UNIVERSITA' DEGLI STUDI ROMA TRE

CORSO DI DOTTORATO DI RICERCA IN MATEMATICA

XXIX CICLO

Graph Algorithms for Analytics and Security

Candidate: Nilakantha Paudel _____

Supervisor: Prof. Giuseppe F. Italiano _____

Coordinator: Prof. Angelo Felice Lopez _____

To my family and friends

Nilakantha Paudel

Acknowledgements

Effective people are proactive and provide a holistic, integrated approach towards the personal and interpersonal potency. They have the clear mission and vision of their life to be, lead the work that drives towards their goal, and takes the personal responsibility of any event.

First and above all, I praise God, the almighty for providing me this opportunity and granting me the capability to proceed successfully. Undertaking this Ph.D. has been an utterly life-changing experience for me, and it would not have been possible to do without the support and guidance that I received from many people. This thesis appears in its current form after having the assistance and guidance from them.

First and foremost, I wish to express my deepest gratitude and indebtedness to my supervisor Professor Giuseppe F. Italiano of University of Roma Tor Vergata. His continuous support, invaluable advice, proper and patient guidance with the clear vision, prior to the investigation in spite of his busy schedule, allowed me to pursue the scientific career and provided the platform to grow as a research scientist. I am delighted to work with Prof. Italiano in this exciting and challenging area. His advice and warm encouragement for research as well as on my career has been priceless. I feel that, I am fortunate enough to have an opportunity to work under guidance of a such supervisor, who cares so much not only about the work, responded to my questions so promptly but also understands the personal problems with high sense of humor.

I would like to express my deepest gratitude to my coauthors for their contribution to this work. Especially, my heartfelt appreciation to the assistant professor Loukas Georgiadis of University of Ioannina, Greece, with whom I spent most of my research time in graph theory, it has been a real pleasure to work and discussing on various interesting topics. He convinced me during our many discussions. His insightful ideas and friendly assistance with various problems during our enjoyable collaboration are essential to establish this work. Besides, he is also a reviewer of the thesis; his brilliant comments and suggestions are helpful to improve the thesis in advance. I am also deeply grateful to Blerina Sinaimeri from Université Lyon I-INRIA, France for being my thesis reviewer. Her invaluable advice and feedback are very much helpful to improve the thesis. I am

also thankful to my friend Gaurav Mani Khanal for his constructive comments during the thesis writing.

I would like to express my deepest gratitude to Prof. Francesco Pappalardi and Prof. Valerio Talamanca for their remarkable comments, which improved my thesis. I am also grateful to Prof. Alberto Paoluzzi, Ph.D. coordinator Prof. Luigi Chierchia, secretary Mrs. Francesca Norrito and the other staff members of the Mathematics Department of Roma Tre University for their valuable assistance in the various administrative issues. A very special thanks to Francesco Pappalardi for his support and parental guidance to my stay in Rome, Italy. I will never forget his support and encouragement during his visit in Nepal that led me to start my Ph.D. His support during the difficult period of my Ph.D. career is always memorable.

Finally, my heartfelt thanks go to my dear parents and lovely wife for always believing and encouraging me to follow my dreams. Their tremendous support, giving me the joy even in times of distress and calm on the most stressful days. There is no word to express how grateful I am to my family for all of their sacrifices that they have made on my behalf. Their prayer and spiritual support for me was what sustained me thus far. I can never be thankful enough. Indeed, this thesis is dedicated to them.

Abstract

In the field of Computer Science, Algorithms constitute the core of any nontrivial computation. Thus, we are interested in the analysis of the design space of recent algorithms that perform well in practice. To do so, we choose two distinct areas of Computer Science, Graph Connectivity, and Authentication Systems. Moreover, we compose several different techniques and present new algorithms to solve efficiently few problems in the above areas. Furthermore, we conduct a thorough empirical analysis by implementing the selected algorithms and highlight their merits and weaknesses.

Graphs are fundamental mathematical structures to represent pairwise relationships between objects. Therefore, a graph $G = (V, E)$ is very convenient tool to describe objects, by vertices V and relations between objects, edges E. The pairwise relationship set E contains ordered (resp., unordered) pair of vertices for directed (resp., undirected) graph. In many real-life applications, such an abstract representation may be needed. Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications. For example, in the construction of reliable communication networks, analysis of the structure of networks, transportation, production, scheduling, power engineering, social networks analysis, etc. Hence, our concern is to analyze the connectivity structure of a given directed graph.

Our work on Graph Connectivity is motivated by recent results on 2-connectivity for directed graphs. In particular, we revisit the problem of computing the 2-edge and the 2-vertex-connected blocks and components of a directed graph G. Specifically, we compare two approaches that give $O(m+n)$-time algorithms, where m is the number of edges and n is the number of vertices of G. The first approach is based on a two-level decomposition of G using auxiliary graphs, and the second approach is based on loop nesting information. Our experiments indicate that the loop-nesting-based algorithms are not only faster in practice but also much more efficient in terms of memory usage, especially for sparse graphs. This makes them suitable for the analysis of large-scale graphs. We also note that the performance of the loop nesting computation degrades as the graph density increases, and we propose variants that alleviate this problem. We believe that these variants may be of independent interest since the loop nesting

information is useful in a variety of applications. Then, we consider the the computation of the 2-edge and the 2-vertex-connected components, and investigate how the recent $O(n^2)$-time algorithms and some of its variants that apply to the hierarchical graph sparsification technique, perform in practice. Despite their superior asymptotical worst-case running times, we observe that these algorithms are competitive with simpler $O(mn)$-time algorithms, based on dominator tree decomposition, only in dense worst-case instances.

In addition, we also consider the critical node detection problem in directed graphs. Given a directed graph G and a parameter k, we wish to remove a set $S \subseteq V$ of at most k vertices of G such that the residual graph $G \setminus S$ has minimum pairwise strong connectivity. This problem is NP-Hard, and thus we are interested in practical heuristics. We present a sophisticated linear-time algorithm for the $k = 1$ case, and, based on this algorithm, give an efficient heuristic for the general case. Then, we conduct a thorough experimental evaluation of various heuristics for the critical node detection problem. Our experimental results suggest that our heuristic performs very well in practice, both in terms of running time and of solution quality.

The next problem that we focused is the Authentication Systems for security models. Authentication System is an essential tool for privacy and security such that it allows the user to get access into the system by verifying the user's identity. There are a large number of applications for Authentication Systems, due to the emerging needs of privacy and security in today's digital society. Many people engage in the digital world without being concerned about the privacy and security of their data. One of the reason can be they are using the devices, which has limited hardware configuration and unable to run the security algorithms. Therefore, we concentrate on designing a new Authentication System for a security model that is suitable for low-end devices.

Handwritten Signature Verification is a biometric security method widely used to verify automatically the authenticity of a user signature. In offline systems, the handwritten signature (represented as an image) is taken from a scanned document, while in online systems, pen tablets are used to record the signature, characterized by several

dynamics (e.g., its position, pressure and velocity). We present a new online Handwritten Signature Verification system that is designed to run on low-end devices. Then, we report the experimental observation of our system on different online handwritten signature datasets with low-end mobile devices.

Contents

List of Tables

List of Figures

1

Introduction

Many researchers are working in the field of Computer Science to solve real life problems. The solution presented for a given problem is strongly influenced by the researcher preferences in designing algorithms. It is a challenging job to provide an efficient solution for a problem. An algorithm is an essence of the problem-solving technique; different researchers can develop several different types of algorithms for a single problem. Hence, even a single problem may have various solutions. In this dissertation, we are interested in the design space of algorithms that perform well in practice. To do so, we choose two different areas of Computer Science, "Graph Connectivity" and "Authentication Systems". We then choose and implement newly proposed algorithms that perform well in practice in these areas. Moreover, we also compose several different techniques and present some novel algorithms to solve efficiently a few problems in those selected areas. Furthermore, we conduct a thorough empirical analysis to highlight the merits and weaknesses of each algorithm.

Graph Connectivity focuses on the behavior of the algorithms that compute the structural property of the graphs, and the Authentication System is about the security and privacy that allows the user to get into the system by using his/her online handwritten signature. In this chapter, we present the historical background of Graph Theory, Graph Connectivity, Security and Authentication Systems. We also provide an overview of available recent algorithms, the motivation for our work, and its contributions. Finally, we outline the organization of this thesis.

1.1 Graph Theory

1.1.1 History and Background

Graph Theory originated from the Seven Bridges of Königsberg problem, in 1735. The great Swiss mathematician Leonhard Euler $(1707-1783)$ studied the Köingsberg bridge connectivity problem introducing the notion of Eulerian graph, which made him the father of graph theory. Based on Euler's formula, L'Huillier [108] introduced the concept of Topology, which was later influenced Listing [109] and Cauchy [31]. After a century, in 1840, A.F Möbius presented the idea of a complete graph and bipartite graph. Eventually, in 1845, the German physicist Gustav Kirchhoff discovered the concept of Tree Graph [97], i.e., a connected graph without cycles, and employed graph theoretical ideas in the calculation of currents in electrical networks or circuits. This established an attractive connection between *Graph Theory* and *Linear Algebra*.

Subsequently, in 1856, William R. Hamilton $(1805-1865)$ and Thomas P. Kirkman $(1806-1895)$ studied the cycles on polyhedral, tours that visited certain sites exactly once and then invented the concept of Hamiltonian graph [98]. Cayley [50] studied on specific analytical forms from differential calculus to study the trees, which had many implications in theoretical Chemistry. His work leads to the invention of Enumerative Graph Theory. Cayley applied his results to trees to the contemporary studies on Chemical Composition [32]. The combination of ideas from Mathematics to Chemistry became part of the standard terminology of Graph Theory. Sylvester $(1806-1897)$ in 1878 drew an equivalence relation between "Quantic Invariants" and Covariants of Algebra and Molecular Diagrams [153]. In 1941, Ramsey worked on **colorations**, which is the root of another branch of graph theory called Extremal Graph Theory [153]. In 1936, Dénes König published the first textbook on Graph Theory [173]. A later, Frank Harary wrote another book in 1969 [80]. His text was enormously popular and enabled mathematicians, electrical engineers, physicist, chemists and social scientists to talk to each other. Moreover, Harary donated all of the royalties from his book to fund the Pólya Prize [135].

Thus, the autonomous development of Topology from 1860 to 1930 fertilized Graph Theory and the common development of Graph Theory and Topology came from the use of the techniques of Modern Algebra. The introduction of probabilistic methods in Graph Theory, precisely in the study of Graph Connectivity, gave escalation to another branch, known as Random Graph Theory, which has been a prolific source of graph-theoretic results. The study of asymptotic Graph Connectivity gave rise to Random Graph Theory.

1.1.2 Motivation

Graphs are very convenient tools to describe objects, by vertices and relations between objects, edges. In many real-life applications, such an abstract representations may be needed. Nowadays, it is in use in many branches of mathematics, for example, Group Theory, Matrix Theory, Numerical Analysis, Probability, Topology, Combinatorics, etc. Also, it has been extensively applied in other scientific areas such as Information Theory, Computer Science, Economics, Physics, Chemistry, Electrical Engineering, Architecture, Operation Research, Sociology, Psychology, Genetics, and so on. The reason is that the graphs help as a mathematical models in several systems involving a pairwise relation. Moreover, graphs have an intuitive and authentic appeal because of their ability of diagrammatic representation of the objects and their relationships. In modern Computer Science, Graph Connectivity has made a tremendous algorithmic development under the influence of the theory of complexity and algorithms.

1.1.3 Basic Notions

Graphs are fundamental mathematical structures to represent pairwise relationships between objects. A graph is defined by $G = (V, E)$, which has the set of *vertices* V of size n and the set of pairwise connectivity relation between the *vertices* called *edges* E of size m. If G is *directed* (resp., *undirected*) then E contains ordered (resp., unordered) pair of *vertices*. Our study focuses on algorithms that evaluate the connectivity structure of directed graphs.

Let $G = (V, E)$ be a directed graph (or simply *digraph*), with m edges and n vertices. For two distinct vertices $u, v \in V(G)$, if there exist an edge $e = (u, v) \in E(G)$, then we say that u is *adjacent* to v. A *path* in a graph G is defined by a sequence of vertices $v_0, v_1, \ldots, v_k$ and $k \geq 1$ such that (v_i, v_{i+1}) is an edge in G for $i = 0, \ldots, k-1$. Two different paths are called *vertex-disjoint* (resp., *edge-disjoint*) if they don't have any common vertices (resp., edges). If there exists a path from a vertex u to a vertex v, then we say that vertex v is *reachable* from vertex u. Two distinct vertices u and v of $V(G)$ are said to be *strongly connected* if they are mutually reachable from each other. If every two distinct vertices $u, v \in V(G)$ are strongly connected, then we say that G is strongly connected.

1.1.4 Connectivity

A strongly connected component of G is a maximal strongly connected subgraph of G such that all of its vertices are strongly connected to each other. Hence, if $G = (V, E)$ is not strongly connected, then it contains several strongly connected components. A vertex (resp., an edge) of G is a *strong articulation point* (resp., a *strong bridge*) if its removal increases the number of strongly connected components of G.

1.1.4.1 2-Edge-Connectivity

In a directed graph $G = (V, E)$, two vertices u and v are 2-*edge-connected*, if there are two edge-disjoint directed paths from u to v, and from v to u. Also note that, a path from u to v and a path from v to u need not to be edge-disjoint. We denote this relation by $u \leftrightarrow_{2e} v$. Equivalently, by Menger's Theorem [121]*, $u \leftrightarrow_{2e} v$, if and only if the removal of any edge from G leaves them in the same strongly connected component. We say that G is *2-edge-connected* if $\forall u, v \in V(G)$, $u \leftrightarrow_{2e} v$. Therefore, if G is *2-edge-connected*, then it does not have any strong bridges. The *2-edge-connected components* of G are its maximal 2-edge-connected subgraphs.

*To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

1.1.4.2 2-Vertex-Connectivity

Let us suppose $G = (V, E)$ be a directed graph, two vertices u and v are 2-*vertex-connected*, if there are two internally vertex-disjoint directed paths from u to v and two internally vertex-disjoint directed paths from v to u. Note that a path from u to v and a path from v to u need not be vertex-disjoint. As we did for the 2-*edge-connected* relation, we denote 2-*vertex-connected* relation by $u \leftrightarrow_{2v} v$. Menger's Theorem [121] also leads to an equivalent definition of the 2-*vertex-connected* of a directed graph as follows: Two vertices u and v in G are $u \leftrightarrow_{2v} v$ only if the removal of any vertex different from u and v leaves them in the same strongly connected component. But unlike the 2-*edge-connected* relation, the converse is not always true. It holds only if u and v are not adjacent to each other. The reason is two mutually adjacent vertices are left in the same strongly connected component by the removal of any other vertex, but they are not 2-*vertex-connected*. We say that G is 2-*vertex-connected* if it has at least three vertices and $\forall u, v \in V(G)$, $u \leftrightarrow_{2v} v$. Therefore, if G is 2-*vertex-connected*, then it does not have any strong articulation points. The *2-vertex-connected components* of G are its maximal 2-*vertex-connected* subgraphs.

1.1.4.3 2-Edge-Connectivity Vs. 2-Vertex-Connectivity

Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications [20, 125] such as the construction of reliable communication networks, in the analysis of the structure of networks, transportation, production, scheduling, power engineering, social networks analysis, etc. Hence, in the context of reliable communication, 2-vertex- and 2-edge-connected components correspond, respectively, to parts of a network that are resilient to single vertex and edge failures. These concepts, however, do not capture the pairwise connectivity among the vertices. Indeed, two vertices may lie in different 2-connected components but still be connected by several disjoint paths as shown in Figure 1.1. This observation motivates the following natural 2-connectivity relations [71, 72, 91, 142]. We define a 2-*vertex-connected block* (resp., 2-*edge-connected block*) of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that

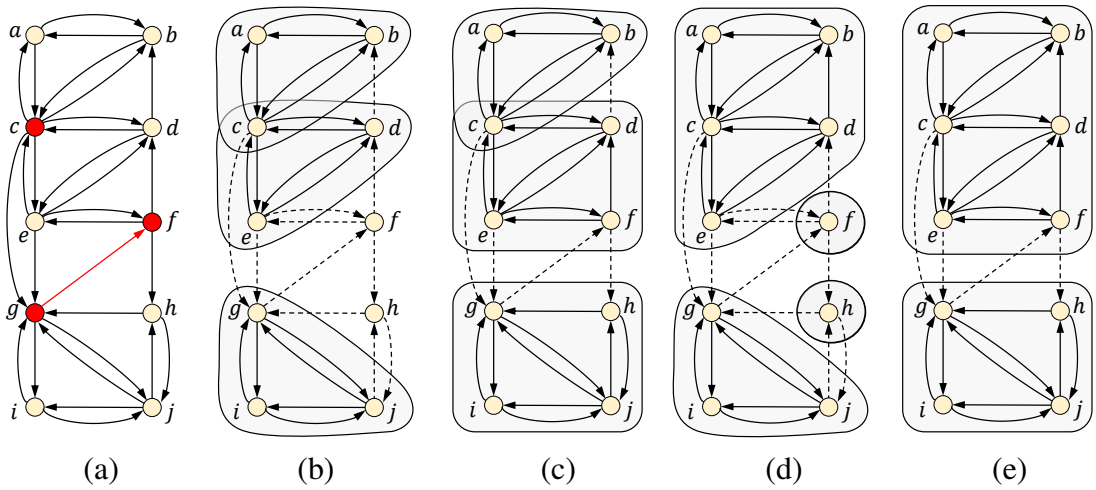


Figure 1.1: (a) A strongly connected digraph G, with strong articulation points and strong bridges shown in red (better viewed in color). (b) The 2-vertex-connected components of G. (c) The 2-vertex-connected blocks of G. (d) The 2-edge-connected components of G. (e) The 2-edge-connected blocks of G. Note that vertices e and f are in the same 2-vertex-connected (resp., 2-edge-connected) block of G since there are two internally vertex-disjoint (resp., edge-disjoint) paths from e to f and from f to e. However, e and f are not in the same 2-vertex-connected (resp., 2-edge-connected) component of G.

$u \leftrightarrow_{2v} v$ (resp., $u \leftrightarrow_{2e} v$) for all $u, v \in B$. Unlike the 2-edge-connected blocks and components, the 2-vertex-connected blocks and components do not define a partition of V, but they can be represented by a tree structure similar to a representation used in [175] for the biconnected components of an undirected graph.

We remark that in digraphs, 2-vertex (resp., 2-edge) connectivity has a much richer and more complicated structure than in undirected graphs. Specifically, the vertex-disjoint (resp., edge-disjoint) paths that make two vertices 2-vertex-connected (resp., 2-edge-connected) in a block, might use vertices that are outside of that block, while in a component, those paths must lie completely inside that component. Hence, two vertices that are 2-vertex-connected (resp., 2-edge-connected) are in a common 2-vertex-connected block (resp., 2-edge-connected block), but not necessarily in a common 2-vertex-connected component (resp., 2-edge-connected component). See, e.g., vertices e and f in Figure 1.1. As a result, 2-connectivity problems on digraphs appear to be

much harder than on undirected graphs. For undirected graphs it has been known for over 40 years how to compute the 2-edge- and 2-vertex- connected components in linear time [154]. In the case of digraphs, however, only $O(mn)$ algorithms were known (see e.g., [91, 92, 112, 127]). It was shown only recently how to compute the 2-edge- and 2-vertex-connected blocks in linear time [71, 72], and the best current bound for computing the 2-edge- and the 2-vertex-connected components is $O(\min\{m^{3/2}, n^2\})$ [36, 84].

1.1.5 Flow Graph

A *flow graph* is a directed graph with a distinguished *start vertex* s such that every vertex is reachable from s. Many algorithms for analyzing a flow graph are based on the *depth first search* (DFS) technique, which explores the graph as deep as possible. In our case, we also use the DFS to create a flow graph G_s from a strongly connected directed graph $G = (V, E)$ by choosing a start vertex s. For example, let us consider a graph shown in Figure 1.2 (i), its flow graph with respected to DFS is shown in Figure 1.2 (ii).

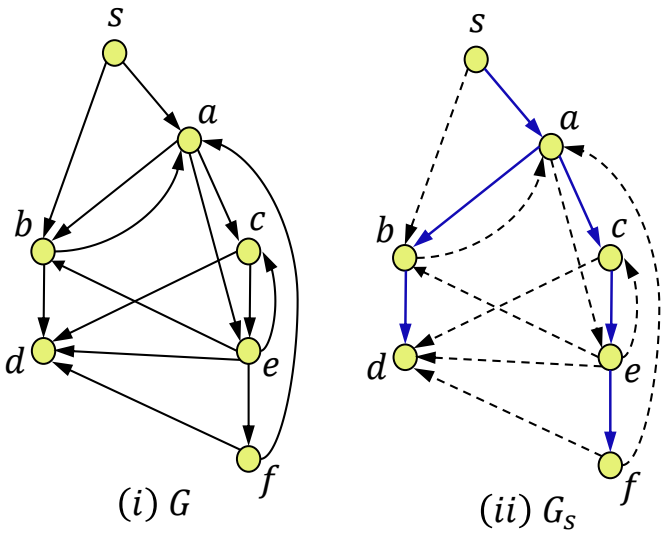


Figure 1.2: (i) Graph G (ii) flow graph G_s of G with respect to depth first search that start from a vertex s, solid edges in blue color represent the DFS edges. (Better viewed in color).

1.1.6 Dominator Tree

In a tree graph, if there exist a path from a vertex u to a vertex v, then we say that u is the *ancestor* of v and v is the *descendant* of u. The vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in G_s contains u as illustrated in Figure 1.3 (i). The dominator relation in G_s is transitive. That is, if vertex x dominates vertex y and y dominates vertex z, then x also dominates z. Thus, we can represent a dominator relation by *tree graph* rooted at s, called *dominator tree* D such that v dominates w if and only if v is an ancestor of w in D. For example, let us consider a graph shown in Figure 1.3 (ii), then its dominator tree is represented by a tree shown in 1.3 (iii). We say that a vertex u $(\neq s)$ is a *non trivial dominator* in D if u dominates at least one vertex v $(\neq u)$.

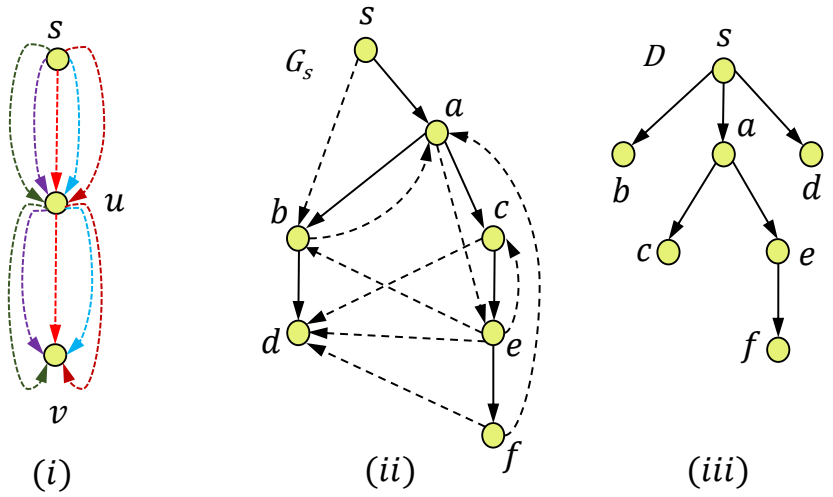


Figure 1.3: (i) Highlevel overview of a dominator relation, (ii) flow graph G_s of a graph G with respect to depth first search that start from a vertex s, solid edges represent the DFS edges, (iii) Dominator tree D of a flow graph G_s. (Better viewed in color).

1.1.7 Loop Nesting Tree

As we already said, in a SCC of a graph G all vertices are strongly connected to each other (i.e., they are mutually reachable from each other). There are several cycles of vertices that can be constructed in a SCC. Moreover, in a strongly connected graph, two

different cycles defined by DFS are either disjoint or one contains the other. Therefore, if we consider a cycle as a loop, then we can represent this relationship between loops in a tree, called loop nesting tree, defined as follows. A *loop nesting tree* represents a hierarchy of strongly connected subgraphs (since a cycle is a strongly connected subgraph) of G_s [161], and is defined with respect to a DFS tree T of G_s as follows. For any vertex u, the *loop* of u, denoted by $loop(u)$, is the set of all descendants x of u in T such that there is a path from x to u in G containing only descendants of u in T. The vertex u is the *head* of $loop(u)$. Any two vertices in $loop(u)$ reach each other. Therefore, $loop(u)$ induces a strongly connected subgraph of G_s; it is the unique maximal set of descendants of u in T that does so. An example is shown in Figure 1.4.

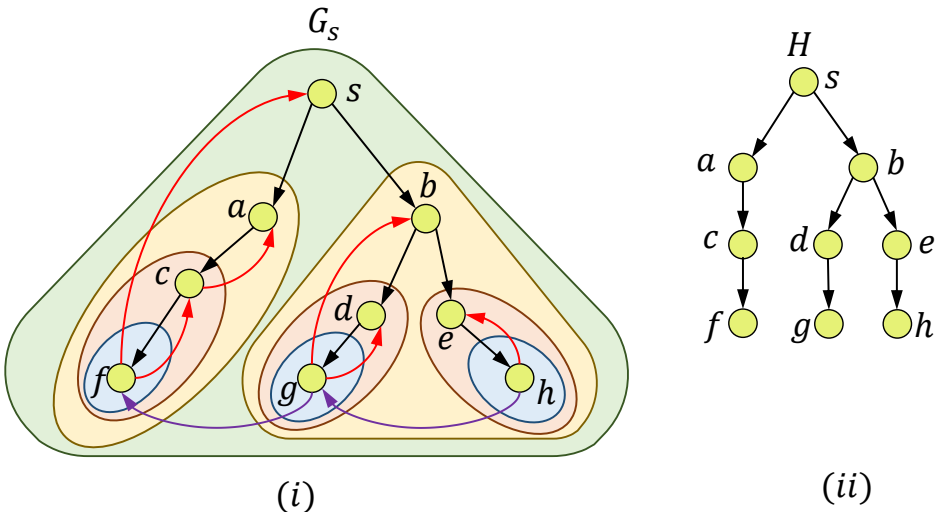


Figure 1.4: (i) flow graph G_s of a graph G with respect to depth first search that start from a vertex s, solid black edges represent the DFS edges, loops $\{e,h\},\{d,g\},\{c,f\},\{b,d,g,e,h\},\{a,c,f\},\{s,a,c,f,b,d,g,e,h\}$ are represented by different color, (ii) loop nesting tree H of G_s (Better viewed in color).

1.1.8 Most Critical Nodes

1.1.8.1 Introduction

We already noticed that if we remove any strong articulation point from a graph, then the graph will be decomposed into several strongly connected components. Let $G = (V,E)$

be a directed graph, and let $C_1, C_2, \ldots, C_\ell$ be its strongly connected components. The *size* $|C_i|$ of a strongly connected component C_i is defined as its number of vertices. We define the *connectivity value of* G as

$$f(G) = \sum_{i=1}^{\ell} \binom{|C_i|}{2}.$$

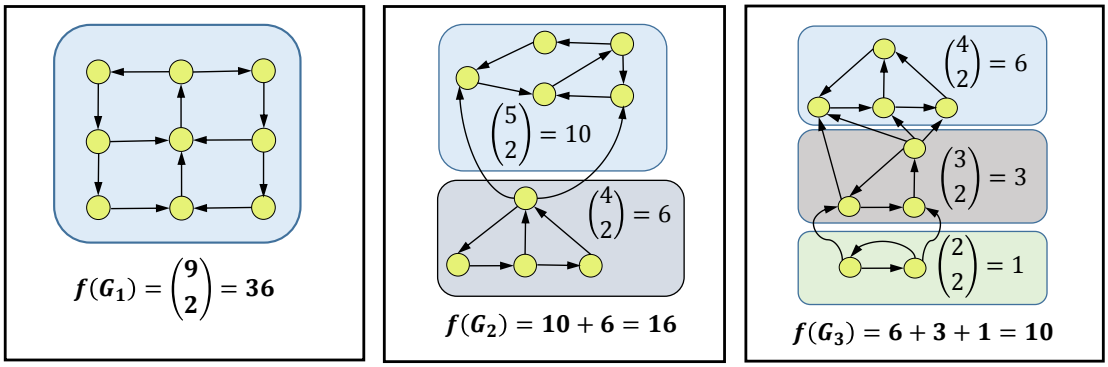


Figure 1.5: Connectivity value of graphs G_1, G_2 and G_3. Even though all of them have the equal number of vertices, their connectivity value are different according to the size and number of SCCs they have.

Note that $f(G)$ equals the number of vertex pairs in G that are strongly connected (i.e., pairwise strong connectivity value). As we can see in Figure 1.5, where all the graphs G_1, G_2 and G_3 have the equal number of vertices but their connectivity value depends on the number of strongly connected vertex pairs. Among the strong articulation points, we observed that there are some distinct vertices whose removal causes the graph to have the minimum pairwise strong connectivity value. These types of particular nodes play the key role in a graph connectivity. We called them highly influential nodes or most critical nodes in a graph.

For example, as shown in Figure 1.6, if we remove any non-strong articulation point like d from G, then it will not affect the number of strongly connected components of G (Figure 1.6 (*vi*)). Therefore, if we remove such non-strong articulation point from a strongly connected graph $G = (V, E)$, and $|V| = n$, then the connectivity value of a graph $f(G)$ will be decreased to $\binom{n-1}{2}$ from $\binom{n}{2}$, which is not a significant decrement. But if we remove any strong articulation points like $\{a, b, c, e, f\}$ from G, then the number

of strongly connected components of G will be increased. Furthermore, it can be seen, if we remove either a or b or c, then the G will have only 2 different strongly connected components. But if we remove the vertex f from G, then G will have the maximum number of strongly connected components, i.e. 5 as shown in Figure 1.6 (iv). Therefore, even though the graph has many strong articulation points, the vertex f is the most critical nodes for the graph presented on Figure 1.6-(i). In different applications of network analysis, we wish to identify the nodes of a network that are *'important'* for a specific task, where the definition of "importance" varies according to the application at hand. For example, one may wish to identify the locations in a network that are useful in order to inhibit the diffusion of contagious [18, 105]. Similarly, critical nodes also help to assess network vulnerabilities [146], or nodes that represent highly influential individuals in a social network [96], etc. Our study considers the problem of detecting a set $S \subseteq V$ of critical nodes such that a directed graph $G \setminus S$ has minimum pairwise strong connectivity. This problem is NP-hard [16, 46], and thus we are interested in practical heuristics.

