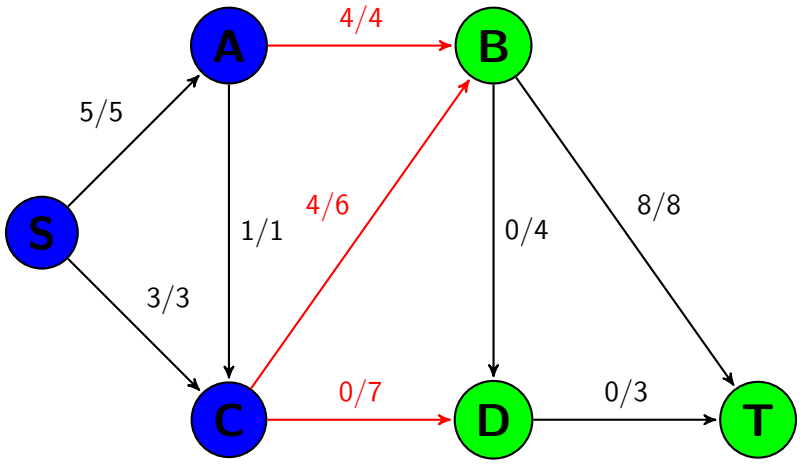

Figure 2.6: The figure illustrates a minimum cut between the source (S) and sink (T), which coincides with the maximum flow. The maximum flow (S, T) equals the flow across edges connecting the two sets of blue and green nodes, totaling 8 units.

Definition 2.2.4. The edge connectivity of a graph $G(V, E)$ is defined as the minimum of the edge connectivities between each pair of vertices in V. If the edge

connectivity of a graph is k, it is considered a k-edge-connected graph.

One straightforward method for computing the edge connectivity of a graph $G(V, E)$ involves invoking a maximum flow algorithm $\binom{V}{2}$ times, considering the capacities of each edge as one, as there are $\binom{V}{2}$ pairs of vertices in a graph. Then, the minimum of these connectivities is determined. Utilizing the Edmonds-Karp algorithm, an $O(V.E^2)$ algorithm can be employed to calculate the edge connectivity between a pair of vertices, while an $O(V^3.E^2)$ algorithm can be utilized for determining the edge connectivity of the entire graph.

Definition 2.2.5. The connectivity of a component in a graph $G(V, E)$ is defined as the minimum edge connectivity between each pair of vertices in the component. In other words, the number of edge-disjoint paths ($\lambda(u, v, G)$) is computed for each pair of vertices (u, v) in a component, and then the minimum of these values is determined. It's important to note that these disjoint paths may include edges and vertices that are present in the graph but not within the component.

In the subsequent section, we present an algorithm for computing k-edge-connected components in a graph.

2.3 Computing k-edge-connected components

Definition 2.3.1. In a graph $G(V, E)$, a subset C of vertices from V is termed a k-edge-connected component if C is a maximal subset of V that has at least k edge-disjoint paths between each pair of vertices. In other words, C is a k-edge-connected component if it is a maximal subset of vertices of V such that for each pair $(u, v) \in C$, we have $\lambda(u, v, G) \geq k$. It's important to note that disjoint paths between u and v can include vertices that are not in component C, but all should be in the graph.

As discussed in the previous section, $\lambda(u, v, G)$, representing the number of disjoint paths between u and v in G, is equal to the minimum cut (u, v) in G and is also equal to the maximum flow (u, v) in G when the capacities of edges are equal to one. Therefore, we can compute it using the Edmonds-Karp algorithm (algorithm 2).

Example 2.3.1. Figure 2.7 depicts a graph with 6 vertices. The subset $\{b, f\}$ forms a 4-edge-connected component in G. This is because $\lambda(b, f, G) = 4$, indicating the existence of 4 edge-disjoint paths: {"baf", "bf", "bef", "$bcdf$"} in G. Notably, only one of these paths ("bf") is contained within the induced subgraph of $\{b, f\}$.

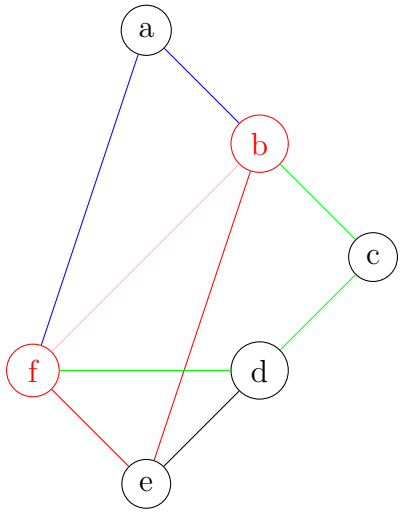

Figure 2.7: An undirected graph with 6 vertices: a, b, c, d, e, f. The subset $\{b, f\}$ forms a 4-edge-connected component because there exist 4 edge-disjoint paths: $\{"baf", "bf", "bef", "bcdf"\}$ in G.

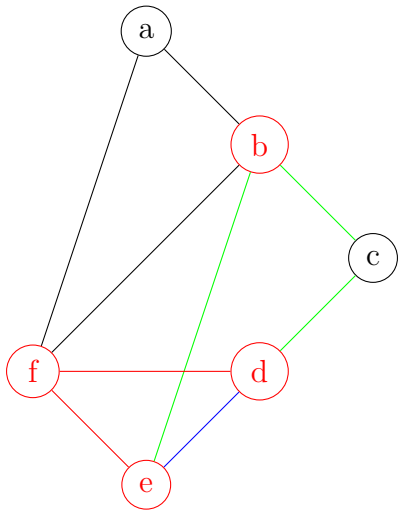

Figure 2.8: $\{b, f, d, e\}$ forms a 3-edge-connected component in G. This is because $\lambda(u, v, G) \geq 3$ for any pair of vertices (u, v) in $\{b, f, d, e\}$. For the pair (d, e), there are 3 disjoint paths ("de", "dcb", and "dfe") shown in different colors (blue, green, and red) in the picture.

Example 2.3.2. In Figure 2.8, $\{b, f, d, e\}$ forms a 3-edge-connected component in G. This is because $\lambda(u, v, G) \geq 3$ for any pair of vertices (u, v) in $\{b, f, d, e\}$. For example, for the pair (d, e), there exist 3 edge-disjoint paths: {"de", "$dcbe$", "dfe"} in G. Notably, although node c is not part of the component $\{b, f, d, e\}$, it appears in one of the disjoint paths.

In the following subsection, a simple algorithm to find k-edge-connected components in graphs for a given k will be introduced.

2.3.1 A simple algorithm to compute k-edge-connected components

In this section, an algorithm for finding k-edge-connected components for a given value of k is presented.

Theorem 2.3.1. In a graph $G(V, E)$, edge connectivity between vertices ($\lambda(u, v, G)$) forms a transitive relation in V. Specifically, if $\lambda(x, y, G) \geq k_1$ and $\lambda(y, z, G) \geq k_2$, then $\lambda(x, z, G) \geq \min(k_1, k_2)$. [26]

Proof. Suppose $\lambda(x, z, G) < \min(k_1, k_2)$. Using contradiction, the theorem can be proved. Let the minimum cut of x and z be C. Then $|C| = \lambda(x, z, G)$ is less than

k_1 and less than k_2, and C is not an edge cut of x and y. Therefore, after removing C, there is still a path from x to y and a path from y to x. Also, we know $|C| < k_2$, therefore C is not an edge cut of y and z, and after removing C, there is still a path from y to z and a path from z to y. Therefore, after removing C, there is still a path from x to z and a path from z to x. Therefore, C could not be a min cut for x and z, which is a contradiction to our assumption. □

Corollary 2.3.2. In a graph $G(V, E)$, for any three vertices $x, y, z \in V$: If $\lambda(x, y, G) = k$ and $\lambda(y, z, G) = k$, then $\lambda(x, z, G) = k$.

Theorem 2.3.3. In a graph $G(V, E)$, the concept of edge connectivity between vertices, denoted as $\lambda(u, v, G)$, forms an equivalence relation in the set of vertices, V.

Proof. The edge connectivity between vertices in a graph exhibits properties of an equivalence relation, namely reflexivity, symmetry, and transitivity.

- Reflexivity: Each vertex is trivially k-edge-connected to itself, indicating that $\lambda(x, x, G)$ equals some value k for any arbitrary k.
- Symmetry: If vertices u and v are k-edge-connected, then it follows that v and u are also k-edge-connected, thereby demonstrating symmetry.
- Transitivity: For any vertices x, y, and z, if $\lambda(x, y, G) = k$ and $\lambda(y, z, G) = k$, then $\lambda(x, z, G) = k$, as established by the previous corollary.

Hence, edge connectivity between vertices in a graph qualifies as an equivalence relation in the set of vertices, V. □

Utilizing the previously established theorems and the corollary, it becomes evident that knowledge of the value of $\lambda(u, v, G)$ for any pair of vertices in a graph enables the determination of k-edge-connected components. This assertion stems from the definition of a k-edge-connected component as a maximal subset of vertices in which for every pair of vertices within it, the edge connectivity $\lambda(u, v, G)$ is at least k.

Algorithm 3 Algorithm for finding k-edge-connected components

Require: Graph $G(V, E)$ and an integer k. C is a 2D array where $C[i][j]$ represents the edge connectivity between vertices i and j.

```
C = 0
for u ← 1 to |V| do
    for v ← 1 to u − 1 do
        C[u][v] ← MaximumFlow(u, v)
    end for
end for
for u ← 1 to |V| do
    C[u][u] ← k {Any singleton vertex is a k-edge-connected component itself}
end for
for u ← 1 to |V| do
    if !mark[u] then
        DFS(u, G, C)
    end if
end for
```

Algorithm 4 DFS algorithm for finding k-edge-connected components

Require: Vertex u, graph G, Connectivity C

DFS(u,G,C)

```
mark[u] ← true
Print(u)
for each neighbour v of u in G do
    if !mark[v] and C[u][v] ≥ k then
        DFS(v, G, C)
    end if
end for
```

Algorithm 3: discovering k-edge-connected components

This algorithm delineates the process of identifying k-edge-connected components within a graph $G(V, E)$. It takes as input an integer k and the graph G, represented as a 2-dimensional vector using an adjacency list format (where $G[i]$ denotes the list of neighbors of vertex i, and $G[i][j]$ indicates the j-th neighbor of vertex i).

The initial phase of the algorithm involves computing the edge connectivity between each pair of nodes in G. To facilitate this, a 2-dimensional array is utilized to store $\lambda(i, j, G)$ for every pair (i, j) within G. This array, denoted as C, initializes all elements to zero to reflect no initial connectivity (line 1).

Subsequently, $\lambda(i, j, G)$ for each node pair within the graph is determined. This task is achieved employing a maximum flow algorithm, specifically algorithm 2, which computes the connectivity for all node pairs in G (lines 2 to 9). Notably, it's worth mentioning that any singleton vertex is inherently k-edge-connected to itself, constituting a k-edge-connected component.

Following the computation of edge connectivities among vertices, the algorithm proceeds to identify the k-edge-connected components. This phase employs a Depth-First Search (DFS) approach, as outlined in algorithm 4. The algorithm iterates through each vertex u, and if unvisited, invokes the DFS function to discern the k-edge-connected component C to which u belongs (lines 10 to 15).

The DFS procedure operates straightforwardly: upon visiting a vertex u, it marks it as visited and prints it as a member of the current k-edge-connected component. Subsequently, it traverses through the neighbors of u, checking for unvisited neighbors v with a connectivity of at least k. If found, DFS is recursively called for v, thereby ensuring all vertices within a k-edge-connected component are identified.

The time complexity of the algorithm is $O(v^2 \cdot F + DFS)$, where F represents the time complexity of the maxFlow algorithm. Since the Edmonds-Karp algorithm (algorithm 2) is used, $F = O(V \cdot E^2)$. Additionally, the Depth-First Search (DFS) operation has a linear time complexity, i.e., $O(DFS) = O(V + E)$. Therefore, the total time complexity amounts to $O(V^3 \cdot E^2)$.

In the following subsection, we present an algorithm to find KECCs with improved efficiency.

2.3.2 An algorithm to compute k-edge-connected components (fix-k algorithm)

In this section, an algorithm to find k-edge-connected components in a graph $G(V, E)$ for a given number $k \leq V$ is introduced. The main ideas of this method come from an algorithm proposed by Hiroshi Nagamochi and Toshimasa Watanabe in 1992 [20].

As defined in previous sections, the induced subgraph of $G(V, E)$ with a vertex set $V_i \subseteq V$ is a k-edge-connected component if it is maximal and for each pair of vertices $(u, v) \in V_i$, we have at least k edge-disjoint paths in G, meaning $\lambda(u, v, G) \geq k$.

A set of k-edge-connected components of a graph partitions V. Leveraging this property, to find k-edge-connected components of G, first identify a k-edge-connected component and then recursively find other k-edge-connected components from the remaining vertices.

The algorithm starts by considering a tree T_i for each k-component with vertices V_i. The main idea is to find these trees and report them as the components at the end.

Each T_i has a root r, and vertices in each T_i have a root $r(v) = r$, indicating that all these vertices belong to the same tree or, equivalently, the same k-component. The algorithm proceeds by assigning a root to each vertex in V.

The initialization step (line 1) sets variables such as i (the current number of k-components), and initializes the roots of vertices to zero. The edges that don't belong to any trees are in T_0, which is initially empty.

At the beginning of each loop iteration, the algorithm checks whether the queue Q is empty. If it is, this signifies the need to find the next k-component. Consequently, the variable i (representing the current component number) is incremented (line 4). Additionally, T_i, representing the tree over the new k-component that has not yet been discovered, is emptied (line 5).

The vertex v serves as the root for component i (where v is equivalent to i at this stage). The current vertex v is assigned as the root of the component, so the assigned root for vertex v should be i, or v (line 6).

Having established the root of T_i for the next k-component, the algorithm proceeds to identify the edges of T_i. To initiate this process, it must explore the neighbors of vertex v. Therefore, it adds unscanned edges incident to v to the queue Q (line 7). In subsequent steps, the algorithm will examine these neighbors to determine which ones are k-edge-connected to v, thereby incorporating them into T_i.

Algorithm 5 Fix-k Algorithm

Require: Integer k and the graph $G(V, E)$

Ensure: k-edge-connected components of the graph: $V_1, V_2, ..., V_p$ are stored in the form of trees (T_i), and the function returns p (number of k-components)

Initialize variables:
for all $v \in V : r(v) \leftarrow 0$
$i \leftarrow 0$
$Q \leftarrow T_0 \leftarrow \emptyset$
for $v = 1$ to n **do**
  if Q is empty **then**
    $i \leftarrow i + 1$
    $T_i \leftarrow \emptyset$
    $r(v) \leftarrow i$
    add unscanned edges incident to v to Q
  else
    Choose an edge (x, y) from Q and remove it from Q
    if $r(x) = r(y) = i$ **then**
      Mark (x, y) and (y, x) as "scanned" {Case 1: Both x and y are already in the same component}
    else
      if $r(x) = i$ and $r(y) = 0$ **then**
        if $\lambda(x, y; D) \geq k$ **then**
          $r(y) \leftarrow i$
          add (x, y) to T_i
          Mark (x, y) and (y, x) as "scanned"
          Add unscanned edges incident to y to Q {Case 2: Enlarge the component by including y into T_i}
        else
          Add (x, y) to T_0
          Find a minimum separating set X such that $|E(X, V - X; D)| = \lambda(x, y; D)$
          Mark all edges in $E(X; A)$ as "scanned" and remove them from Q {Case 3: $\lambda(x, y; D) < k$, create a new component}
        end if
      end if
    end if
  end if
end for
$p \leftarrow i$
$V_j \leftarrow \{v \in V \mid r(v) = j\}$ for $j = 1, 2, ..., p$

If Q is not empty (line 8), it signifies that the algorithm is in the midst of constructing T_i. At this point, it must select an edge (x, y) and remove it from Q (line 9). The algorithm then checks whether vertex y is already inside T_i. This verification can be accomplished by examining their respective roots. If the condition in line 10 is true, indicating that y and x are in the same component T_i, the algorithm proceeds to mark the edge (x, y) as scanned (line 11). This edge cannot be added to T_i because doing so would result in the creation of a loop within the tree.