1.1.8.2 Applications

The critical node detection problem (CNDP) has many applications as already observed in [16]. For instance, it is important in social network analysis, where it can yield a better understanding of several properties, such as centrality, importance and cohesion of specific nodes [21]. It was also applied to the study of covert (or terrorist) networks [103], network immunization [37].

Similarly, CNDP is an essential tool to estimate the vulnerability of supply chain networks. It also has a use for jamming and suppressing on a network. For the jamming, it helps to select those nodes such that whose removal creates the maximum network disruption, and for the suppress, it has a use to determine the nodes that we have to protect from enemy disruptions. Moreover, it helps to neutralize the terrorist activity in the today's digital world. When we collect the data from the social network or by some other intelligence source, it helps to determine the active individuals whose "neutral-

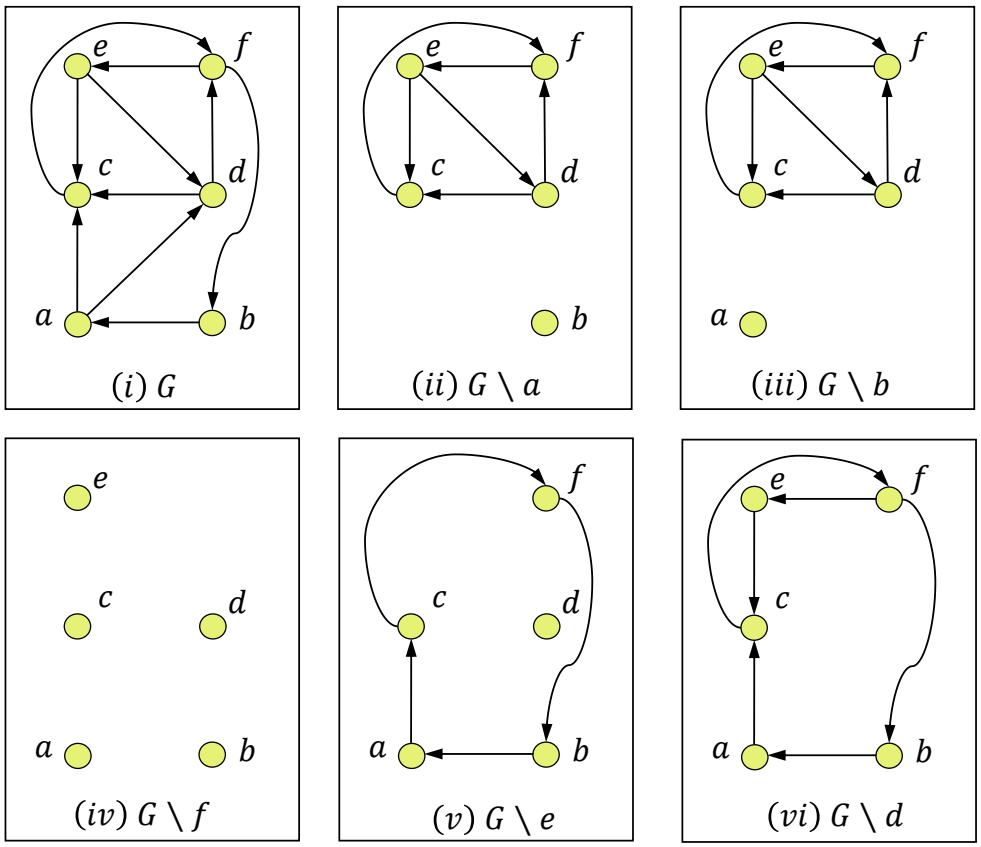


Figure 1.6: A strongly connected digraph G (i), The strongly connected components of $G \setminus v$, for $v \in \{a,b,f,c,d\}$, are shown in figures $(ii),(iii),(iv),(v)and(vi)$ respectively.

ization" will maximally disrupt the communication. This leads us to break down the communication in covert networks. There are some particular social groups of populations, who have the high rates of transmissibility of viruses for which mass vaccination is very expensive. In that case, it would help to determine the appropriate set of individuals to vaccinate so that the spread of the disease or virus could be minimized. In addition, CNDP also has a large number of applications in drug designs. Examining the protein-protein interaction maps, one can determine which proteins cells need to be targeted to destroy the network. That will be the key reference to identify the aggressive cancer cells that have to be removed to slow down the growth rate of cancer. Furthermore, it will help to determine critical roadways to fortify or to repair first, enable mass evacuation of first responders,in the event of a natural disaster, for example, hurricane,

earthquake or flood.

1.2 Authentication Systems

Another problem that we covered in this thesis is the Authentication System for a security model. Authentication [129] is an essential tool of a security model. It is the process that allows the users (or, in some cases, the machines [3]) to get access to the system by confirming their identity.

1.2.1 Categories

Authentication mechanisms are divided into four different categories given below, and each category can follow the different type of methods for the authentication. (See also [166].)

i. **Single Factor:** Single Factor Authentication System is the weakest level of Authentication type, where only a single component is used to verify an individual's identity. Therefore, this type of authentication model is not recommended for most of systems, for example, bank or other financial institution, health or personally relevant organizations that need a higher level of security.

ii. **Two-Factor:** In Two-Factor Authentication System, two different elements are used to verify the user identities, for example, bankcard and a Personal Identification Number (PIN). Moreover, when a user needs to access a very-high-security system physically, then it might also check the height, weight, and biometric security as the face, or the retina scan, or the fingerprint, etc.

iii. **Multi-Factor:** Multi-Factor Authentication System is a better option to enhance the security level than the two-factor authentication level.

iv. **Strong-Factor:** Strong-Factor Authentication System is very similar to Multi-Factor Authentication System or Two-Factor Authentication System, but exceeding those by other rigorous requirements [166]. According to its definition, its

implementation varies and depends on of institution. For example, the U.S. Government's National Information Assurance Glossary [79] defines strong authentication as a layered authentication approach relying on two or more authenticators to establish the identity of an originator or receiver of information. Whereas, the European Central Bank has defined strong authentication in [51] "a procedure based on two or more authentication factors." The using factors must be mutually independent of each other, and at least one factors must be "non-reusable and non-replicable," (except in the case of an inherence factor) and also incapable of being stolen on Internet.

1.2.2 Types

Several methods have been used for the authentication techniques, for example, Remote, IPsec, Network, Logon, etc. [48, 147]. We focus our research on Logon Meth-

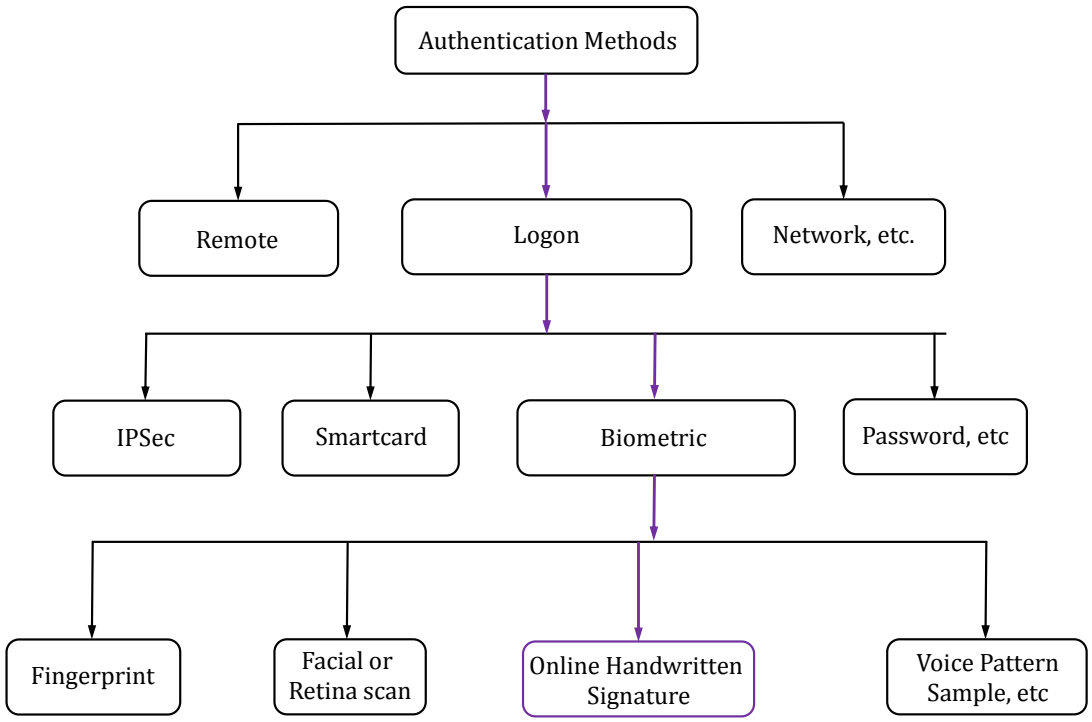


Figure 1.7: High-level ideas of the authentication methods with its hierarchies for a security model.

ods. The Logon Method uses different ways to verify the user identity, for example, password, smart card, biometric recognition, etc. The biometric recognition includes the fingerprint, facial or retinal scan, voice pattern sample, or online handwritten signature. Our study is centered on the online handwritten signature verification for the authentication system illustrated in Figure-1.7. There are tremendous applications of authentication systems, due to the emerging needs of privacy and security in today's digital society (see also the [117]). Many people engage in the digital world without being concerned about the privacy and security of their data because they are using the devices, which has limited hardware configuration and unable to run the security algorithms. Therefore, we concentrate on designing a new authentication system for a security model that is suitable for low-end devices (i.e., devices with limited hardware configuration).

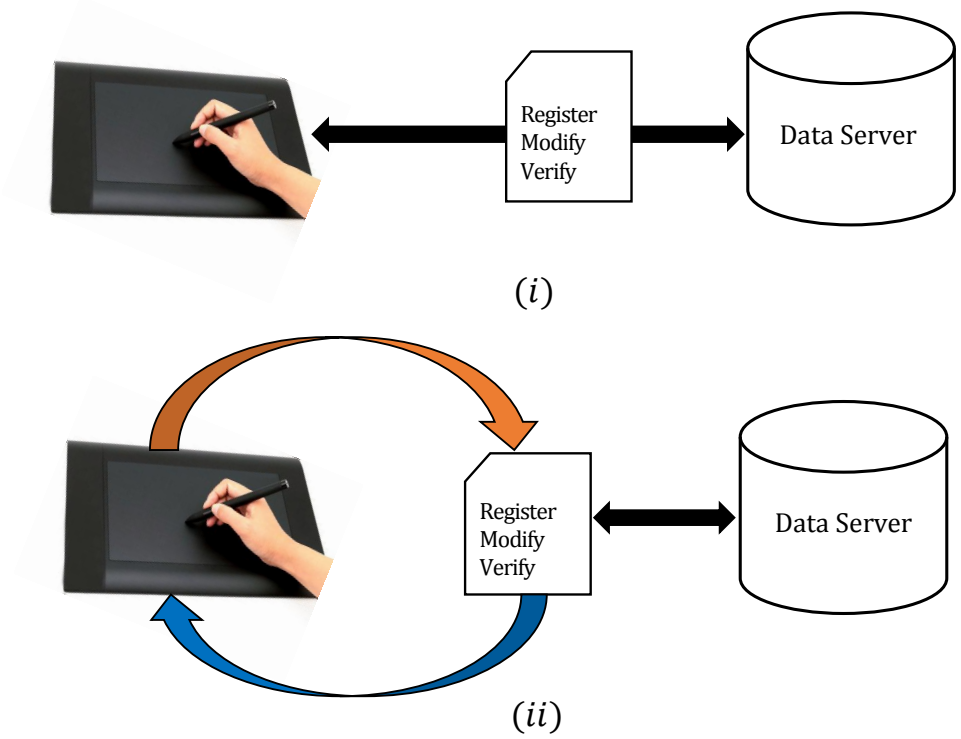


Figure 1.8: Overview of a Handwritten Signature Verification Process.

1.2.3 Online Handwritten Signature Verification

In general, online handwritten signature verification works as follows: a client device takes the data of handwritten signature of a user and then send it to the server for verification. The server processes the input stream and decides whether the input stream corresponds to a genuine signature, thus granting the authorization to the user as shown in Figure 1.8 (i). In our security model, the client itself checks the user identity through an application that does not require to send the data to server for verification as shown in Figure 1.8 (ii). Moreover, if the user modifies the signature, then the changes will be notified to the server.

1.3 Contributions

This thesis examines the efficiency of recent algorithms and presents novel and more efficient algorithms in the field of Graph Connectivity and Authentication systems. They will be detailed in the following subsections.

1.3.1 Graph Connectivity

In the field of Graph Connectivity, we modified the existing algorithms that compute the loop nesting forest, 2-*edge-connected* (resp., 2-*vertex-connected*) *components* (resp., *blocks*) of a directed graph and boost their performances both concerning the memory and the running time. We also have done some experimental observations between the newly available algorithms of 2-*connectivity* of the digraphs, compared their performances reporting our results.

We revisited the problem of computing the 2-edge and the 2-vertex-connected blocks and components of a directed graph G in practice by taking into account the recent theoretical advances in these areas. In particular, we explore the design space of algorithms that perform well in practice by implementing and engineering new existing algorithms. We do this by comparing new implementations against the fastest existing implementations in [44] with a thorough empirical analysis that highlights the merits and weak-

nesses of each technique. Specifically, we present an efficient implementation of new linear-time algorithm for computing the 2-edge-connected and the 2-vertex-connected blocks of G. That are based on loop nesting information [73]. We then compared these algorithms against the algorithms based on a two-level decomposition of G using auxiliary graphs [71, 72] implemented in [44]. To the best of our knowledge, these are the best existing ones.

We consider the computation of the 2-vertex-connected components of G and investigate how recent $O(n^2)$-time algorithms by Henzinger et al. [84], and some of its variants that apply the hierarchical graph sparsification technique of [84], perform in practice. For the computation of blocks, our experiments indicate that the loop-nesting-based algorithms are not only substantially faster in practice but also much more efficient in terms of memory usage, especially for sparse graphs. That makes them suitable for the analysis of large-scale real-world graphs, which are known to be inherently sparse. Furthermore, the loop-nesting-based algorithms are conceptually simpler to implement. Our experiments also highlight that the performance of the loop nesting computation degrades substantially as the graph density increases, and propose variants that alleviate this problem. We believe that these variants may be of independent interest since the loop nesting information is useful in a variety of applications [140, 161]. For the computation of 2-vertex-connected components, our experimental results suggest as following. Even though the algorithms based on hierarchical sparsification presented in [84] are asymptotical superior, they are competitive with simpler $O(mn)$-time algorithms based on dominator tree decomposition presented in [44] only in dense worst-case instances. On the other hand, in general, for the real world graph, simpler $O(mn)$-time algorithms, based on dominator tree decomposition performed better than the hierarchical sparsification algorithms.

1.3.2 Most Critical Nodes

We design a new algorithm to compute highly influential vertex (also called most critical node) of a directed graph in linear time and such that after the removal of a most

critical node, the graph will decompose into minimum pairwise strong connectivity. After removing a most critical node, a strongly connected component of a graph will be decomposed into several strongly connected components. Let us consider a directed graph $G = (V,E)$. We want to find a set $S \subseteq V$ of most critical nodes such that the residual graph $G \setminus S$ has minimum pairwise strong connectivity. This problem is ***NP-Hard*** [16, 46]. A graph can have several strongly connected components. In terms of most critical node, all SCCs are independent of each other. Hence, a critical node detection problem follows the independent set reduction process. Thus, we are interested in practical heuristics. We compared our algorithm with other available heuristics by performing experimental observations. This is a far-reaching step, especially as it was the first real progress on this important natural problem since the foundational work done 15 years ago for undirected graphs.

We present a sophisticated linear-time algorithm to find a *most critical node* of *directed graphs*. That is, given a directed graph $G = (V,E)$ with n vertices and m edges, we identify the most critical node of G in $O(m+n)$ time. As highlighted by several recent results, connectivity-related problems for digraphs are notoriously harder than those for undirected graphs, and indeed many notions for undirected connectivity do not translate to the directed case; see, e.g., [71, 83]. Our algorithm is based on the recent framework of [73] for answering strong connectivity queries in a directed graph under an edge or a vertex failure. A natural extension of this algorithm is to repeatedly remove the most critical node of the current graph G, until we have removed k vertices. Within this process, we obtain an efficient heuristic for the general case that runs in $O(k(m+n))$ time. We assess the performance of our algorithms experimentally. We show that the linear-time algorithm performs very well in practice, while the näive approach of computing $f(G \setminus v)$ for all vertices v is not competitive even for graphs of small size. Furthermore, our heuristic is shown to achieve a much better fragmentation of the input graph compared to selecting nodes by other popular heuristics, such as Page Rank [27], Betweenness Centrality [24], and Maximum Degree.

1.3.3 Handwritten Signature Verification

We designed and developed a new authentication system based on handwritten signature verification method that runs on low-end devices.

Our authentication system is based on the following aspects. Firstly, we defined a method for the verification of signature dynamics which is compatible with wide range of low-end mobile devices, in terms of computational overhead and verification accuracy that doesn't require the special hardware; secondly our new method makes use of several technical features that, to the best of our knowledge, have not been previously used for handwritten signature recognition; finally, in order to assess the verification accuracy of our handwritten signature verification (HSV) system, along with the average computational time, we conduct an experimental study whose results are reported for different data sets of signatures.

1.4 Outline

Chapter 2 contains a brief description of the necessary tools, techniques, and notions on graphs. It also discusses the related previous work, which has already been done by other researchers.

In Chapter 3, we explain the fundamental algorithms and relevant notation that we use throughout our analysis of the graph algorithms. We describe tree graphs, flow graphs, and how to create a flow graph from a given directed graph. After that, we explain the notion of dominators tree; we describe available algorithms to compute the dominator trees of a digraph and its applications. Finally, we provide the reasons why and which algorithm we choose to compute the dominator tree in our analysis. We also discuss the algorithms to compute the strongly connected components of a digraph. That is mainly the algorithms by Tarjan [154] and by Gabow [60]. Similarly, we provide the definition of a loop nesting forest that is defined in [156]. Next, we outline the application, and we explain the available algorithms to compute the loop nesting forest. In addition, we also present a new memory efficient algorithm to compute the loop nest-

ing forest of a directed graph. This is derived from the single pass Tarjan's Streamline version [30]. Then we present the result of empirical studies on those algorithms.

Chapter 4 involves an experimental study of various algorithms that are used to compute the 2-*edge-connected blocks* of a digraph within a linear time bound, available in [71] and in [73] respectively. We discussed their nature and the ideas to process the graphs during the computation. We also explain how to achieve efficient implementations of such standard algorithms. Furthermore, we introduce a new memory efficient version of the algorithm that is inherited from the algorithm presented in [73]. After that, we present a thorough empirical analysis report of these algorithms by using both the real world graphs which are taken from different application domains and synthetic benchmark graphs. All the results of this chapter are ready to be published.

Next, in Chapter 5, we consider the problem to compute the 2-*vertex-connected blocks* of a digraph. There are only two algorithms available with the linear time bound to compute the 2-*vertex-connected blocks* of a digraph. That are presented in [72] and in [73]. We compare the main ideas behind those algorithms and explain why the 2-*vertex-connected block* computation is more complex than the 2-*edge-connected blocks* computation. In addition, we present a linear time memory efficient algorithm that is derived from [73]. After that, we perform an experimental evaluation of the algorithm presented in [72] and in [73] along with our memory efficient version. Finally, we report the results of our experiments. A paper with all the results of this Chapter along with Chapter 4 is ready for publication.

Chapter 6 analyzes recent algorithms that are used to compute the 2-*vertex-connected component* of a directed graph. More precisely, our analysis compares the algorithms available in [44] which has $O(mn)$ time complexity to the quadratic time $O(n^2)$ algorithm presented in [83]. We partly merge these algorithms and design a new hybrid algorithm, which also runs in quadratic time $O(n^2)$. We implement all of these algorithms by using uniform data structures. Our analysis reports thorough the experimental observation shows that the $O(mn)$ time algorithm performs better than other quadratic time algorithms for the real world and normal artificial random graphs. All the results

of this Chapter along with the Chapters 4 and 5 are ready for the publication.

In Chapter 7, we explain our novel sophisticated linear time algorithm to compute most critical node of a directed graph for the $k = 1$, where k is the number of most critical nodes that are supposed to be removed. We also propose two heuristics named as “a maximum number of children in *loop nesting tree*" and “a maximum number of children in *dominator tree*" to find the *most critical node* of a digraph. Then, we conduct a thorough experimental evaluation of various other heuristics such as *Maximum Degree, Page Rank, Betweenness Centrality* for critical node detection problem. The preliminary version of the algorithm, heuristics that are proposed in this Chapter and the experimental reports were presented at the “17^{th} International Conference on Algorithm Engineering and Experiments [132]". A journal publication containing all the results is in preparation.

Next in Chapter 8, we present a new authentication system based on online handwritten signature verification (HSV). We implement the proposed algorithm and perform the experiments of the signatures from various languages on Android version ≥ 4.0. We choose three different testing datasets: on the SigComp2011 Dutch and Chinese datasets [110]; on the SigComp2013 Japanese dataset [113]. The experimental observation produces 95% of correct results for the Chinese, Japanese, and Dutch signatures executed under one second. A preliminary version of this Chapter was presented at the 2^{nd} International Conference on Information Systems Security and Privacy [131]. Moreover, the presented algorithm is published as a book Chapter in “Communications in Computer and Information Science Series” by Springer Publications [133].

Chapter 9 concludes the thesis. It gives a summary of the achieved results for the graph connectivity, critical nodes and security system by pointing out the novelties introduced by the algorithms, their performance, and the produced result. It also provides the open issues that remain to be covered, representing the future works we intend to pursue and some open problems.

The Appendix section contains a brief description of Ackermann’s function, tree traversal methods, pointer machine model, random access model, statement of Menger’s

theorem and a short note of asymptotic notations of algorithms.

Finally, we provide the reference of resources that we used in our work in the bibliography section.

2

Preliminaries and State of the Art

2.1 Introduction

A *graph* G consists of a non-empty set $V(G)$ of elements called *vertices* and a set $E(G)$ which contains the pairwise connectivity relation between the *vertices* called *edges*. We call $V(G)$ (or simply V) the *vertex set* and $E(G)$ (or simply E) the *edge set* of G. We will often write $G = (V,E)$ which means that V and E are the vertex set and edge set of G, respectively. There are two different types of graphs, *directed graph* (or just *digraph*) and *undirected graph*. Throughout the whole thesis, we will use the term *directed graph* and *digraph* interchangeably. We say that G is a *directed* (resp., *undirected*) graph if E contains the *order* (resp., *unordered*) pair of vertices. Moreover, if $e = (u,v)$ is an *edge* of a digraph $G = (V,E)$, then the first vertex u is its *tail* and the second vertex v is its *head*. We also say that the e *leaves* (or *outgoes*) from u and *enters* (or *incomes* or *incidents*) to v. In both (i.e., both the directed and the undirected) type of the graph, we call u and v the *end-vertices*; we say that the end-vertices are *adjacent* i.e., u is adjacent to* v and v is adjacent to u. Furthermore, we say that e is *associated* to the end-vertices.

The above definition allows a digraph G to have edges with the same end-vertices. For example, $e_1 = (u,v)$ and $e_2 = (u,v)$. Here, e_1 and e_2 are called the *parallel* (or *multiple*) edges, that is pairs of edges with the same tail and the same head. If G has parallel edges, then it is called a directed *multigraphs*. Moreover, if an edge $e = (u,u)$ (i.e., edge whose head and tail are coincide), then e is a *loop*. If G has parallel edges and loops, then we called G is a directed *pseudographs*. A simple digraph G does not have any parallel edges and loops. In our work, the definitions and algorithm can be extended

*Some authors use the convention that u is adjacent to v to mean that there is an edge from u to v, rather than just that there is an edge (u,v) or (v,u) in G. We also do the same in our thesis.

to multigraphs or pseudographs, but for simplicity, our input digraphs are simple.

The *reverse graph* G^R results from the *graph* G after reversing all edge directions. G^R is identitical to G for undirected graph.

2.1.1 Size, Degree, Sparse Graph and Complete Graph

The ***size*** $|V|$, of a *graph* $G = (V, E)$, is given by the number of its *vertices*. Therefore, it can be either finite or infinite. Unless otherwise specified, we assume, throughout the whole thesis that our *graphs* are finite (therefore, $|V| = n$, $n \in \mathbb{N}$ and $|E| = m$, $m \in \mathbb{N}$).

The ***degree*** (or *valency*) of a *vertex* is dentoed by $d_G(v)$ and is defined as the number of *edges* associated to it (i.e., $d_G(v) = |\{e \in E : e = (u, v) \text{ or } e = (v, u) \text{ for } u \in V\}|$). For *undirected graph*, it is equal to the number of edges incident to it. For *directed graph*, the degree $d_G(v)$ of a vertex v equals the sum of the number of edges that outgoes (or leaves) from v (i.e., edges that have v on their tail) called *out degree* $\text{Outdeg}_G(v)$ of v plus the number of edges that incidents (or enters) to v (i.e., edges that have the v on their head) called *in degree* $\text{Indeg}_G(v)$ of v. If a *vertex* v has degree 0 (i.e., there is no edges associated with v), then v is called a *isolated vertex*. The *minimum degree* and *maximum degree* of G are given by the numbers $\delta(G) := \min \{d_G(v) | v \in V\}$ and $\Delta(G) := \max \{d_G(v) | v \in V\}$ respectively. If all of the vertices of a graph G have the *equal degree* k, then G is called a *k-regular graph*, or simply *regular graph*. For example, a *cubic* graph is a 3-regular graph. The *average degree* of G is given by the number

$$d_G(G) := \frac{1}{|V|} \sum_{v \in V} d_G(v)$$

The *average degree* globally measures the number of edges of G per vertex. We can see that $\delta(G) \leq d_G(G) \leq \Delta(G)$. The *graph density* is defined by the ratio of edges to vertices i.e., $\varepsilon(G) := \frac{|E|}{|V|}$. Sometimes, the graph density is useful in the analysis of an *algorithm*.

Handshaking Lemma. [†] In a graph, the sum of degrees of *vertices* is equal to twice the number of edges. The quantities d_G and ε are intimately related as following:

$$|E| = \frac{1}{2}\sum_{v \in V} d_G(v) = \frac{1}{2} d_G(G).|V| \implies \varepsilon(G) = \frac{1}{2} d_G(G).$$

A ***sparse graph*** or a minimally connected graph is a graph such that $m = \Theta(n)$[‡], i.e., it has the minimum number of *edges* to connect all *vertices*.

For two distinct vertices u and v of G, if there exist edges $e_1 = (u,v)$ and $e_2 = (v,u)$ in $E(G)$, then we say that G is the *edge-maximal* graph. Therefore, if G is edge-maximal, then G has a property such that for all vertices $u \in V(G)$, no graph $G+(u,v)$ or $G+(v,u)$ does for $v \in \{V(G) - \{u\}\}$. If G is *edge-maximal*, then G is called a ***complete graph*** and $m = \Theta(n^2)$.

2.1.2 Subgraph and Induce Subgraph

We can perform the *set operations* between two graphs illustrated in Figure 2.1.

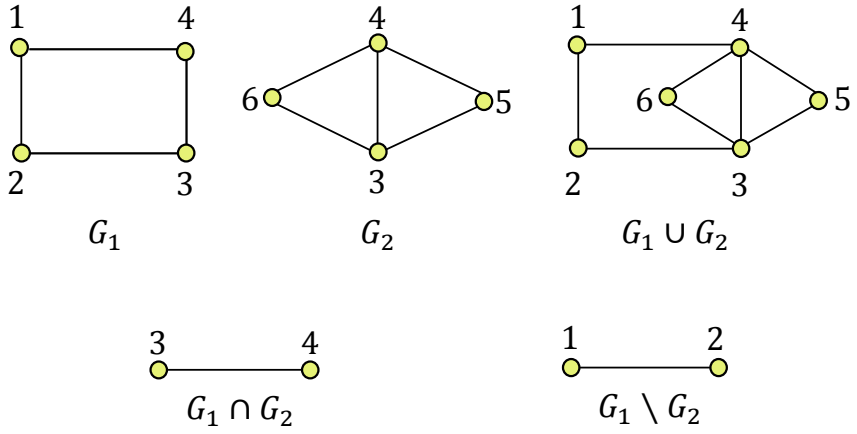


Figure 2.1: Example of *set* operations between the *graphs* G_1 and G_2

[†] The Handshaking Lemma has its origins in Leonhard Euler's famous 1736 analysis of the "Bridges of Königsberg" problem.

[‡] To see the details of the asymptotic notation, please refer the Appendix A.1.

Let $G = (V,E)$ and $G' = (V',E')$ be two graphs. If there exists a bijective function $\phi : V \to V'$ such that for all $u,v \in V$, $(u,v) \in E \iff (\phi(u),\phi(v)) \in E'$. We then called map ϕ is an *isomorphism* and we say that G is *isomorphic* to G'. Furthermore, if $G = G'$ then ϕ is called an *automorphism*. We defined the set $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$. We say that G and G' are *disjoint* if $G \cap G' := \emptyset$. If $V' \subseteq V$ and $E' \subseteq E$ then we say that G' is a ***subgraph*** of G (and G is ***supergraph*** of G'). In this case, we use the notion of $G' \subseteq G$ (i.e., G contains G'). Similarly, if $G' \subseteq G$ and $G' \neq G$, then we say that G' is a ***proper subgraph*** of G. Furthermore, if $G' \subseteq G$ and G' is *isomorphic* to G (i.e., G' contains *all* the edges $(u,v) \in E$ with $u,v \in V'$), then we say that G' is an ***induced subgraph*** of G; we also say that V' *induces* or *spans* G' in G. Figure 2.2 illustrates the example of *subgraph* and *induced subgraph*.

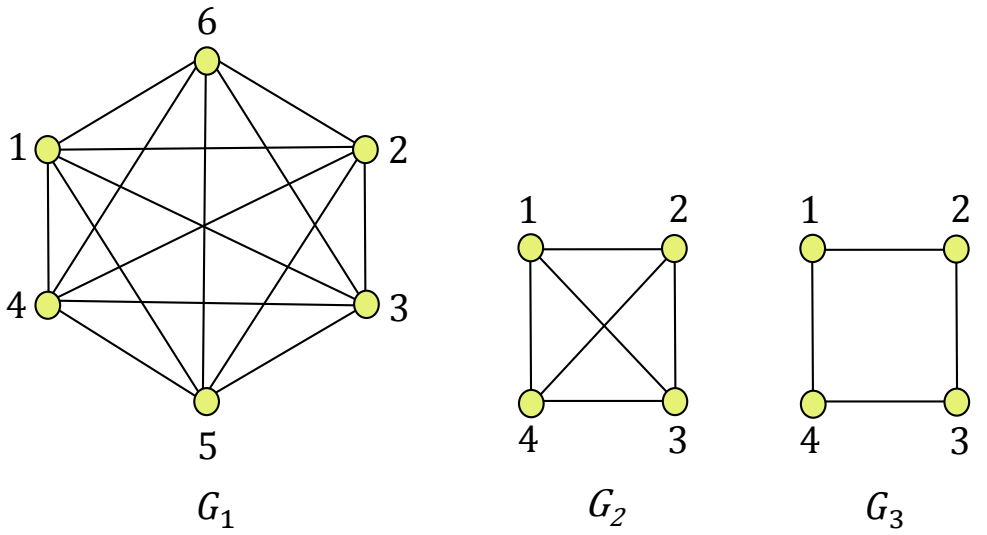


Figure 2.2: Example of subgraph and induce graph, *graph* G_1 with its *subgraph* G_2 and G_3, where G_2 is an *induce subgraph*, but G_3 is only subgraph.

2.1.3 Path and Cycle

A ***path*** in a graph $G = (V,E)$ is defined by a sequence of *vertices* $v_0, v_1, \ldots, v_k$ and $k \geq 0$ such that (v_i, v_{i+1}) is an edge in G for $i = 0, \ldots, k-1$. The *path length* k is defined as its

number of edges. In a graph $G = (V, E)$, if there exists a path from a vertex v to a vertex w, then we say that vertex w is *reachable* from vertex v. Two different paths are called *edge-disjoint* if they don't have any common edge. Similarly, we say that two different paths are *vertex-disjoint* if they don't have any vertex in common.

A ***cycle*** in G is a *path* such that its first and last vertices are same $(v_0 = v_k)$. If two different cycles are cyclic permutations of each other, then it can be considered as a single cycle. A *directed acyclic graph* (or DAG) is a digraph that has no cycles.

2.1.4 Graph Contraction.

Let S be a set of vertices in G, and v be a vertex of G but $v \notin S$. A contraction of S into v means forming a new graph by replacing each edge (x, w) with $x \neq v$ and $w \in S$ by (x, v). Each arc (w, y) with $y \neq v$ and $w \in S$ by (v, y) and then deletes all vertices in S and the remaining arcs leaving a vertex in S. Figure 2.3 illustrates the concept of contraction.

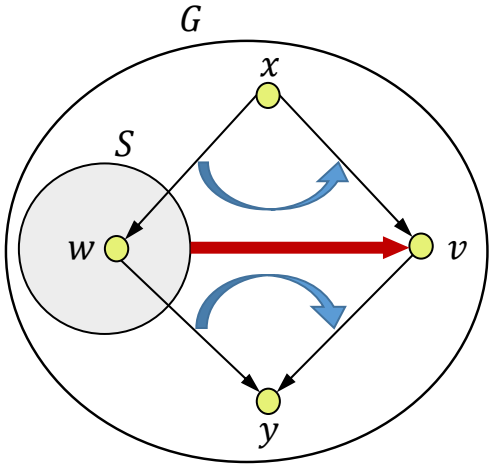


Figure 2.3: Contraction of a set $S \subseteq V(G)$ to a vertex $v \in \{V(G) \setminus S\}$ in digraph $G = (V, E)$.

2.2 Graph Connectivity

2.2.1 Definition.

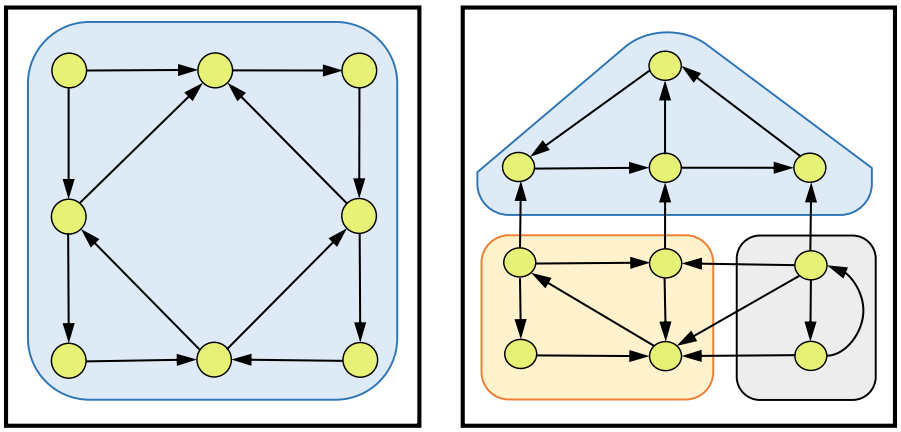
Let $G = (V, E)$ be a directed (resp., undirectd) graph. Two distinct vertices v and w of $V(G)$ are said to be *strongly connected* (resp., *connected*) if there exists a *path* from v to w and a path from w to v (i.e., v and w are mutually reachable from each other). If every two distinct vertices v and w in G are strongly connected (resp., connected), then G is called a strongly connected (resp., connected) graph. Throughout the whole thesis, we assume that our *graphs* are *strongly connected* directed graph unless defined.

2.2.2 Strongly Connected Component

Let us consider a *digraph* $G = (V, E)$. According to the above definition, two distinct vertices v and w in G are strongly connected if they are mutually reachable from each other. Moreover, for every two distinct vertices v and w of G, if v and w are strongly connected, then G is strongly connected. Sometimes, all the vertices of G might not be strongly connected each other (i.e., G is not strongly connected). Therefore, the conecept of *strongly connected component* (SCC) is arised. A strongly connected component of G is a maximal strongly connected subgraph of G such that all of its *vertices* are *strongly connected* to each other. Hence, G is either *strongly connected* or has several *strongly connected components*. For example, the graphs shown in 2.4 (a) is a strongly connected digraph and Figure 2.4 (b) shows three different strongly connected components.

2.2.3 Reverse Digraph

Let $G = (V, E)$ be a digraph, then the ***reverse digraph*** of G results from G by inverting the direction of all the edges, denoted by $G^R = (V, E^R)$. For example, Figure 2.5 illustrates the example of *digraph* and its corresponding *reverse digraph*. The connectivity property of G^R is identitical to that of G. This is proved by following Lemma 2.2.1.



(a) (b)

Figure 2.4: Example of a strongly connected graph (a), and a graph with several strongly connected components (b).

Lemma 2.2.1. *If a digraph $G = (V,E)$ is strongly connected then its reverse digraph $G^R = (V,E^R)$ is also strongly connected.*

Proof. Let $G = (V,E)$ be a strongly connected digraph. Choose two distinct vertices $u,v \in V(G)$. Since G is strongly connected, u and v are also stongly connected, i.e, there exists a path P_1 from u to v and another path P_2 from v to u. When G will be converted into G^R by reversing all edges directions, then all paths will also be reversed. Thus, in G^R, P_1 connects v to u and P_2 connects u to v. This implies that u and v are also strongly connected in G^R. □

2.2.4 Flow Graphs, Dominators and Loop Nesting Tree

A *flow graph* is a digraph with a distinguished *start vertex* s such that every vertex is reachable from s. We let G_s be the flow graph of $G = (V,E)$ with start vertex $s \in V$. Several different techniques are available to explore a flow graph. Typically, we explore a flow graph by executing a *depth-first-search* (DFS), which chooses any vertex $s \in V(G)$ of a graph $G = (V,E)$ and then scan the graph as deep as possible. For example, let us consider a graph shown in Figure 2.6 (i), its flow graph with respected to DFS is

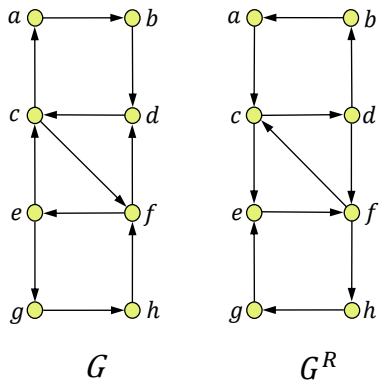


Figure 2.5: Example of graph G and its reverse graph G^R that results from G by inverting the direction of all edges.

shown in Figure 2.6 (ii).

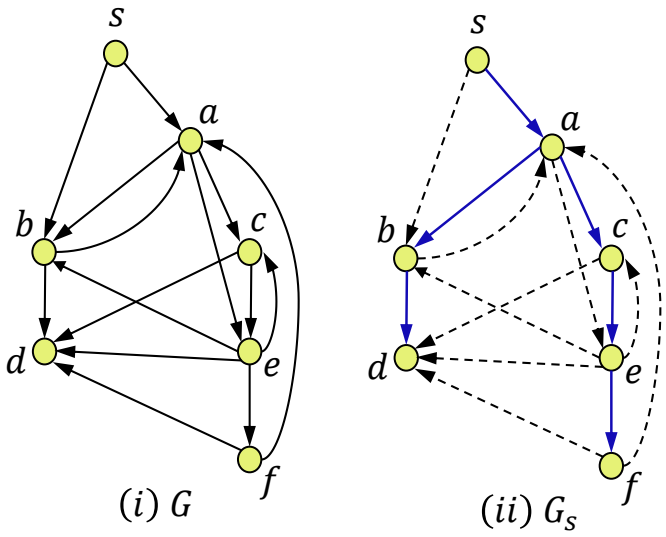


Figure 2.6: (i) Graph G (ii) flow graph G_s of G with respect to depth first search that start from a vertex s, solid edges in blue color represent the DFS edges. (Better viewed in color).

In a tree graph, if there exist a path from a vertex u to a vertex v, then we say that u is the *ancestor* of v and v is the *descendant* of u. A vertex v is a *dominator* of a vertex w (v *dominates* w) in G_s, if every path from s to w contains v as illustrated in Figure 2.7 (i). The dominator relation in G_s is transitive. That is, if $u \in V$ dominates $v \in V$ and v dominates $w \in V$, then u also dominates w. Thus, a dominator relation can be

represented by a *tree graph* rooted at s, called *dominator tree* D such that v dominates w if and only if v is an ancestor of w in D. For example, let us consider a flow graph shown in Figure 2.7 (ii), then Figure 2.7 (iii) represents the dominator tree of a flow graph shown in Figure 2.7 (ii). We say that a vertex v $(\neq s)$ is a *non trivial dominator* in D if v dominates at least one vertex w $(\neq v)$.

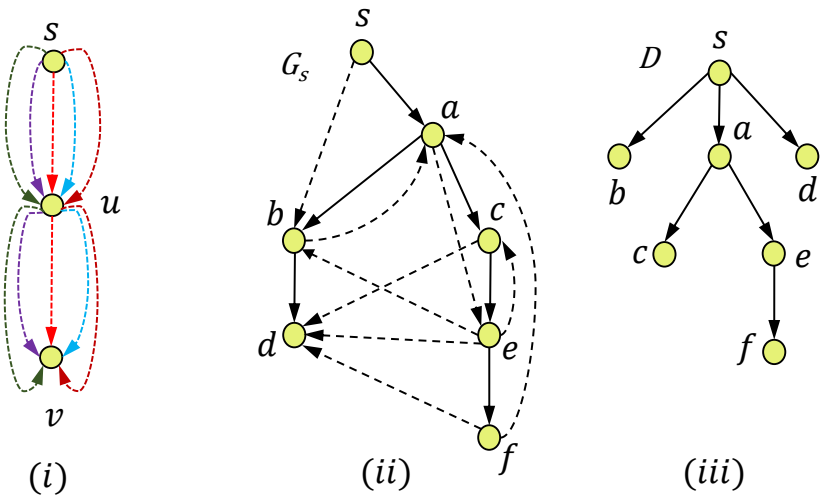


Figure 2.7: (i) Highlevel overview of a dominator relation, (ii) flow graph G_s of a graph G with respect to depth first search that start from a vertex s, solid edges represent the DFS edges, (iii) Dominator tree D of a flow graph G_s. (Better viewed in color).

As we already said all vertices are mutually reachable from each other in a SCC. Several different cycles of vertices can be constructed in a SCC and in a strongly connected graph, two different cycles defined by DFS are either disjoint or one contain other. Therefore, if we consider a cycle as a loop, then we can represent this relationship of loops by a tree called *loop nesting tree*, which represents a hierarchy of strongly connected subgraphs (since a cycle is a strongly connected subgraph) of flow graph G_s [161], and is defined with respect to a *depth-first-search* (DFS) tree T of G_s as follows. For any vertex u, the *loop* of u, denoted by $loop(u)$, is the set of all descendants x of u in T such that there is a path from x to u in G containing only descendants of u in T. The vertex u is the *head* of $loop(u)$. For example, Figure 2.8 (ii) represents a loop nesting tree of a flow graph shown in Figure 2.8 (i). Any two vertices in $loop(u)$ reach

each other. Therefore, $loop(u)$ induces a strongly connected subgraph of G_s; it is the unique maximal set of descendants of u in T that does so. Hence, two differents loops are either disjoint or one contains another.

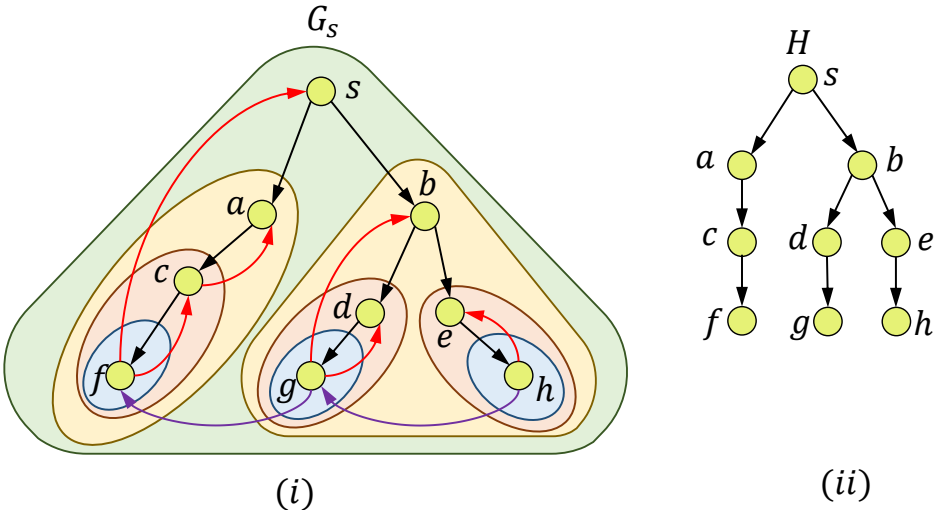


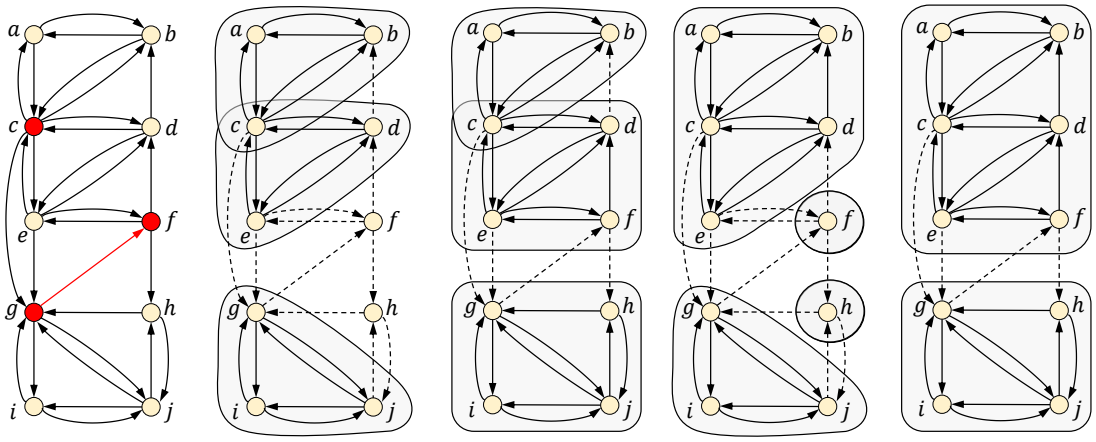
Figure 2.8: (i) flow graph G_s of a graph G with respect to depth first search that start from a vertex s, solid black edges represent the DFS edges, loops $\{e,h\},\{d,g\},\{c,f\},\{b,d,g,e,h\},\{a,c,f\},\{s,a,c,f,b,d,g,e,h\}$ are represented by different color, (ii) loop nesting tree H of G_s (Better viewed in color).

Next Chapter gives a brief description of a tree graph and flow graph, explains the available algorithms to compute the dominators and the loop nesting forest of a flow graph.

2.2.5 Edge Connectivity

Strong Bridge. Let $G = (V,E)$ be a directed graph. An *edge* $e \in E(G)$ is a ***strong bridge*** (SB) in G, if its removal increases the number of SCCs of G, as shown in Figure 2.9-(a), where an edge (g,f) (red color) is a strong bridge. A digraph of size $|V(G)|=n$ can have at most $2n-2$ strong bridges [88]. The SBs of a directed graph can be computed in a linear time [88]. The next Chapter explains the available algorithms to compute the strong bridges, and their relation to the dominator tree.

2-Edge-Connected. Let $G = (V, E)$ be a directed graph. Two vertices $v, w \in V(G)$ are called 2-*edge-connected*, if there are two distinct edge-disjoint paths from v to w and two distinct edge-disjoint paths from w to v. Also note that, a path from v to w and a path from w to v need not be edge-disjoint. We denote the 2-*edge-connected* relation by $v \leftrightarrow_{2e} w$. If $\forall u, v \in V(G)$, $u \leftrightarrow_{2e} v$, then G is said to be 2-*edge-connected*. Furthermore, if G is 2-*edge-connected* then it does not have any strong bridges. Menger's Theorem [121][§] also leads to an equivalent definition of the 2-*edge-connected* of a graph as follows: Two vertices v and w in G are 2-*edge-connected*, if and only if the removal of any edge from G leaves them in the same strongly connected component. Following the Menger's Theorem, it is easy to see that $v \leftrightarrow_{2e} w$ if and only if the removal of any edge leaves v and w in the same strongly connected component.



(a) G (b) $2VCC(G)$ (c) $2VCB(G)$ (d) $2ECC(G)$ (e) $2ECB(G)$

Figure 2.9: An overview of the 2-connectivity of a digraph.

2-edge-connected components. The ***2-edge-connected components*** (2ECC) of a digraph G are its maximal 2-*edge-connected* subgraphs as shown in Figure 2.9-(d).

2-edge-connected block. We define a ***2-edge-connected block*** (2ECB) of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $\forall u, v \in B$, $u \leftrightarrow_{2e} v$. Therefore, in 2ECB, two vertices u and v are in same 2ECB by using the edges, which belong to other 2ECBs

[§]To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

as shown in Figure 2.9-(e).

2.2.6 Vertex Connectivity

Strong Articulation Point. Let $G = (V, E)$ be a directed graph. A vertex $v \in V(G)$ is a *strong articulation point* (SAP), if its removal increases the number of strongly connected components of G. For a digraph, strong articulation points are also called the 1-vertex cuts as shown in Figure 2.9-(a), where the vertices c, f, g (red color) are SAPs. A digraph of size $|V(G)| = n$, could have at most n strong articulation points. It can be realized by the graph of a simple cycle where each vertex is a SAP. The process to compute the strong articulation points of a graph is analogous to the computation of the strong bridges. The only difference is that for SBs, we need to use the connectivity relation between the edges whereas for SAPs we need to use the connectivity relation between the vertices. Therefore, SAPs of a digraph can also be computed in a linear time [88]. We will explain the available algorithms to compute the SAPs and its relation with dominators in next Chapter.

2-Vertex-Connected. Let $G = (V, E)$ be a directed graph, two different vertices $v, w \in V(G)$ are called 2-*vertex-connected*, if there are two internal vertex-disjoint paths from v to w and two internal vertex-disjoint paths from w to v. Note that, a path from v to w and a path from w to v need not be vertex-disjoint. We denote this 2-*vertex-connected* relation between two vertices v and w by $v \leftrightarrow_{2v} w$. If $\forall v, w \in V(G)$, $v \leftrightarrow_{2v} w$, then G is said to be 2-*vertex-connected*. Furthermore, if G is 2-*vertex-connected*, then it does not have any *strong articulation point*. Equivalently, by Menger's Theorem, $v \leftrightarrow_{2v} w$, only if the removal of any vertex different from v and w leaves them in the same strongly connected component. But unlike the 2-*edge-connected* relation, the converse is not always true. It holds only if v and w are not adjacent to each other. The reason is two mutually adjacent vertices are left in the same strongly connected component by the removal of any other vertex, but they are not 2-*vertex-connected*.

2-vertex-connected components. The ***2-vertex-connected components*** (2VCC) of a digraph G are its maximal 2-*vertex-connected* subgraphs as shown in Figure 2.9-(b).

2-vertex-connected block. We define a ***2-vertex-connected block*** (2VCB) of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $\forall u, v \in B$, $u \leftrightarrow_{2v} v$. Therefore, the paths between two vertices u and v of a same 2VCB may contain the vertices and edges that belong to other 2VCBs as shown in Figure 2.9-(c).

2.3 Undirected Graphs Vs Directed Graphs

As we noticed in Figure 2.9, the 2-*edge-connected* (resp., 2-*vertex-connected*) blocks are not identical with the 2-*edge-connected* (resp., 2-*vertex-connected*) components in directed graphs but they are same for the undirected graph. In the directed graph, two different vertices may be 2-*edge-connected* but may lie in different 2-*edge-connected component*. Similarly, two vertices from the different 2-*vertex-connected components* may be in a same 2-*vertex-connected block*. Hence, these characteristics of 2-*connectivity* have the much richer and complicated structure in digraphs. Let us take the example of a 2-*edge-connected*; in the undirected connected graphs, if we remove all the bridges then left connected components are the 2-*edge-connected components* (i.e., also identical with 2-*edge-connected blocks*). But in the case of directed graph, the 2-*edge-connected components*, the 2-*edge-connected blocks*, and the strongly connected components left after the removal of all strong bridges are not necessarily the same. These observations are better explained in Figure 2.10. Therefore, many notions for undirected connectivity do not translate to the directed case; see, e.g., [71, 83]. Moroever, connectivity-related problems for digraphs are notoriously harder than those for undirected graphs. There are very few properties can be translated from undirected graph to directed graph, but they are much more complex to implement in the directed graph.

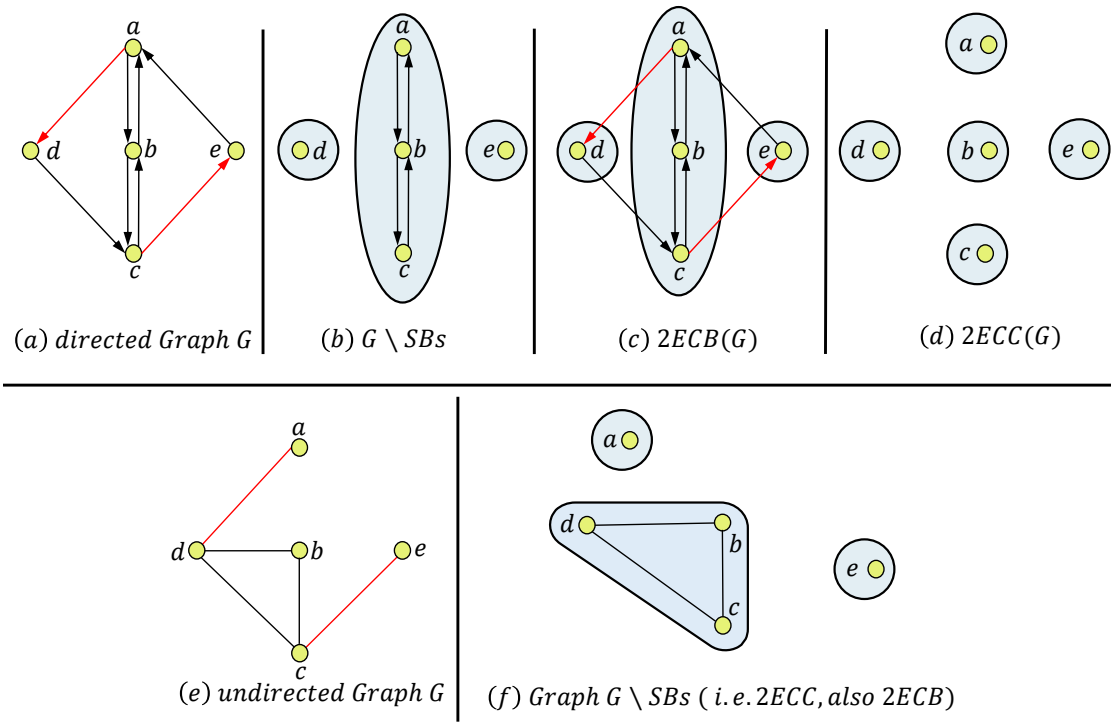


Figure 2.10: (a) A strongly connected digraph G, and its strong bridges, shown in red; (b) The strongly connected components left after removing all the strong bridges from G; (c) The 2-edge-connected blocks of G; (d) The 2-edge-connected components of G; (e) An undirected graph G, and its bridges shown in red; (f) The connected components left after the removal of all bridges of G corresponding to the 2-edge-connected components of G, and to the 2-edge-connected blocks of G. (Viewed better in color.)

2.4 Critical Node

Let G be a directed graph, and $C_1, C_2, \ldots, C_\ell$ be its strongly connected components. The *size* $|C_i|$ of a strongly connected component C_i is the number of vertices in C_i. We define the *connectivity value* of G as

$$f(G) = \sum_{i=1}^{\ell} \binom{|C_i|}{2}.$$

Note that $f(G)$ equals the number of vertex pairs in G that are strongly connected. We can see an example in Figure 2.11, where the graphs G_1, G_2 and G_3 have different connectivity values even if they have the equal number of vetices.

We already explained that the removal of a strong articulation point from a digraph G, disconnects G. If we scruntinize this property of SAPs, we discover the existence

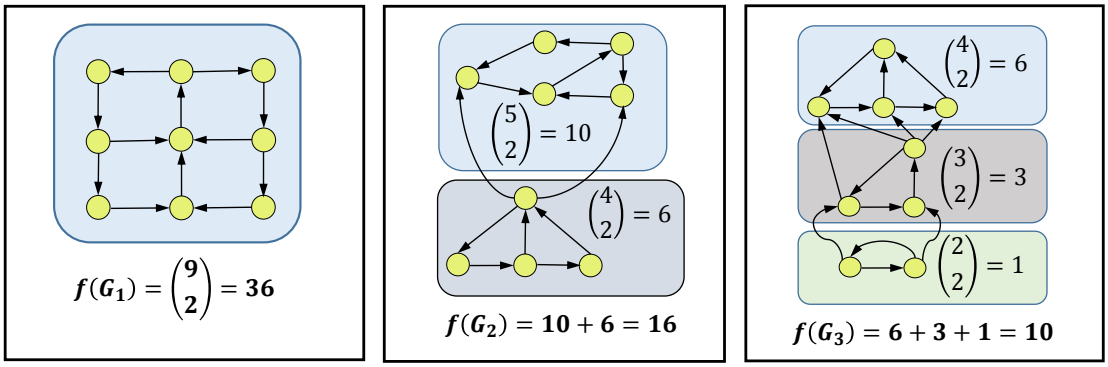


Figure 2.11: Connectivity value of graphs G_1, G_2 and G_3. Even though all of them have the equal number of vertices, their connectivity value are different according to the size and number of SCCs they have.

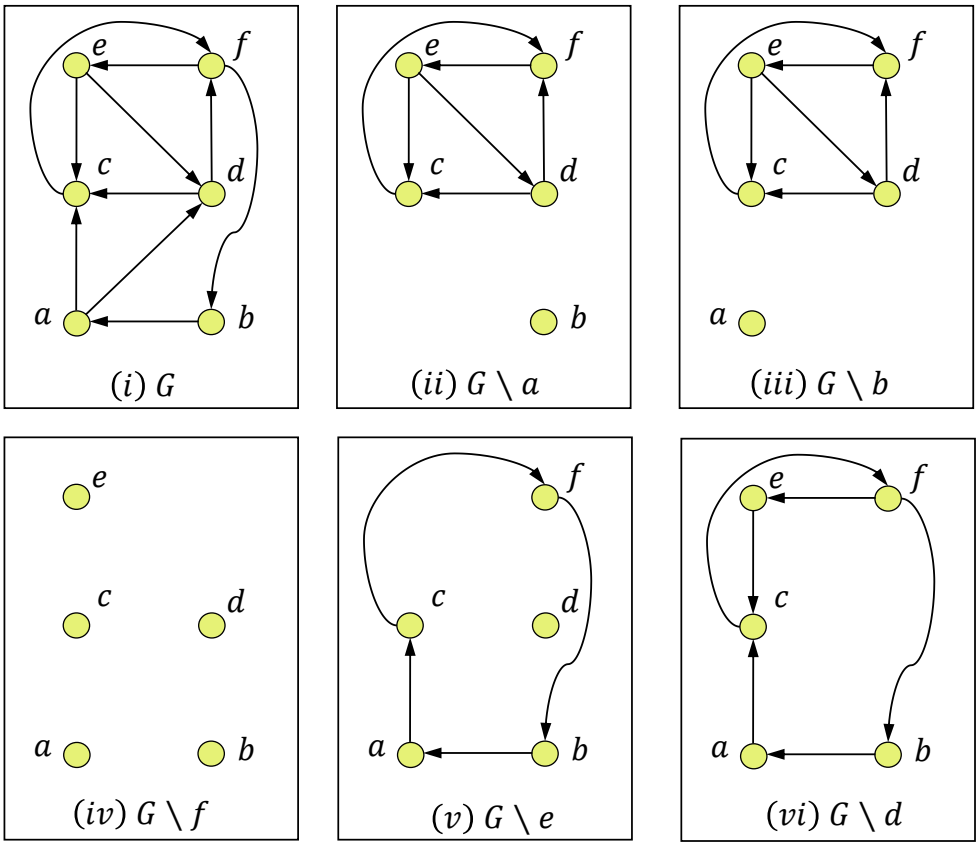


Figure 2.12: A strongly connected digraph $G\,(i)$, The strongly connected components of $G \setminus v$, for $v \in \{a, b, f, c, d\}$, are shown in figures $(ii), (iii), (iv), (v) and (vi)$ respectively.

of some specific vertex whose removal causes the graph to have the minimum pairwise strong connectivity. This type of particular node is called a *highly influential node* or a *most critical node* of the graph. For example, as shown in Figure 2.12, if we remove a non strong articulation point d from G, then G will still be strongly connected and connectivity value of G will be decreased to $\binom{n-1}{2}$ from $\binom{n}{2}$, which is not a significant decrement (Figure 2.12 (*vi*)). But if we remove any of the SAPs $\{a,b,c,e,f\}$, then G will be decomposed into several SCCs. Moroever, if we remove either a or b or c, or e, then G will have only 2 different SCCs (Figure 2.12 (*ii*), (*iii*), (*v*)), but if we remove a vertex f, then G will have the maximum number of strongly connected components, i.e., 5 (Figure 2.12 (*iv*)). Therefore, even though G has several SAPs, here, the vertex f is a most critical node.