If the roots of x and y are not equal (line 12) and $r(y) = 0$ (line 13), it indicates that the algorithm has not yet assigned any root to y, and further investigation is required to determine whether y belongs to the same k-component. To ascertain this, the algorithm computes the connectivity between x and y (line 14), a task achievable using any maximum flow algorithm. Here, we utilize the EdmondKarp algorithm (algorithm 2) to determine $\lambda(x, y; D)$. If the calculated value exceeds k (line 14), it indicates that y should indeed be included in the same k-component. Consequently, the algorithm assigns i to $r(y)$ (line 15), adds the edge (x, y) to the tree T_i (line 16), and marks the edge as scanned (line 17). Following this, the algorithm proceeds to explore the neighbors of y to expand T_i. Hence, it adds unscanned edges incident to y to Q (line 18).

If the connectivity between x and y (line 19) is less than k, it indicates that y belongs to another k-component, and the edge (x, y) does not belong to any trees. Consequently, the algorithm includes the edge (x, y) in T_0 (line 20). Following this, the algorithm identifies the set X to which x belongs via the minimum cut operation (line 21) and marks all edges originating from set X as scanned (line 22). These edges are deemed unnecessary by the algorithm as they do not belong to any T_i.

At the conclusion of the algorithm, the variable "i" represents the total number of k-components. This value is stored in p (line 28). As previously discussed, each vertex v is assigned a root. Consequently, all vertices sharing the same root $r(v) = j$ are part of V_j, denoting component j (line 29).

The algorithm exhibits a time complexity of $O(V)$ multiplied by the complexity of the MaxFlow algorithm. This complexity arises due to the loop over vertices, where the primary operations involve computing connectivity between pairs of vertices (as seen in lines 14 and 21), a task efficiently performed by a maximum flow algorithm. Utilizing the EdmondKarp algorithm (algorithm 2), the total time complexity amounts to $O(V^2.E^2)$, a significant improvement compared to the previous algorithm by a factor of V.

The question often posed is whether we can achieve further optimization. The answer is affirmative! In the subsequent section, we introduce three novel algorithms boasting superior time complexities!

2.4 Enhancements to the fix-k algorithm

In this section, we introduce three new algorithms with time complexities of $O(V^2 \cdot E)$, $O(V^3 \cdot \log^{O(1)}(V))$, and $O(V \cdot E^{1+o(1)})$ for computing KECCs in a graph $G(V, E)$. The core concept of these algorithms resembles algorithm 5, but they primarily diverge in the choice of maximum flow method employed.

Given that the fix-k algorithm (algorithm 5) utilizes a maximum flow algorithm, enhancing the MaxFlow algorithm and leveraging the specific properties of the problem can lead to improvements in the fix-k algorithm.

2.4.1 Improved algorithm using the Ford-Fulkerson method

The fix-k algorithm (algorithm 5) exhibits a time complexity of $O(V \cdot$MaxFlow algorithm$)$, and by utilizing the Edmond-Karp algorithm (algorithm 2), the overall time complexity becomes $O(V^2 \cdot E^2)$.

Although the Edmond-Karp algorithm, with a time complexity of $O(V \cdot E^2)$, surpasses the Ford-Fulkerson algorithm, which operates in $O(E \cdot f)$ where f denotes the maximum flow in the graph $G(V, E)$, employing the Ford-Fulkerson algorithm instead of Edmond-Karp can significantly enhance the efficiency of finding KECCs.

This improvement stems from the fact that, in our problem, the maximum flow f in the network graph G is bounded. While in a typical graph G, the maximum flow f does not have an upper bound and can be any value, in our scenario, it is constrained.

To ascertain the edge connectivity between a pair of vertices, each edge's capacity is set to one. Consequently, the maximum flow f in the graph can at most equal the maximum degree in the graph, which is $V - 1$ in the worst-case scenario. Hence, the Ford-Fulkerson algorithm, in our context, boasts a time complexity of $O(E \cdot V)$, representing an improvement by a factor of E compared to Edmond-Karp!

As previously mentioned, the time complexity of algorithm 5 is $O(V \cdot$MaxFlow algorithm$)$, thus yielding an algorithm that operates in $O(V^2 \cdot E)$ time.

This enhancement results in a remarkable improvement over algorithm 5, offering a performance boost by a factor of E. Particularly in dense graphs, this advancement

could lead to a time saving of nearly $O(V^2)$ compared to algorithm 5!

2.4.2 Improved algorithm using randomization

Algorithm 6 Karger-Stein Algorithm

Let n be the number of vertices in G
if $n \leq 6$ **then**
  return Run Karger's algorithm on G and return the minimum cut
end if
Choose a random edge (u, v) uniformly from G
Contract edge (u, v) by merging vertices u and v into a single vertex
Remove self-loops created by the contraction
Remove parallel edges
$G' \leftarrow$ Resulting graph after contraction
if number of vertices in $G' \leq \frac{n}{\sqrt{2}}$ **then**
  // Merge Phase
  Let G'' and G''' be two copies of G'
  $c_1 \leftarrow$ KargerStein(G'')
  $c_2 \leftarrow$ KargerStein(G''')
  return $\min(c_1, c_2)$
else
  // Continue with contraction phase
  return KargerStein(G')
end if

Another approach to enhancing the algorithm is through randomization. Unlike the deterministic nature of the current fix-k algorithm, leveraging randomization can offer improvements. This enhancement involves substituting the existing maxflow algorithm with a more efficient one.

By integrating Karger–Stein's min-cut algorithm [14], which runs in $O(V^2 \cdot \log^{O(1)}(V))$ time, we can upgrade the fix-k algorithm to operate in $O(V^3 \cdot \log^{O(1)}(V))$ time using randomization.

Karger-Stein's algorithm represents an advancement over Karger's algorithm for identifying a minimum cut in an undirected graph. It employs randomization by iteratively contracting edges until only two vertices remain. Each edge contraction merges its two incident vertices into a single vertex, effectively reducing the graph's size. Through repeated iterations, the algorithm eventually identifies the minimum cut.

Algorithm 7 Karger's Algorithm

Let n be the number of vertices in G
if $n \leq 2$ **then**
    return Run BFS or DFS to find the minimum cut
end if
Choose a random edge (u, v) uniformly from G
Contract edge (u, v) by merging vertices u and v into a single vertex
Remove self-loops created by the contraction
Remove parallel edges
$G' \leftarrow$ Resulting graph after contraction
return Karger(G')

2.4.3 Another improved algorithm

In a recent publication [5], Li Chen and colleagues introduced an algorithm with almost linear time complexity for solving the maxFlow problem. They achieved a time complexity of $O(E^{1+o(1)})$ for max flow computation.

Considering that the fix-k algorithm can be solved in $O(V \cdot \text{MaxFlow Algorithm})$, theoretically, we could potentially solve the fix-k algorithm in nearly $O(V \cdot E^{1+o(1)})$ time, which is remarkable!

It's worth noting that the non-existence of an algorithm to solve the maximum flow problem in $O(E)$ indicates that $O(V \cdot E)$ could serve as a lower bound for enhancing the fix-k algorithm.

Chapter 3

Algorithms for finding all k-edge connected components

In this chapter, our focus shifts to discussing algorithms designed to identify k-edge-connected components for all values of k.

In contrast to the preceding chapter, where we introduced algorithms specifically tailored for finding k-edge-connected components for a fixed value of k, here we broaden our scope. We delve into algorithms capable of identifying k-edge-connected components for all $1 \leq k \leq V$ in the graph $G(V, E)$.

We commence this chapter with an overview of the problem and key definitions.

Definition 3.0.1. A k-edge-connected component in a graph $G(V, E)$ is a subgraph C of G that is maximal, meaning it cannot be further extended without violating the k-edge-connectivity property. For any pair of vertices within this component, there must exist at least k edge-disjoint paths in G. It's essential to note that these paths can traverse vertices or edges outside of C.

Example 3.0.1. Consider the undirected graph depicted in Figure 3.1. Here, $\{a, d\}$ forms a 2-edge-connected component since there exist two edge-disjoint paths, "abd" and "acd", connecting vertices a and d.

Similarly, $\{b, c\}$ constitutes a 3-edge-connected component as there exist three edge-disjoint paths, "bc", "bdc", and "bac", between vertices b and c.

Furthermore, the entire graph is 2-edge-connected because the removal of any two edges is sufficient to disconnect it.

In the following, we explain how to find all k-edge-connected components for all $k \leq V$ in a graph.

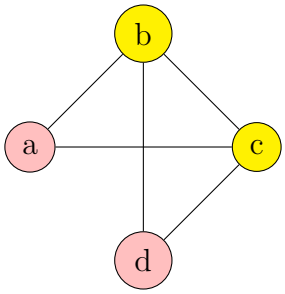

Figure 3.1: A 2-edge-connected graph consists of $\{a, d\}$ as a 2-edge-connected component and $\{b, c\}$ as a 3-edge-connected component in the graph.

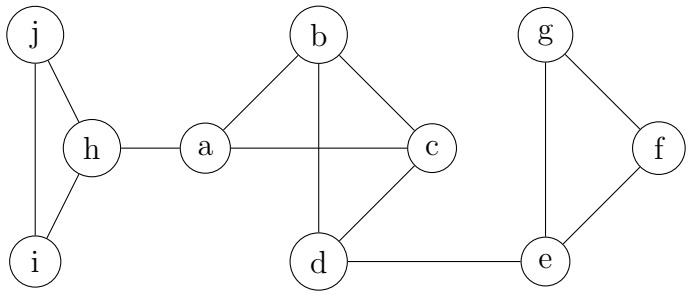

Figure 3.2: Undirected graph with 10 vertices: a, b, c, d, e, f, g, h, i, j.

3.1 An algorithm for computing all k-edge-connected components

One approach to finding k-edge-connected components for all $k \leq V$ in a graph $G(V, E)$ is by utilizing the algorithms introduced in the previous chapter, which are specifically designed to compute k-edge-connected components for a fixed value of k. By employing these algorithms iteratively for each k from 1 to V, we can determine all k-edge-connected components. If the time complexity of the algorithm for finding the k-edge-connected components in G (referred to as the fix-k algorithm) is denoted as $O(T)$, then the overall time complexity to compute all k-components is $O(T \cdot V)$ due to the loop over k from 1 to V.

However, we may ponder whether there exists a more efficient approach. Is it necessary to invoke the fix-k algorithm for all $k \leq V$? Can we devise a more optimized solution? These are questions worth exploring to potentially improve the efficiency of the algorithm.

Before addressing this question, let's consider the following example.

Example 3.1.1. In Figure 3.2, you can see a 1-edge-connected graph (or a connected graph) because removing edge (d, e) (or edge (h, a)) renders the graph disconnected. Therefore, for $k = 1$, $\{a, b, c, d, e, f, g, h, i, j\}$ is a 1-edge-connected component. For $k = 2$, $\{a, b, c, d\}$, $\{h, i, j\}$ and $\{e, f, g\}$ are 2-edge-connected components. For $k = 3$, $\{b, c\}$, $\{a\}$, $\{d\}$, $\{e\}$, $\{f\}$, $\{g\}$, $\{h\}$, $\{i\}$ and $\{j\}$ are 3-edge-connected components. For $k \geq 4$, each singleton (vertex) is a k-edge-connected component with size 1.

Each singleton (a vertex) is k-edge-connected to itself, making it a k-edge-connected component for any k. As evident from the previous example, in a graph with ten vertices, we only need to compute k-edge-connected components for $k \leq 4$, as for $k \geq 4$,

the k-components are singletons. If we can determine this critical k value (in the example, it is 4), we can significantly improve the running time.

Algorithm 8 All-k-Algorithm($G(V, E)$)

```
1: low ← 0
2: h ← |V|
3: while low < h do
4:    mid ← ⌊(low+h)/2⌋
5:    Fix-K-Algorithm(mid) {Compute k-edge-connected components for k = mid}
6:    if all computed k-components are singletons then
7:       h ← mid − 1
8:    else
9:       low ← mid + 1
10:   end if
11: end while
12: for k = 1 to k = h do
13:   Fix-k-Algorithm(k) {Print k-edge-connected components}
14: end for
```

One method to determine this critical value of k is by employing binary search. Algorithm 8 provides the pseudocode for this approach. The first step is to find h, which represents the smallest k such that for $i \geq k$, i-connected components are singletons. The algorithm accomplishes this by employing a binary search (lines 1 to 11). Subsequently, it iterates over all k values less than h and identifies k-edge-connected components using the fix-k algorithm (lines 12 to 14).

The time complexity of the algorithm, $O(h \cdot \text{fix-}k \text{ algorithm})$, is contingent upon the graph structure and the value of h (the critical k). If h is small and $O(\log(V))$, the runtime will be $O(\log(V) \cdot \text{fix-}k \text{ algorithm})$. Conversely, if the graph is densely connected and the critical k is close to V, the time complexity becomes $O(V \cdot \text{fix-}k \text{ algorithm})$.

3.2 A Simple Algorithm for Finding All k-Edge-Connected Components

The algorithms presented in this section are adapted from the work of Wang et al. as outlined in their paper[26]. The task of identifying all k-edge-connected components is tackled using a two-step approach.

1-edge-connected components: {a, b, c, d, e, f, g, h, i, j}

2-edge-connected components: {a, b, c, d}, {e, f, g}, {h, i, j}

3-edge-connected components: {a}, {b, c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}

4-edge-connected components: {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}

Figure 3.3: Hierarchical representation of k-edge-connected components of the graph in figure 3.2

In the first step, a data structure called an auxiliary graph is constructed. This structure stores the connectivity information between $V - 1$ pairs of vertices in the graph. The time complexity for constructing the auxiliary graph is $O(F \cdot V)$, where F represents the time taken to compute the maximum flow between each pair of vertices in the graph.

In the second step, the k-edge connected components can be extracted from the auxiliary graph for any given k, employing an $O(V)$ algorithm.

EdmondKarp's algorithm (algorithm 2) is employed to compute the maximum flow between $V - 1$ pairs of vertices. Given that constructing the auxiliary graph is $O(F \cdot V)$, the total time complexity amounts to $O(V^2 \cdot E^2)$.

3.2.1 Auxiliary graph

k-edge connected components in the graph $G(V, E)$ form a partition of V [26]. Another interesting property of k-edge-connected components is that they are unions of $(k + 1)$-edge-connected components. Leveraging these properties, finding all k-components for all k can be achieved recursively.

The process begins by identifying 1-components. Subsequently, 2-components are derived from each 1-component. This pattern continues to unfold: 3-components are derived from 2-components, and so forth. The algorithm concludes once each component reduces to a singleton (a vertex).

Example 3.2.1. Figure 3.3 illustrates the hierarchical structure of k-edge-connected components in the graph shown in Figure 3.2.

To identify all k-edge-connected components of the graph, we begin with $k = 1$. The 1-edge-connected components of a graph are simply its connected components. Since the graph is connected, there exists only one 1-edge-connected component: $\{a, b, c, d, e, f, g, h, i, j\}$.

Moving to 2-edge-connected components, exploring within each 1-edge-connected component is sufficient, as each 1-component comprises 2-edge-connected components. Thus, the union of $\{a, b, c, d\}$, $\{e, f, g\}$ and $\{h, i, j\}$ yields $\{a, b, c, d, e, f, g, h, i, j\}$.

Similarly, to identify 3-edge-connected components, we can examine each 2-edge-connected component $\{a, b, c, d\}$, $\{e, f, g\}$, and $\{h, i, j\}$ and subdivide them into sets of 3-edge-connected components: $\{\{a\}, \{b, c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}\}$. This process continues until we reach singleton vertices ($k = 4$).

Another approach to identify all k-components is to initiate the search from $k = h - 1$ (the highest k), proceeding in descending order until reaching $k = 1$. This method involves constructing a data structure called the auxiliary graph.

Let's define h as the smallest k for which (h)-components consist of singletons. Assume $(k + 1) = h$. We aim to find all k-components where $1 \leq k < h$.

Example 3.2.2. In the previous example, if $h = 4$, all 4-components are singletons, forming a forest of vertices denoted as A_h (or A_4 in this scenario).

To discover k-components with $k = h-1$, we begin by identifying a pair of vertices (s, t) where $\lambda(s, t, G) = k$. In the previous example, (b, c) with a connectivity of 3 represents such a pair. Subsequently, the algorithm incorporates (b, c) with weight 3 into the forest. However, since there are no other vertices u satisfying $\lambda(b, u, G) = 3$ or $\lambda(c, u, G) = 3$, we cannot expand the 3-component further, leading to the discovery of $\{b, c\}$ as one of the 3-components in the graph.

As there are no other pairs (u, v) in the graph with $\lambda(u, v, G) = 3$, $\{b, c\}$ emerges as the sole 3-component. Consequently, A_k for $k = h - 1$ (or A_3) comprises a forest where b and c are connected, while other vertices remain singletons.

To compute $h - 2$ or 2-components, the algorithm explores the possibility of expanding one of the 3-components in the graph. For instance, it examines whether there exists any vertex v such that $\lambda(b, v, G) = 2$. In our case, vertices d and a fulfill this criterion. Therefore, the algorithm incorporates edges (b, a) and (b, d) with weight 2 into the forest, forming $\{a, b, c, d\}$ as one of the 2-components.

Similarly, the algorithm identifies 2-components of $\{i, j, h\}$ by adding edges (i, h) and (j, h) with weight 2, as well as $\{e, f, g\}$ by including edges (e, g) and (e, f) with weight 2 (by expanding the singletons $\{i\}$ and $\{e\}$). Consequently, A_2 constitutes the forest comprising $\{a, b, c, d\}$, $\{i, j, h\}$, and $\{e, f, g\}$.

The final step involves identifying A_1, which represents the auxiliary graph. If the algorithm identifies any connected pair of vertices from one component to another, it

adds an edge with weight 1 between the 2-components. In our example, edges (h, a) and (d, e) are added, generating a 1-component consisting of $\{a, b, c, d, e, f, g, h, i, j\}$. This step completes the construction of the auxiliary graph. Figure 3.4 illustrates the resultant auxiliary graph derived from Figure 3.2.

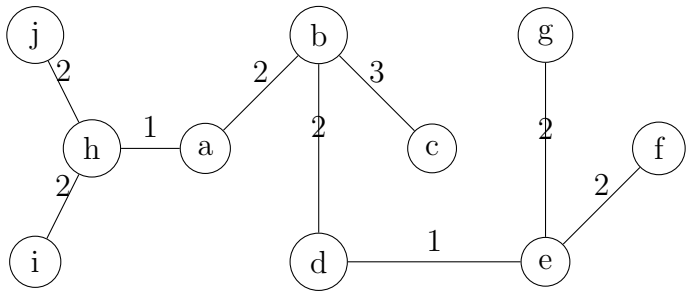

Figure 3.4: The auxiliary graph of figure 3.2.

Definition 3.2.1. The auxiliary graph A of a graph $G(V, E)$ is a weighted spanning tree over the vertices V, storing edge-connectivity information of G. Two vertices (u, v) are k-edge-connected in G if there exists a path with a minimum weight of at least k between u and v in A. This implies that if two vertices are k-connected, they are also i-connected for all $i \leq k$.

Example 3.2.3. In the graph of Figure 3.2, vertices j and f are 1-edge-connected because the minimum weight edge in the path $jhabdef$ in the auxiliary graph (Figure 3.4) is equal to 1. On the other hand, vertices b and c are 3-edge-connected because the minimum weighted edge in the path bc in the auxiliary graph is equal to 3.

In the next subsection, an algorithm for constructing the auxiliary graph will be elucidated.

3.2.2 Construction of the Auxiliary Graph

As mentioned earlier, the Auxiliary graph serves as a weighted tree over the graph $G(V, E)$, sharing the same set of vertices. It encapsulates the connectivity details of G, facilitating the identification of k-edge-connected components for all k within the graph.

Algorithm 9 illustrates the process of constructing the auxiliary graph for any directed graph $G(V, E)$. If graph G is undirected, it's essential to include both edges

Algorithm 9 Construction$(G(V, E), s, N)$

Require: Graph G, Vertex s, Set of vertices N

if $N = \{s\}$ **then**
  return
end if
Randomly pick a vertex t from $N - \{s\}$.
$(x, S, T) \leftarrow$ MAX-FLOW(G, s, t)
$(x', T', S') \leftarrow$ MAX-FLOW(G, t, s)
if $x' < x$ **then**
  $x \leftarrow x', S \leftarrow S', T \leftarrow T'$
end if
Add edge (s, t) with weight x to A.
CONSTRUCTION$(G, s, N \cap S)$
CONSTRUCTION$(G, t, N \cap T)$

(u, v) and (v, u) in the graph data structure for each edge (u, v) (since if u is connected to v, v is also connected to u).

As demonstrated in the example of the previous section, each k-connected component is the union of certain $(k + 1)$-edge-connected components.

The main idea of the construction of an auxiliary graph is to add edges to the forest of $(k + 1)$-edge-connected-components to form k-components.

To commence, we initiate the function for the graph $G(V, E)$, a random vertex s, and a set of vertices N (where N is initially identical to V).

The construction process (algorithm 9) operates recursively, terminating when the size of the contained vertex set reduces to 1 (lines 0 to 1), as a single vertex forms a singleton and constitutes a k-edge-connected component for any k.

Next, a sink vertex t is selected from the set of vertices $N - \{s\}$ (line 2). Subsequently, the algorithm determines the connectivity (s, t) in G ($\lambda(s, t, G)$). This step involves invoking the maximum flow algorithm from source s to sink t.

As discussed earlier, finding the maximum flow between two vertices s and t is tantamount to identifying the minimum cut (s, t). EdmondKarp's algorithm (algorithm 2) is employed for this purpose.

Given that the graph is directed, the algorithm computes two maximum flows: one from s to t and the other from t to s. As the connectivity between t and s in a directed graph corresponds to the minimum of $\lambda(s, t, G)$ and $\lambda(t, s, G)$, the algorithm determines this minimum value, denoted as x. Additionally, the algorithm maintains the minimum cuts in the sets S and T (lines 3 to 6).

Subsequently, the algorithm adds the edge (s, t) with weight x (representing the connectivity of s and t) to the auxiliary graph A (line 8). It then recursively calls itself on vertex s and the set $N \cap S$, and on vertex t and the set $N \cap T$.

Since the auxiliary graph is a tree (it has V_1 edges), and in each step of the algorithm, the maxflow algorithm is called two times, the running time of the algorithm is $O(V \cdot \text{maxflow algorithm})$. The EdmondKarp algorithm was used in this project; hence the total construction time is $O(V^2.E^2)$.

In the following, the explanation of how to compute k-edge-connected components for a given k from the auxiliary graph in $O(V)$ time will be provided. Therefore, the total time of finding all k-edge-connected components using the auxiliary graph approach will be $O(\textit{construction}) + O(V \cdot h) = O(V^2.E^2)$. ($h \leq V$ is the smallest number that h-edge-connected components in a graph are singletons.)

3.2.3 Compute k-edge-connected components from the auxiliary graph

Finding k-edge-connected components for a given k from the constructed auxiliary graph can be accomplished in linear time. It's worth noting that if two vertices are k-edge-connected, they are also i-edge-connected for any $i \leq k$. Leveraging this insight along with the properties of the auxiliary graph, we can identify all k-edge-connected components by removing edges with $weight < k$.

Indeed, the rationale behind this lies in the fact that a $(k+1)$-component is a k-component, whereas a $(k-1)$-component does not meet the criteria of a k-component. Consequently, upon removal of all edges with weights less than k, the auxiliary graph transforms into a forest consisting solely of k-components. The process of identifying these k-components can be executed efficiently using a Depth-First-Search (DFS) algorithm. The pseudocode for this procedure is provided in Algorithm 10.

Algorithm 10 Find k-edge-connected components

Require: Auxiliary graph A, edge weight threshold k
Ensure: A List of k-edge-connected components

```
1: Remove all edges in A with weight < k
2: Initialize an empty list components
3: for each vertex v in V do
4:    if not mark[v] then
5:       C ← DFS(v, A)
6:       Add C to components
7:    end if
8: end for
9: return components
```

In the subsequent section, optimization techniques are introduced aimed at enhancing the performance of the all-k algorithm.

3.3 Improvement on the All-k algorithm

All the optimization techniques outlined at the end of Chapter 2 can be applied to any algorithm that employs a maximum flow algorithm to compute edge connectivity. This includes the all-k algorithm (algorithm 9) introduced in this chapter.

As mentioned in the previous section, the all-k algorithm exhibits a time complexity of $O(V \cdot \text{MaxFlow algorithm})$. Given that we utilized the EdmondKarp algorithm

as the maximum flow algorithm, our implemented approach has a time complexity of $O(V^2 \cdot E^2)$.

3.3.1 Improvement on All-k algorithm using Ford-Fulkerson algorithm

As discussed in Chapter 2, the Ford-Fulkerson Maximum Flow algorithm (algorithm 1) has a time complexity of $O(E \cdot f)$, where f represents the maximum flow in the graph. In our problem, where each edge has a capacity of 1, the flow for each vertex can be at most $V-1$. Consequently, we can infer that the maximum flow in the graph cannot exceed $V-1$. Therefore, by utilizing the Ford-Fulkerson algorithm instead of Edmonds-Karp, we substantially reduce the time complexity to $O(E \cdot V^2)$.

3.3.2 Improvement on All-k algorithm using randomization

As discussed in Chapter 2, by employing the Karger-Stein Maximum Flow algorithm (algorithm 6) for computing edge connectivity [14], we can significantly reduce the time complexity of our implemented algorithm from $O(V^2 \cdot E^2)$ to $O(V^3 \cdot \ln^{O(1)}(V))$. This reduction arises from the fact that the all-k algorithm can be implemented in $O(V \cdot$ max flow algorithm$)$, and the Karger-Stein algorithm has a time complexity of $O(V^2 \cdot \ln^{O(1)}(V))$.

3.3.3 Enhancements to the All-k Algorithm Utilizing Recent MaxFlow Research

In a recent publication by Li Chen et al. [5], a groundbreaking advancement in solving the maxFlow problem was introduced, achieving an almost linear time complexity of $O(E^{1+o(1)})$.

Considering that the all-k algorithm can be solved in $O(V \cdot$ MaxFlow algorithm$)$, theoretically, it becomes feasible to solve the all-k algorithm in nearly $O(V \cdot E^{1+o(1)})$, which is indeed remarkable.

Given the impossibility of solving the maximum flow problem in $O(E)$, it follows that $O(V \cdot E)$ could potentially act as a lower bound for improving the all-k algorithm in graph scenarios.

Chapter 4

Experiments, Evaluation, Analysis, and Comparisons

In this chapter, experiments were conducted to evaluate the performance of two algorithms: the fix-k algorithm (algorithm 5) and the all-k algorithm (algorithm 9). These experiments aimed to assess how these algorithms performed across various types of graphs, including both real-world and artificially generated ones. To aid in the evaluation process, several functions were implemented, including one for generating random graphs based on user-defined parameters such as the number of vertices (N) and edges (E). Subsequently, the results obtained from running these algorithms in C++ were recorded in a file, and additional Python scripts were utilized to generate visualizations for analysis.

The experiments were divided into six categories. Experiment 1 focused on examining how different values of k influenced the running time of the fix-k algorithm and the number of k-edge-connected components in a graph. The results revealed that as k increased, both the running time and the number of k-edge-connected components increased until reaching a certain k value. This critical k value corresponds to the largest k for which there exists a k-edge-connected component with a size greater than 1 in the graph. Alternatively, if h represents the smallest number such that h-edge-connected components are singletons, then this critical k value equals $h - 1$.

Experiment 2 aimed to measure the running time of the fix-k algorithm on various types of graphs. Charts were created to illustrate how the running time changed with increasing values of N (number of vertices) and E (number of edges) in the graph.

In experiment 3, the performance of the all-k algorithm on different types of

graphs, including sparse and dense ones, was evaluated. Similar to experiment 2, charts were generated to visualize the running time of the algorithm with varying numbers of vertices or edges.

Experiment 4 compared the performance of the all-k algorithm with the fix-k algorithm for solving the all-k problem. The fix-k algorithm was enhanced by efficiently finding h, which denotes the largest k where a k-edge-connected component exists with a size greater than 1 in the graph.

Experiment 5 aimed to determine h, the maximum k for which a k-edge-connected component with a size greater than 1 exists in the graph. In essence, it sought to establish the upper limit for k in solving the all-k problem using the fix-k algorithm. This maximum k corresponds to the smallest value where each k-edge-connected component has a size of one. Algorithm 8 in chapter 3 employs the fix-k algorithm alongside a binary search approach to compute all k-edge-connected components in the graph.

Experiment 6 aimed to address the fix-k problem and evaluate whether the all-k algorithm outperforms the fix-k algorithm. To achieve this, both algorithms were compared in their ability to solve the fix-k problem across various types of graphs.

4.1 Experiments on the fix-k algorithm

This section presents a series of experiments conducted on the fix-k algorithm (algorithm 5). The experiments are categorized into three types. The first type involves varying the parameter k on a graph to observe how changes in k impact the algorithm's runtime and the number of k-edge-connected components in the graph. The second type focuses on modifying the graph while keeping k fixed to analyze how increasing the number of vertices or edges affects the runtime. Lastly, the third type of experiment aims to determine h, which represents the largest useful k on a graph. For any given graph, there exists a certain k value such that running the algorithm to find k-edge-connected components yields n components equivalent to the graph's vertices. This smallest k, denoted as h, holds significance as it signifies that for any $i \geq h$, the i-edge-connected components in the graph reduce to individual vertices.

4.1.1 Experiments on fix-k algorithm with varying k

In this subsection, experiments were conducted on a graph to observe the effect of increasing the parameter k on the running time of the fix-k algorithm. Random graphs were utilized for these experiments to ensure a comprehensive evaluation. To facilitate this evaluation, a function was implemented to generate random graphs with N vertices and E edges.

Performance of the fix-k algorithm for $k \leq 10$

The following results depict the performance of the fix-k algorithm when executed on a random graph containing 500 vertices and 10000 edges for values of k ranging from 1 to 10. This experiment focuses on small values of k, with the assumption that for larger k, the number of components will match the number of vertices, rendering further exploration unnecessary.

Table 4.1: Execution time and number of components for different values of k on a random graph with 500 vertices and 10000 edges

k	Execution time (Second)	Number of components
1	28.1703	1
2	27.7959	1
3	27.6585	1
4	27.4036	1
5	27.434	1
6	28.7214	1
7	28.2045	1
8	27.4667	1
9	27.9807	1
10	27.5725	1

However, it's worth noting that for all $k \leq 10$, the number of k-edge-connected components is 1, indicating that this graph is strongly connected. As observed, the running times for $k \leq 10$ are nearly identical, averaging about 27.5 seconds. Given that the entire graph constitutes a single k-edge-connected component for all $k \leq 10$, the algorithm needs to be executed for larger values of k to ascertain where the number of components starts to increase.

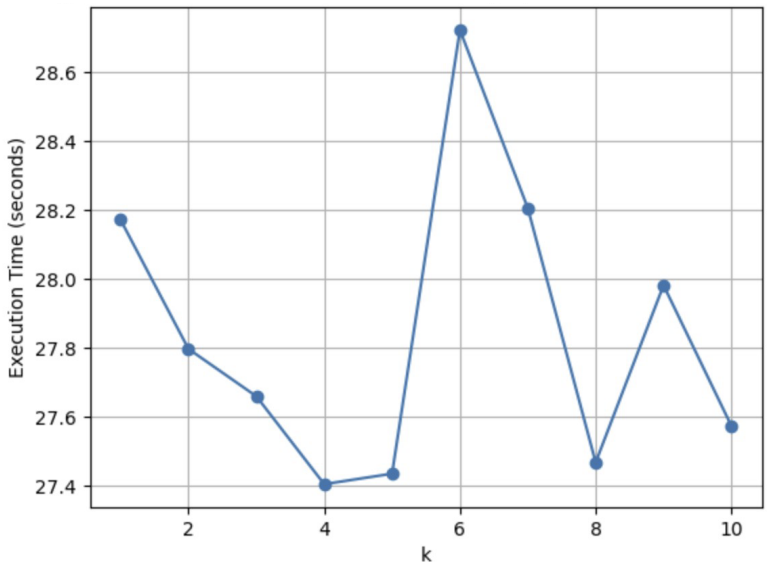

Figure 4.1: Execution time of the fix-k algorithm for a random graph with $N = 500$ and $E = 10000$.