Let v be a strong articulation point and let $G \setminus v$ denotes the digraph obtained after deleting the vertex v together with all its incident edges. Let $S \subseteq V$ be a set of at most k vertices. Then $G \setminus S$ denotes the digraph obtained after deleting all vertices in S and their associated edges. If the connectivity value of residual graph $G \setminus S$ is maximally minimized, i.e., if $S = \min_{S \subseteq V} f(G \setminus S)$, then S will be the set of most critical nodes of G.

2.5 Security and Authentication System

The field of Biometrics examines the unique physical or behavioral traits that can be used to determine a person's identity. Biometric recognition is the automatic recognition of an individual based on one or more of these traits. This method of authentication ensures that the person is physically present at the *point-of-identification* and makes unnecessary to remember a password or to carry a token. The most popular biometric traits used for authentication are the face, voice, fingerprint, iris and handwritten signature.

In our study, we focus on "*Handwritten Signature Verification*" (HSV), which is a most common and trusted method for user identity verification. HSV can be broadly classified into *online* and *offline* signature verification, based on the device used and on the method used to acquire the data related to the signature. Offline methods pro-

cess handwritten signatures taken from scanned documents, which are, therefore, represented as images. This means that offline HSV systems only process the 2D spatial representation of the handwritten signature (i.e., its shape). Conversely, online systems use specific hardware, such as pen tablets, to register pen movements during the act of signing. For this reason, online HSV systems can process dynamic features of signatures, such as the time series of the pen's position and pressure. They do so by using specific hardware, such as pen tablets, in order to record pen movements and other variables during the act of signing.

2.6 Related Work

We already explained in section 2.3 such that same problem in the directed graph is notoriously harder to implement than the undirected graph. For undirected graphs, all bridges (resp., articulation points) and 2-*edge-connected* (resp., 2-*vertex-connected*) components can be computed in linear time, essentially by using the same algorithm which used the depth first search presented by Tarjan [154] 40 years ago. Therefore, when the same problem is considered for digraphs, it becomes much more complex and challenging. Italiano et al. [88] presented the algorithms that compute all strong bridges and strong articulation points of a digraph in linear time in 2012. The running time bound for 2-*vertex-connected* (resp., 2-*edge-connected*) components is still not linear. The very recent optimal time bound for 2-*vertex-connected components* (resp., 2-*edge-connected components*) is $O(\min\{m^{3/2}, n^2\})$ [36, 84]. The algorithm presented in [84] improved a previous algorithm where $O(mn)$ time bounds algorithms appeared in [90, 126]. All the algorithms compute the 2-vertex-connected (resp., 2-edge-connected) components of a digraphs following three different ideas: (i) they repeatedly remove the strong articulation point (resp., strong bridge) of a strongly connected component and break it into many strongly connected components until and unless they will not have the strong articulation points (resp., strong bridges). At the end, the remaining graph will become the 2-vertex (resp., 2-edge) connected components; (ii) they repeatedly calculate the forward and reversed dominator tree to find the non-trivial vertex (resp.,

edge) dominators and remove them from the graph to break into its strongly connected components. The process continue until the graph has a non-trivial vertex (resp., edge) dominators. When the graph does not have any non-trivial vertex (resp., edge) dominator, then it would become the 2-vertex-connected (resp., 2-edge-connected) component graph; (iii) they use hierarchical sparsification technique, searching for the strong articulation point (resp., strong bridge) within certain vertices of the graph. If they found any strong articulation points (resp., strong bridge), then break the graph by removing such strong articulation points (resp., strong bridges) to get a 2-vertex-connected (resp., 2-edge-connected) components of the graph.

Italiano et al. [88] presented an alogorithm such that strong articulation points (resp., strong bridges) can be computed in linear time $O(m+n)$. Furthermore, Italiano et al. [88] also proved that, a digraph $G = (V,E)$ of vertex size n can have at most $2n-2$ strong bridges. In case of strong articulation points, we can realized that G can be a simple cycle where each vertex $v \in V(G)$ is a SAP. Hence, G can have at most n strong articulation points. Therefore, if the algorithm follows the (i) technique to get the 2-*vertex-connected components* (resp., 2-*edge-connected components*) of a digraph, then there can be at most $O(n)$ rounds and thus the total time taken by the algorithm is $O(mn)$. Erusalimskii and Svetlov [49] proposed an algorithm that reduces the problem of computing the 2-*vertex-connected components* of a digraph to the computation of the 2-*vertex-connected components* in an undirected graph. For every vertex v, the reduction process repeatedly computes the strongly connected components of all subgraphs $G \setminus v$ and deletes the edges that connect different strongly connected components. This process is repeated until and unless no edge is removed in all current subgraphs $G \setminus v$; the 2-*vertex-connected components* of the resulting digraph G are identical to the 2-*vertex-connected components* of the undirected version of G. On the other hand, Erusalimskii and Svetlov [49] did not analyze the running time of their algorithm. Later, Jaberi [90] showed that the algorithm of Erusalimskii and Svetlov [49] has $O(nm^2)$ running time. Jaberi [90] also proposed two different algorithms to compute the 2-vertex-connected components of a digraph with $O(mn)$ running time. The first algorithm follows the (i)

technique (i.e., it decomposes the digraph by repeatedly removing a strong articulation point at a time). The second algorithm follows the (ii) technique to get the 2-vertex-connected components, dividing the digraph using a dominator tree [106]. Later Di Luigi et al. [44] also proposed an algorithm to compute the 2-*vertex-connected components* of a digraph in $O(mn)$ time using the dominator tree. Though the algorithms presented in [90] and [44] have the same asymptotic behavior, the algorithm proposed by Di Luigi et al. [44] produced better results in practice.

Henzinger et al. [85] first introduced the hierarchical graph sparsification for undirected graphs. Chatterjee et al. [34] and Chatterjee and Henzinger [33] extended this technique to directed graph and to *game graphs* (Consider a directed graph $G = (V,E)$ with a partition (V_1,V_2) of V, which is called a game graph [35].), respectively. The sparsification technique allows to replace the 'm' in the $O(mn)$ running time by an 'n', yielding $O(n^2)$. Henzinger et al. [84] define a 2-*isolated set* of a digraph $G = (V,E)$, where G is not necessarily strongly connected, to be a set of vertices $S \subseteq V$ such that (a) cannot be reached by the vertices of $V \setminus S$ or (b) can be reached from $V \setminus S$ only through one vertex v. Every 2-*vertex-connected component* of G contains either only vertices of $S \cup \{v\}$ or only vertices of $V \setminus S$. Hence, if such a set S is found, we can compute recursively the 2-vertex-connected components in the subgraphs induced by $S \cup \{v\}$ and $V \setminus S$ respectively.

The algorithm of Henzinger et al. [84] is based on a fast computation of 2-isolated sets using subgraphs of the input digraph G. As shown in [84], a 2-isolated set S of type (a) can be found by computing strongly connected components. Similarly, a 2-isolated set S of type (b) can be found by computing dominators in a suitably defined flow graph. In order to find 2-isolated sets fast, Henzinger et al. [84] apply the *hierarchical sparsification*. They start the search for a 2-isolated set in a subgraph G' of G such that $\forall v \in G'$, $\text{Indeg}_{G'}(v) = \text{Indeg}_G(v)$ for the first 2^i incoming edges in E. If no 2-isolated set is found, they repeatedly increase i by 1 until the search is successful or $G' = G$. In this way, they showed that the search takes time $O(n)$ per vertex in the 2-isolated set, which gives an $O(n^2)$ total time bound.

We can extend the notion of 2-*connectivity* to the notion of *k-connectivity* define as follows. Any digraph is called the *k-vertex-connected*, if any nonadjacent vertices of the graph will be still in the same strongly connected components after removing the $k-1$ vertices. Similarly, if the vertices are in same strongly connected components after removing the $k-1$ edges, then the graph is called the *k-edge-connected*. The computation of the k-edge-connected components of a digraph was first considered by Matula and Vohra [115], where they gave an $O(n^3)$ time bound algorithm. Henzinger et al. [84] also extended their quadratic time algorithm to compute the k-vertex-connected (resp.,k-edge-connected) components of a digraph which has the $O(n^3)$ (resp., $O(n^2 \log n)$) time bound.

The algorithms for the 2-vertex-connected (resp., 2-edge-connected) blocks were developed in 2014. Jaberi [89] proposed the algorithms to compute the 2-vertex-connected blocks of any digraph $G = (V,E)$ in $O(mn)$ time. Jaberi [89] also presented an algorithm to compute the 2-edge-connected blocks of any digraph in $O(n \min\{m, b^* n\})$ time where b^* is the number of strong bridges in G. Later, Georgiadis et al. [71] proposed the three different algorithms for the 2ECB computation in a digraph G; (i) simple iterative algorithm; (ii) recursive algorithm with $O(mn)$ time complexity; (iii) fast algorithm with linear time $O(m+n)$ bound. Their fast algorithm with linear time bound is the first algorithm to compute the 2-edge-connected blocks of a digraph in linear-time. Again in 2015, Georgiadis et al. [73] proposed an algorithm for the 2ECB computation based on *loop nesting tree* and *dominator tree* information. For the 2 vertex-connected block computation, in 2014, Georgiadis et al. [72] presented two different algorithms, (i) simple with $O(mn)$ time bound and (ii) fast with linear time bound $O(m+n)$. Again in 2015, Georgiadis et al. [73] presented the algorithm to compute the 2VCB of a digraph in linear time, as in the case of 2ECB, the algorithm uses the *loop nesting tree* and *dominator tree* information.

Di Luigi et al. [44] performed the first experimental study on 2-vertex-connected (resp., 2-edge-connected) components (resp., blocks) of a directed graph. They compared linear-time algorithm for computing the 2-edge-connected blocks to simple $O(mn)$-

time algorithms that are presented in [71]. Similarly, for the 2-vertex-connected blocks, their experimental observation compared the linear time algorithm with the simple $O(mn)$ time algorithms that are available in [72]. In addition, Di Luigi et al. [44] also compared the running time of the algorithms that compute the 2-vertex-connected component that are presented in [49], [92], and [44] and have the running time $O(m^2n)$, $O(mn)$, and $O(mn)$ respectively.

For the critical node detection problem (CNDP), to the best of our knowledge, we present a pioneer algorithm to find the most critical node (one vertex cut) off a directed graph. The previous algorithms were designed for the undirected graphs and we already explained that not all the properties of the undirected graph can be translated to directed graph. In fact, the properties which can be transformed from undirected graph to directed graph are much more complex in the latter case. Nevertheless, here we are going to discuss the historical development of the algorithms to compute the most critical node of the undirected graph. The CNDP problem is related to variety of graph partitioning problems in the literature and has been extremely active research area from last 15 years. The graph will be divided into several partitions after the removal of highly influential nodes.

Addis et al. [2] define a *dynamic programming* recursion that solves the CNDP in polynomial time when the graph has bounded *treewidth*. The treewidth of G is the minimum width of a tree decomposition of G. The worst-case complexity of their algorithm is $O(n^3k^2)$, where k is the number of critical nodes that are supposed to be removed from the undirected graph. Di Summa et al. [45] presented an integer linear programming model of the branch and cut algorithms, where non-polynomial numbers of constraints are provided. Again Veremyev et al. [171] and Veremyev et al. [172] reformulated a CNDP algorithm that requires $\Theta(n^2)$ constraints and ascertained the optimal solutions for graphs. Apart from the algorithms, many heuristics are available to decompose the graph by removing the most critical nodes. Arulselvan et al. [17] used a solution to the maximum independent set problem and tested it on a limited number of network structures with promising results. The maximum independent set problem starts from

a local search; it repeats the process until and unless the desired termination criteria is reached. Two stochastic search algorithms and randomized rounding-base algorithm are developed in [167] and [169, 170] respectively. Moreover, Dinh et al. [47] proposed a pseudo-approximation algorithm with $O(\log n \log(\log n))$. Note that, the given time bound is for the approximation ratio. Very recently, Ventresca and Aleman [168] presented a linear-time algorithm for the $k = 1$ case in undirected graphs. Their algorithms exploits the relation between depth-first search DFS and articulation points and biconnected components of an undirected graph [154].

Regarding the online handwritten signature verification(HSV), it suffers from several limitations. In fact, handwritten signatures are usually acquired using digitizing tablets connected to a computer. Due to the limited hardware configuration capacity of common low-end mobile devices (such as mobile phones), they may not be able to support the verification algorithms or may be too slow to run the verification algorithm (due to limited computational power). As a result, the range of possible uses of the verification process is strongly limited by the hardware needed. The systems available in [41, 111, 151, 165, 176] can address only partially these issues: they are supported by mobile devices, but they are not inherently designed for common low-end devices such as mobile phones; several approaches make use of pen pads (special purpose hardware for handwriting), signature tablets (special purpose desktop and mobile hardware for signing), interactive pen displays (complete instruments for working in digital applications), Kiosk systems and PC Tablets. As for the online HSV systems described in [23, 104, 120], even if experiments related to online HSV were carried out on low-end devices in order to evaluate the verification accuracy, no analysis addressing the computational time is used in the algorithm design (which is particularly important, due to the limited computational power of mobile devices). In our work, we develop a new algorithm for low-end devices and performed an experimental analysis, which is specially focused on the computational time in those types of devices.

3
Fundamental Algorithms

3.1 Tree Graphs

A ***Tree graph*** is a graph, for which there exist a vertex called *source* (or *root*) such that there is a unique simple path from it to every other *vertex*. The *root* vertex is the head of the edges; it is unique for directed graphs but there could be several for undirected graphs. Let T be a *directed tree graph*, v and w be the vertices of T, and $e = (v,w)$ be an *edge* of T. Then the notation " $v \rightarrow w$ in T" (or $v = t(w)$) means that v is the *father* (or *parent*) of w and w is a *son* (or *child*) of v. Every *vertex* except the *root* has its unique parent in T. Moreover, the notation " $v \xrightarrow{*} w$ in T" (or $T[v,w]$) implies that there exist a *path* from v to w in T. In this case, v is an *ancestor* of w (*proper ancestor* if $v \neq w$, denoted by $T(v,w]$ or $v \xrightarrow{+} w$), and w is a *descendant* of v (*proper descendant* if $v \neq w$, denoted by $T(v,w]$). Every vertex is a *non-proper* ancestor (resp., descendant) of itself.

Let T and T' be two tree graphs such that $T' \subseteq T$, then T' is called a *subtree* of T. For any rooted *tree* T and *vertices* $u,v \in V(T)$, T_u (resp., $T(v)$) denote the *subtree* of T rooted at u (resp., a tree contains v). Similarly, $A(v)$ (resp., $\widetilde{A}(v)$) and $D(v)$ (resp., $\widetilde{D}(v)$) represent the set of an *ancestor* (resp., *proper ancestor*) and the set of *descendants* (resp., *proper descendants*) of v in T. Figure 3.1 illustrates these notations.

We can extend the *ancestor* (resp., *descendant*) relationship to edges illustrate in Figure 3.2 as following. Let us consider the distinct *vertices* $u,v,w,z \in V(T)$ and *edges* $e_1 = (u,v)$, $e_2 = (w,z) \in E(T)$. The *edge* information give us that, u and w are the parent of v and z respectively. Moreover, If z is the *proper ancestor* of u then also of v. It implies that w,z and e_2 are the *proper ancestor* of e_1 and e_1 is the *proper descendant* of w,z and e_2, see Figure 3.2 (i). Similarly, if w is the *proper descendant* of v then also of u, which implies that w,z and e_2 are the *proper descendants* of e_1 and e_1 is the *proper*

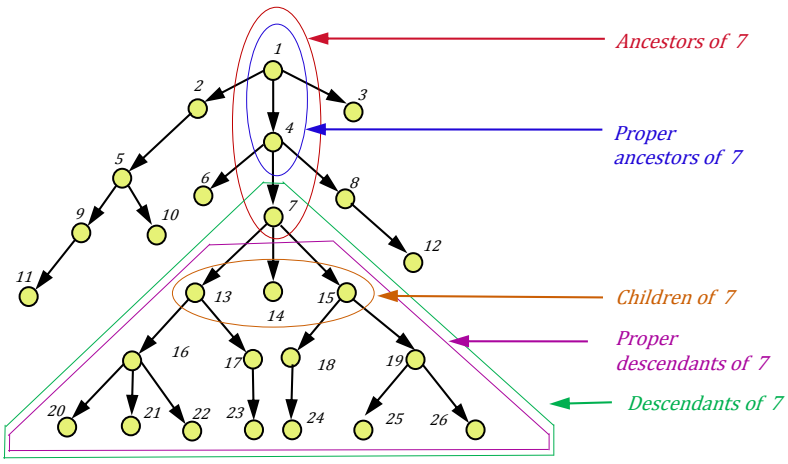


Figure 3.1: Tree notations, Let us consider the vertex 7, then the ancestors $A(7) = \{1,4,7\}$, proper ancestors $\widetilde{A}(7) = \{A(7) - \{7\}\} = \{1,4\}$, children $C(7) = \{13,14,15\}$ descendants, $D(7) = \{7,13,14,15,16,17,18,19,20,21,22,23,24,25,26\}$, and proper descendants, $\widetilde{D}(7) = \{D(7) - \{7\}\} = \{13,14,15,16,17,18,19,20,21,22,23,24,25,26\}$. (Better viewed in color.)

ancestor of w,z and e_2 as shown in Figure 3.2 (ii). If T is a *tree* and contains all the *vertices* of a graph G, then T is called the ***spanning tree*** of G.

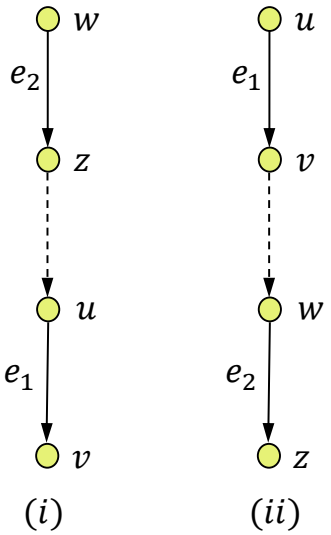


Figure 3.2: Extended notions of a tree graph

3.2 Flow Graph

We construct the *flow graph* of *strongly connected digraph* $G = (V,E)$ by choosing a *start vertex* $s \in V(G)$. If G is not *strongly connected*, then we can do the same process for each strongly connected component of G separately. Without loss of generality, let us assume that all graphs are *strongly connected*. Under consideration, for all the vertices $v \in \{V(G) - \{s\}\}$, there exists a *path* from s to v and vice-versa in both G and G^R. In the whole thesis, we fixed the arbitrary *start vertex* s for the *flow graph*, and to avoid the ambiguities; we denote the flow graphs of G and G^R (i.e., reverse graph of G) by G_s and G_s^R respectively unless otherwise stated. To create the *flow graph* we have to explore the outgoing edges of the newly visited *vertex*. Many algorithms for analyzing a flow graph are based on the *depth-first search* technique. We also explore a flow graph by executing a depth-first search procedure that we are going to explain in next section.

3.3 Depth-First Search

The ***depth-first search*** (DFS) is a fundamental tool to design efficient algorithms in graph theory. It has been used for finding solutions to problems in combinatorial theory and artificial intelligence [78, 128]. In this method, for any graph $G = (V,E)$, all the *vertices* are unexplored at the beginning. To start the process, it chooses a *vertex* $s \in V(G)$ as a *root* and then starts to explore the *graph* in "*depth*" as much as possible by seeking the *outgoing edges* from the most recently discovered vertex. It marks the vertex w as visited when it meets the w at the first time. After that, if there is an unexplored edge $e = (v,w) \in E(G)$ being explored then it ignores the both e and w. The process will continue until and unless all the *vertices* have been discovered. Procedure CreateDFS and DFS explain the pseudocode of this process. It is evident that, all the *vertices* that are reachable from the root will be discovered. If the graph is not strongly connected, then some vertices will remain to find. In that case, it will select a new vertex as a root from the remaining vertices and repeats the same process. But in our case, it will not happen, since we already considered only strongly connected graphs

for the simplicity.

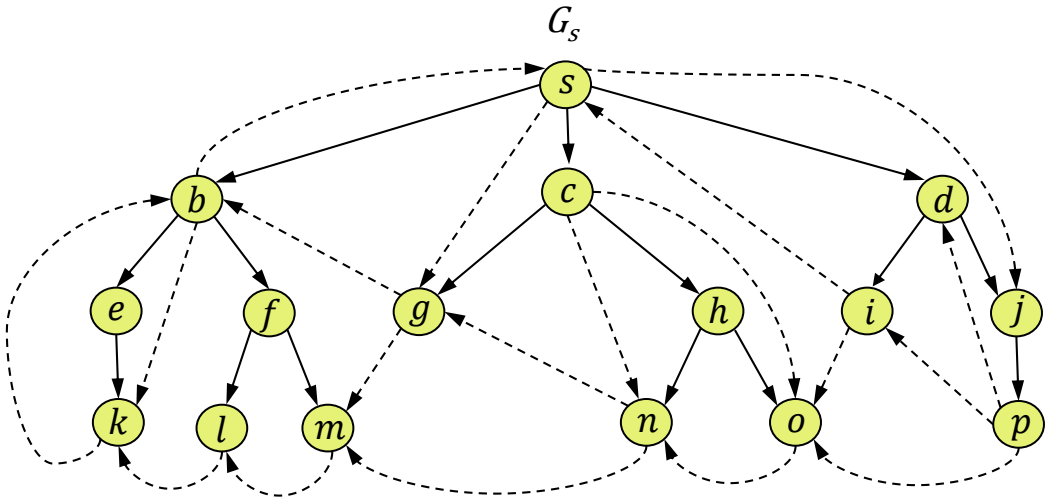


Figure 3.3: A flow graph G_s. The solid edges in G_s are the edges of depth first search trees with root vertex s.

Procedure CreateDFS(G)

```
1 foreach v ∈ V(G) do
2     visited(v) = false
3 DFS(s) // s is a source vertex
```

Procedure DFS(v)

```
1 visited(v) = true
2 for all edges (v,w) ∈ E(G) do
3     if visited(w) = false then
4         DFS(w)
```

In DFS, each edge is explored only once but it yields much more information than list of reachable vertices from the root. We can study the connectivity structure and edge properties through this process. That will help us to extend the DFS in order to find a dominator tree, find the strongly connected component in a graph, and to create a loop nesting forest. We will explain these on next section. For the moment, let us

observe the edge properties by adding few more parameter in DFS computation. We can assign the *preorder*[*] number for each vertex $v \in V(G)$ according to its appearance in the tree formation, such that the *descendants* of v will be ordered consecutively with v first. When the DFS visit a new vertex v, then v became a new input parameter for the search and start to explore the outgoing edge from it. Let us keep all the vertices on stack at beginning. During the DFS execution, if all the outgoing edges of a vertex has been explored then we unstacked it. The vertices are listed on a stack in order, which help us to determine the path from the source to the current processed vertex. If we keep track of the time of each vertex when it is unstacked (so called *sorder(v)*), then Tarjan [154] define the following categories for the edges, such that every *edge* $e = (v,w) \in E(G)$ in DFS tree will lies in one of them.

i. $e = (v,w)$ with w unvisited when (v,w) is explored, is called a ***tree edge***.

ii. $e = (v,w)$ with w stacked when (v,w) is explored, is called a ***backward edge***.

iii. $e = (v,w)$ with $pre(v) < pre(w)$ and w unstacked when (v,w) is explored, is called a ***forward edge***.

iv. $e = (v,w)$ with $pre(v) > pre(w)$ and w unstacked when (v,w) is explored, is called a ***cross edge***.

Procedures CreateDFSCategory and DFSCategory explain the extended DFS algorithm in detail. Figure- 3.4 illustrates the process taken by the Procedure DFSCategory for the graph shown in Figure 3.3. Lemmata 3.3.1 - 3.3.5 from [154, 158] explain the basic properties of the variables calculated by the DFSCategory. Tarjan [154] contains the proofs such that these calculation require the $O(|V|+|E|)$ time and space. We will use this extended version of the DFS in our analysis later.

Lemma 3.3.1 (Path Lemma [154]). *Let T be a DFS tree of a strongly connected digraph $G = (V,E)$ and let $pre(v)$ denote the preorder number of vertex v in T. If v and w are vertices such that $pre(v) < pre(w)$, then any path from v to w must contain a common ancestor of v and w in T.*

[*]For the details of preorder, inorder and postorder traversal technique please refer the Appendix A.3.3

Procedure CreateDFSCategory(G)

```
   /* nextp and nexts are the last preorder number and stack number
      respectively, that are assigned to the vertices.  The preorder(v) = 0 ⟺ v
      has not been visited yet and sorder(v) = 0 ⟺ v has not been unstacked
      yet.  Similarly, the descendants(v) counts the total descendants of v    */
1  nextp = 0 , nexts = |V(G)|+1
2  foreach v ∈ V(G) do
3      preorder(v) = 0
4      descendants(v) = 0
5      sorder(v) = 0
6  DFSCategory(s) // s is a source vertex
```

Procedure DFSCategory(v)

```
   /* v is the most recently visited vertex                                   */
1  descendants(v) = 1
2  nextp = preorder(v) = nextp+1
3  for all edges (v,w) ∈ E(G) do
4      if preorder(w) = 0 then
5          label(v,w) is a tree edge
6          DFSCategory(w)
7          descendants(v) = descendants(v) + descendants(w)
8      else if sorder(w) = 0 then
           /* w is stacked                                                    */
9          label(v,w) is a backward edge
10     else if preorder(v) < preorder(w) then
11         label(v,w) is a forward edge
12     else
13         label(v,w) is a cross edge
14     if w  is a last adjacent vertex of v in adjacency list then
15         nexts = sorder(v) = nexts − 1
```

Proof. Let us suppose T is a DFS tree of a strongly connected digraph G. Then there are three different cases, illustrated in Figure 3.5, (i) $v \xrightarrow{*} w$ in T (ii) $w \xrightarrow{*} v$ in T (iii) Neither $v \xrightarrow{*} w$ nor $w \xrightarrow{*} v$ in T.

For the case (i) and (ii), it is trivial that DFS process ancestor at first and then descendant later. It means that DFS assigns the *preorder* number to the ancesor at first

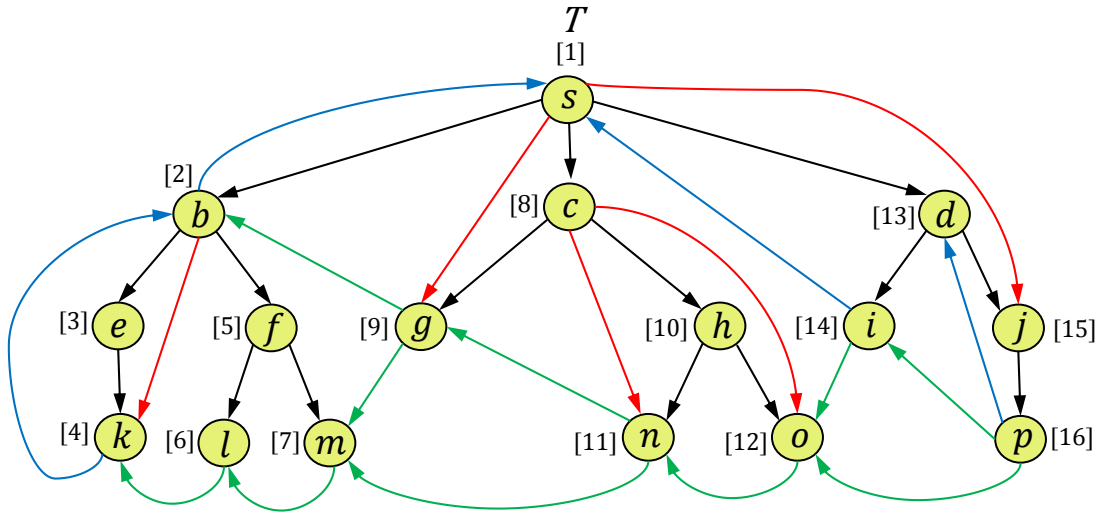


Figure 3.4: Extended DFS algorithm in a flow graph G_s with root s. Black edges are the tree edges, red are the forward edges, blue are the backward edges and green are the cross edges. (Better viewed in color.)

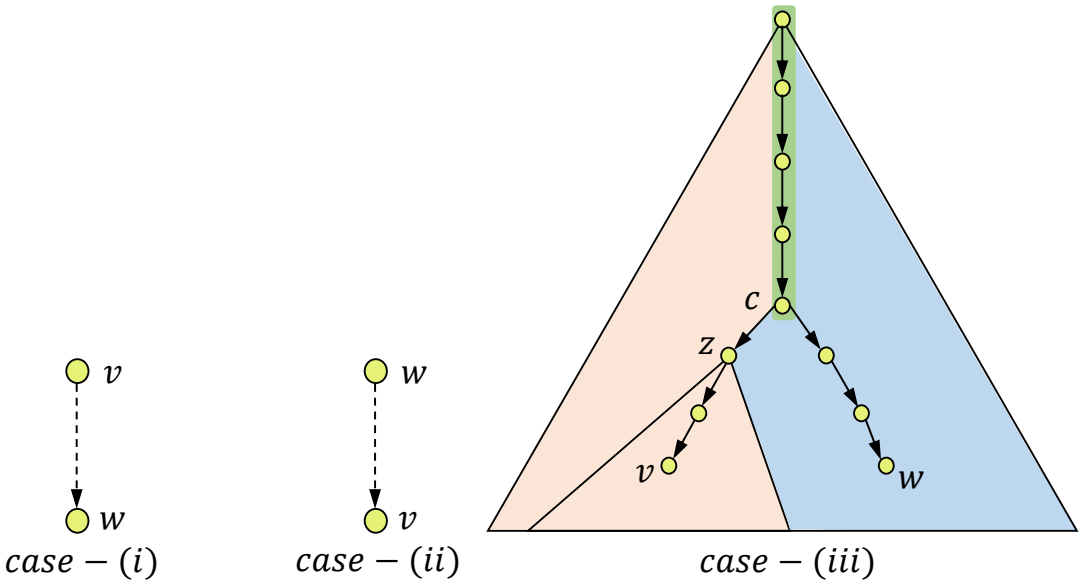


Figure 3.5: Illustration the cases of Path Lemma 3.3.1, for case - (iii), sets A (shown in orange) and B (shown in blue).

and then descendans later. Hence, for case (i), $(pre(v) < pre(w))$ and for case (ii), $(pre(w) < pre(v))$.