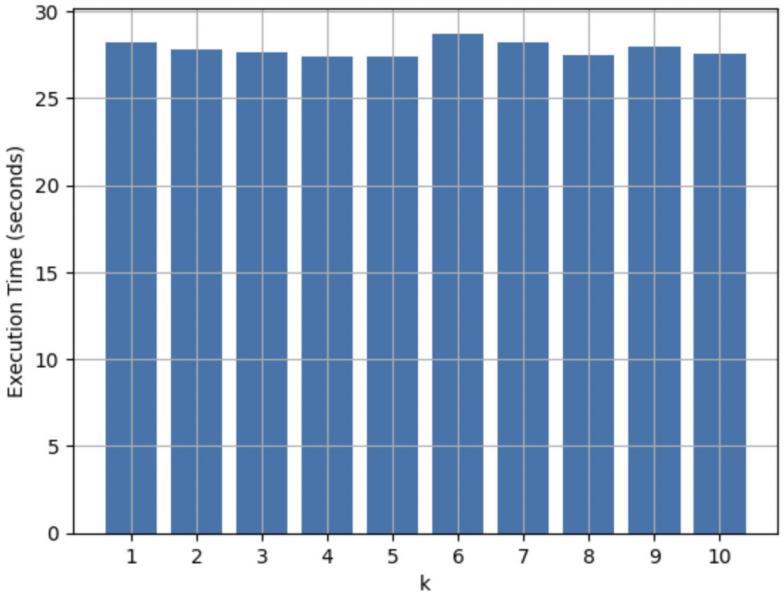

Figure 4.2: Execution time of the fix-k algorithm for a random graph with $N = 500$ and $E = 10000$.

Fix-k algorithm running on $k = 1, 10, 100, 1000$

In this experiment, a random graph with 1000 vertices and 20000 edges is generated. Subsequently, the fix-k algorithm is applied to the graph for various values of $k = 1, 10, 100, 1000$. The objective is to analyze how the algorithm's running time and the number of k-edge-connected components in the graph evolve as k increases from a small value (1) to the largest possible value (equal to the number of vertices, V).

Table 4.2: Execution time and number of components for a graph with 1000 vertices and 10000 edges for different values of $k = 1, 10, 100, 1000$.

K	Execution time (Second)	Number of components
1	174.615	1
10	179.114	1
100	632.414	1000
1000	649.794	1000

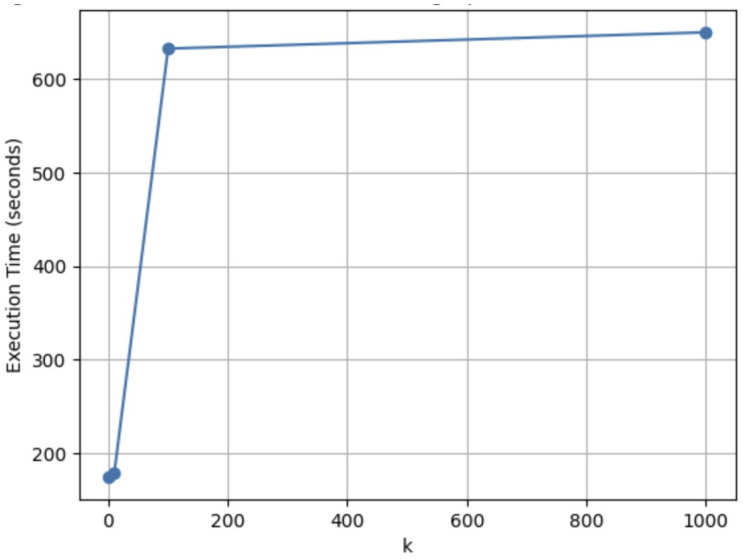

Figure 4.3: The execution time of the fix-k algorithm for a random graph with $V = 1000$ vertices and $E = 20000$ edges for different values of $k = 1, 10, 100, 1000$.

The results are intriguing! Upon closer examination, it becomes apparent that for $k = 1$ and $k = 10$, the number of k-edge-connected components in the graph was one, and the execution time was nearly identical. Similarly, for $k = 100$ and $k = 1000$, the running time exhibited minimal variation, with both cases resulting in approximately the same execution time. Additionally, in both instances, the number of k-edge-connected components in the graph was 1000. These findings suggest a

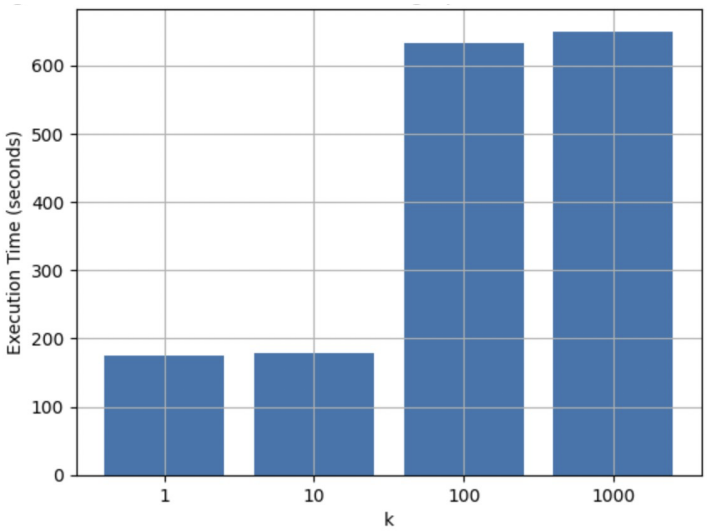

Figure 4.4: The execution time of the fix-k algorithm for a random graph with $V = 1000$ vertices and $E = 20000$ edges for different values of $k = 1, 10, 100, 1000$.

hypothesis: the running time of the algorithm may be more closely correlated with the number of k-edge-connected components in the graph rather than the specific value of k. However, further experiments are necessary to draw a more definitive conclusion.

Experiments on fix-k algorithm varying $k = \{1, 21, 41, ..., 461\}$

The objective of this experiment is to analyze how altering the value of k from 1 to N in a graph with N vertices influences both the running time of the fix-k algorithm and the number of k-edge-connected components in the graph. For this purpose, a random graph with $N = 500$ vertices and $E = 1500$ edges is generated using the random graph generator function. Subsequently, the fix-k algorithm is executed for $k = \{1, 21, 41, ..., 461\}$, and the corresponding running time is recorded for each value of k. It is noteworthy that once k exceeds 20, the running times become nearly identical, each totaling 44 seconds. Moreover, for $k > 20$, the number of k-edge-connected components remains constant at 500, equivalent to N.

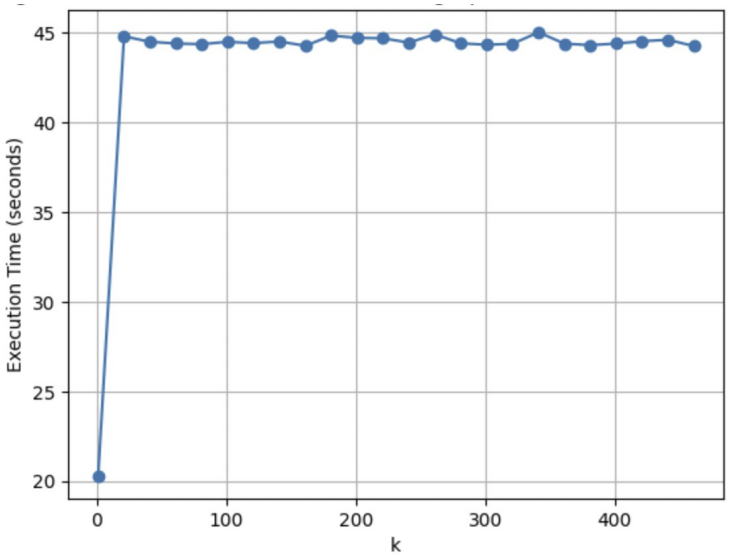

Figure 4.5: The fix-k algorithm execution time for a random graph with $N = 500$ vertices and $E = 1500$ edges for different values of $k = \{1, 21, 41, ..., 461\}$.

Interestingly, this experiment further illustrates that the running time of the fix-k algorithm is more closely tied to the number of k-edge-connected components in the graph rather than the value of k. For all $k \geq 21$, the running time consistently remained at 44 seconds, while the number of k-edge-connected components remained

Table 4.3: Execution time and number of k-edge-connected components for a graph with 500 vertices and 1500 edges for various values of k

K	**Execution time (Second)**	**Number of k-edge-connected components**
1	20.2818	1
21	44.7841	500
41	44.4842	500
61	44.3858	500
81	44.3524	500
101	44.4791	500
121	44.4071	500
141	44.4992	500
161	44.2595	500
181	44.827	500
201	44.7007	500
221	44.6745	500
241	44.4391	500
261	44.8938	500
281	44.3955	500
301	44.319	500
321	44.3713	500
341	45.0111	500
361	44.3788	500
381	44.2967	500
401	44.3833	500
421	44.5139	500
441	44.5938	500
461	44.2435	500

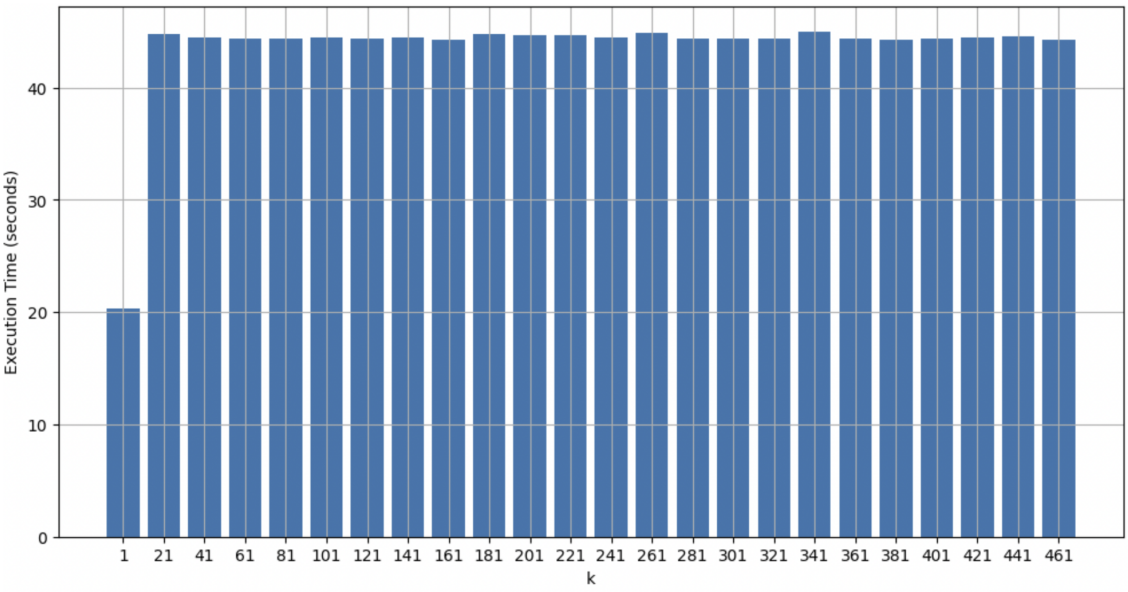

Figure 4.6: The fix-k algorithm execution time for a random graph with $N = 500$ vertices and $E = 1500$ edges for different values of $k = \{1, 21, 41, ..., 461\}$.

at 500. This observation suggests that there is no need to execute the algorithm for $k \geq 21$. Given that k increases by 20 in each iteration and the minimum k is one, we can infer that the largest k required to compute k-edge-connected components is less than 21. Therefore, it appears that h, the maximum k needed, is small compared to N.

In the subsequent experiments, the fix-k algorithm is executed starting from $k = 1$ until $k = h$, where h represents the smallest value such that all k-edge-connected components reduce to singletons, signifying that the number of h-edge-connected components equals N. This threshold is reached because for larger values of k, the graph exhibits the same outcome.

Experiments on fix-k algorithm with a graph of 700 nodes and 2100 edges

This experiment is conducted on a random graph with $N = 700$ vertices and $E = 2100$ edges. The fix-k algorithm is executed on this graph for values of k ranging from 1 onwards. At each iteration, k is incremented by one, and both the running time and the number of k-edge-connected components are recorded. The experiment continues until the number of k-edge-connected components equals the number of vertices, which is $N = 700$ in this case. At this point, further iterations are unnecessary since there will be no changes in the components for larger values of k. This is because when the number of components equals N, each component is a singleton and thus a k-edge-connected component for any k.

As anticipated, the value of h for this graph was small, specifically, it was found to be 13. Another noteworthy observation from the chart is that the execution time does not consistently increase with increasing k. This could be attributed to the fact that h is relatively small in this scenario. However, further experiments are required to provide a more comprehensive evaluation.

Figure 4.8 illustrates that as k increases, the number of k-edge-connected components increases until a certain point. In this graph, for $k > 13$, the number of k-edge-connected components remains constant and equals $N = 700$. This occurs when each component reduces to a singleton. Consequently, there is no necessity to execute the algorithm for larger values of k.

Table 4.4: Execution time and number of k-edge-connected components for various values of k in a graph with 700 vertices and 2100 edges.

k	**Execution time (second)**	**Number of k-edge-connected components**
1	55.1692	1
2	55.9651	12
3	61.5776	44
4	79.4298	115
5	102.074	194
6	126.426	301
7	138.459	423
8	120.114	528
9	105.176	606
10	101.159	659
11	111.334	690
12	115.123	698
13	117.998	700

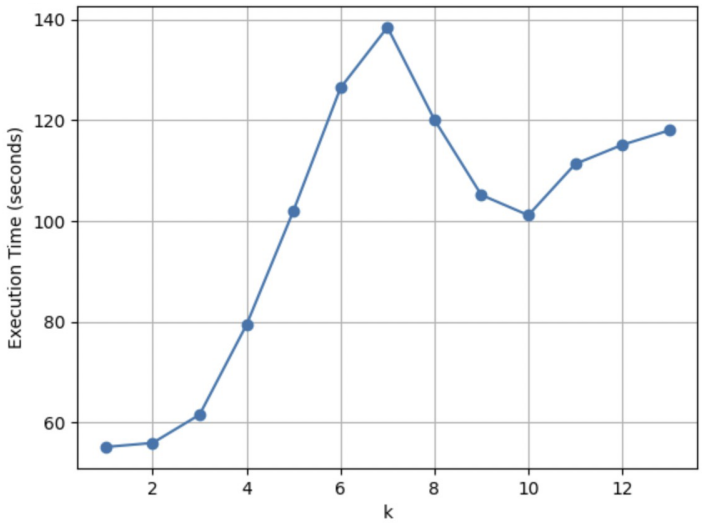

Figure 4.7: The fix-k algorithm execution time for a random graph with $N = 700$ vertices and $E = 2100$ edges for different values of $k = \{1, 2, 3, ..., 13\}$.

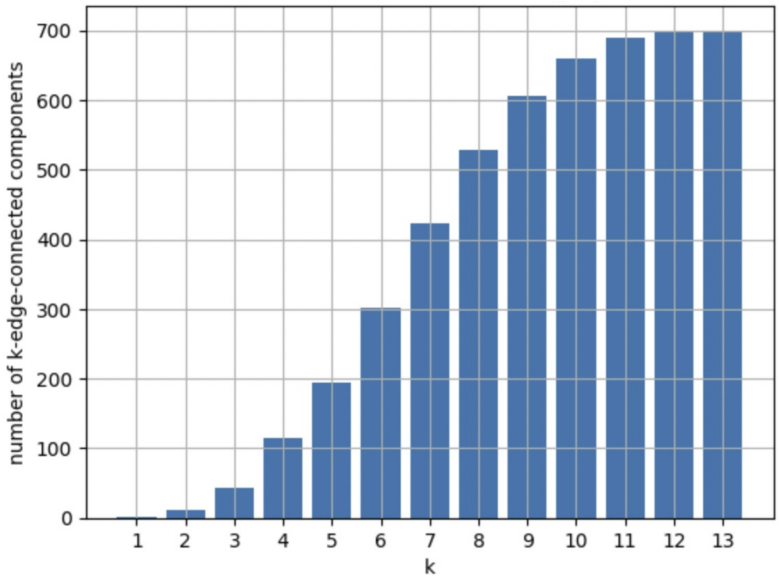

Figure 4.8: Number of k-components in a random graph with $N = 700$ vertices and $E = 2100$ edges for different values of $k = \{1, 2, 3, ..., 13\}$ using the fix-k algorithm.