For case (*iii*), without loss of generality, let us consider that DFS process the vertex v before the vertex w (i.e., $pre(v) < pre(w)$).

We are going to prove it by contradiction.

Let P be a path that connects v to w in G. Path P exists because G is strongly connected. Moreover, let us suppose on the contrary that P does not pass through any common ancestor of v and w in T. Let c be the lowest common ancestor of v and w and a vertex z is lies in a path (c,w) (i.e., $c \xrightarrow{*} z$ and $z \xrightarrow{*} v$) in T. Let A be the set of vertices that are visited by DFS before z in T or lies in a subtree T_z (i.e., tree rooted at z), and B be the set of vertices visited after z in T. Thus, $v \in A$ and $w \in B$. Let $x \in A$ be the last vertex of P in A and y be the successor of x on P (i.e., the edge $(x,y) \in P$). Since P does not have a common ancestor of v and w, therefore, the edge (x,y) must belong to $A \times B$. Hence, the following relationship must hold true -$pre(x) \leq pre(z) < pre(y)$. But such a relationship is not possible because all the out-neighbors of x must be visited before DFS traversal finishes for vertex x. Hence we get a contradiction. □

Note. Path Lemma 3.3.1 gives the concept of the nearest common ancestor (or also called the lowest common ancestor) of two different vertices, which is used by the algorithms in our analysis later.

Lemma 3.3.2 (Tarjan [158])**.** *Suppose that all vertices of a directed graph G are reachable from the source s, and that the edges of G are divided into categories using Procedure CreateDFSCategory. Then:*

i. The tree edge form a directed tree T with root s which contains all vertices of G.

ii. If $e = (v,w)$ is a tree edge in T, then $\implies v \to w$ in T (or v is the parent *of w in T) $\implies pre(v) < pre(w)$.*

iii. If $e = (v,w)$ is a backward edge in T, then $w \xrightarrow{*} v$ in T (or v is the descendant *of w in T) $\implies pre(v) > pre(w)$*

iv. If $e = (v,w)$ is a forward edge in T, then $v \xrightarrow{*} w$ in T (or v is an ancestor *of w in T) $\implies pre(v) < pre(w)$*

v. If $e = (v,w)$ is a cross edge in T, then neither $v \xrightarrow{*} w$ in T nor $w \xrightarrow{*} v$ in T and $pre(v) > pre(w)$.

Lemma 3.3.3 (Tarjan [158]). *If $e = (v,w)$ is a tree edge, a forward edge or a cross edge, $sorder(v) < sorder(w)$. Similarly, if $e = (v,w)$ is a backward edge $sorder(v) > sorder(w)$.*

Lemma 3.3.4 (Tarjan [158]). *Let v be a vertex in G. Then the number of descendants of v in the spanning tree T is given by $descendants(v) = 1 + \sum_{v \to w} descendants(w)$*

Lemma 3.3.5 (Tarjan [158]). *Statements i, ii, iii, and iv below are equivalent.*

i. $v \xrightarrow{*} w$ in T.

ii. $pre(v) \leq pre(w)$ and $pre(w) < pre(v) + descendants(v)$

iii. $sorder(v) \leq sorder(w)$ and $sorder(w) < sorder(v) + descendants(v)$

iv. $pre(v) \leq pre(w)$ and $sorder(v) \leq sorder(w)$

We already have the algorithm that counts the total descendants of each vertex. However, if a vertex u is the ancestor of v and v is the ancestor of w, then w is the descendant of both vertexes u and v. So, we can collect the descendants of each vertex in $O(n)$ time from the DFS tree given by the procedure ComputeDescendants.

3.4 Strongly Connected Component

Let us recall the definition of strongly connected component (SCC) from Chapter 2 - section 2.2.2. A strongly connected component of a digraph $G = (V,E)$ is a maximal strongly connected subgraph of G such that all of its vertices are strongly connected to each other. In this section, we are going to illustrate some existing algorithms that are used to compute the strongly connected components of a given graph. Two different algorithms are available to find the SCC, one proposed by Tarjan [154] and one by Gabow [60]. Both of them require $O(m+n)$ time and space.

Procedure ComputeDescendants(G)

```
1  INTEGER index = 0
   /* k is the highest preorder number                                   */
2  Create three arrays start, end and descendants of size |V(G)| = n = k
3  for i ← k to 1 do
4      v = label(i)
5      index = index + 1
6      start(v) = end(v) = index
7      descendants(index) = v
8      foreach children c of v in T do
9          if start(c) < start(v) then
10             start(v) = start(c)
   /* we can get the common descendants of any vertex v as following     */
11 for i ← start(v) to end(v) do
12     print descendants(i)
```

3.4.1 Tarjan's Algorithm

In 1972, Tarjan [154] presented an algorithm to compute the SCC, which is incorporated in Procedure SCC-TARJAN and is based on the following lemmas (3.4.1 to 3.4.4), proofs are available in [154]. Moreover, Tarjan's algorithm to compute the SCCs of a given graph is also included in text books [5, 25, 26, 53, 87, 101, 114, 118, 144, 174].

Lemma 3.4.1. *Let $G = (V,E)$ be a directed graph. We may define an equivalence relation on the set of vertices as follows: two vertices v and w are equivalent if there is a closed path $P: v \overset{*}{\Rightarrow} v$, which contains w. Let V_i, $1 \leq i \leq n$ be the distinct equivalence classes under this relation. Let $G_i = (V_i, E_i)$, where $E_i = \{(v,w) \in E(G) | v,w \in V_i\}$. Then:*

i) Each G_i is strongly connected.

ii) No G_i is a proper subgraph of a strongly connected subgraph of G.

Lemma 3.4.2. *Let v and w be vertices in G which lies in the same strongly connected component. Let F be a spanning forest of G generated by repeated depth-first search. Then v and w have a common ancestor in F. Further, if u is the highest numbered*

common ancestor of v and w, then u lies in the same strongly connected component of v and w.

Lemma 3.4.3. *Let C be strongly connected component in G. Then the vertices of C define a subtree of a tree in F, the spanning forest of G. The root of this subtree is called the root of the strongly connected component of C.*

Proof. The problem of finding the SCCs of a graph G can be reduce to the problem of finding the roots of SCCs. We can construct a simple test to determine if a vertex is the root of a SCC as following.

$LOWLINK(v) = \min(\{v\} \cup \{w \mid \exists$ a cross edge $c \in E$ such that $v \xrightarrow{*} c \rightarrow w$ and $\exists u \in V$ ($u \xrightarrow{*} v$ & $u \xrightarrow{*} w$ & u and w are in the same strongly connected component of $G)\})$.

That is $LOWLINK(v)$ is the smallest vertex which is in the same component as v and is reachable by traversing zero or more *tree edges* followed by at most one backward or cross edge. □

Procedure SCC-TARJAN(G)

```
1 INTEGER i = 0
2 create the empty global stacks LOWLINK and NUMBER of size |V(G)| = n
  // Empty stack of points
3 foreach v ∈ V do
4     LOWLINK (v) = 0
5     NUMBER (v) = 0
6 foreach w ∈ V(G) do
7     if w is not yet numbered then
8         STRONGCONNECT-TARJAN(w)
```

Lemma 3.4.4. *Let G be a directed graph with LOWLINK defined as above relative to some spanning forest F of G generated by depth-first search. Then v is the root of some strongly connected component of G if and only if $LOWLINK(v) = v$.*

Procedure STRONGCONNECT-TARJAN(w)

```
1  LOWLINK(v) = NUMBER(v) = i = i + 1
2  put v on stack of points
3  foreach (v,w) ∈ E(G) do
4      if w is not yet numbered then
           // (v,w) is a tree edge
5          STRONGCONNECT-TARJAN(w)
6          LOWLINK(v) = min( LOWLINK(v), LOWLINK(w) )
7      else if NUMBER(w) < NUMBER(v) then
           // (v,w) is a backward or cross edge
8          if w is on stack of points then
9              LOWLINK(v) = min( LOWLINK(v), NUMBER(w) )
10 if LOWLINK(v) = NUMBER(v) then
       // v is the root of the component, start new strongly connected component
11     while w on top of point stack satisfies NUMBER(w) ≥ NUMBER (v) do
12         delete w from point stack and put w in current component.
```

3.4.2 Gabow's Algorithm

Process. Gabow [60] proposed an algorithm to compute the SCC based on the idea that SCC is the finest acyclic contraction (for the Contraction, please refer Chapter 2 section 2.1.4) of a graph $G = (V,E)$. We provide the pseudocode details of this algorithm in Procedure SCC-GABOW, which is adapted from [60]. The algorithm maintains a graph H that contains the contracted information of G with some deleted vertices. It also maintains a path P in H. Initially H is the given graph G. If H is an empty set (i.e., it does not have any vertices) then the algorithm stops. Otherwise, it starts a new path P by choosing a vertex v and setting a path $P = (v)$. After that it continues as $P = (v_1, \ldots, v_k)$ by choosing an edge (v_k,w), which is directed from the last vertex of P and do the following process.

- If $w \notin P$, add w to P, making it the new last vertex of P. Continue growing P.
- If $w \in P$, say $w = v_i$, contract the cycle v_i, v_{i+1}, $\ldots$, v_k, both in H and in P, P is now a path in a new graph H. Continue growing P
- If no edge leaves v_k, output v_k as a vertex of the strong component graph. Delete

Procedure SCC-GABOW(G)

```
1 create the empty global stacks S,B,I of size |V(G)|= n
2 INTEGER c = n
3 foreach v ∈ V do
4 |   I[v] = 0
5 foreach v ∈ V do
6 |   if I[v] = 0 then
7 |   |   STRONGCONNECT-GABOW(v)
```

v_k from both H and P. If P is now non-empty then growing P continuously. Otherwise try to start a new path P.

In the path P, we can notice that, if the v_k does not have any leaving edges, then v_k is a vertex of the finest acyclic contraction. Thus, the algorithm forms a finest acyclic contraction of G and computes the SCC.

Explanation Let assume that the graph has n vertices and algorithm numbers the SCC starting from $n+1$. Two stacks S and B are used to represent the path P. Stack S contains the sequence of (original) vertices in P and stack B contains the boundaries between contracted vertices. More precisely, S and B correspond to $P = (v_1, \ldots, v_k)$ where $k = TOP(B)$, for $i = 1, \ldots, k$,

$$v_i = \{S[j] : B[i] \leq j < B[i+1]\} \tag{3.1}$$

When $k > 0$ we have $B[1] = 1$. Also when $i = k$ in 3.1, it interpret $B[k+1]$ to be $TOP(S)+1$. An array $I[1 \ldots n]$ is used to store stack indices. It also stores the corresponding SCC number of a vertex when that number is known. More precisely, for a given vertex v, the corresponding number of SCCs is calculated as following.

$$I[v] = \begin{cases} 0, & \text{If } v \text{ has never been in } P. \\ j, & \text{If } v \text{ is currently in } P \text{ and } S[j] = v. \\ c, & \text{If the strong component containing } v \text{ has been deleted and numbered as } c\ . \end{cases} \tag{3.2}$$

Since there are only n vertices, hence no confusion between an index j and a com-

Procedure STRONGCONNECT-GABOW(v)

```
1  PUSH(v,S) // add v to the end of DFS path P.
2  I[v] = TOP(S)
3  PUSH(I[v],B)
4  for edges(v,w) ∈ E do
5      if I[w] = 0 then
6          STRONGCONNECT-GABOW(w)
7      else
           /* Following loop do the contractions and handle the deleted
              vertices.                                                  */
8          while B[TOP(B)] > I[w] do
9              POP(B)
10 if B[TOP(B)] = I[v] then
11     POP(B)
12     c = c + 1
13     while (TOP(S) ≥ I[v]) do
14         I[ POP(S) ] = c
```

ponent number c in 3.2. A variable c is used to keep the track of numbers of SCCs formed. Gabow [60] claims that, because of using the variable I in the procedure STRONGCONNECT-GABOW for multiple proposes, it pays off the speed by 20% [61, p.19].

Remark. Tarjan's LOWPOINT method [154] for strong components is presented in texts [5, 52, 101, 114, 118, 144]. Also, Gabow presented a linear-time implementations of SCC [60] that uses only stacks and arrays as data structures. A line-by-line comparison in the pseudocodes of Gabow's algorithm and Tarjan's algorithm showed that both approaches are similar in terms of lower level resource usage. Moreover, performance differences are likely to be small or platform-dependent [60]. In our analysis, we used both of them to compute the SCC of a given digraph.

3.5 Dominators

3.5.1 Introduction

The concept of dominator was evolved from the flow graph. If G_s is the flow graph of G, then the dominator relation between the *vertices* (resp., *edges*) in G_s is defined as following. A vertex u (resp., edge e_1) is a dominator of vertex v (resp., edge e_2), if every path from the source s to v (resp., edge e_2) includes u (resp., edge e_1). The dominator relation of the vertices (resp., edges) of G_s forms a tree called the vertex dominator tree (resp., edge dominator tree) rooted at s. We mainly focus on the vertex dominator tree (since both of them have the almost identical properties) that is denoted by D_s or simply D. The dominator relation (also known as dominance relation) between the vertices in D is transitive closure ($a \rightarrow b$, $b \rightarrow c \implies a \rightarrow c$) and hence one single vertex v could have many dominators which is denoted by dom(v). Thus, the dominator relation can be represented in compact form as a tree as shown in Figure 3.6, which also has the other characteristics given below.

i. For each vertex $v(\neq s) \in V$, s and v are the trivial dominators.

ii. u dominates $v \iff u$ is an ancestor of v in D.

iii. If v has non-trivial dominators, then v has a unique immediate dominator called parent of v in D.

iv. If u is the dominator of v, then all the non-trivial dominators of u also dominates v.

3.5.2 Applications

The dominators problem has occurred in many application areas and hence they used the dominator relation. For example, electronics circuit testing, control flow, biological computation, compiler code generation, program optimization, social network analysis, and etc. In the electronic circuit testing, VLSI test uses the dominator relation for

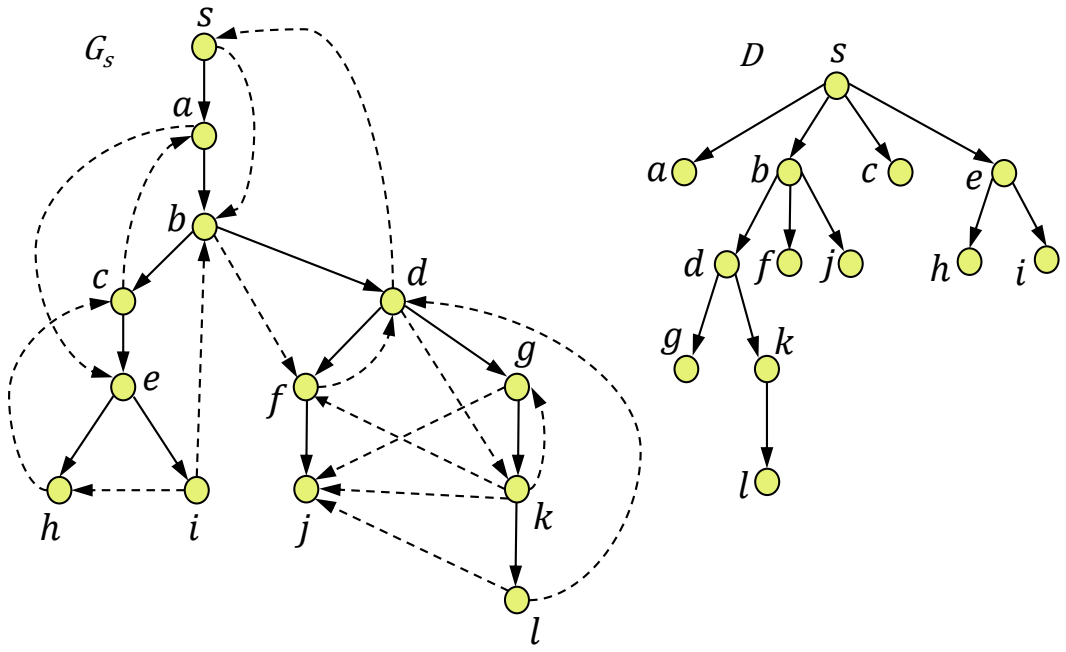


Figure 3.6: A flow graph G_s and its corresponding dominator trees D. The solid edges in G_s are the edges of depth first search trees with root vertex s. The corresponding digraph G is strongly connected.

identifying the pairs of equivalent line faults in logic circuits [15]. Similarly, the post-dominator information [56] is used to calculate the control dependencies in program dependence graph. The dominator analysis is also being applied in theoretical biology [10, 11] to analyze the extinction of species in trophic models (or foodwebs). The program compiler uses the dominator information extensively in global flow analysis, program analysis and optimization, code generation [7]. The Best-known application of dominators could be a loop optimization, which enables a host of natural loop optimization [123]. Apart from the code generation, the dominator relation has been used in structural analysis [145], scheduling algorithm [152] and memory profiling [116]. In addition, the dominator trees are also used to implement the generalized reachability constraints in the field of constraint programming, which are helpful in the solution of ordered disjoint-paths problem [138]. Moreover, they are also needed in the computation of dominance frontiers [42], which are needed for efficiently computing program dependence graphs and static single-assignment forms. Furthermore, Graph Connectiv-

ity, Path Determination Problems [65, 66, 88], and an analysis of Diffusion Networks [81] are also used the dominator relation.

3.5.3 Algorithms

Development. In the previous section, we discussed the concept of dominator, dominator tree and its application. Now in this section, we are going to discuss the methodology to find the dominators. During the last 40 years, many researchers proposed different kinds of algorithms to solve this problem. In 1972, Allen and Cocke [8] gave a solution that, given set of data-flow equations, computes iteratively the dominator relation in $O(mn^2)$ worst-case time bound. In the same time bound, Cooper, Harvey, and Kennedy [40] presented a clever tree-based space-efficient implementation of the iterative algorithm in 2006. Their algorithm is much more efficient in practice, however does not improve the $O(mn^2)$ worst-case time bound. Still in 1972, but after Allen and Cocke [8], Purdom and Moore [136] found an another straight-forward algorithm which has $O(mn)$ time complexity. The algorithm selects a root and performs the search in $G \setminus \{v\}$ for each $v \in V(G)$ in a way that collects the vertices w that are not reachable from the root. Two years after in 1974, Tarjan [158] proposed a solution with $O(n \log n + m)$ time complexity. The algorithm uses the depth-first-search and efficient algorithms for computing disjoint set unions and manipulating priority queues.

In 1979, the algorithm proposed by Tarjan [158] was improved by Lengauer and Tarjan [106], they proposed two different solutions with $O(m \log_{(m/n+1)} n)$ and $O(m\alpha(m,n))$ time complexity, where $\alpha(m,n)$ is an extremely slow-growing functional inverse of the Ackermann function [160] [†]. The algorithm performs very well in practice and have been used in many applications.

Subsequently, on the basis of the Lengauer-Tarjan algorithm, the truly linear-time algorithms were discovered. They achieved linear time by incorporating several other techniques, including the pre computation of answers to small sub problems. Buchsbaum et al. [28] gave a simpler algorithm in 1998, its corrigendum is appeared later in

[†]Please refer the appendix section A.2.2 for the details of Ackermann function

2005. One year after in 1999, Alstrup et al. [14] presented a linear-time algorithm for the random-access model of computation. After that in 2004, Georgiadis and Tarjan [67] proposed a linear-time algorithm for pointer machine computation model. Again in 2008, Buchsbaum et al. [29] proposed a new algorithm, where they replace random-access table look-up by a radix sort, and partitioning a tree. The genesis of their algorithm was the discovery of a subtle error in the analysis of a previous allegedly linear-time algorithm for finding dominators and provides the systematic study.

In the recent years, Gabow [62] and Fraczak et al. [57] presented linear-time algorithms, which are based on a different approach, and require only the simple data structures and a data structure for static tree set union [64]. Gabow's algorithm uses the concept of minimalset posets defined in [59, 63], while the algorithm of Fraczak et al. [57] uses vertex contractions. The linear-time algorithms, proposed in [14, 28, 57, 62] use bit-manipulation techniques, so they run on the random-access-machine[‡](RAM) model of computation. Nevertheless, the algorithms presented in [30, 67] are implementable on less powerful pointer-machine model[§][163].

In addition, Ramalingam and Reps [141] proposed an incremental algorithm for finding dominators in an acyclic graph. Their algorithm uses a data structure that computes nearest common ancestors in a tree that grows by leaf additions. (More on [13], [140].) To overcome the incremental nearest common ancestors problem, Gabow [58] and Alstrup and Thorup [12] give the $O(m)$- time RAM algorithm and $O(m\log(\log n))$-time pointer machine algorithm respectively. Ramalingam [140] showed a way to reduce the problem of computing dominators in an arbitrary flow graph to computing dominators in an acyclic graph. The reduction process uses static-tree disjoint set union, that leads to run it in in $O(m\alpha(m,n))$ time on a pointer machine [160], and in $O(m)$ time on a RAM [59]. Therefore, the combination of any linear-time algorithm that computes the dominators in an acyclic graph with Ramalingam's reduction produce a linear-time RAM algorithm to compute the dominators of a graph.

[‡]For the details of Random Access Machine Model, please see Appendix Section A.3.1

[§]The Appendix Section A.3.2 contains the note for Pointer Access Machine Model

Selection: As we explained in development section, many algorithms are available to compute the dominators. It is very difficult to say which one is better among them because the efficiency not only depends on the running time and storage space but also on the graph structure. Theoritical and experimental comparision of these algorithms has been carried out [28, 40, 70, 106].

Lengauer and Tarjan found the $O(m\alpha(m,n))$-time version of their algorithm (LT) to be faster than the simple $O(m\log n)$ version (SLT) even for small graphs [106]. They also showed that the Purdom-Moore [136] algorithm gives better result only if the graphs has less than 20 vertices. Moreover, in their experimental observation results they also show that, a bit-vector implementation of the iterative algorithm, by Aho and Ullman [4], is 2.5 times slower than the LT for graphs that has more than 100 vertices.

Buchsbaum et al. [28] reported that their claimed linear-time algorithm has low constants, being only about 10% to 20% slower than their implementation of LT for graphs with more than 300 vertices. Nevertheless, this algorithm was later shown to have the same time complexity as LT [67]. The corrected linear-time version of [28] is more complicated (refer to the Corrigendum). Cooper et al. [40] presented a clever tree-based space- and time-efficient implementation of the iterative algorithm, which they claimed to be 2.5 times faster than SLT. However, a careful implementation of SLT later led to different results [39].

Georgiadis et al. [70] presented an experimental study that compares the algorithm proposed by Cooper et al. [40] (and some variants) with careful implementations of both versions of the Lengauer-Tarjan algorithm and with a new hybrid algorithm. The results suggest that, although the performance of all the algorithms is similar, the most consistent and fast algorithms are the Simple Lengauer -Tarjan Algorithm and the hybrid algorithm, and their advantage is directly proportional to the graph's size and structure. The idea behind the Lengauer-Tarjan algorithm is to find the minima of a function defined on the paths of a depth-first search spanning tree of the graph [162]. This can be achieved efficiently by using the data structure, which support the link and eval operation. The link eval technique resemble the unite and find operation of a disjoining set

union data structure [160] but involve a more elegant use of path compression and tree balancing. The algorithm runs in $O(m \log_{(m/n+1)} n)$ and $O(m\alpha(m,n))$ time bound with a simple linking and complicated balanced linking strategy respectively. Therefore, after reviewing the existing algorithms, their complexity and available experimental observation reports, we decided to use the SLT for our analysis. In the following sections, we will give a high level overview and the pseudocode of the Iterative, LT and SLT algorithms.

Common Notations. Let allow us to introduce some notions before we explain the algorithms. The *immediate dominator* of a vertex $v \neq r$ (root vertex), denoted by $idom(v)$, is the unique vertex $u \neq v$ that dominates v and is dominated by all vertices in $\{Dom(v)\text{-}v\}$ (root vertex has no immediate dominator). The (immediate) dominator tree is a directed tree I rooted at r and formed by the edges $\{(idom(v),v)|v \in \{V-r\}\}$. A vertex u dominates $v \iff u \xrightarrow{*} v$ in I, so computing the immediate dominators is enough to determine all dominance information. Similarly, for any directed graph $G = (V,E)$, we say that v is a *successor* of u (and that u is a *predecessor* of v). The set of all successors of v is denoted by $succ(v)$ and the set of all predecessors of v is denoted by $pred(v)$. Also, for any subset $S \subseteq V$ and a tree T, NCA(T,U) denotes the nearest common ancestor (in some paper, it also known as a lowest common ancestor LCA) of $U \cap T$ in T.

3.5.3.1 Iterative Algorithms

There exist two different types of iterative algorithm. We are going to explain their structures as following.

i. Data-flow Equation, Boolean Vector In 1970, Allen [9] defined the computation of dominance relations by a data-flow equation 3.3. The equation provides a procedure that finds the dominators in an interval. An interval is the maximal single entry subgraph, where all cycles in the subgraph contain the entry vertex. An entry vertex v is a

vertex in the program control flow graph C, if it contains a program entry point.

$$\forall v \in V(G) : Dom'(v) = \left(\bigcap_{u \in pred(v)} Dom'(u) \right) \cup \{v\} \qquad (3.3)$$

Allen and Cocke [8] in 1972 came up with the idea that one can solve this equation iteratively. Hence, to find the dominators in a directed graph we start by initializing $Dom'(r) = \{r\}$ and $Dom'(v) = V(G)$ for $v \neq r$, and then apply the following steps repeatedly: Find a vertex v such that the equation 3.3 is false and replace $Dom'(v)$ by the expression on the right side of the equation 3.3.

Algorithm 1: Boolean Vector

Input: A strongly connected digraph $G = (V, E)$, root vertex r
Output: Dominator set for each vertex

1 $Dom'(r) \leftarrow \{r\}$
2 **while** *changes occured in any* $Dom'(v)$ **do**
3 **for** $v \in \{V - \{r\}\}$ **do**
4 $Dom'(v) \leftarrow \left(\bigcap_{u \in pred(v)} Dom'(u) \right) \cup \{v\}$

A näive way to perform this iteration is to cycle repeatedly through all the vertices of V until no $Dom'(v)$ changes. It is not necessary to initialize all the sets $Dom'(v)$; rather suffices to initialize $Dom'(r) = \{r\}$ and exclude uninitialized sets from the intersection in 3.3. If it is uninitialized, then an iterative step will be applied to a vertex v only if a value has been computed for at least one $u \in pred(v)$. It is also possible to initialize the sets $Dom'(v)$ more accurately. In addition, the $Dom'(v)$ can be represented as a boolean vector of size $|V(G)|$. The entry value of w in the boolean vector for vertex v is true if vertex w dominates v. Otherwise w does not dominate v and entry value is false (see in Algorithm 1).

ii. Tree Base Iterative Algorithm Cooper [39] significantly improved the efficiency of this algorithm complexity for memory space and running time in practice. They observed that dominator set $Dom'(v)$ for a given vertex v can be represented as a list of

immediate dominators $idom(v)$ such that:

$$Dom'(v) = \{\{v\} \cup idom(v) \cup idom(idom(v)) \cup idom(idom\ldots(v))\}$$

Algorithm 2: Tree Base Iterative Algorithm

Input: A strongly connected digraph $G = (V, E)$, root vertex r
Output: Dominator tree

```
1 Create any tree T as subgraph of G rooted at r
2 while changes occured in T do
3     for v ∈ V do
4         for v ∈ V do
5             if u ≠ parent_T(v) and parent_T(v) ≠ NCA(u, parent_T(v)) then
6                 parent_T(v) ← NCA(u, parent_T(v))
```

Since there is an edge from the $idom(v)$ to v, we can represent this data structure in the dominator tree. Cooper [39] also provide the pseudocode implementation (we refer the reader to [70] for a better explanation, nevertheless, the algorithm is incorporated in Algorithm 2). The idea behind this technique is: we can represent all sets of $Dom'(v)$ by a single tree and perform an iterative step as an update of the tree. We can start the process with any tree T and root vertex r, that is a subgraph of G and repeat the step given below till it will no longer applies:

Find a vertex v such that

$$pred(v) \cap T \neq \emptyset \text{ and } parent_T(v) \neq \text{NCA}(T, pred(v)),$$

replace $parent_T(v)$ by $\text{NCA}(T, pred(v))$.

The relation between this algorithm and the original algorithm is as follwoing. For each vertex $v \in T(v)$, $Dom'(v)$ is the set of ancestors of v in T. The common vertices between $Dom'(u)$ and $Dom'(v)$ is the set of ancestors of $\text{NCA}(T, \{u, v\})$ in T. Therefore, once the iteration stops, the current tree T will became the dominator tree I. Georgiadis et al. [70] explained that one can also perform the iteration not only through vertex-by-vertex but also through arc-by-arc.

3.5.3.2 Lengauer-Tarjan Algorithm

The algorithm introduced a new term in dominance relation called *semidominators*. Afterward, the main computation of the algorithm followed the *link-eval* data structure, introduced by Tarjan [162].

Semidominator A *semidominator* is an ancestor of a vertex v in a DFS tree D of graph $G = (V, E)$ that gives an initial approximation to *immediate dominator* (or *idom*) of v. A semidominator (or *sdom*) of a vertex v (denoted by $semi(v)$) is the smallest starting vertex in a semidominator path. Any path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ in G become a *semidominator* path if $v_i > v$ for $1 \leq i \leq k$–1. The semidominator of v is defined as

$$semi(v) = \min \{u \mid \text{there is a semidominator path from } u \text{ to } v\}$$

There is a relation between any vertex $v \neq r$ to $idom(v)$ and $semi(v)$ in the following way.

$$idom(v) \xrightarrow{*} semi(v) \xrightarrow{+} v.$$

Lengauer and Tarjan [106] showed that, for any vertex $w \neq r$, we can compute the semidominators and immediate dominators by finding minimum *semi* values on paths of D,

$$semi(w) = \min \{S_w(v) \mid v \in pred(w)\} \tag{3.4}$$

where the function $S_w(v)$: $pred(w) \mapsto V$ is defined as

$$S_w(v) = \begin{cases} v, & v \leq w \\ \min \left\{ semi(u) \mid \text{NCA}\,(D, \{v, w\}) \xrightarrow{+} u \xrightarrow{*} v \right\}, & v > w \end{cases}$$

The immediate dominator can be found similarly, by evaluating the function: $R_d(v)$: V–$\{r\} \mapsto V$, defined by

$$R_d(v) = \arg\min \left\{ semi(v) \mid semi(v) \xrightarrow{+} u \xrightarrow{*} v \right\}$$

$$idom(v) = \begin{cases} semi(v), & \text{if } semi(v) = semi(R_d(v)) \\ idom(R_d(v)), & \text{Otherwise} \end{cases} \tag{3.5}$$

link-eval data structure The *link eval data structure* is defined by Tarjan [162] as following. Let us suppose a tree T has been constructed from the graph $G = (V, E)$. Then for each $v \in V$, the link-eval data structure maintains a forest F, which is a subgraph of T and subjected to the operations given below.