Fix-k algorithm execution on a graph with 1000 nodes and 3000 edges

This experiment involves running the fix-k algorithm on a random graph with $N = 1000$ vertices and $E = 3000$ edges. The parameter k is incremented from 1 to larger values (increased by 1 each time) until $k = h$, where h represents the point at which the number of k-edge-connected components in the graph equals N.

Table 4.5: Execution time and number of k-edge-connected components in a graph with 1000 vertices and 3000 edges.

k	**Execution time (Second)**	**Number of k-edge-connected components**
1	158.607	2
2	160.387	12
3	177.371	63
4	228.139	157
5	292.06	293
6	354.111	458
7	376.392	613
8	361.417	740
9	297.365	858
10	285	931
11	314.162	982
12	341.439	999
13	341.589	1000

Although the graphs in this experiment differ from those in the previous one, the results are somewhat similar, suggesting that the number k, the number of vertices N, and edges E may have a more significant impact on the outcomes than the specific graph structure. However, this is only a hypothesis, and further experiments are required to draw more definitive conclusions.

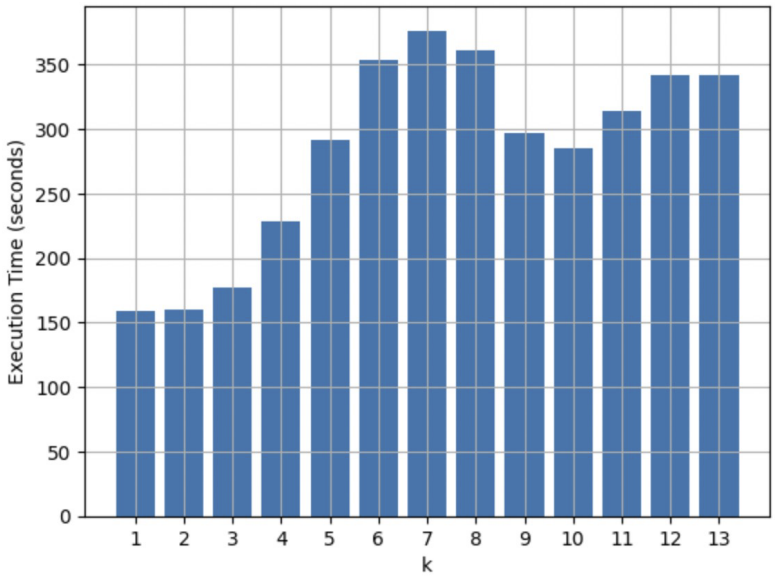

Figure 4.9: The fix-k algorithm execution time for a random graph with $N = 1000$ vertices and $E = 3000$ edges for different values of $k = \{1, 2, 3, ..., 13\}$.

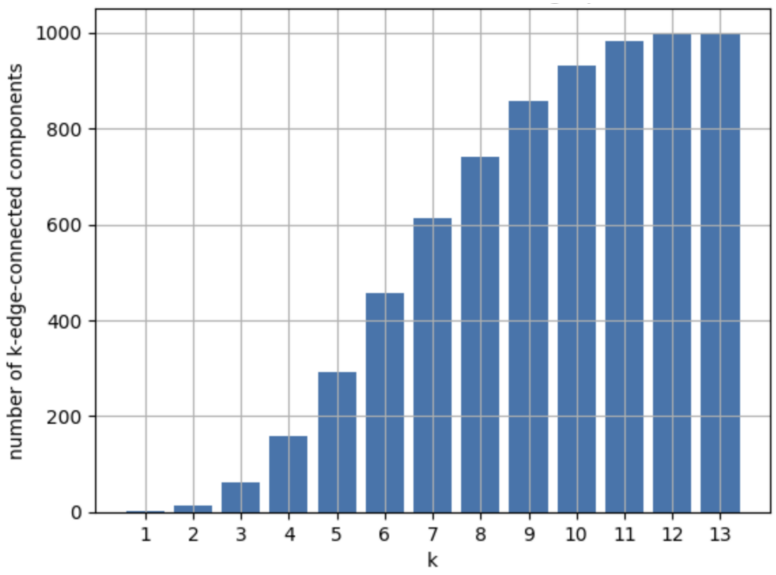

Figure 4.10: Number of k-edge-connected components in a random graph with $N = 1000$ vertices and $E = 3000$ edges for different values of $k = \{1, 2, 3, ..., 13\}$ computed using the fix-k algorithm.

Fix-k algorithm execution on a graph with 1000 nodes and 7000 edges

In this experiment, the fix-k algorithm is executed for various values of k on a graph comprising 1000 vertices and 7000 edges. The algorithm is iteratively run starting from $k = 1$, and the value of k is incremented by 1 each time until $k = h$. Here, h represents the smallest k at which the number of k-edge-connected components equals the total number of vertices N, indicating that each k-component becomes a singleton.

Table 4.6: Execution time of the fix-k algorithm and the number of k-edge-connected components in a graph with 1000 vertices and 7000 edges for various values of k.

k	**Execution Time (Second)**	**Number of k-edge-connected components**
1	168.17	1
2	168.278	1
3	167.991	1
4	167.88	3
5	170.811	5
6	175.613	8
7	192.53	21
8	219.639	46
9	265.114	72
10	337.018	115
11	439.337	180
12	554.565	261
13	666.109	362
14	764.302	461
15	784.117	560
16	770.413	651
17	668.128	746
18	566.848	830
19	484.073	878
20	378.012	926
21	391.757	957
22	450.887	979
23	480.899	992
24	485.904	998
25	503.903	999
26	493.83	1000

As depicted in the charts, in a graph with more edges, h is larger compared to a graph with the same number of vertices but fewer edges. This observation is logical

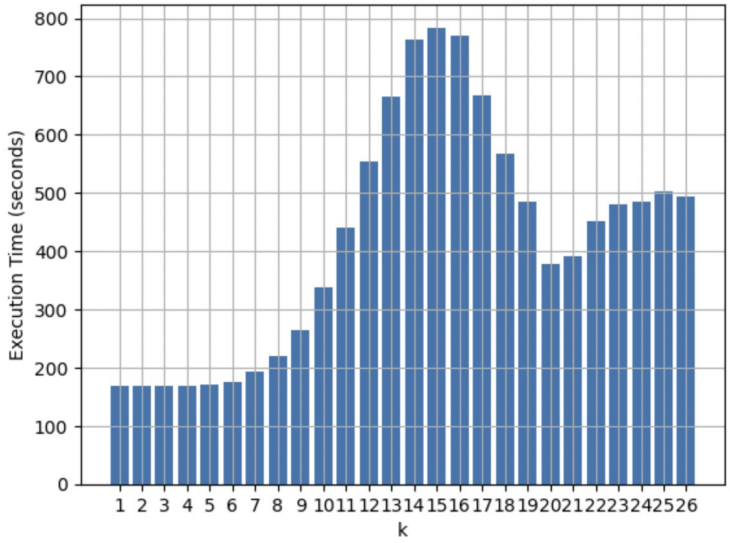

Figure 4.11: Execution time of the fix-k algorithm in a graph with 1000 vertices and 7000 edges for different values of k.

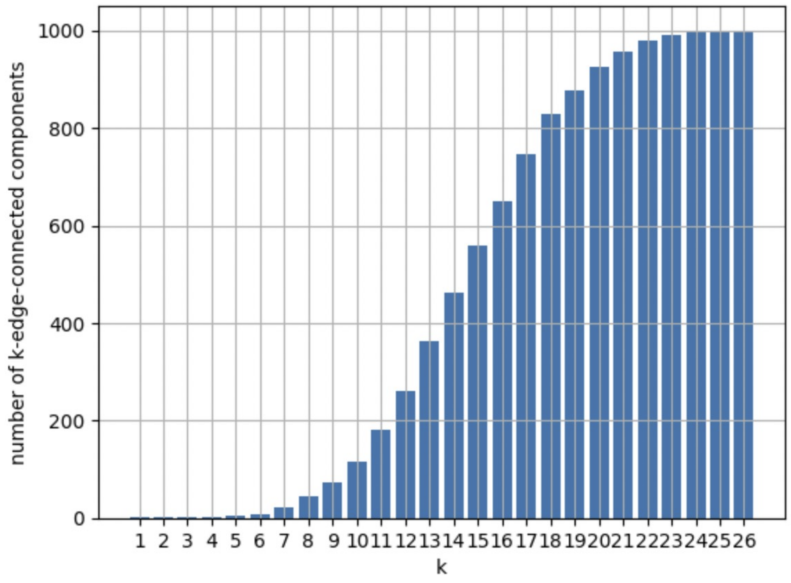

Figure 4.12: Number of k-edge-connected components using fix-k algorithm in a graph with 1000 vertices and 7000 edges for different values of k.

because as the number of edges increases while maintaining the same number of vertices, the graph typically becomes denser and may exhibit greater connectivity.

Fix-k algorithm execution on a dense graph

This experiment assesses the performance of the fix-k algorithm on an extremely dense graph. The graph comprises $N = 500$ vertices and $E = 24950$ edges. The fix-k algorithm is applied to the graph, beginning with $k = 1$ and incrementally increasing until it reaches h, denoting the smallest k where the number of k-edge-connected components equals N. In this graph, h is found to be 134. The results encompass all $k = \{1, 2, 3, ..., 134\}$; however, for brevity, only a subset of these results (specifically, those where $k = 4l + 1$) is reported here due to space constraints.

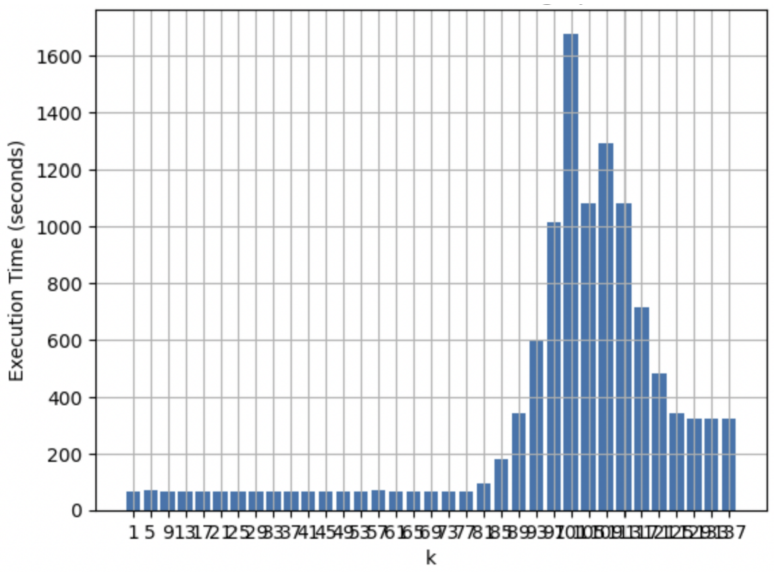

Figure 4.13: Execution time of the fix-k algorithm running on a graph with 500 vertices and 24950 edges for different values of k.

Table 4.7: Execution time of the fix-k algorithm on a graph with 500 vertices and 24950 edges, along with the number of k-edge-connected components for various values of k.

k	**Execution Time (Second)**	**Number of k-edge-connected Components**
1	66.756	1
5	69.4337	1
9	66.59	1
13	66.9806	1
17	67.1751	1
21	66.5186	1
25	66.9397	1
29	66.9187	1
33	67.5815	1
37	66.5797	1
41	66.5183	1
45	66.4827	1
49	67.177	1
53	67.4687	1
57	68.2756	1
61	66.763	1
65	67.1006	1
69	67.1063	1
73	66.654	1
77	66.7413	1
81	94.7516	3
85	180.159	11
89	341.158	26
93	599.337	54
97	1013.43	94
101	1679.77	175
105	1080.36	255
109	1290.97	331
113	1080.6	390
117	716.78	429
121	480.578	457
125	340.92	483
129	324.322	497
133	325.32	498
137	325.376	500

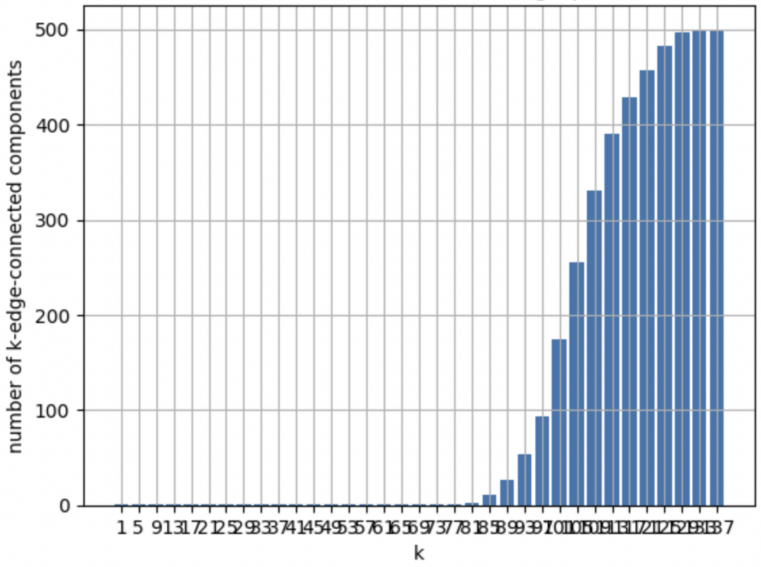

Figure 4.14: Number of k-edge-connected components using fix-k $a$lgorithm on a graph with 500 vertices and 24950 edges for different values of k.

You can observe that this graph is 77-edge-connected. For $k \leq 77$, the number of k-edge-connected components is 1. These results illustrate that for all graphs, there is a particular k value beyond which the number of k-edge-connected components begins to increase, indicating the graph's edge-connectivity. When the number of k-edge-connected components reaches N, the corresponding k value would be considered as h. The largest k for which there exists at least one k-edge-connected component with a size greater than 1 in the graph is $k = h - 1$.

4.1.2 Experiments for determining h in solving the all-k problem with the fix-k algorithm

As demonstrated in previous experiments, it is unnecessary to invoke the fix-k algorithm N times (for all $k \leq N$) to solve the all-k problem efficiently. An intriguing question arises: what is the largest required k for a given graph? This value represents the smallest number needed to execute the fix-k algorithm to identify all k-edge-connected components for all $k \leq N$. The results from experiments in section 4.1.1 indicate that as k increases, the number of k-edge-connected components in the graph also increases. The smallest k at which the number of k-edge-connected components in the graph reaches N (or each k-component becomes a singleton) is the value we seek, which we denote as h.

This experiment investigates how h changes with increasing the number of vertices N in a graph. I developed a function capable of determining h for any given graph. The underlying concept is that for $i < h$, the number of i-edge-connected components is less than N, the number of vertices in the graph. Conversely, for $i \geq h$, the number of i-edge-connected components in the graph equals N. Hence, I employed binary search to identify this critical value for any graph.

In this experiment, random graphs with $N = \{100, 400, 700, ..., 4000\}$ and $E = 4 \times N$ were generated using our random graph generator. For each graph, h was determined and recorded. The fix-k algorithm was then executed, and the corresponding running time for $k = h$ was recorded in the following table.

Table 4.8: Execution time of the fix-k algorithm for $k = h$ on different graphs with N vertices and $E = 4 \times N$ edges

N	h **(Biggest** k**)**	**Execution time for** $k = h$ **(Second)**
100	14	0.437018
400	15	20.5887
700	17	93.7142
1000	17	282.705
1300	19	620.693
1600	18	1021.36
1900	19	1906.53
2200	19	2815.77
2500	18	4211.53
2800	18	5898.5
3100	18	8075.98
3400	19	10631.2
3700	19	13298.5
4000	19	16841.7

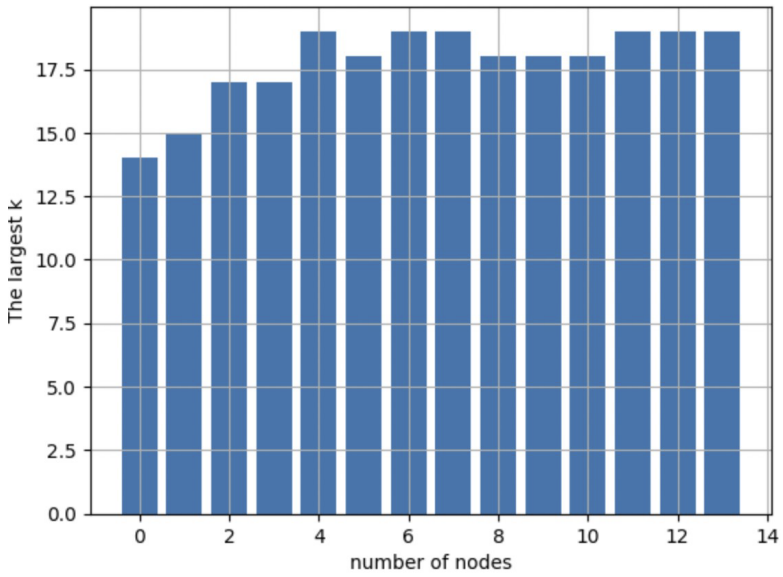

Figure 4.15: Changes of $k = h$ for different graphs with N vertices and $E = 4 \times N$ edges.

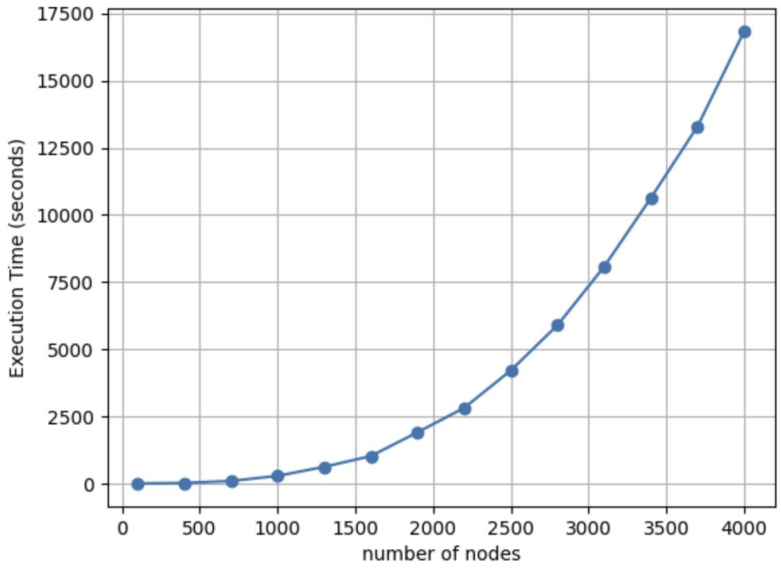

Figure 4.16: Execution time of the fix-k algorithm on $k = h$ for different graphs with N vertices and $E = 4 \times N$ edges.

The results from this experiment indicate that h is small. Specifically, for graphs with a linear number of edges where $N \leq 4000$ vertices, h is less than 20.

4.1.3 Execution of the fix-k algorithm on graphs with varied vertex and edge counts

Execution of the fix-k algorithm on sparse graphs with varied vertex and edge counts

For this experiment, nine graphs are generated with varying numbers of vertices, ranging from $N = \{100, 1100, ..., 8100\}$, where each graph has $E = 3 \times N$ edges. The fix-k algorithm is then applied to these graphs with $k = 5$.

In the first two charts, the relationship between the running time of the fix-k algorithm and the number of vertices (N) is depicted, showcasing how the algorithm's execution time increases with increasing N.

The third chart illustrates the number of 5-edge-connected components for each graph. Notably, this function exhibits a linear trend, providing insights into the behavior of edge-connected components concerning graph size.

Table 4.9: Execution time of the fix-k algorithm for different graphs with N vertices and $E = 3 \times N$ edges, alongside the corresponding number of 5-edge-connected components.

N	**Execution Time (Second)**	**Number of 5-edge-connected components**
100	0.218698	25
1100	275.077	312
2100	1878	599
3100	5877.53	889
4100	13997	1183
5100	26588.4	1497
6100	44736.5	1743
7100	71114.4	2058
8100	106279	2318

The chart in Figure 4.18 illustrates that as the number of vertices in the graph increases, the running time of the fix-k algorithm also increases. This relationship appears to follow a polynomial function of approximately degree 2

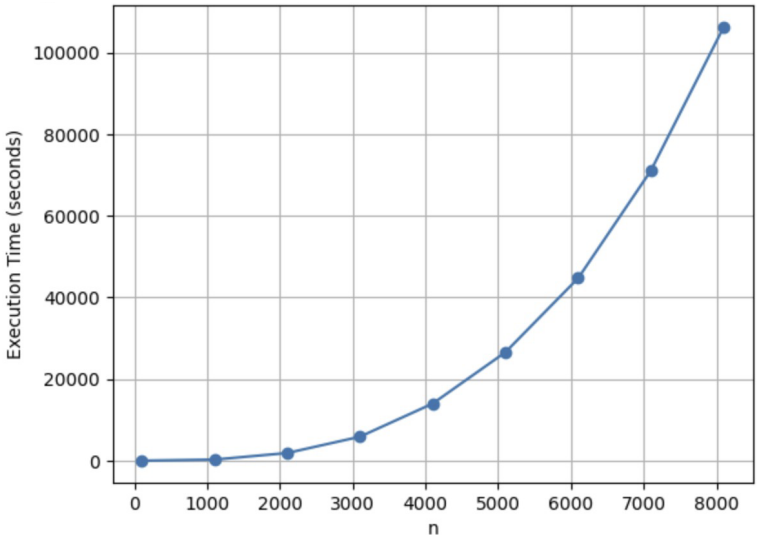

Figure 4.17: Execution time of the fix-k algorithm for different graphs with N vertices and $E = 3 \times N$ edges with $k = 5$.

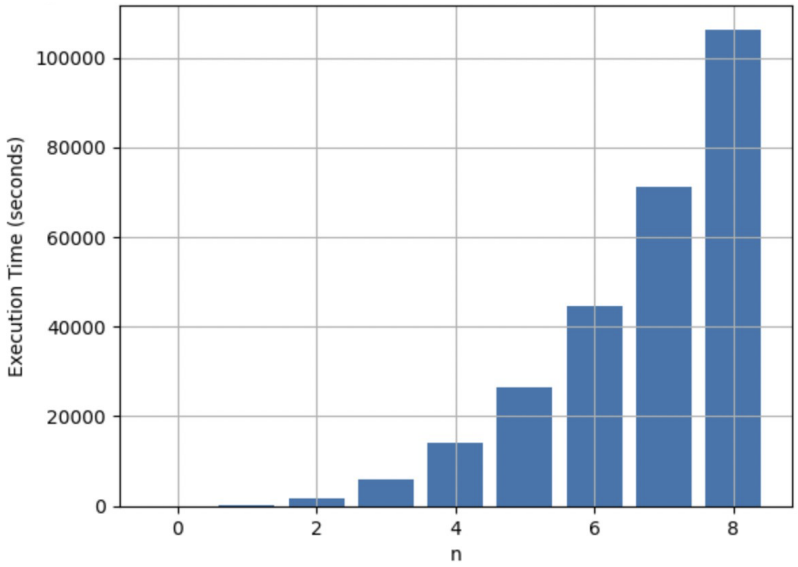

Figure 4.18: Execution time of the fix-k algorithm for different graphs with N vertices and $E = 3 \times N$ edges with $k = 5$.

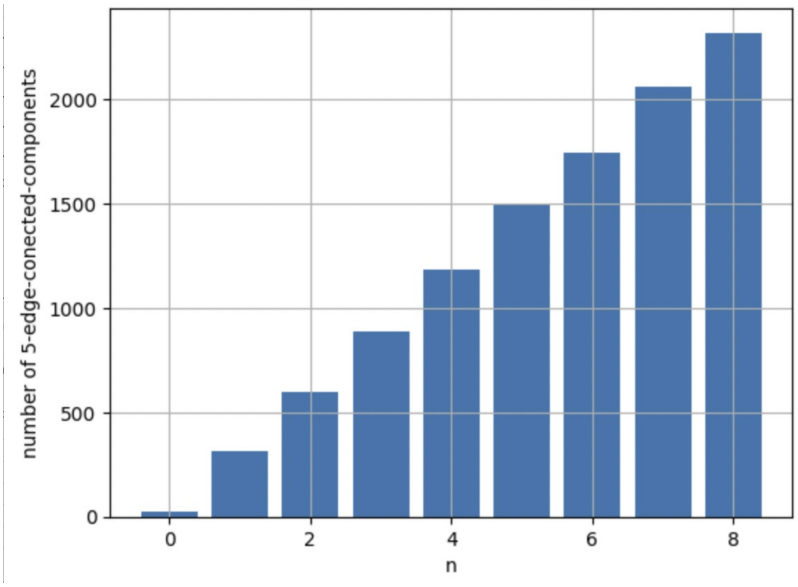

Figure 4.19: Number of 5-edge-connected components using fix-k algorithm for different graphs with N vertices and $E = 3 \times N$ edges.

The chart in Figure 4.19 depicts how the number of 5-edge-connected components increases with the number of vertices in the graph. This relationship appears to follow a linear function.

Execution of the fix-k algorithm on dense graphs with varied vertex and edge counts

This experiment aims to assess the impact of increasing the number of vertices and edges in a dense graph on the running time of the fix-k algorithm.

The charts below depict the running time of the fix-k algorithm for $k = 5$ across various random graphs with a range of vertices, $N = \{100, 200, 300, ..., 2600\}$. The number of edges, E, is determined using the formula: $E = \frac{N \times N}{10}$.

Table 4.10: Execution time of the fix-k algorithm on different graphs with N vertices and $E = \frac{N \times N}{10}$ edges.

Number of nodes	Execution time (Second)
100	0.332751
200	3.36539
300	14.9921
400	42.344
500	104.437
600	201.875
700	349.392
800	598.359
900	907.562
1000	1388.08
1100	2001.83
1200	2782.6
1300	3867.24
1400	4964.36
1500	6749.13
1600	8613.03
1700	10998.2
1800	13958.2
1900	17118.7
2000	20997.8
2100	24976.1
2200	30227.5
2300	35943.1
2400	42711.4
2500	50368.5
2600	58814.7

As observed, with the increase in both N and E, the running time escalates, exhibiting a polynomial function of degree approximately 2. Given the density of

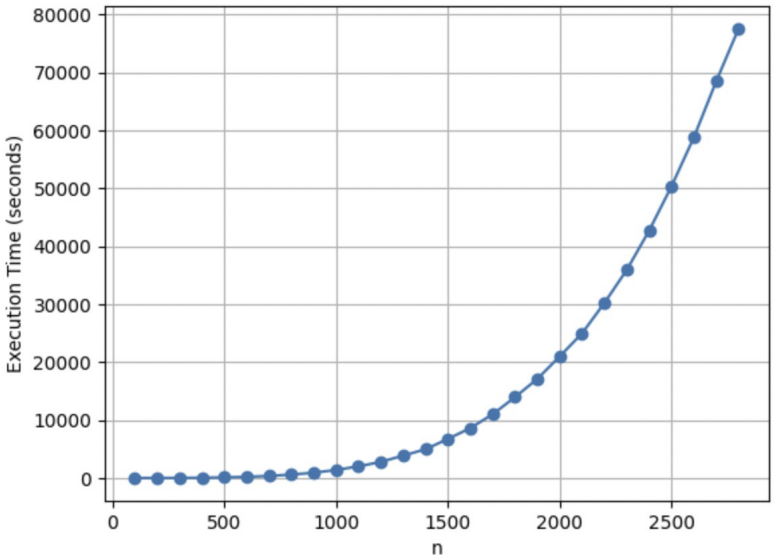

Figure 4.20: Execution time of the fix-k algorithm on different graphs with N vertices and $E = \frac{N \times N}{10}$ edges.

these graphs, all of them possess just one 5-edge-connected component (they are at least 5-edge-connected), hence the number of such components is omitted in the table.

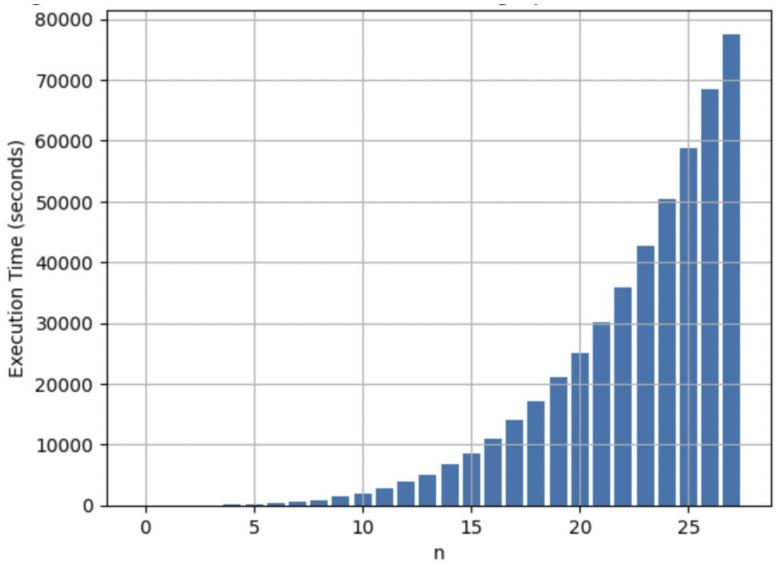

Figure 4.21: Execution time of the fix-k algorithm on different graphs with N vertices and $E = \frac{N \times N}{10}$ edges.

4.2 Experiments on the all-k algorithm

In this section, we conduct experiments on the all-k algorithm using graphs with varying numbers of nodes and edges generated by our random graph generator. We measure the execution time of the algorithm for each graph. These experiments cover both dense and sparse graphs to provide comprehensive insights.

4.2.1 Experiments for all-k algorithm on sparse graphs

In this part, results for running the all-k algorithm on sparse graphs with N vertices and $3 \times N$ edges for $N = 100, 200, 300, .., 2900$ are presented.

As anticipated, as the number of vertices in the graph increases, the running time of the all-k algorithm also increases, exhibiting a behavior akin to that of a polynomial function of approximately degree 2.

Table 4.11: Execution time of the all-k algorithm on sparse graphs with N vertices and $3 \times N$ edges.

Number of nodes	Execution time (Second)
100	0.249848
200	1.4407
300	4.62859
400	10.7545
500	21.0751
600	35.9221
700	56.089
800	83.5247
900	117.884
1000	161.516
1100	214.412
1200	278.321
1300	353.249
1400	439.484
1500	538.243
1600	652.049
1700	781.998
1800	927.166
1900	1089.46
2000	1271.07
2100	1469.36
2200	1688.57
2300	1928.52
2400	2196.64
2500	2474.78
2600	2777.63
2700	3109.16
2800	3466.36
2900	3851

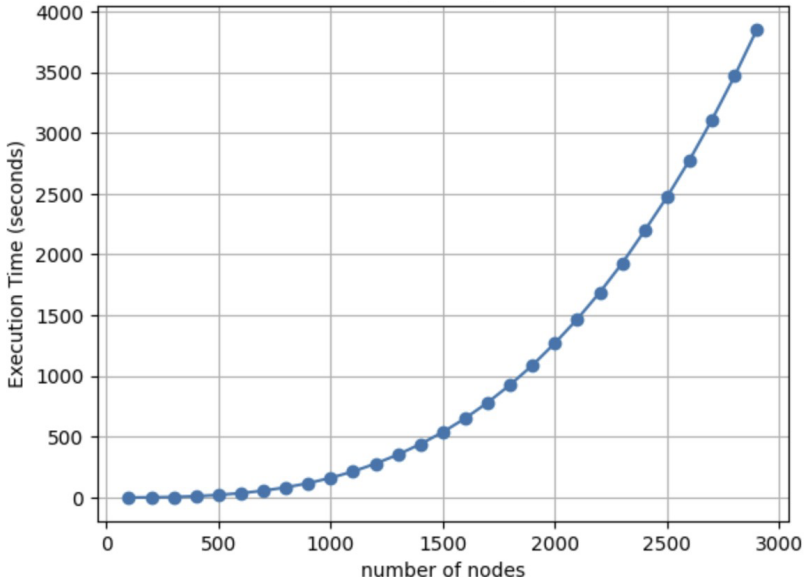

Figure 4.22: Execution time of the all-k algorithm on sparse graphs with N vertices and $3 \times N$ edges.