> $link(v, w)$: Add edge (v, w) to F. This makes v the parent of w in F.
>
> $eval(v)$: If $v = root_F(v)$, return v. Otherwise, return any vertex of minimum value among the vertices u that satisfy the condition $root_F(v) \xrightarrow{+}_F u \xrightarrow{*}_F v$, where the notation $u \xrightarrow{*}_F v$ (resp., $u \xrightarrow{+}_F v$) denotes the u is an ancestor (resp., proper ancestor) of v in forest F.

Procedure EVAL-SIMPLE(v)

```
1 if ancestor(v) = 0 then
2     return v
3 else
4     COMPRESS(v)
5     return label(v)
```

Tarjan [162] provided the detail description to implement the simple and sophisticated way of LINK and EVAL data structure. To calculate the EVAL, simple method uses the *path compression* and two arrays, *ancestor* and *label* for the representation of forest F which is build by LINK operation. Initially, for each $v \in V$, $ancestor(v) = 0$ and $label(v) = v$. In general, if v is a root of any tree in F then $ancestor(v) = 0$. The algorithm maintains the $label(v)$ so that they satisfy the given property. Let v be any vertex, r be the root of the tree in F that contain v, and let $r = v_0, v_1, \ldots v_{k-1}, v_k = v$ be such that ancestor(v_i) = v_{i-1} for $1 \leq i \leq k$. Let u be a vertex such that $semi(u)$ is minimum among the vertices $u \in \{label(v_i) \mid 1 \leq i \leq k\}$, then

u is a vertex such that $semi(u)$ is minimum among vertices u satisfying $r \xrightarrow{+} u \xrightarrow{*} v$. (3.6)

The algorithm assigns $ancestor(w) = v$ to perform the LINK(v, w) operation. To calculate the EVAL(v), it follows ancestor pointers to determine the sequence $r = v_0, v_1, \ldots$

Procedure EVAL-ADVANCED(v)

```
1 if ancestor(v) = 0 then
2     return label(v)
3 else
4     COMPRESS(v)
5     if semi(label(ancestor(v))) ≥ semi(label(v)) then
6         return label(v)
7     else
8         return label(ancestor(v))
```

$v_{k-1}, v_k = v$ such that $ancestor(v_i) = v_i - 1$ for $1 \leq i \leq k$. If $v = r$ then v is returned. Otherwise, it will start the path compression process by assigning $ancestor(v_i) = r$ for $2 \leq i \leq k$ and simultaneously update the labels to maintain the equation 3.6 as given in procedure EVAL-SIMPLE which is taken from [162]. With this simpler implementation, time require for $(n-1)$ LINKs and $(m+n-1)$ EVALs is $O(m \log n)$ (for more details, please refer on [162]).

The advance version uses path compression to calculate EVAL(v) but implements the LINK instruction. The path compression is carried out only on balanced trees as shown in procedure EVAL-ADVANCED, adapted from [162]. This technique requires two additional arrays, *size* and *child*. Initially, $\forall v \in V$, $size(v) = 1$ and $child(v) = 0$. With this version of implementation, the time required for $(n-1)$ LINKs and $(m+n-1)$ EVALs is $O(m\alpha(m,n))$, where α is the functional inverse of Ackerman's function.

Procedure COMPRESS(v)

```
1 if ancestor(v) = 0 then
2     return
3 COMPRESS(ancestor(v))
4 if semi(label(ancestor(v))) < semi(label(v)) then
5     label(v) = label(ancestor(v))
6 ancestor(v) = ancestor(ancestor(v))
```

Process In general, Lengauer-Tarjan Algorithm follows the steps given below.

Procedure LINK(v,w)

```
   /* this procedure assumes for convenience size(0) = label(0) = semi(0) = 0   */
1  s = w
2  while semi(label(w)) < semi(label(child(s))) do
3      if (size(s) + size(child(child(s)))) ≥ (2 * size(size(child(s)))) then
4          parent(child(s)) = s
5          child(s) = child(child(s))
6      else
7          size(child(s)) = size(s)
8          s = parent(s) = child(s)
9  label(s) = label(w)
10 size(v) = size(v) + size(w)
11 if size(v) < 2 * size(w) then
12     s = child(v)
13 while s ≠ 0 do
14     parent(s) = v
15     s = child(s)
```

1. Perform the DFS on graph G and assign the preorder number to each vertex.
2. Compute sdom of all vertices by applying the equation 3.4 in reverse preorder.
3. Implicitly define the idom of each vertex by using the equation 3.5.
4. Explicitly define the idom of each vertex in forward preorder number.

Now, we are going to explain the steps in details, expanded by Algorithm 3, which is taken from [106]. The algorithm starts from the depth-first search on G from the root vertex r. We already explain in section 3.3, during the DFS, each vertex get its corresponding preorder number. The parent of each vertex in a DFS tree D can be represent by an array *parent*. If the vertex v has a lower preorder number than vertex u, then we denote their relation by $v < u$. Initially, it has $T = D$, and for each $v \in V$, $value(v) = v$. After processed the first time, $value(v) = semi(v)$. Each vertex w has to be processed three times. The first time, it executed the $eval(u)$ for each $u \in pred(w)$ to compute the $semi(w)$ and then $S_w(u)$. In the second process, $link(w, semi(w))$ is performed by inserting the w into a bucket associated with vertex $semi(w)$. In the third round, it again processes the w after $semi(v)$ has been computed, where v satisfies the condition

$parent[v] = semi(w)$ and $v \xrightarrow{*} w$; at this time it performs the operation $eval(w)$, thus computing $R_d(w)$. Then, immediate dominators are derived from relative dominators in a preorder pass.

Implementation issues. Georgiadis et al. [70] remarked some implementation issues for the Lengauer and Tarjan Algorithm. During the implementation of our analysis, we also consider their remarks as following.

The algorithm process the $bucket(parent(w))$ at the end of the iteration that deals with w; hence the same bucket may be processed several times. A better alternative is to process the $bucket(w)$ at the beginning of the iteration that deals with w; each bucket is now processed exactly once, so it need not be emptied explicitly.

They observe that buckets have very specific properties: (1) every vertex is inserted into at most one bucket; (2) there is exactly one bucket associated with each vertex; (3) vertex i can only be inserted into some bucket after bucket i itself is processed. Properties (1) and (2) ensure that buckets can be implemented with two $n-$sized arrays, $first$ and $next$: $first(i)$ represents the first element in bucket i, and $next(v)$ is the element that succeeds v in the bucket it belongs to. Property (3) ensures that these two arrays can actually be combined into a single array $bucket$. They also remarked an another measure that is relevant in practice to avoid the unnecessary bucket insertions: a vertex w for which $parent(w) = semi(w)$ is not inserted into any bucket because we already know that $idom(w) = parent(w)$. Also, we note that the last bucket to be processed is the one associated with the root r. For all vertices v in this bucket, we just need to set $idom(v) \leftarrow r$; there is no need to call EVAL for them.

3.5.4 Relation to Strong Articulation Points and Strong Bridges

Several relations can be defined between the strong articulation points (resp., strong bridges) and vertex (resp., edge) dominator tree. In particular, Italiano et al. [88] explored this type of connections through the following lemmata. Furthermore, they provide the algorithms to compute the strong articulation points and the strong bridges in

Algorithm 3: Lengauer-Tarjan Algorithm

Input: A strongly connected digraph $G = (V, E)$, root vertex r
Output: An array which contains the parent of a vertex in dominator tree

```
   /* step 1, create the DFS tree and initialize the basic variables          */
 1 n = 0
 2 foreach v ∈ V do
 3 |   pred[v] = ∅
 4 |   semi(v) = 0
 5 DFS-LT(r)
   /* process the vertices in reverse preorder                                */
 6 for i ← n to 2                          /* n is the highest preorder number */
 7  do
 8 |   w = vertex(i)
   |   /* step 2, calculate the semidominator of w by equation 3.4           */
 9 |   foreach v ∈ pred[w] do
10 |   |   u =EVAL-SIMPLE(v)        /* use EVAL-ADVANCE(v) for sophisticated
   |   |    version */
11 |   |   if semi(u) < semi(w) then
12 |   |   |   semi(w) = semi(u)
13 |   add w to bucket(vertex(semi(w)))
   |   /* LINK(parent(w),w) is only for sophisticated version                 */
14 |   LINK(parent(w),w)
   |   /* step 3, implicitly find the immediate dominator of w by equation 3.5
   |      */
15 |   foreach v ∈ bucket(parent(w)) do
16 |   |   delete v from bucket(parent(w))
17 |   |   u = EVAL-SIMPLE(v)        /* use EVAL-ADVANCE(v) for sophisticated
   |   |    version */
18 |   |   if semi(u) < semi(v) then
19 |   |   |   idom(v) = u
20 |   |   else
21 |   |   |   idom(v) = parent(w)
   /* step 4, calculate the immediate dominator explicitly                    */
22 for i ← 2 to n do
23 |   w = vertex(i)
24 |   if idom(w) ≠ vertex(semi(w)) then
25 |   |   idom(w) = idom(idom(w))
26 idom(r) = 0
```

Procedure DFS-LT(v)

```
1 semi(v) = n = n + 1
2 vertex(n) = v
3 foreach (v, w) ∈ E(G) do
4     if semi(w) = 0 then
5         parent(w) = v
6         DFS(w)
7     add v to pred[w]
```

linear time (Algorithm 4 and Algorithm 5 respectively).

Lemma 3.5.1. *(Italiano et al. [88]) Let $G(V,E)$ is a strongly connected graph, and let G_s be a flow-graph with start vertex s. If u is a non-trivial dominator of a vertex v in G_s, then u is a SAP in G .*

Proof. Let $G(V,E)$ a strongly connected graph, G_s be a flow-graph of G with start vertex s and u is a non-trivial dominator of v in G_s $\implies$ all the paths from s to v in G must include u $\implies$ $G \setminus \{u\}$ is not strongly connected $\implies$ u must be a SAP in G. □

Lemma 3.5.2. *(Italiano et al. [88]) Let $G(V,E)$ strongly connected graph. If a vertex u is a SAP in a G, then there should be the vertices $s, v \in V$ such that u is the non-trivial dominator of v in the flow graph G_s with start vertex s.*

Proof. Let $G(V,E)$ strongly connected graph and u is a SAP in G, then by **Lemma 3.5.1**, there must be two distinct vertices $s \neq u$, $v \neq u$, such that every path from s to v contains u. Hence u must be a non-trivial dominator of vertex v in the flow graph G_s. □

Lemma 3.5.3. *(Italiano et al. [88]) Let $G(V,E)$ be a strongly connected graph, and let s be any vertex in G. Let G_s be the flow-graph with start vertex s. If (u,v) is an edge dominator in G_s, then (u,v) is a strong bridge in G.*

Proof. It is completely analogous to **Lemma 3.5.1** □

Algorithm 4: Find All Strong Articulation Points

Input: A strongly connected graph $G(V,E)$, with n vertices and m edges.
Output: Strong Articulation Points of G.

1 Choose arbitrarily a vertex $s \in V$ in $G(V,E)$, and test whether s is a strong articulation point in G or not. If s is an articulation point, output s.
2 Compute D_s, the set of non-trivial vertex dominators in the flow graph rooted at s, G_s.
3 Compute the reversal graph $G^R = (V,E^R)$.
4 Compute D_s^R, the set of non-trivial vertex dominators in the reverse flow graph rooted at s, G_s^R.
5 Output $D_s \cup D_s^R$.

Algorithm 5: Find All Strong Bridges

Input: A strongly connected graph $G(V,E)$, with n vertices and m edges.
Output: Strong bridges of G.

1 Choose arbitrarily a vertex $s \in V$ in $G(V,E)$.
2 Compute D_s, the set of edge dominators in the flow graph rooted at s, G_s.
3 Compute the reversal graph $G^R = (V,E^R)$.
4 Compute D_s^R, the set of edge dominators in the reverse flow graph rooted at s, G_s^R.
5 Output the union of edges in D_s and reversal of the edges in D_s^R, $D_s \cup D_s^R$.

Lemma 3.5.4. *(Italiano et al. [88]) Let $G(V,E)$ be a strongly connected graph. If (u,v) is a strong bridge in G, then there must be a vertex $s \in V$ such that (u,v) is an edge dominator in the flow-graph G_s.*

Proof. It is similar with **Lemma 3.5.2** □

3.6 The Loop Nesting Forest

3.6.1 Introduction

The ***Loop Nesting Forest*** of a flow graph G_s is a hierarchical representation of strongly connected subgraphs of G. Several different ways are available to define such a forest [81, 106, 140, 148, 149, 156]. In our analysis we follow the process which was first presented by Tarjan [156] and later rediscovered by Havlak [81]. It is defined with respect to a depth-first spanning tree T of G rooted at s as follows. For any vertex

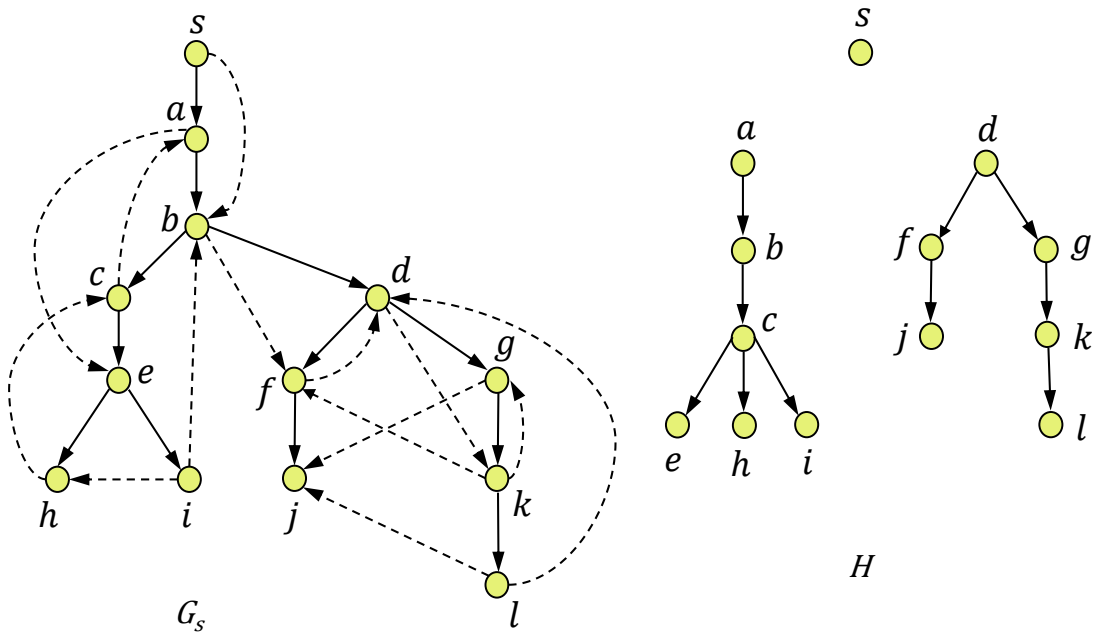


Figure 3.7: A flow graph G_s with a depth-first spanning tree T shown with solid arcs; non-tree arcs are shown dashed; vertices are numbered in reverse postorder (in brackets). The corresponding digraph G is not strongly connected. Loop nesting forest H of G with respect to T.

v, the loop of v, denoted by $loop(v)$, is the set of all descendants x of v in T such that there is a path from x to v containing only descendants of v in T. Vertex v is the head of $loop(v)$. Any two vertices in $loop(v)$ are mutually reachable. Therefore $loop(v)$ induces a strongly connected subgraph of G; which is the unique maximal set of descendants of v in T. For any two vertices u and v, $loop(u)$ and $loop(v)$ are either disjoint or nested (i.e., one contains another). We can extend this property to define a loop nesting forest H of G, with respect to T, as a forest, in which, parent of any vertex w is the nearest proper ancestor v of w in T such that $w \in loop(v)$ if there is such a vertex v, null otherwise. Then $loop(v)$ is the set of all descendants of v in H (see Figure 3.7). Since we only consider strongly connected graphs in our analysis and hence we have a single tree in a loop nesting forest called ***Loop Nesting Tree***. An entry to $loop(v)$ is an $e = edge(w,z)$ such that $z \in loop(v)$ and $w \notin loop(v)$. Since T is a depth-first spanning tree, every cycle contains a back arc [154]. More precisely, every cycle C contains a vertex v that is a common ancestor of all other vertices w on the cycle

[154], which means that: any $w \in C$ is in $loop(v)$. Thus, every cycle of G is contained in a loop. A loop is reducible if all its entries enter its head. A flow graph is reducible [82, 159] if all of its loops are reducible. If $loop(v)$ is reducible, then v dominates all vertices in the loop. A reducible flow graph G_s has the property such that if we delete all of its backward edges with respect to any spanning tree, then it produces an acyclic graph, which has the same dominators as flowgraph G_s [159].

3.6.2 Applications

Two fundamental tools in flow graphs are the loop nesting forests and the dominator trees. We already provided many applications of dominator trees before. Loop nesting forest gives the efficient algorithms for the following tasks: in dominator computation [29] and its verification [68], in testing whether a graph is reducible or not and then identify the reducible loops [139, 159]; in computing the bridges as well as in finding the maximally edge-disjoint spanning trees [156]; in computing a low-high order which can be used to construct the two independent spanning trees and then certify the dominator tree of a flow graph [69]. Georgiadis et al. [77] provided a technique by which we can use the loop nesting forests in order to get an efficient solutions to various problems related to 2-vertex and 2-edge connectivity of a graph.

3.6.3 Algorithms

In 1976, Tarjan [161] presented an $O(m\alpha(n,m/n))$ time pointer-machine algorithm to compute a loop nesting forest by using the disjoint set union, here α is a functional inverse of Ackermann function [160] [¶]. Later on in 1985, Gabow's and Tarjan's static tree disjoint set union algorithm [64] reduces the running time of this algorithm to $O(m)$ on a random access model[‖]. In Recent year, Buchsbaum et al. [30] gave an $O(m)$-time pointer-machine algorithm [**] in 2008, which is also called the streamlined version of Tarjan's algorithm. It also has the same asymptotic behavior but needs less storage

[¶] Please refer the appendix section A.2.2 for the details of Ackermann function

[‖] To see the description of random access model, please refer the the Appendix section A.3.1

[**] Appendix section A.3.2 contains the details of pointer machine model

space. Georgiadis et al. [77] provides the first experimental study of Tarjan's loop nesting forest algorithm, and the first implementation of its streamlined version.

3.6.3.1 Tarjan's Algorithm

Tarjan's algorithm [161] is based on *contraction* and *intervals*. Let us introduce some terminology, which will help us to explain the algorithm. Suppose T is a depth-first spanning tree, rooted at vertex s.

Contraction. Please refer Chapter 2, section 2.1.4.

Reducibility. Let consider the following two sets.
$C(w)$: $\{\ v \,|\, (v,w)$ is a cycle arc in $T\ \}$, and
$I(w)$: $\{\ v \,|\, w \xrightarrow{*} v$ and $\exists z \in C(w)$, such that $\exists$ a path from v to z which contains only descendants of w in T.
Then, it is trivial that the subgraph of G induced by the vertices of $I(w)$ is strongly connected. Let imagine that we compute the $I(n)$ in $G = G(n)$ and contract $I(n) - \{n\}$ into n and created a new graph $G(n-1)$. Again, compute the $I(n-1)$ in $G(n-1)$ and contract $I(n-1) - \{n-1\}$ into $\{n-1\}$ and created a new graph $G(n-2)$, and so on, until we reach to root vertex. Gradually G will be contracted into an acyclic graph $G(0)$ whose vertices correspond to the maximal strongly connected subgraph of G. This technique, presented in Tarjan [155], gives an efficient way to test the reducibility of G and find a pair of edge-disjoint spanning trees given by Tarjan [159].

Interval. Without loss of generality, we can assume that the root vertex $(= 1)$ has no incoming edges. Let $I(k)$ be defined in $G(k)$ for $2 \leq k \leq n$. Then the set $I(k) - \{k\}$ partition the set $\{i \,|\, 2 \leq i \leq n\}$. Therefore, the set $I(k)$ is known as an *interval*. Furthermore, the graph $T_I = \{\{1 \leq i \leq n\}, \{(h(i), i) \,|\, 2 \leq i \leq n\}\}$ is a tree, where $h(i)$ is the header or parent of i, called the *interval tree* (also called the *loop nesting tree*) of G. In other words, if T is a depth-first spanning tree of G, and H is the corresponding loop nesting forest, then for any vertex $v \in V$, the interval of v is defined by, $I(v) = \{v\} \cup \{$ children of v in $H\}$ and the tree rooted at v, called the *interval tree* (or *loop nesting tree*)

of G.

Process. To compute intervals, Tarjan's algorithm modifies the existing algorithm that tests the reducibility of the flow graphs presented in [155] and systematically computes the sets $I(w)$ by using a backward search from the vertices in $C(w)$. To represent the sets $I(w)$ and to contracts the graphs $G(n), G(n-t) \ldots G(1)$, it uses the disjoint set union method [86, 160]. Similarly, to restrict the backward searches within the vertices of $C(w)$, it uses the lowest (or nearest) common ancestor of v and w, denoted by LCA(v,w) (or NCA(v,w)) as following. For any edge (v,w), find a vertex x such that $x \xrightarrow{*} v$, $x \xrightarrow{*} w$ in T, and for any vertex y satisfying $y \xrightarrow{*} v$, $y \xrightarrow{*} w$ also satisfies $y \xrightarrow{*} x$. Aho et al. [6] provides an algorithm to compute the LCA(v,w) for each edge (v,w) that uses depth first search and the set union method available in [86, 160]. The complexity of LCA algorithm is $O(m, \alpha(m,n))$ [160], where $\alpha(m,n)$ is a very slowly growing function related to a functional inverse of Ackermann's function. The Algorithm 6 contains the detail the steps that computes $h(k)$ for all $k > 1$ and $I(i)$ for all i, which is taken from [161].

Explanation. The algorithm computes H by contracting intervals, as following. At first, it numbers the vertices from 1 to n in reverse postorder of T and identifies vertices by number. Note that, the vertex numbers correspond to a reverse postorder numbering of G (hence 1 is the start vertex). After that it process the vertices in increasing order. (Indeed, any bottom-up order of T will do.) If vertex v is processed then it computes the interval $I(v)$ of v and then contract $I(v)v$ into v as shown in Figure 3.8. For any vertex $v \in V$, if $G(v)$ is the resulting graph after contracting $I(w) \setminus w$ into w for all vertices $w > v$, then $I(v)$ is found by a backward search from v in $G(v)$ that visits only descendants of v in T. Graph $G(v)$ is computed implicitly with the use of a disjoint set union data structure available in [160], which achieves the effect of contractions. As well as maintains a collection of disjoint sets, each with a name, and supports the following operations:

make-set(x)**:** Create a new set $\{x\}$ with representative x. Element x must be in no

Algorithm 6: LOOP NESTING FOREST TARJAN ORIGINAL

Input: A strongly connected digraph $G = (V, E)$, root vertex $r(= 1)$
Output: Interval and parent (or header) of each vertex

```
/* first pass                                                    */
 1: procedure SEARCH(v)
        h(v) = i
        I(i) = I(i) ∪ {v}
        UNION(i, v)
        foreach (v, w) ∈ E(G) do
            if FIND(w) ≠ i and h(FIND(w)) = 1 then
                SEARCH(FIND(w))
 2: end procedure
/* second pass                                                   */
1  for i ← 1 to n do
2      create a set {i} named i
3      h(i) = 1
4      I(i) = {i}
/* third pass                                                    */
// delete all cross edges and forward edges in T
5  for i ← n to 2 do
6      foreach cross edge or forward edge (v, w) with LCA(v, w) = i do
7          add (v, FIND(w)) to E
8      foreach cycle edge (v, i) do
9          if h(FIND(v)) = 1 then
10             SEARCH(FIND(v))
11 for i ← 2 to n do
12     if h(i) = 1 then
13         I(1) = I(1) ∪ {i}
```

existing set.

find(x)**:** Return the representative element of the set containing element x.

unite(x, y)**:** Unite the sets containing elements x and y and give the new set the representative of the old set containing x.

The algorithm begins by executing the *make-set(v)* before processing the vertex v. To contract $I(v)$ into v, it executes the $unite(v, w)$ for all vertices w that are visited during

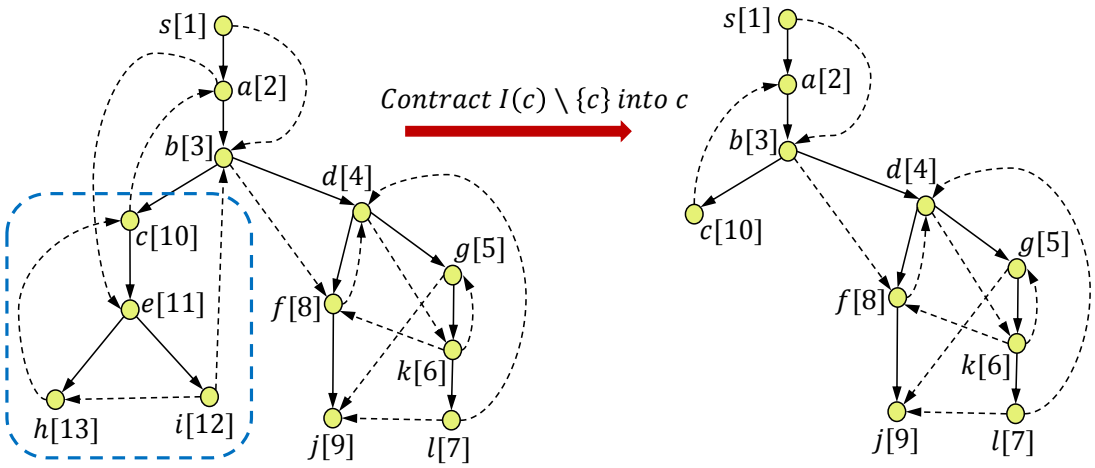


Figure 3.8: Loop nesting tree contraction example. A flow graph G_s with a depth-first spanning tree T shown with solid arcs; non-tree arcs are shown dashed; vertices are numbered in reverse postorder (in brackets). The corresponding digraph G is not strongly connected.

the backward search from v. These operations create a set which contains exactly the vertices in $loop(v)$ and assign the set name v. In order to bound the running time of the backward search from v, the algorithm need to avoid examining edges whose first end is not a descendant of v in T. To accomplish this, it computes the set $E(v)$ of each edges $(x,y) \in E$ such that v is the lowest common ancestor of x and y in T and (x,y) is not a forward edge. (Actually, forward edge can be removed without affecting the resulting loop nesting forest.) Aho et al. [6] explained that these computations can be performed during the *depth-first-search* that generates T, yielding in a two-pass algorithm or in a separate pass through the vertices, yielding in a three-pass algorithm. The algorithm runs the backward search from v in a subgraph $G'(v)$ of $G(v)$. $G'(v)$ has no edges for $v = n$. For $v < n$, it construct the $G'(v)$ from $G'(v+1)$ as following. At first, it contracts $I(v+1)\{v+1\}$ into $v+1$. Then for each edge $(x,y) \in E(v)$, if $y = v$, then (x,y) is a backward edge, and x is inserted into a set B, if $y \neq v$, then insert the edge $(find(x), find(y))$ into $G'(v+1)$. To compute $I(v)$, pop up a vertex x from B until and unless it will not be empty, but during the pop up, execute a backward search from x, collect the vertices that have not been visited already and then for each such vertex w,

assign $h(w) \leftarrow v$ and execute $unite(v,w)$.

3.6.3.2 Streamlined Version

The streamlined version of Tarjan's algorithm [30] computes a loop nesting forest by a single *depth-first-search*, which also produces T, but avoids the computation of lowest common ancestors and requires less storage space. To perform the backward search, it maintains the incoming arcs of each vertex w by $in(w)$. At the beginning, it initializes all $in-sets$ to be empty. When a vertex v is visited at the first time during the depth-first search, insert v into $in(find(w))$ for each backward or cross edge (v,w). Then, we can get the interval of v as following. When v is visited in postorder, and $in(v)$ is not empty, then remove a vertex x from $in(v)$ and compute $w \leftarrow find(x)$. Then, while $w \neq v$, compute $in(v) \leftarrow in(v) \cup in(w)$, execute $unite(v,w)$, and set $w \leftarrow find(p(w))$. The $in-sets$ can be represented by singly-linked circular lists, so that insertions, deletions, and unions can be done in $O(1)$ time. Thus, the streamlined algorithm has the same asymptotic running time, $O(m\alpha(n,m/n))$, as original version.

3.6.3.3 Memory Efficient Version

We modify the streamline version of the Tarjan's algorithm by the following observation. Every insertion of v into a $in(find(w))$ list, done during the first DFS visit to v, indicates that v has a path to $find(w)$ using only descendants of w in T, and therefore, a backward search is triggered when v will be found in the list of $find(w)$. However, if another back or cross edge (z,w) was already inserted in the list $in(find(w))$, such that z is a descendant of v in T, then the backward search (using the edges indicated from the vertices in the list $in(find(w))$) from the vertex $find(w)$ will also visit v. Thus, in this case we can avoid inserting vertex v into $in(find(w))$ if we have an efficient way to test if there exists a descendant of v that has a cross or back edge to a vertex z, where $find(z) = find(w)$. We can implement a simple test that discovers such cases, as follows. For each vertex v, we keep a variable $last[v]$ that stores the last vertex that was inserted into $in(v)$. We also change the order in which vertices are inserted into the in lists.

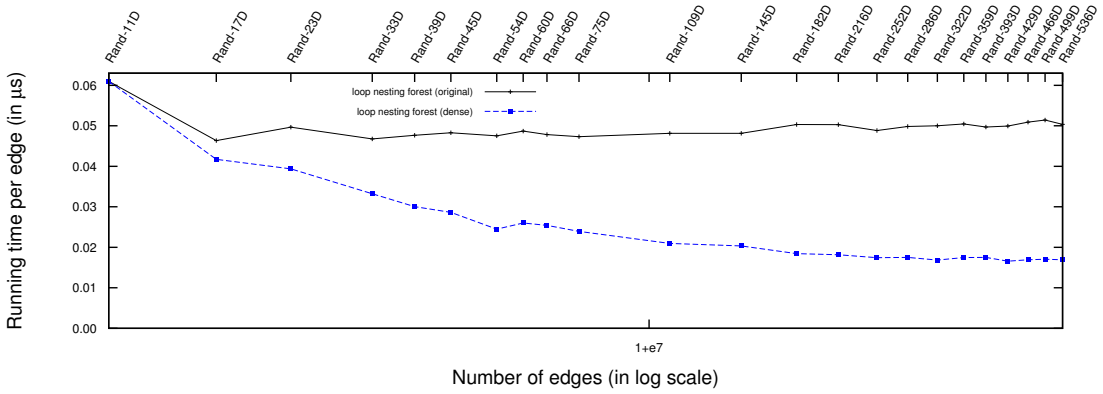


Figure 3.9: Running times in μs per edge to form the loop nesting tree by the original 1 pass algorithm and the memory efficient algorithm. Data are given in Table 3.1. (Better viewed in color.)

Instead of doing the insertions of v in preorder, we do them in postorder, we just start the backward search from v. In this way, when v is visited in postorder, the vertices that were processed just before v are descendants of v in T. Hence, if such a descendant of v has an outgoing cross of back edge to another vertex w, then this will be stored as the last vertex that was inserted into $in(find(w))$; if another vertex was inserted later into $in(find(w))$ it will be again a descendant of w. This allows us to reduce the total number of elements that are inserted into the *in* lists and hence uses the less memory storage for the computation.

We implemented both the streamline and the memory version of the algorithms to compute the loop nesting forest of a given directed graph $G = (V, E)$. We use the same data structure to represent the graph and for the computations. After that, we did the experimental observations to know their difference in running time. For the analytical observation, we created the random graphs with a fixed number of vertices $(100K)$ and edge to vertex ratio (density) in a range that spans from 11 to 536. Figure 3.9 plots the experimental reports of their running time, data are presented in Table 3.1. It showed that, if the graph density is increasing then the memory efficient version pays off its performance over the streamline version.

Graph	Vertices	Edges	original algorithm	memory efficient algorithm
Rand-11D	100K	1.1M	0.068	0.068
Rand-17D	100K	1.7M	0.080	0.072
Rand-23D	100K	2.3M	0.116	0.092
Rand-33D	100K	3.3M	0.152	0.108
Rand-39D	100K	3.9M	0.184	0.116
Rand-45D	100K	4.5M	0.216	0.128
Rand-54D	100K	5.4M	0.256	0.132
Rand-60D	100K	6.0M	0.292	0.156
Rand-66D	100K	6.6M	0.316	0.168
Rand-75D	100K	7.5M	0.356	0.180
Rand-109D	100K	10.9M	0.524	0.228
Rand-145D	100K	14.5M	0.700	0.296
Rand-182D	100K	18.2M	0.916	0.336
Rand-216D	100K	21.6M	1.084	0.392
Rand-252D	100K	25.2M	1.232	0.440
Rand-286D	100K	28.6M	1.424	0.500
Rand-322D	100K	32.2M	1.612	0.544
Rand-359D	100K	35.9M	1.812	0.628
Rand-393D	100K	39.3M	1.952	0.688
Rand-429D	100K	42.9M	2.144	0.712
Rand-466D	100K	46.6M	2.372	0.788
Rand-499D	100K	49.9M	2.568	0.852
Rand-536D	100K	53.6M	2.700	0.908

Table 3.1: Running time comparison between original algorithm and memory efficient algorithm, to create the loop nesting forest, time are in seconds. We keep fixed the number of vertices to 100K while we increase the edge density value.

4

2-Edge-Connected Blocks

4.1 Introduction

Let us recall the definition of *2-edge-connected blocks* (2ECB) from Chapter 2. Given a digraph $G = (V, E)$, we say that two vertices $u, v \in V$ are *2-edge-connected*, and denote this relation by $u \leftrightarrow_{2e} v$, if there are two *edge-disjoint* directed paths from u to v and two *edge-disjoint* directed paths from v to u. Note that, the paths from u to v and the paths from v to u need not be *edge-disjoint*. Moreover, two paths from u to v or two paths from v to u can share the vertices. Menger's Theorem [121]* also states the equivalent

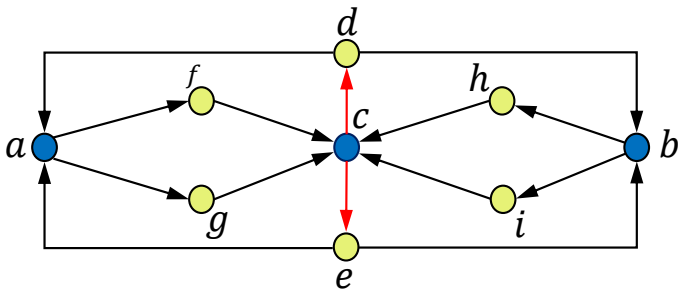


Figure 4.1: Example of *2-edge connected blocks*, the vertices $\{a,b,c\}$ are in same 2ECB, but the paths from a to b ($\{a,f,c,d,b\}$,$\{a,g,c,e,b\}$) and b to a ($\{b,h,c,d,a\}$,$\{b,i,c,e,a\}$) share the edges (c,d) and (c,e), as well as contain the vertices d,e,f,g,h, which are not in the same 2ECB. Also two *edge-disjoint* path between the vertex a and b share the vertex c. (Better viewed in color.)

defintion of the 2-edge-connected relation in a graph as follows: Two vertices u and v in G are 2-edge-connected, if and only if the removal of any edge from G, leaves them in the same strongly connected component. If G is *2-edge-connected*, then it does not contain any *strong bridge*. A 2ECB of a digraph $G = (V, E)$ is defined as a maximal

*To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

subset $B \subseteq V$ such that $\forall u,v \in B$, $u \leftrightarrow_{2e} v$. Hence, if u,v be vertices of a 2ECB then the paths of $u \leftrightarrow_{2e} v$ may contain the vertices that do not belong to the same 2ECB as illustrates in Figure 4.1, where, the vertices a,b and c are in the same 2ECB. But two paths from a to b and two paths from b to a share a vertex c. Also, the paths from a to b ($\{a,f,c,d,b\},\{a,g,c,e,b\}$) and b to a ($\{b,h,c,d,a\},\{b,i,c,e,a\}$) share the edges (c,d) and (c,e), as well as contain the vertices d,e,f,g,h, which are not in the same 2ECB. Furthermore, two distinct 2ECBs may have only one vertex in common that is going to explain by following leema.

Lemma 4.1.1. *Two different 2ECBs do not have any vertex in common.*

Proof. We proceed by contradiction.

Let us consider that A and B be two different 2ECBs in a digraph $G = (V,E)$, and $w \in V(G)$ such that $A \cap B = w$ as illustrated in Figure 4.2.

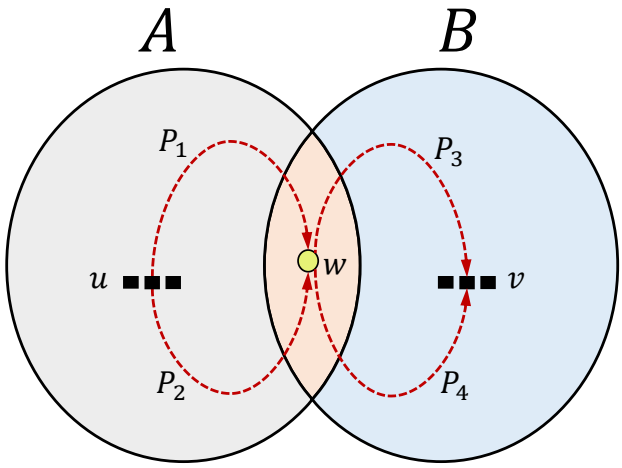


Figure 4.2: Maximum vertices between two different 2ECBs of digraph $G = (V,E)$, A and B have common vertices a and b. By definition, they are not 2ECBs because for two distinct vertices $x,y \in G$; they are not the maximal subset of G such that $x \leftrightarrow_{2e} y$.

Since $w \in A \implies$ for each vertex $u \in A \setminus \{a\}$, $u \leftrightarrow_{2e} w \implies$ there exist two distinct *edge-disjoint* paths P_1 and P_2 that start from u and end at w. Similarly, $w \in B \implies$ for

each vertx $v \in B \setminus \{a\}$, $a \leftrightarrow_{2e} v \implies$ there exist two distinct *edge-disjoint* paths P_3 and P_4. Both paths are start from w and end at v.

Therefore, there exist two distinct edge-disjoint paths $\mathscr{P}_1 = P_1 \cup P_3$ (or $P1 \cup P_4$) and $\mathscr{P}_2 = P_2 \cup P_4$ (or $P_2 \cup P_3$) from any vertex $u \in A$ to every vertex $v \in B$.

Analogously, we also have the two edge-disjoint paths from v to u as well. Then, by the definition of 2ECB, i.e., a maximal subset $B \subseteq V$such that $\forall u, v \in B,\ u \leftrightarrow_{2e} v$, and here we find A and B are not maximal, so, they cannot be the 2ECBs, it is a contradiction.

Note: This lemma elaborates the relations between two different 2ECBs, which will be helpful to explain the algorithms in next section.

□

4.2 Related Work

In 2014, Jaberi [89] presented an algorithm to compute the 2-edge-connected blocks of any digraph $G = (V,E)$ with $O(n \min\{m, b^* n\})$ time complexity where b^* is the number of strong bridges in G. Nevertheless, it may have $O(mn)$ time complexity for the worst case graph, which has $n-1$ strong bridges as shown in Figure 4.3. In the same year 2014, Georgiadis et al. [71] proposed the three different algorithms for the 2ECB computation of the digraph G. (i) A simple iterative algorithm with $O(mn)$ time complexity, (ii) recursive algorithm which also has $O(mn)$ time complexity, and (iii) fast algorithm with linear time bound $O(m+n)$ that combine the iterative and the recursive techniques. Georgiadis et al. [71] presented the Pioneer algorithm to compute the 2-edge-connected blocks of a digraph in linear-time bound. That was the first real progress on this extremely natural problem, starting from the foundational work done 40 years ago for undirected graphs. After one year, in 2015, Georgiadis et al. [73] again presented an algorithm for 2ECB computation based on *loop nesting tree* and *dominator tree* informations. On the basis of their algorithm, we also present a memory efficient version algorithm to compute the 2ECBs. Our algorithm modifies the existing loop nesting forest computation and produce the better performance for the dense graph, which is already explained in Chapter 3 - Section 3.6.3.3.

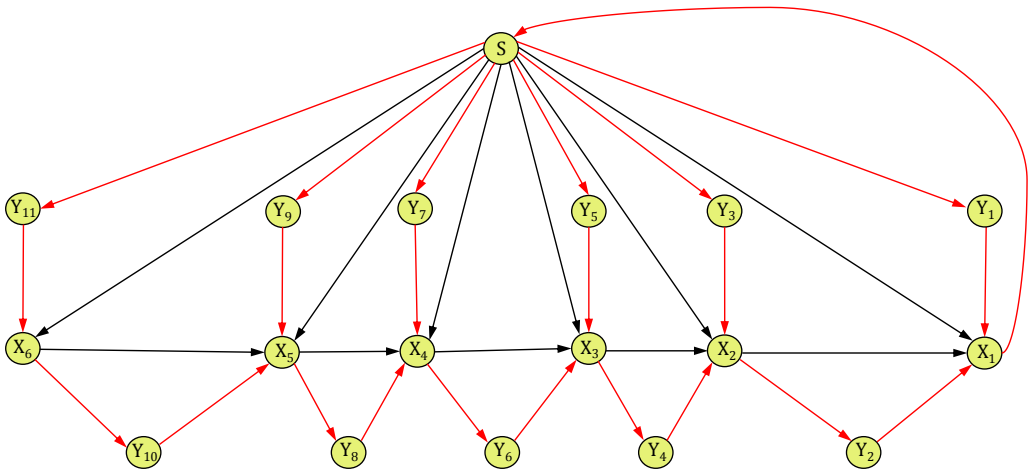


Figure 4.3: A strongly connected digraph with n vertices and greater than n bridges where it causes k recursive calls of Recursive Algorithm, Vertices $X_1, X_2, \ldots, X_k$ are not 2-edge-connected, but Recursive Algorithm requires k recursive calls (in this case $k = 6$) to separate them into different blocks. (Better viewed in color.)

There is only one experimental study is available for 2ECBs computation of the digraphs, which is done by Di Luigi et al. [44]. They compared the linear-time algorithms [71] to the simple $O(mn)$-time algorithms. Their experimental results show that simple algorithms are not competitive with the more sophisticated linear-time algorithms. Furthermore, Di Luigi et al. [44] didn't include the linear time algorithm that is presented in [73]. We have done the comparative empirical analysis between all available linear time algorithms. We did not include the algorithms of Jaberi [89] because of their extensive requirements in storage space. Moreover, our key focus is to observe the *running time* of linear time algorithms rather than other algorithms. In next sections, we will explain the high-level idea of these algorithms and then report our experimental observation later.

4.3 Algorithms

Before starting to explain the algorithms, let us define some common notions as following. Let G_s (resp., G_s^R) be the flow-graph of G (resp., G^R) and let D (resp., D^R)

denotes the dominator trees of flow-graph G_s (resp., G_s^R). Similarly, let T_u denotes the tree rooted at u and $T(v)$ tells the tree contains the vertex v. Also, let the symbol $[v]_{2e}$ represent a 2-edge-connected block contains the vertex $v \in V(G)$.

4.3.1 Simpler Algorithm

If the graph does not have any strong bridges, then it will be 2-edge-connected. Therefore, for every strong bridge e, we can obtain the 2-edge-connected blocks of $G = (V,E)$ by computing the strongly connected components of $G \setminus e$. A strong bridge e separates the distinct vertices u and v, if all paths from u to v contain an edge e or all the paths from v to u contain an edge e. Hence, u and v must be lie in two different strongly connected components of $G \setminus e$. This trivial observation gives a characterization of the 2ECB in terms of the strong bridges. The details steps of this process is available in Algorithm- 7, which is taken from [71]. Since Italiano et al. [88] already proved that, any digraph $G = (V,E)$ may have maximum $2n-2$ strong bridges, where n is the number of vertices, hence, Algorithm- 7 may have $O(mn)$ time bound in the worst case scenario.

Algorithm 7: Simpler

Input: A strongly connected digraph $G = (V,E)$
Output: 2-edge connected blocks of G

1 Initialize the 2-edge-connected blocks as $[v]_{2e} = V$ (start from the trivial partition containing only one block).
2 Compute the strong bridges of G.
3 **foreach** *strong bridge of* $e \in E(G)$ **do**
4 Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus e$
5 Let $\{[v_1]_{2e}, \ldots, [v_l]_{2e}\}$ be the current 2-edge-connected blocks. Refine the partition into blocks by computing the intersections $[v_i]_{2e} \cap S_j$ for all $i = 1, \ldots, l$ and $j = 1, \ldots, k$.

4.3.2 Recursive Algorithm

This algorithm is based on the bridge decomposition of the dominator trees and auxiliary graphs. Let us suppose a 2ECB, which has a specific vertex v, then any vertex

$w(\neq v)$, will be in same 2ECB $\iff$ $v \leftrightarrow_{2e} w$ (i.e., there are two edge-disjoint path from v to w and two edge-disjoint path from w to v). We can divide this computation into two parts. Find the set of vertices $[v] \rightarrow_{2e}$ that are 2-edge-connected from v and calculate the set of vertices $[v] \leftarrow_{2e}$ that are 2-edge-connected to v. Then, $[v]_{2e}$ is formed by the intersection of these two sets $[v] \rightarrow_{2e}$ and $[v] \leftarrow_{2e}$. To perform such computation in an advance way, this algorithm uses the bridge decomposition and auxiliary graphs.

Algorithm 8: Recursive

Input: A strongly connected digraph $G = (V, E)$
Output: 2-edge-connected blocks of G

1 Choose an arbitrary ordinary vertex $s \in V^o$ as a start vertex. Compute the dominator trees D and D^R and the bridges of the flow graphs G_s and G_s^R.
2 Compute the number b of bridges (x, y) in G_s such that y is an ancestor of an ordinary vertex in D. Compute the number b^R of bridges (x, y) in G_s^R such that y is an ancestor of an ordinary vertex in D^R.
3 **if** $b = b^R = 0$ **then**
4 ⌊ return $[s]_{2e} = V^o$
5 **if** $b^R > b$ **then**
6 ⌊ swap G_s and G_s^R
7 Find the bridge decomposition of D into the subtrees T_r and compute the corresponding auxiliary graphs Gr. Compute recursively the 2ECB for each auxiliary graph G_r with at least two ordinary vertices.

Let us consider the computation of $[v] \rightarrow_{2e}$. When we compute the vertex dominator tree D, then we could easily identify the bridges of G. Since Italiano et al. [88] proved that, bridges in flow graph are also bridges in Graph G. In addition, each bridge $e = (u, w)$ is also an edge in D such that $e = (u = d(w), w)$. Therefore, let find all such vertices w in D and marked them, after that, let us follow the concepts given by following lemmas, which are taken from Georgiadis et al. [71].

Lemma 4.3.1 (Georgiadis et al. [71]). $z \in [v] \rightarrow_{2e} \iff$ *marked vertex does not dominates z in* G_s.

We can compute the $[v] \leftarrow_{2e}$ by the same way, but we have to operate it in reverse graph G_s^R, and its dominator tree D^R. Note that, a vertex marked in D may not be

marked in D^R and vice versa. However, the common set will be same that hinges by the following lemma.

Lemma 4.3.2 (Georgiadis et al. [71]). *$z \in [v]_{2e} \iff$ any marked vertex does note dominates z in G_s and in G_s^R. Moreover, $[v]_{2e}$ can be computed in $O(m)$ time.*

According to Lemma 4.3.2, for a single vertex v, $[v]_{2e}$ takes $O(m)$ times, which implies that it will take $O(mn)$ time for n vertices. Now, let describe it in sophisticated way that avoids repeated computation of 4.3.2. However, it also has the $O(mn)$ time complexity, but will be a useful ingredient to explain the linear-time algorithm. Let remove all edges $(d(v), v)$ (resp., $(d^R(v), v)$) such that v is marked in D (resp., D^R). This decomposes the dominator trees D (resp., D^R) into forest $\mathscr{F}$ of rooted trees, where each tree is rooted either at a marked vertex or at the start vertex s as shown in Figure 4.4 (c), known as a *bridge decomposition* of D (resp., D^R) into many subtrees $T(v)$ (resp., $T^R(v)$). Then, we can proceed to the next step by using the following lemma, which is taken from [71].

Lemma 4.3.3 (Georgiadis et al. [71]). *If v and w are the distinct vertices in $V(G)$, then $[v]_{2e} = [w]_{2e} \iff T(v) = T(w)$ and $T^R(v) = T^R(w)$.*

Lemma 4.3.3 gives the the necessary but not sufficient condition for two distinct vertices v and w to be 2-edge-connected. Because they may lie in the same subtree in both bridge decompositions of D and of D^R, which can be separated by a strong bridge. To overcome this problem, it will use the parent property of dominator trees [69], and structural properties for paths [71] that connect vertices in different subtrees that are going to state below.

Lemma 4.3.4. (Parent property of the dominator tree [69].) *$\forall e = (v,w) \in E(G)$, $d(w)$ is an ancestor of v in D.*

Lemma 4.3.5. (Structural property of a path in dominator tree [71].) *Let $e = (u,v) \in E(G)$ such that $T(u) \neq T(v)$ and let r_v be the root of $T(v)$. Then either $u = d(v)$ and e is a bridge in G_s, or u is a proper descendant of r_v in D.*

Lemma 4.3.6. (Structural property of a path in dominator tree [71].) *Let r be a marked vertex in D and v be any vertex that is not a descendant of r in D. Then there is path from v to r that does not contain any vertex in $T(r) \setminus r$. Moreover, all simple paths from v to any vertex in $T(r)$ contain the edge $(d(r), r)$.*

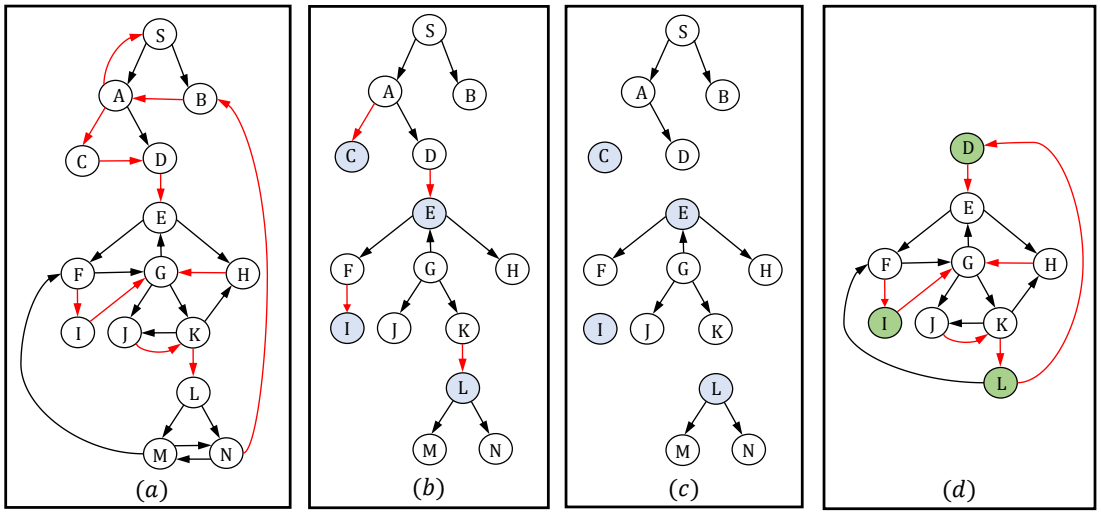


Figure 4.4: (a) A flow graph G_s, (b) Dominator tree D of G_s, (c) Bridge decomposition into the subtrees $T(v)$ induced by the bridges of G_s, and (d) together with the auxiliary graph of vertex E. Strong bridges of G and bridges of G_s are shown in red; marked vertices are shown in light blue (Example is taken from [71] and better viewed in color.)

Auxiliary graphs. Auxiliary graphs were defined in [71], to decompose the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original 2-*edge-connected blocks of* G. For each tree $T_r \in \mathscr{F}$, we construct an auxiliary graph $G_r = (V_r, E_r)$ as follows. The vertex set of G_r consists of ordinary and boundary vertices. All the vertices $v \in T_r$ are the ordinary vertices and v became a boundary vertex in V_r if it has a marked child in D. Let w be a marked child of a boundary vertex v, then contract all the descendants of w in D into w. Still, if there exist any vertices in $V \setminus T_r$, which are not the descendants of r $(\neq s)$ are contracted into $d(r)$. In these contractions process, all the parallel edges are eliminated, as you can see on Figure 4.4 (d) taken from [71]. We can compute all auxiliary graphs G_r in $O(m)$ time, and each auxiliary graph is strongly connected. Thus, by using bridge decomposition and auxiliary graph,

we can get the 2ECB of any digraph $G = (V, E)$ in $O(mn)$ time. The details steps are shown in Algorithm 8, which is adapted from [71].

Algorithm 9: AUXE

Input: A strongly connected digraph $G = (V, E)$
Output: 2-edge-connected blocks of G

1 Choose an arbitrary ordinary vertex $s \in V$ as a start vertex. Compute the dominator trees D and the bridges of the flow graphs G_s.
2 Partition D into subtrees T_r and compute the corresponding auxiliary graphs G_r.
3 **foreach** auxiliary graph $H = G_r$ **do**
4 Compute the dominator tree D_H^R and the bridges of H^R. Let $d_H^R(q)$ be the parent of $q \neq r$ in D_H^R.
5 Partition D_R^H into the subtrees $T_H^R(q)$. Compute the corresponding auxiliary graphs H_q^R with $q \neq r$.
6 Set $[r]_{2e}$ to consist of the ordinary vertices in $T_H^R(r)$.
7 **foreach** auxiliary graph H_q^R *with* $q \neq r$ **do**
8 Compute the strongly connected components $S_1, S_2, \ldots, S_k$ of $H_q^R \setminus (d_H^R(q), q)$.
9 Partition the ordinary vertices of H_q into blocks according to each $S_j, j = 1, \ldots, k$; For each ordinary vertex v, $[v]_{2e}$ contains the ordinary vertices in the strongly connected component of v.

4.3.3 Linear Time Algorithm Through Auxiliary Graph

A careful integration of Simple and Recursive algorithm gives a linear time algorithm. The critical observation shows that if a strong bridge separates different pairs of vertices in successive recursive calls (which create the worst-case scenario for Recursive Algorithm as shown in Figures 4.3), then it will appear as the strong bridge entering the root of a subtree in the bridge decomposition of a dominator tree. Algorithm 9 described the detailed steps of this combination that provide the linear time algorithm (see [71] for the verification). We refer this algorithm as AUXE. The main idea is that it executes the Recursive Algorithm but stops the recursion at the second level. Two vertices that are not 2-edge-connected but have not been separated yet (i.e., they are ordinary vertices of an auxiliary graph computed at recursion depth level 2) can be separated by running

the Simpler Algorithm for the specific auxiliary graph. It suffices to remove only one strong bridge of that particular auxiliary graph, and we can do that by executing the step 3 of the Simpler Algorithm.

Algorithm 10: LNFE

Input: A strongly connected digraph $G = (V, E)$
Output: 2-edge connected blocks of G

/* Initialization */
Choose an arbitrary vertex $s \in V$ as a start vertex. Compute the reverse graph G^R
Compute the dominator trees D and D^R of the flow graphs G_s and G_s^R, respectively.
Compute the loop nesting trees H and H^R of the flow graphs G_s and G_s^R, respectively.
Compute $\mathscr{F}$ (the bridge decomposition of D) and D (the bridge decomposition of D^R).
foreach $v \in V$ **do**
    Find the roots r_v and r_v^R in the bridge decomposition
    Find the nearest boundary vertices h_v and h_v^R
    Set $label(v) = \langle r_v, h_v, r_v^R, h_v^R \rangle$
/* Computation of 2-edge-connected blocks */
Sort the tuples $\langle label(v), v \rangle$ lexicographically by their labels
Partition the vertices into blocks, where $u, v \in V$ are in the same block if and only if $label(u) = label(v)$

4.3.4 Linear Time Algorithm Through Loop Nesting Tree and Dominator Tree

Recently, Georgiadis et al. [73] presented new linear-time algorithms to compute the 2-edge-connected blocks, based on loop nesting trees. We refer to this algorithm as LNFE. Along with the previous notations, let H (resp., H^R) denotes the loop nesting tree of the flow-graph G_s (resp., G_s^R). The algorithm assigns a label to each vertex v, $label(v) = \langle r_v, h_v, r_v^R, h_v^R \rangle$. Where r_v (resp., r_v^R) is the root of the trees that contain v in the bridge decompositions of D (resp., D^R) respectively. Also, h^v (resp., h_v^R) is the nearest ancestor w of v in loop nesting tree H (resp., H^R) such that $h(w) \in D_w$ (resp., $h^R(w) \in D_w^R$). The vertices that have identical label are in same 2-edge-connected block

of G (proof is available in [73, page 32]). Details of the steps are presented in Algorithm 9 that is taken from [73].

4.3.5 Memory Efficient Version

We already explained in the Chapter-3, Section-3.6.3.3 that we can improve the memory usage in loop nesting tree computation by avoiding the unnecessary edge insertions in the dynamic lists. Since the vertices of a loop are contracted into a single vertex, when the algorithm tries to insert parallel edges in the dynamic lists, then by our identification, algorithm can avoid such insertions. We engineer the implementation of LNFE by our concept and refer as LNFE-ME.

4.4 Experimental Analysis

In this section, we are going to report our experimental observations that we obtained, using the algorithms AUXE, LNFE and LNFE-ME. We implemented all algorithms in plain `C++` without using any external graph library. In particular, we implemented the loop-nesting-tree based algorithms, LNFE and its memory-efficient versions LNFE-ME. Moreover, we note that all these three algorithms, AUXE, LNFE, LNFE-ME were implemented within a uniform framework, use the same data structures for representing graphs. We compiled our source codes by `g++ v.4.8.4` with full optimization (flag `-O3`). The experiments were conducted in a 64−bit GNU/Linux machine running on Ubuntu `14.04LTS`. The machine has an `3696MHz` Intel `i7-4790` octa-core processor, `16GB` of RAM, `16MB` of L3 cache, and each core has a `2MB` private L2 cache. We measured CPU running time using the `getrusage` function, and memory consumption using Valgrind [†] (`v.3.11`). All experiments are executed on a single core without using any *parallelization*. All the running times in our experiments were averaged over ten different executions.

[†] `http://valgrind.org/`

Implementation issue. Both the loop nesting tree and dominator tree use the depth-first search (DFS) tree for their computation. Therefore, in the combined computation, we can use the single depth-first search, which makes it possible to deallocate the adjacency list of the forward graph during the computation of the dominator tree (because we compute loop nesting during the DFS formation and dominator tree later). We use the same idea for the reverse graph as well. Furthermore, we reversed the graph simply by swapping the roles of forward and reverse adjacency lists rather than making a complete new copy.

Dataset. We considered several real-world graphs whose characteristics are summarized in Table 4.1. Most of them are taken from the 9^{th} *DIMACS* implementation challenge [43] and from the Stanford Large Network Dataset Collection [107]. We also generated random graphs with specific properties in order to analyze in more depth the performance of some algorithms. The characteristics of random graphs are presented in Table 4.3.

Analysis. We start our evaluation by applying the algorithms AUXE, LNFE, and LNFE-ME on the datasets presented in Table 4.1. Figure 4.5 (top) plots the running times, reported in Table 4.2. We observe that, on average, algorithm LNFE is about 2.04 times faster than AUXE and almost have the same level of performance with LNFE-ME because LNFE is around 0.08% faster than LNFE-ME. The result is expected for sparse input graphs, since in this case LNFE-ME does not get the chance to avoid the insertion of many parallel edges in the dynamic lists used by the loop nesting tree computations, but still spends the time to maintain two additional arrays that are necessary for filtering all the edges.