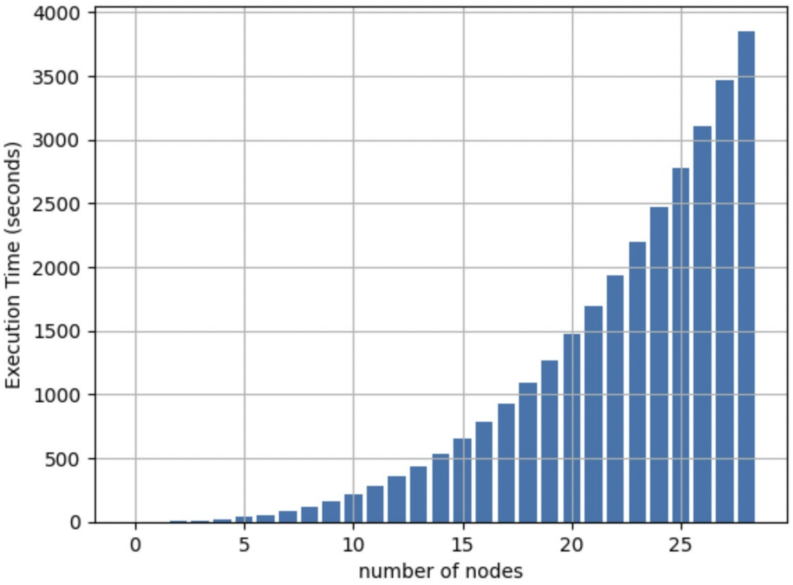

Figure 4.23: Execution time of the all-k algorithm on sparse graphs with N vertices and $3 \times N$ edges.

4.2.2 Experiments for the all-k algorithm on dense graphs

Below are the outcomes obtained from executing the all-k algorithm on graphs with varying numbers of vertices and $E = \frac{N \times N}{20}$ edges, where N ranges from 100 to 1600.

Table 4.12: Execution time of the all-k algorithm on dense graphs with N vertices and $E = \frac{N \times N}{20}$ edges.

Number of nodes	Execution time (Second)
100	0.288231
200	2.054
300	7.14571
400	17.9679
500	36.1744
600	66.4365
700	110.089
800	172.253
900	256.344
1000	366.502
1100	505.227
1200	685.74
1300	898.41
1400	1159.05
1500	1477.74
1600	1840.76

As observed, the execution time increases with the number of nodes in the graphs, following a polynomial function of approximately degree 2. Notably, compared to the sparse graph experiment, the all-k algorithm exhibits higher execution times on dense graphs.

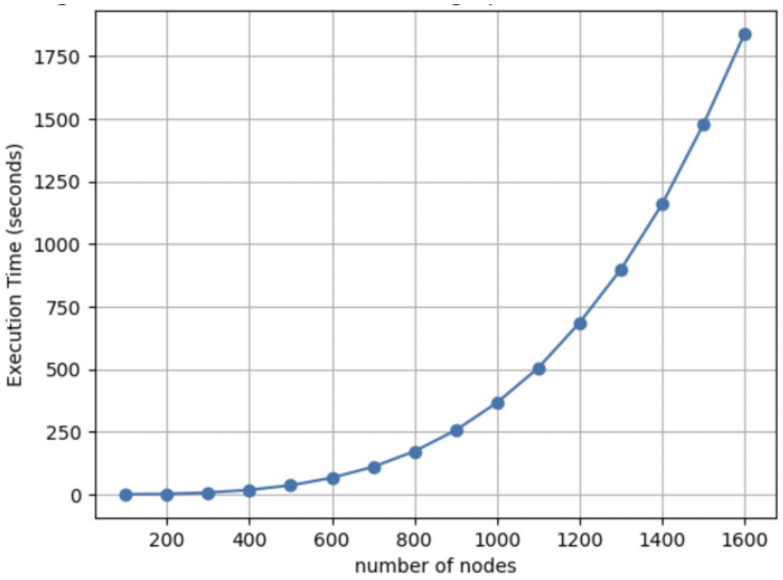

Figure 4.24: Execution time of the all-k algorithm on dense graphs with N vertices and $E = \frac{N \times N}{20}$ edges.

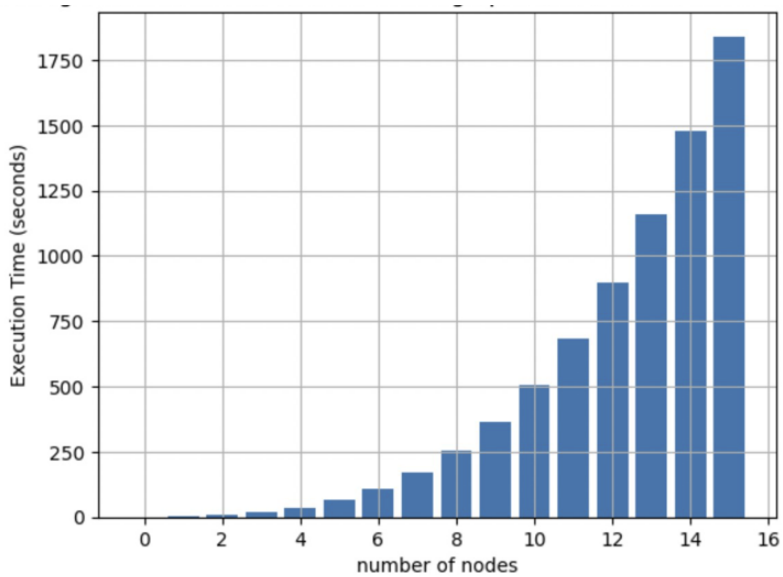

Figure 4.25: Execution time of the all-k algorithm on dense graphs with N vertices and $E = \frac{N \times N}{20}$ edges.

4.3 Experiments for comparing the all-k algorithm with the fix-k algorithm

4.3.1 Experiments for comparing the all-k algorithm with the fix-k algorithm to solve the all-k problem

The experiments conducted in this section are designed to measure how the fix-k algorithm's running time is comparable to the all-k algorithm to find k-edge-connected components for all $k \leq N$.

Experiments comparing the all-k algorithm with the fix-k algorithm for dense graphs

The following results depict the performance of the all-k and the fix-k algorithms on graphs with N vertices and $E = \frac{N \times N}{10}$ edges, where $N = \{100, 200, 300, 400, 500\}$. For each graph, the running time of the all-k algorithm is compared with the cumulative running time of the fix-k algorithm for all values of k ranging from 1 to N.

Table 4.13: Comparison of execution times between the all-k algorithm and the fix-k algorithm for graphs with N vertices and $E = \frac{N \times N}{10}$ edges.

N	Time for all-k (Second)	Time for fix-k (Second)
100	0.396903	32.5399
200	3.265	615.518
300	12.5576	4680.74
400	33.1257	21968.7
500	72.2242	38671.7

It's evident from the comparison that the all-k algorithm outperforms the fix-k algorithm in practice for solving the all-k problem. However, it's worth noting that in the comparison chart, the all-k algorithm's running time is pitted against the cumulative running time of the fix-k algorithm for all $k \leq N$. This approach overlooks the fact that the fix-k algorithm doesn't need to be executed for all $k \leq N$.

The primary objective is to identify the value of h, which represents the largest k in the graph that encompasses a k-edge-connected component containing more than one vertex. Subsequently, the fix-k algorithm should be run for all $k < h$. As for $k \geq h$, every component is a singleton, implying that the fix-k algorithm only needs

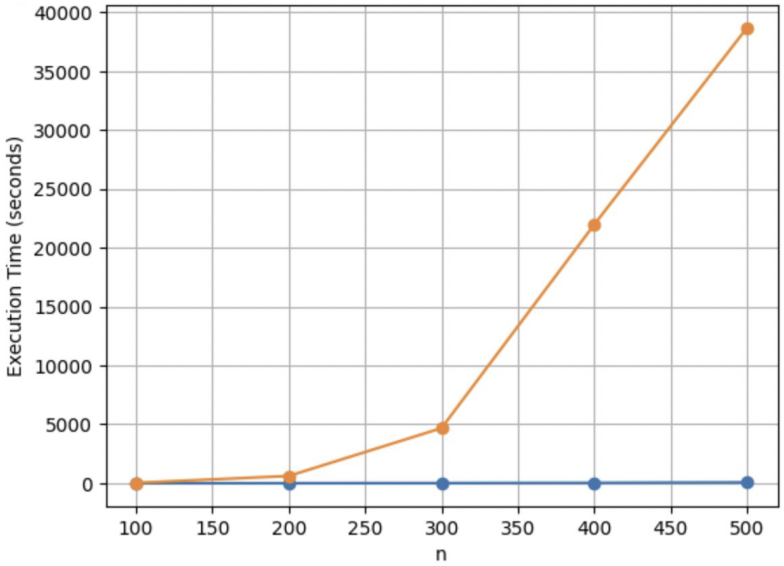

Figure 4.26: Comparison of execution times between the all-k algorithm and the fix-k algorithm for graphs with N vertices and $E = \frac{N \times N}{10}$ edges.

to be executed until the graph is divided into N components, where N represents the number of vertices in the graph $G(V, E)$.

Comparing the all-k algorithm with the fix-k algorithm on sparse graphs

In the following section, we present experiments comparing the fix-k algorithm with the all-k algorithm for solving the all-k problem on sparse graphs.

Comparing the all-k algorithm with the fix-k algorithm on graphs with N vertices and $E = 5 \times N$

In this experiment, we compare the performance of the all-k algorithm with the fix-k algorithm on graphs with varying numbers of vertices and edges. The graphs are generated with $N = \{100, 300, 500, ..., 1500\}$ vertices and $E = 5 \times N$ edges using our random graph generator. For each graph, the all-k algorithm is executed once, while the fix-k algorithm is executed h times, where h represents the largest k such that there exists at least one $(k-1)$-edge-connected component with size greater than one. In other words, h is the smallest k for which all k-edge-connected components are singletons.

Table 4.14: Comparison of the execution times of the all-k algorithm and the fix-k algorithm calling for all $k \leq h$ for graphs with N vertices and $E = 5 \times N$.

N	Time for all-k (Second)	Time for fix-k (Second)	h for fix-k
100	0.283721	7.92826	16
300	4.98351	192.274	18
500	21.6653	939.384	20
700	58.3679	5420.35	19
900	124.384	5107.77	19
1100	221.455	9000.81	19
1300	358.463	14403	19
1500	547.794	27472.9	21

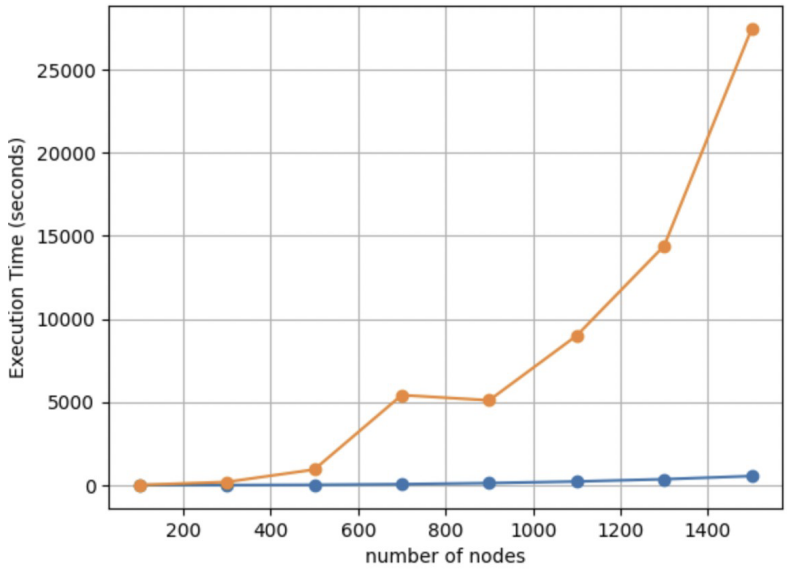

Figure 4.27: Comparison of the execution times of the all-k algorithm and the fix-k algorithm calling for all $k \leq h$ for graphs with N vertices and $E = 5 \times N$.

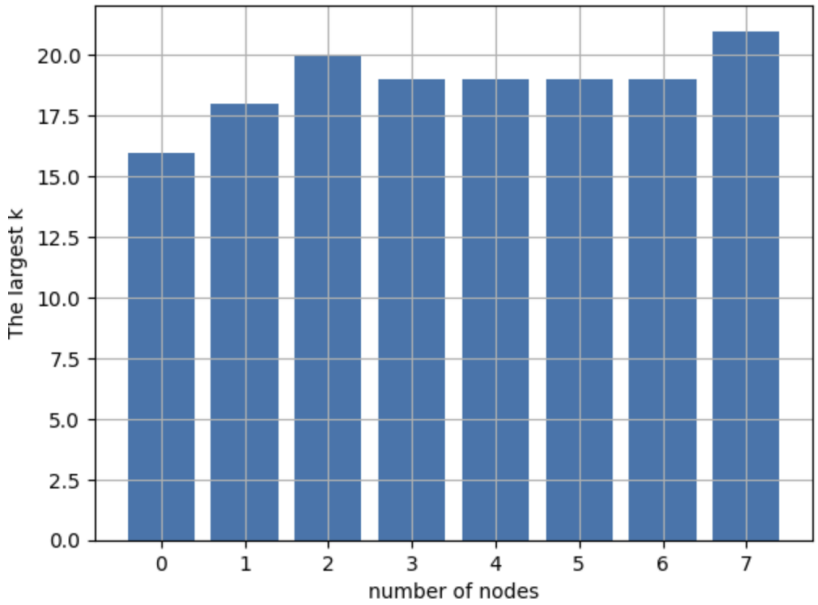

Figure 4.28: Changes of h (the largest k) in the fix-k algorithm for graphs with N vertices and $E = 5 \times N$ edges.

Comparing the all-k algorithm with the fix-k algorithm for graphs with $E = 3 \times N$

In this experiment, the graphs with $N = \{100, 300, 500, ..., 1300\}$ and $E = 3 \times N$ are created using our random graph generator. Then the all-k algorithm is executed one time and the fix-k algorithm is executed h times (for all $1 \leq i \leq h$, all i-edge-connected components are computed) for each of the graphs. $k = h - 1$ is the largest number where there exists at least one k-edge-connected component with more than 1 vertex in the graph.

Table 4.15: Comparisons of the execution times of the all-k algorithm and the fix-k algorithm calling for all $k \leq h$ for graphs with N vertices and $E = 3 \times N$.

N	Time for all-k (Second)	Time for fix-k (Second)	h for fix-k
100	0.251864	4.57	12
300	4.68265	87.56	11
500	21.0056	497.946	14
700	56.0803	1402.78	14
900	118.94	3264.48	15
1100	217.698	5269	14
1300	352.171	8051.97	13

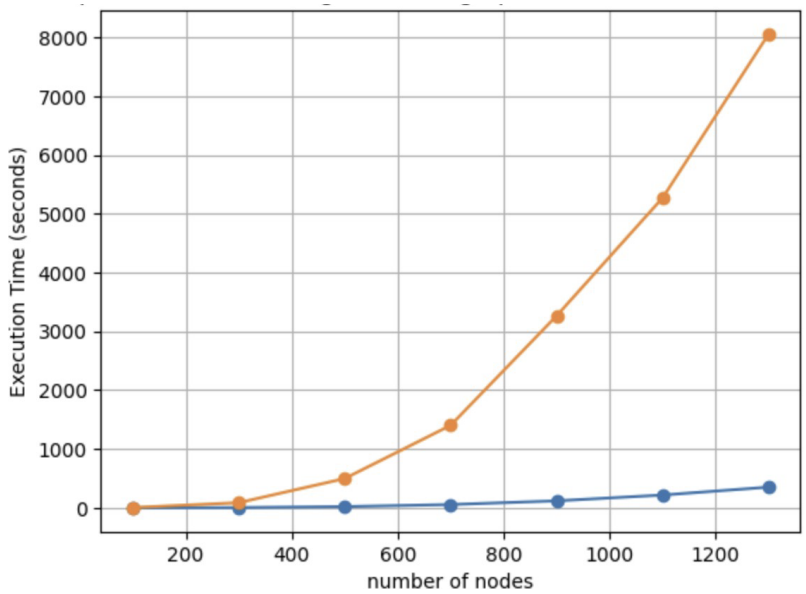

Figure 4.29: Comparisons of the execution times of the all-k algorithm and the fix-k algorithm calling for all $k \leq h$ for graphs with N vertices and $E = 3 \times N$.

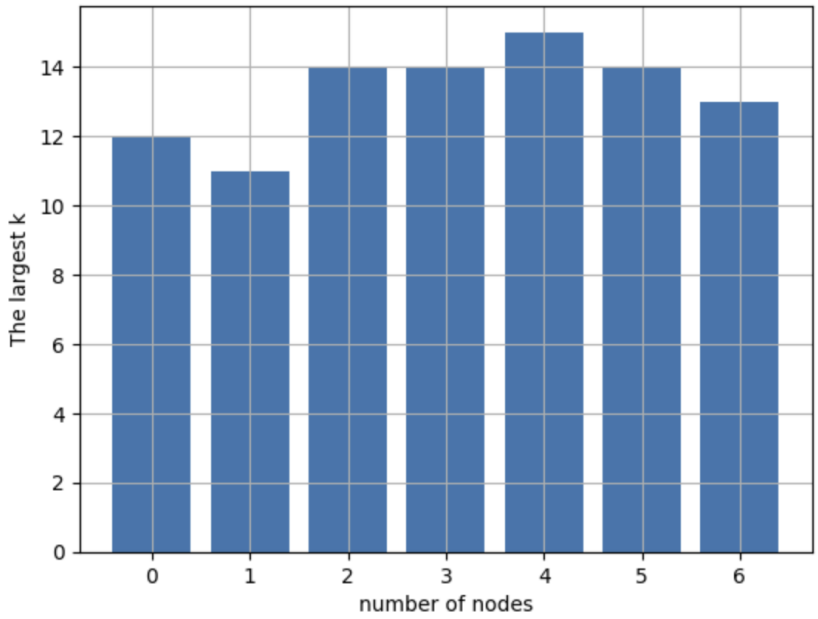

Figure 4.30: Variation of h (the largest k) in the fix-k algorithm for graphs with N vertices and $E = 3 \times N$ edges.

4.3.2 Experiments comparing the all-k algorithm with the fix-k algorithm for solving the fix-k problem

In this section, we employ both the all-k algorithm and the fix-k algorithm to identify k-edge-connected components within a graph for a given k. To accomplish this, we generate graphs with varying numbers of vertices and edges using our random graph generator. Subsequently, we execute both algorithms on these graphs and record their respective running times. The ensuing results are presented below.

Experiments on dense graphs: comparing the all-k algorithm with the fix-k algorithm for solving the fix-k problem

The experiments in this section were conducted on random graphs with varying numbers of vertices, ranging from 100 to 1700, and edges were set to $E = \frac{N \times N}{10}$. For each graph, both the all-k and fix-k algorithms were applied to find the 10-edge-connected components.

Table 4.16: Execution times of the all-k algorithm versus the fix-k algorithm for graphs with N vertices and $E = \frac{N \times N}{10}$ edges, with $k = 10$.

N	Time for all-k algorithm (Seconds)	Time for fix-k algorithm (Seconds)
100	0.397512	0.33202
200	3.33769	3.17864
300	12.6341	12.7251
400	33.4277	35.0479
500	72.7507	77.4947
600	137.637	150.248
700	238.513	261.972
800	386.12	435.652
900	591.244	673.188
1000	866.269	1007.33
1100	1230.92	1419.62
1200	1690.2	1997.25
1300	2278.57	2709.05
1400	3016.79	5450.19
1500	3925.15	4793.57
1600	5012.81	7651.78
1700	6243.97	7485.9

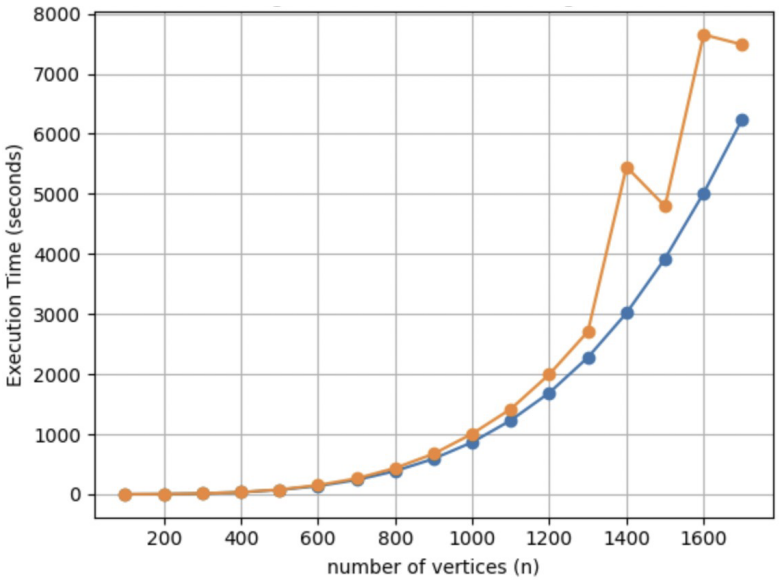

Figure 4.31: The execution times of the all-k algorithm versus the fix-k algorithm for graphs with N vertices and $E = \frac{N \times N}{10}$ edges, with $k = 10$. The orange curve represents the fix-k algorithm.

Comparing sparse graphs: all-k algorithm versus fix-k algorithm for solving the fix-k Problem

For this experiment, we generated graphs with varying numbers of vertices ($N = 100, 300, 500, ..., 3100$) and $E = 5 \times N$ edges using our random graph generator. Subsequently, we executed both the all-k and fix-k algorithms for each graph, with $k = 10$.

Table 4.17: Execution times of the all-k algorithm and the fix-k algorithm for graphs with N vertices and $E = 5 \times N$ edges, with $k = 10$.

N	Time for all-k algorithm (Seconds)	Time for fix-k algorithm (Seconds)
100	0.286285	0.232837
300	5.64356	6.49999
500	22.8506	21.5704
700	57.7319	57.1139
900	120.892	120.396
1100	219.001	217.793
1300	359.302	356.118
1500	548.002	544.804
1700	795.093	789.602
1900	1106.22	1099.26
2100	1489.27	1479.1
2300	1951.01	1939.48
2500	2501.8	2482.11
2700	3145.33	3129.47
2900	3891.77	4012.88
3100	6997.09	7542.7

Indeed, for sparse graphs, the running time of the two algorithms appears to be very similar. However, it's worth noting that there might be some differences for very large graphs, where the characteristics of the graph could have a more significant impact on the algorithms' performance.

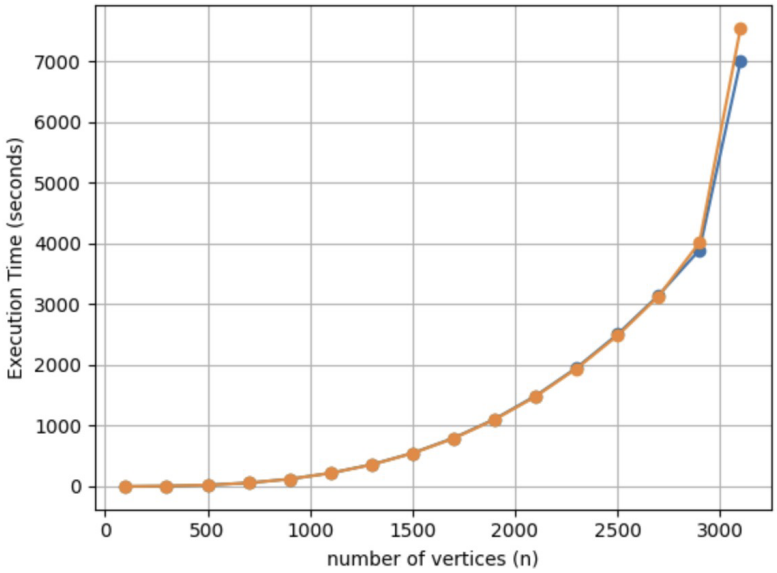

Figure 4.32: Execution times of the all-k algorithm and the fix-k algorithm for graphs with N vertices and $E = 5 \times N$ edges, with $k = 10$. The Orange curve represents the fix-k algorithm.

Chapter 5

Conclusions

In our research, we embarked on a comprehensive study of k-edge-connected components in graphs, presenting algorithms aimed at identifying these components. Algorithm 3 and algorithm 5, introduced in chapter 2, address the problem for a specific value of k, whereas algorithm 8 and algorithm 10 detect k-edge-connected components for all $1 \leq k \leq V$ in a graph $G(V, E)$ with V vertices and E edges.

We established that for any graph $G(V, E)$, there exists a number $1 \leq h \leq V$, such that all k-edge-connected components with $k \geq h$ are singletons (with a size of 1). Leveraging this insight, to solve the all-k problem, we only need to find k-edge-connected components for $1 \leq k \leq h$. Additionally, in chapter 2, we presented an algorithm to compute edge connectivity in graphs.

Almost all algorithms employed in this project utilized Maximum Flow algorithms to compute edge connectivity in graphs. Consequently, we provided a thorough explanation of the concept of maximum flow and the algorithms utilized in this context. Each algorithm presented in our research was implemented and illustrated comprehensively using various examples.

Furthermore, we introduced three different techniques to enhance the time complexity of the fix-k and all-k algorithms, yielding three new algorithms for each problem with time complexities of $O(E \cdot V^2)$, $O(V^3 \cdot \ln^{O(1)}(V))$, and $O(V \cdot E^{1+o(1)})$.

In chapter 4, for a more robust evaluation, our implemented algorithms underwent extensive testing through hundreds of executions across six main categories of experiments. To streamline the evaluation process, we developed an algorithm to generate large random graphs for all experiments.

The experiments of type one evaluated the fix-k algorithm (algorithm 5) by varying the parameter k on various types of graphs, including dense and sparse graphs. The

results highlighted that as k increases, the number of k-edge-connected components in the graph increases until $k = h$. For larger k values ($h \leq k \leq V$), the number of components remains the same, equal to V, and each k-component is a singleton.

Our findings indicate that as the parameter k increases, the execution time of the fix-k algorithm initially increases until a certain point $t = t_1$, then declines until reaching another point $t = t_2$ ($1 \leq t_2 \leq t_1$). Subsequently, it increases again until $t = t_3$ ($t_2 \leq t_3 \leq t_1$) when $k = h$. For $k \geq h$, the execution times remain relatively constant, hovering around the value of t_3.

Experiments of type two assessed how h and the execution time of the fix-k algorithm (with $k = h$) change by increasing the number of vertices in a graph. The results revealed a consistent increase in execution time with the number of vertices, and h for each graph generally increased, albeit with occasional minor fluctuations.

Type three experiments evaluated the fix-k algorithm across various graphs with differing numbers of vertices and edges, demonstrating that both the execution time of the fix-k algorithm and the number of k-components (for a fixed k) increase with the number of vertices in the graph.

Similarly, experiments of type four evaluated the all-k algorithm across graphs with varying numbers of vertices and edges, revealing that the execution time of the all-k algorithm increases with the number of vertices in the graph.

Moreover, experiments were conducted to compare the fix-k algorithm with the all-k algorithm. Type five experiments demonstrated that the all-k algorithm outperforms the fix-k algorithm for solving the all-k problem in both dense and sparse graphs. This is logical, considering that solving the all-k problem using the fix-k algorithm entails running the fix-k algorithm $O(h) = O(V)$ times. Comparing the running time of the all-k algorithm, which is $O(V^2 \cdot E^2)$, with V times the running time of the fix-k algorithm, which is $O(V^3 \cdot E^2)$, the all-k algorithm is $O(V)$ times faster.

Lastly, type six experiments confirmed that the all-k algorithm behaves similarly to the fix-k algorithm in terms of execution time for solving the fix-k problem in both dense and sparse graphs. This outcome is expected because even though the all-k algorithm solves the problem for all k values, it shares the same time complexity as the fix-k algorithm implemented in this project, with both having a time complexity of $O(V^2 \cdot E^2)$.

5.1 Future Works

In Chapters 2 and 3, we introduced novel techniques to enhance the fix-k and all-k algorithms. Through these advancements, we successfully upgraded our implemented algorithm from $O(V^2 \cdot E^2)$ to $O(E \cdot V^2)$, $O(V^3 \cdot \ln^{O(1)}(V))$, and $O(V \cdot E^{1+o(1)})$ utilizing three distinct approaches.

However, a lingering question persists: can we further optimize these algorithms?

A primary avenue for future exploration involves delving deeper into refining the algorithms for identifying k-edge-connected components in graphs. This endeavor holds the potential to unlock even more efficient solutions, thereby pushing the boundaries of what is achievable in graph connectivity analysis.

Additionally, an avenue for improving our algorithms involves leveraging parallelization and threads. By identifying segments of the algorithms amenable to parallel computation, we can define multiple threads to execute them concurrently. This approach holds promise for significantly reducing the execution time of the algorithms.

Furthermore, another potential enhancement lies in employing Linear Programming (LP). By modeling the problem of finding k-edge-connected components as an optimization problem and formulating an LP for it, we may uncover the most efficient algorithm in terms of time complexity. While I have explored this direction, my model remains incomplete, and as such, I have not presented it in this project.

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, page 909–918, New York, NY, USA, 2013. Association for Computing Machinery.

[2] Lijun Chang and Zhiyi Wang. A near-optimal approach to edge connectivity-based hierarchical graph decomposition. *Proceedings of the VLDB Endowment*, 15(6):1146–1158, 2022.

[3] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 205–216, 2013.

[4] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1900–1918. SIAM, 2017.

[5] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Commun. ACM*, 66(12):85–92, nov 2023.

[6] Ye Dinitz and Jeffery Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20:242–276, 1998.

[7] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[8] Shimon Even and R Endre Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 4(4):507–518, 1975.

[9] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[10] Zvi Galil and Giuseppe F Italiano. Reducing edge connectivity to vertex connectivity. *ACM Sigact News*, 22(1):57–61, 1991.

[11] Italiano G. F. Kosinas E. Georgiadis, L. Computing the 4-edge-connected components of a graph in linear time. *ArXiv. /abs/2105.02910*, 76:513–517, 2021.

[12] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[13] Camille Jordan. A note on finding the bridges of a graph. *Journal für die reine und angewandte Mathematik*, 1869(70):185–190, 1869.

[14] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, jul 1996.

[15] Evangelos Kosinas. Computing the 5-edge-connected components in linear time, 2023.

[16] Evangelos Kosinas. Computing the 5-edge-connected components in linear time. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1887–2119. SIAM, 2024.

[17] D. R. Fulkerson L. R. Ford. *Flows in networks*. Rand Corporation Research Studies Series. PUP, 1962.

[18] David W Matula. Determining edge connectivity in 0 (nm). In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 249–251. IEEE, 1987.

[19] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math*, 9(163), 1992.

[20] Hiroshi Nagamochi and Toshimasa Watanabe. Computing k-edge-connected components of a multigraph (special section on discrete mathematics and its applications). *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 76:513–517, 1993.

[21] Heli Sun, Jianbin Huang, Yang Bai, Zhongmeng Zhao, Xiaolin Jia, Fang He, and Yang Li. Efficient k-edge connected component detection through an early merging and splitting strategy. *Knowledge-Based Systems*, 111:63–72, 2016.

[22] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[23] R Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974.

[24] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).

[25] Yung H Tsin. A simple 3-edge-connected component algorithm. *Theory of Computing Systems*, 40(2):125–142, 2007.

[26] Chin FYL Ting H-F Tsin YH Poon S-H Wang T, Zhang Y. A simple algorithm for finding all k-edge-connected components. *PLoS ONE 10(9): e0136264*, 2015.

[27] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. I/o efficient ecc graph decomposition via graph reduction. *Proceedings of the VLDB Endowment*, 2016.

[28] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. Finding maximal k-edge-connected subgraphs from a large graph. In *Proceedings of the 15th international conference on extending database technology*, pages 480–491, 2012.