Furthermore, we noticed that both LNFE and LNFE-ME are more robust than AUXE, in the sense that their running times are less sensitive to the structure of the graphs. The performance of AUXE, on the other hand, is more dependent on the graph structure, which affects the number and size of the auxiliary graphs. More specifically, the AUXE algorithm is favored in graphs with few 2-edge-connected blocks because it cre-

Graphs				2-edge connected blocks		
Name	Type	$n = \|V(G)\|$	$m = \|E(G)\|$	Max-size	Avg-size	Total #
p2p-Gnutella31	P2P	14.1K	50.9K	8.0K	8K	1.0
web-NotreDame	WG	54.0K	296.2K	16.5K	35.6	760.0
soc-Epinions1	SN	32.2K	443.5K	18.1K	261.0	70.0
Amazon0302	PCP	241.8K	1.1M	140.2K	24.6	7.3K
WikiTalk	SN	111.9K	1.5M	50.3K	8.4K	6.0
web-Stanford	WG	150.5K	1.6M	58.6K	76.9	1.2K
Amazon0601	PCP	395.2K	3.3M	305.9K	87.7	3.7K
web-Google	WG	434.8K	3.4M	225.0K	56.2	4.8K
web-BerkStan	WG	334.9K	4.5M	128.2K	64.4	2.9K
SAP-4M	MP	4.1M	11.9M	141.5K	39.6	5.3K
Oracle-6M	MP	6.4M	15.9M	389.4K	19.0	44.9K
SAP-11M	MP	11.1M	36.4M	751.8K	36.7	25.8K
USA-USA	RN	23.9M	57.7M	16.1M	158.7	105.7K
LiveJournal	SN	3.8M	65.3M	2.9M	747.7	39.5K
SAP-32M	MP	32.3M	81.8M	264.1K	22.3	27.0K
SAP-70M	MP	69.8M	214.9M	1.3M	8.7	44.8K

Table 4.1: The characteristics of the real-world graphs that we considered; n and m refers to the number of vertices and the number of edges, respectively. The graph types are encoded as follows: road network (RN), peer to peer (P2P), web graph (WG), social network (SN), production co-purchase (PCP), memory profiling (MP). The graphs are sorted in increasing order according to their number of edges. Additionally, we report the statistics of their 2-edge-connected blocks, whose size refers to the number of their vertices.

ates fewer auxiliary graphs (even of large size). To investigate more this effect, we executed AUXE on the following type of artificial graphs. For a fixed number of vertices (100K) we marginally increase the edge-density by 0.5 each time creating three different types of graphs: (*i*) a graph that is 2-edge-connected, (*ii*) a graph containing 44 2-edge-connected blocks with sizes in the range from 2 to $n/5$, and (*iii*) a graph containing 10K blocks of equal size. We present the plot of this experiment in Figure

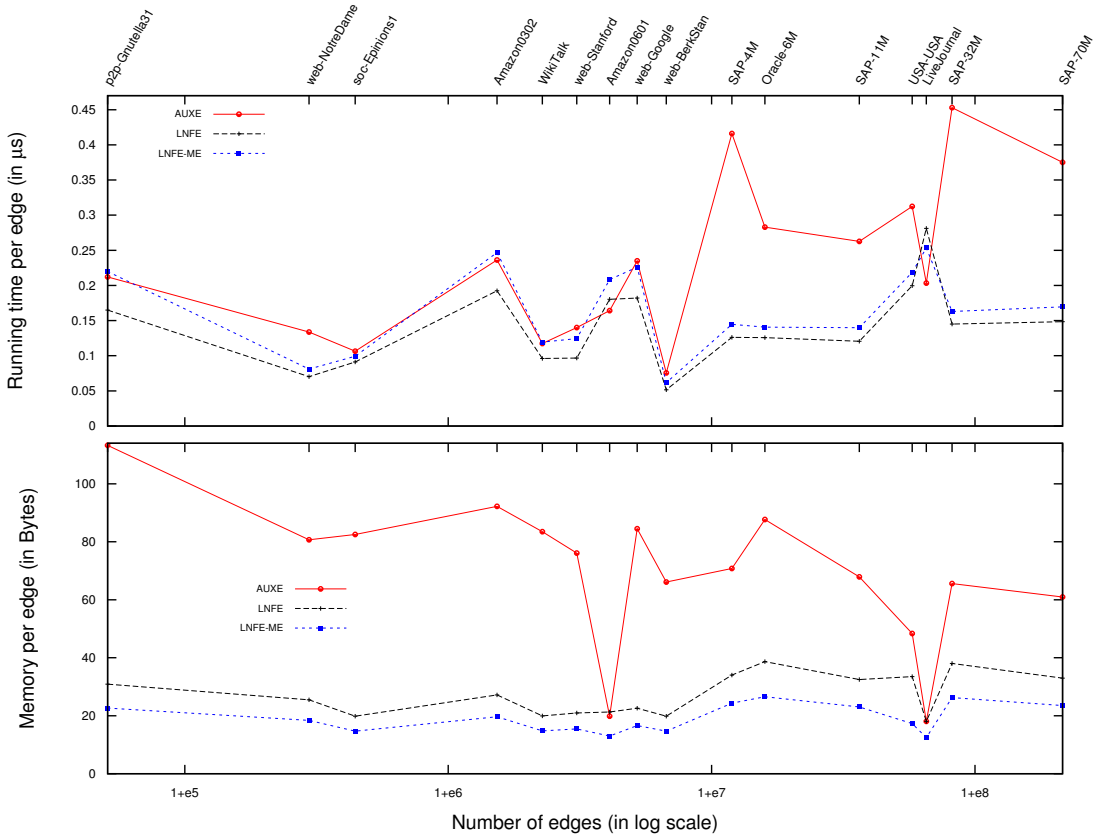


Figure 4.5: Running times per edge in μs (top) and Memory usage per edge in Bytes (bottom) of the algorithms AUXE, LNFE and LNFE-ME on the real world graph datasets presented in Table 4.1. (Better viewed in color.)

4.6 and the data are in Table 4.4. As we notice that the running time of AUXE is consistently larger in the case of type (*iii*) compared to type (*i*) graphs, which verifies that the existence of many 2-edge-connected blocks (of equal size) slows down the algorithm. Additionally, on the type (*ii*) graphs the running time is in between the other two cases and gets closer to type (*i*) graphs as the density increases.

Next, we analyze the memory consumption of the algorithms AUXE, LNFE and LNFE-ME. Figure 4.5 (bottom) plots the memory usage of the algorithms, reported in Table 4.2. From this plot, it is clear that LNFE and LNFE-ME significantly require less memory for all input graphs. Notice that the AUXE algorithm requires much more memory than the algorithm that uses the loop nesting tree (in this case by a factor of 2.62 than LNFE-ME). The LNFE-ME algorithm improves the memory consumption of LNFE

Graphs	Running times in seconds			Memory consumption in MBytes		
	AUXE	LNFE	LNFE-ME	AUXE	LNFE	LNFE-ME
p2p-Gnutella31	0.011	0.008	0.011	5.5	1.5	1.1
web-NotreDame	0.040	0.021	0.024	22.8	7.2	5.2
soc-Epinions1	0.047	0.040	0.044	34.9	8.4	6.2
Amazon0302	0.267	0.218	0.279	99.5	29.3	21.2
WikiTalk	0.174	0.142	0.176	117.7	28.1	20.8
web-Stanford	0.221	0.152	0.196	114.4	31.5	23.2
Amazon0601	0.542	0.595	0.688	62.5	67	40.8
web-Google	0.803	0.623	0.773	275.6	73.7	54.1
web-BerkStan	0.342	0.233	0.279	285.3	85.6	63.3
SAP-4M	4.968	1.506	1.728	806.2	387.7	277.1
Oracle-6M	4.504	2.002	2.238	1331.2	587.1	402.8
SAP-11M	9.551	4.388	5.086	2355.2	1126.4	798.8
USA-USA	18.024	11.517	12.622	2662.4	1843.2	951
LiveJournal	13.292	18.372	16.554	1126.4	1126.4	777.1
SAP-32M	37.062	11.864	13.338	5120	2969.6	2048
SAP-70M	80.614	31.897	36.426	12492.8	6758.4	4812.8

Table 4.2: Running times in seconds and Memory consumption in MBytes respectively of all the algorithms for computing the 2-edge-connected blocks executed on the real world graphs of Table 4.1

by about 46.5%, on average. Since all the real-world graphs that we consider are sparse, therefore, we do additional experiments to compare the performance of the algorithms AUXE, LNFE and LNFE-ME on dense graphs in order to highlight the full potential of the LNFE-ME algorithm. We considered random graphs with a fixed number of vertices (100K) and density in a range that spans from 11 to 536. The results are plotted in Figure 4.7 (top). The observation is that LNFE gradually loses the advantage over LNFE-ME, and it even becomes significantly slower for very dense graphs. The bottleneck for dense graphs is that the loop nesting tree computation needs many memory writes when

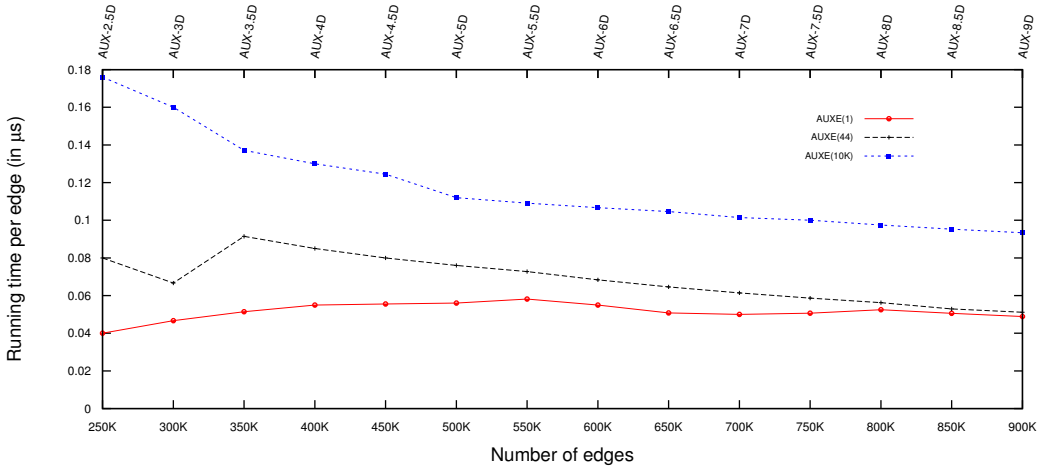


Figure 4.6: Running times in seconds of the AUXE algorithm on the following type of random graphs. The number of vertices is fixed (100K) in all graphs and we marginally increase the density by 0.5, each time creating three graphs: (i) a graph that is 2-edge-connected (represented by the line AUXE(1)), (ii) a graph containing 44 2-edge-connected blocks with sizes in the range from 2 to $n/5$ (represented by the line AUXE(44)), and (iii) a graph containing $10K$ 2ECBs of equal size (represented by the line AUXE(10K)). The data are presented in Table 4.4. (Better viewed in color.)

it inserts edges into dynamic lists. On the contrary, LNFE-ME was designed to filter many unnecessary insertions in the dynamic lists maintained by the loop nesting tree algorithm, and hence, it performs consistently faster than LNFE. In the case of AUXE, it has to make only one auxiliary graph, so as we explain before; it is faster than LNFE and LNFE-ME.

Also, we present separately the comparison between the two versions of the algorithm that uses the loop nesting tree LNFE and LNFE-ME. Because, both the algorithms use the same steps to compute the 2 edge-connected blocks. We analyze the running time in different steps as following. (i) set-up time (time to read the graph and create the adjency list), (ii) dfs-time (time to create the depth first search), (iii) lnf-time (time to create the loop nesting forest), (iv) dom-tree time (time to create the dominator tree) and (v) processing time(time to assign the label to each vertex and then create the blocks). Figure 4.8 represents the graphical representations of these time fractions. It shows that the LNFE-ME pays off over LNFE when the graphs get denser.

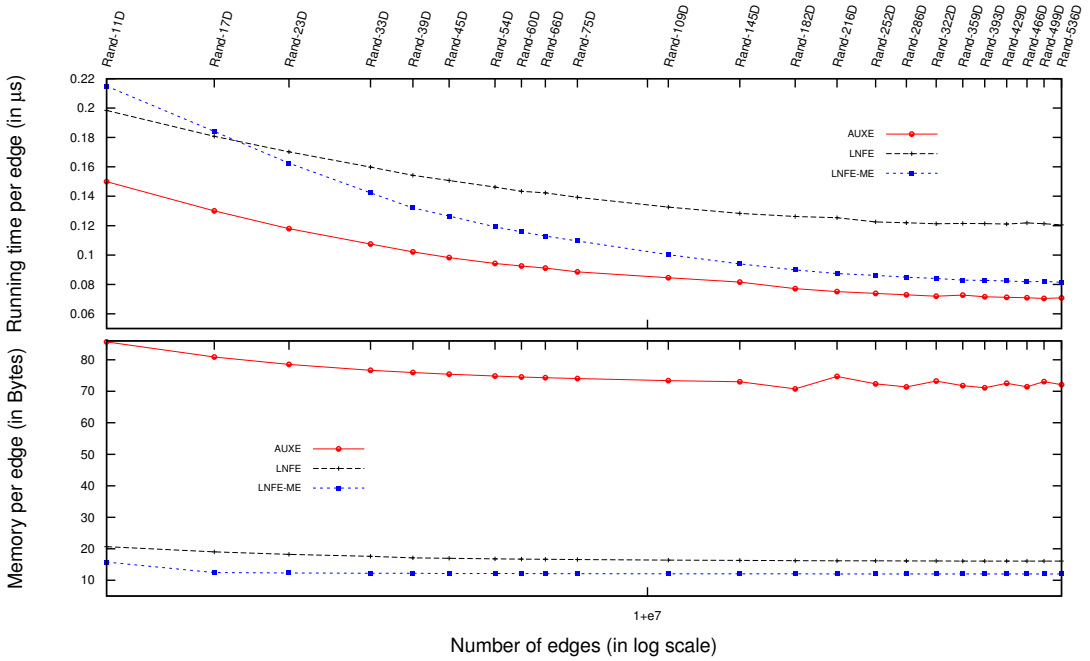


Figure 4.7: Running times per edge in μs (top) and memory usage per edge in Bytes (bottom) of the algorithms AUXE, LNFE and LNFE-ME on the random graphs summarized in Table 4.3. (Better viewed in color.)

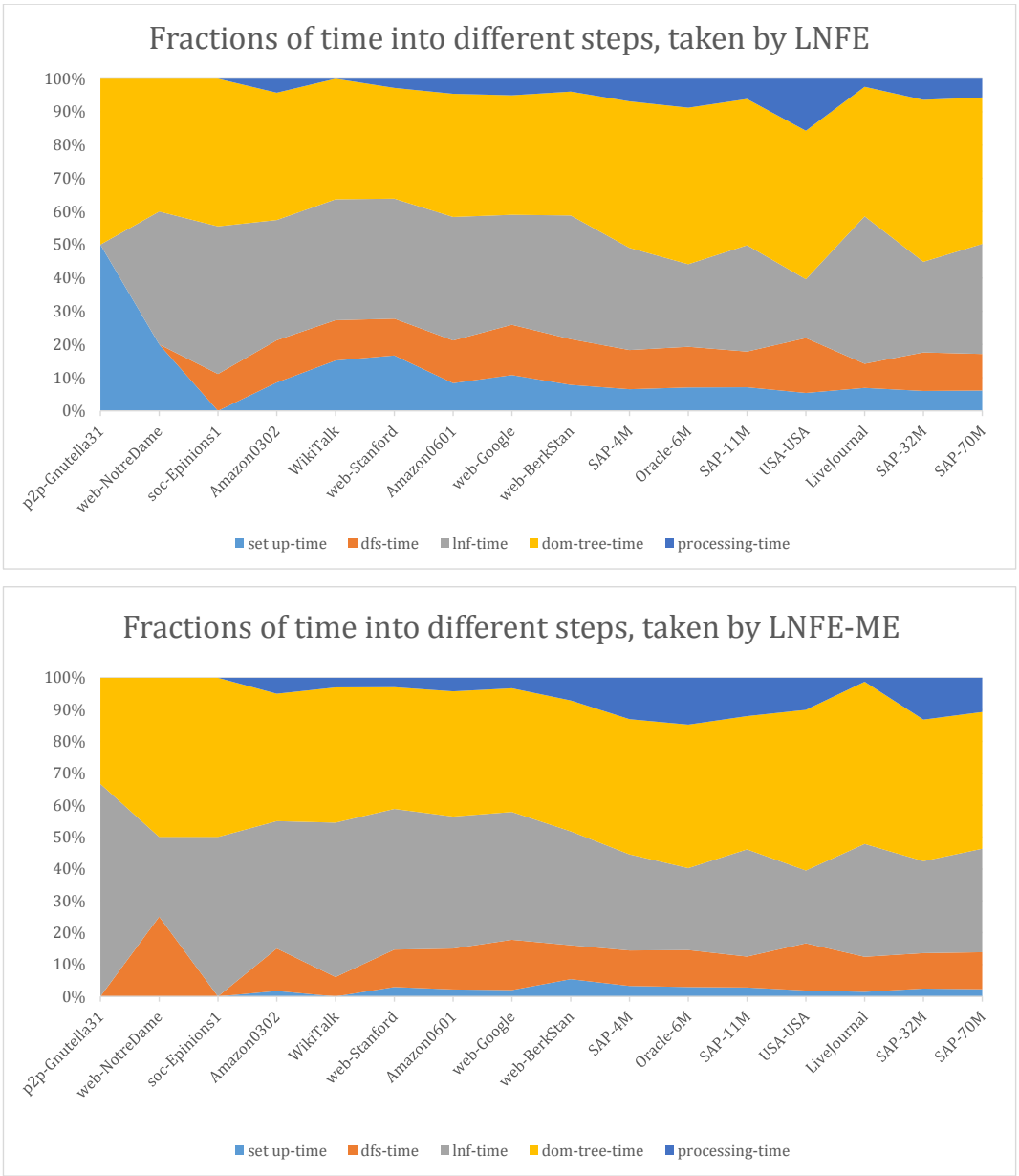


Figure 4.8: Fraction of running time in seconds, that has taken by LNFE(top) and LNFE-ME(bottom) in their different steps during the execution on real world graphs presented in Table 4.1 (Better viewed in color.)

Graphs ($n = 100$K)		Running times in seconds			Memory in MBytes		
Name	m	AUXE	LNFE	LNFE-ME	AUXE	LNFE	LNFE-ME
Rand-11D	1.1M	0.1672	0.2212	0.2396	91.1	22.0	16.8
Rand-17D	1.7M	0.2244	0.3120	0.3176	133.1	31.3	20.5
Rand-23D	2.3M	0.2756	0.3976	0.3796	175.0	40.6	27.5
Rand-33D	3.3M	0.3496	0.5196	0.4628	237.8	54.6	38.0
Rand-39D	3.9M	0.3944	0.5956	0.5100	279.8	63.1	45.0
Rand-45D	4.5M	0.4392	0.6740	0.5660	321.7	72.4	51.9
Rand-54D	5.4M	0.5080	0.7880	0.6420	384.5	86.4	62.4
Rand-60D	6.0M	0.5548	0.8600	0.6944	426.4	95.7	69.4
Rand-66D	6.6M	0.6024	0.9408	0.7460	468.3	105.0	76.4
Rand-75D	7.5M	0.6660	1.0480	0.8248	531.2	119.0	86.9
Rand-109D	10.9M	0.9196	1.4428	1.0904	761.7	170.2	125.3
Rand-145D	14.5M	1.1864	1.8668	1.3660	1013.2	226.1	167.2
Rand-182D	18.2M	1.4032	2.2976	1.6368	1228.8	282.0	209.1
Rand-216D	21.6M	1.6196	2.7040	1.8844	1536.0	333.2	247.5
Rand-252D	25.2M	1.8640	3.0916	2.1736	1740.8	389.1	289.4
Rand-286D	28.6M	2.0844	3.4860	2.4280	1945.6	440.3	327.8
Rand-322D	32.2M	2.3224	3.9120	2.7108	2252.8	496.2	369.7
Rand-359D	35.9M	2.6124	4.3604	2.9784	2457.6	552.0	411.6
Rand-393D	39.3M	2.8112	4.7652	3.2488	2662.4	603.3	450.1
Rand-429D	42.9M	3.0584	5.1968	3.5356	2969.6	659.2	492.0
Rand-466D	46.6M	3.3056	5.6772	3.8116	3174.4	715.0	533.9
Rand-499D	49.9M	3.5212	6.0600	4.1000	3481.6	766.3	572.3
Rand-536D	53.6M	3.7992	6.4604	4.3640	3686.4	822.1	614.2

Table 4.3: The characteristics of random graphs, where we keep fixed the number of vertices to 100K while we increase the edge density value.

AUXE-Nature, running time in seconds				
Graph ($n = 100$K)	Edges	AUXE(1)	AUXE(42)	AUXE(10K)
AUX-2.5D	250K	0.010	0.020	0.044
AUX-3D	300K	0.014	0.020	0.048
AUX-3.5D	350K	0.018	0.032	0.048
AUX-4D	400K	0.022	0.034	0.052
AUX-4.5D	450K	0.025	0.036	0.056
AUX-5D	500K	0.028	0.038	0.056
AUX-5.5D	550K	0.032	0.040	0.060
AUX-6D	600K	0.033	0.041	0.064
AUX-6.5D	650K	0.033	0.042	0.068
AUX-7D	700K	0.035	0.043	0.071
AUX-7.5D	750K	0.038	0.044	0.075
AUX-8D	800K	0.042	0.045	0.078
AUX-8.5D	850K	0.043	0.045	0.081
AUX-9D	900K	0.044	0.046	0.084

Table 4.4: Running times in seconds of the AUXE algorithm on the following type of random graphs. The number of vertices is fixed (100K) in all graphs and we marginally increase the density by 0.5, each time creating three graphs: (i) a graph that is 2-edge-connected (represented by the line AUXE(1)), (ii) a graph containing 44 2-edge-connected blocks with sizes in the range from 2 to $n/5$ (represented by the line AUXE(44)), and (iii) a graph containing $10K$ 2ECBs of equal size (represented by the line AUXE(10K)).

5 2-Vertex-Connected Blocks

5.1 Introduction

Let us recall the definition of *2-vertex-connected blocks* (2VCB) from Chapter 2. Given a digraph $G = (V, E)$, we say that two vertices $u, v \in V$ are 2-*vertex-connected*, and denote this relation by $u \leftrightarrow_{2v} v$, if there are two vertex-disjoint directed paths from u to v and two vertex-disjoint directed paths from v to u. Note that, a path from u to v and a path from v to u need not be *vertex-disjoint*. Menger's Theorem [121]* also states the

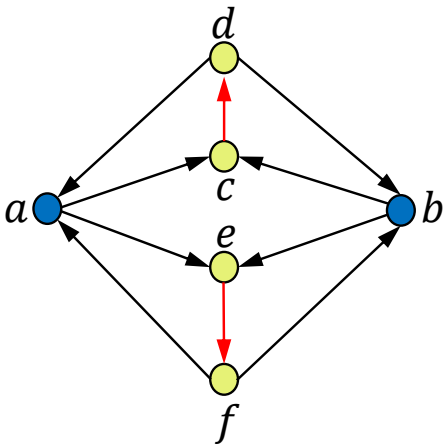


Figure 5.1: Example of *2-vertex-connected blocks*, vertices $\{a, b\}$ are in same 2VCB, but the paths from a to b ($\{a, c, d, b\}$,$\{a, e, f, b\}$) and b to a ($\{b, c, d, a\}$,$\{b, e, f, a\}$) share the vertices $\{c, d, e, f\}$, which are not in same 2VCB. (Better viewed in color.)

equivalent definition of 2-*vertex-connected*, two different vertices $v, w \in G$ are 2-*vertex-connected*, only if the removal of any vertex different from v and w leaves them in the same strongly connected component. But unlike the 2-*edge-connected* relation, the converse is not always true. It holds only if v and w are not adjacent to each other. Since

*To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

two mutually adjacent vertices are left in the same strongly connected component by the removal of any other vertex, but they are not 2-*vertex-connected*. A 2VCB of a digraph $G = (V,E)$ is defined as a maximal subset $B \subseteq V$ such that $\forall u,v \in B$, $u \leftrightarrow_{2v} v$. Thus, if u,v be the vertices of a 2VCB then, the paths that connect the vertices u and v may contain the vertices that do not belong to the same 2VCB as shown in Figure 5.1, where the vertices a and b are in the same 2VCB. But the paths from a to b ($\{a,c,d,b\}$,$\{a,e,f,b\}$) and b to a ($\{b,c,d,a\}$,$\{b,e,f,a\}$) share the vertices $\{c,d,e,f\}$ which do not belong to the same 2VCB.

Lemma 5.1.1. *Two different 2VCBs can have at most one vertex in common.*

Proof. Let us consider A and B be the two different 2VCBs of a digraph $G = (V,E)$. Let us assume on the contrary that $|A \cap B| = 2$ and $A \cap B = \{x,y\}$. Furthermore, let us suppose $u \in A \setminus \{x,y\}$ and $v \in B \setminus \{x,y\}$ as illustrated in Figure 5.2.

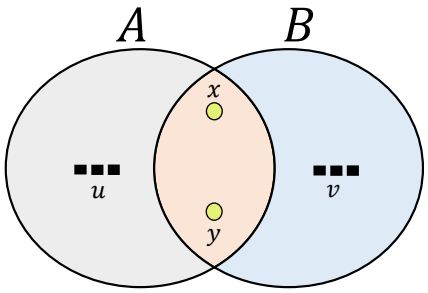


Figure 5.2: Two different 2VCBs A and B share a vertex w.

Since $u,x,y \in A$ implies that $u \leftrightarrow_{2v} x$, $u \leftrightarrow_{2v} y$ and $x \leftrightarrow_{2v} y$. Similarly, $v,x,y \in B$ implies that $x \leftrightarrow_{2v} v$ and $y \leftrightarrow_{2v} v$. Now, we have to show that $u \leftrightarrow_{2v} v$. Assume, for contradiction, that u and v are not 2-vertex-conneted. Then, there is a strong articulation point w such that every path from u to v contains w, or every path from v to u contains w (or both). Without loss of generality, suppose that w is contained in every path from u to v. Since x and y are distinct, we can assume that $w \neq x$. (If $w = x$ then we swap the role of x and y.) Then, $u \leftrightarrow_{2v} x$ implies that there is a path P from u to x that avoids w, and similarly, $x \leftrightarrow_{2v} v$ implies that there is a path Q from x to v that avoids w. So, P

followed by Q gives a path from u to v that does not contain w, a contradiction. Hence $u \leftrightarrow_{2v} v$.

□

Remark. This lemma help us to explain the algorithms in next section.

5.2 Related Work

In July 2014, Jaberi [89] proposed an algorithm to compute the 2VCBs of a digraph $G = (V,E)$ that has the $O(mn)$ time complexity. Later on, in the same year, Georgiadis et al. [72] presented two different algorithms for the 2VCB computation of a digraph G; Simpler with $O(mn)$ time complexity and Faster with linear time bound $O(m+n)$. Recently in 2015, Georgiadis et al. [73] again presented the algorithm to compute the 2VCB of a digraph in linear time by using *loop nesting tree* and *dominator tree* information. Furhtermore, we also modify the algorithm presented in [73] and present its memory efficient version. In particular, our algorithm boost the computation of loop nesting tree formation and uses the less memory for the dense graph that we already explained in Chapter 3 - Section 3.6.3.3 Thus, algorithms for 2VCBs computation were developed from 2014. To the best of our knowledge, only one previous experimental study is done between these algorithms by Di Luigi et al. [44]. But they did not include the linear time algorithm present in Georgiadis et al. [73] since this algorithm was available after their experimental observations. We perform the empirical analysis between the linear time algorithms presented in [72], [73], and modified version of [73]. Our experimental observation did not incorporate the algorithms of Jaberi [89] because of its extensive requirements in storage space. Moreover, we centered our research to observe the complexity of linear time algorithms rather than $O(mn)$ time algorithms. In the next sections, we will explain the high-level idea of algorithms and then report our experimental observations later.

5.3 Algorithms

Notations. Let us define the common notation that will be used by the 2VCB algorithms. Let G_s (resp., G_s^R) is the flow-graph of G (resp., G^R). Let vertex dominator trees and loop nesting tree of the flow-graph G_s (resp., G_s^R) is denoted by D (resp., D^R) and H (resp., H^R) respectively. Moreover, let $d(v)$ (resp., $d^R(v)$) denote the parent of $v \neq s$ in D (resp., D^R) and let $C(v)$ (resp., $C^R(v)$) denotes the set of children of a vertex v in D (resp., D^R). Similarly, let $\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$) represents the set of proper descendants of a vertex u in D (resp., D^R). Let T_u and $T(v)$ denote a tree rooted at u and a tree that contains a vertex v respectively. Also, given a tree T and $v \in V(T)$, we let $C_T(v)$ represents the set containing v and its children in T.

vertex resilient block. By Menger's Theorem [121][†], any two distinct vertices v and w are 2-*edge-connected* in digraph G if and only if the removal of any edge from G, v and w are still in same strongly connected component. However, for the 2-*vertex connectivity* is more complicated. Because any two adjecent vertices v and w are 2-*vertex-connected* if removal of any vertex different from v and w leaves them in same strongly connected component, while the converse holds only when v and w are not adjacent. To overcome this complication, Georgiadis et al. [72] defined a term *vertex-resilient* for the intermediary relation, denoted by $v \leftrightarrow_{\text{vr}} w$ such that any two vertices v and w are said to be vertex-resilient if the removal of any vertex different from v and w leaves v and w in the same strongly connected component. Hence, *vertex-resilient block* (VRB) of a digraph $G = (V,E)$ is defined as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{\text{vr}} v$ for all $u, v \in B$ as illustrated in Figure 5.3. Moreover, as a special case, if $|B| = 1$, it is considered as a trivial vertex-resilient block. However, we will not consider such trivial B in our 2VCB computation. Therefore, in terms of vertex resilient block, if the two vertices v and w are not adjacent then $v \leftrightarrow_{2\text{v}} w$ if and only if $v \leftrightarrow_{\text{vr}} w$. Thus, two vertices v and w that are vertex-resilient need not be necessarily 2-*vertex-connected*. VRB also has the many other properties defined in [72]; few of them are explained

[†]To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

through the following lemmas (proofs are available in [72]).

Lemma 5.3.1. *The number of vertex-resilient blocks in a digraph G is at most $n-1$.*

Lemma 5.3.2. *The total number of vertices in all vertex-resilient blocks is at most $2n-2$.*

The following lemma help us to compute the 2VCBs by using 2ECB and VRB.

Lemma 5.3.3. ([72]) *For any two distinct vertices v and w, $v \leftrightarrow_{2v} w \iff v \leftrightarrow_{vr} w$ and $v \leftrightarrow_{2e} w$.*

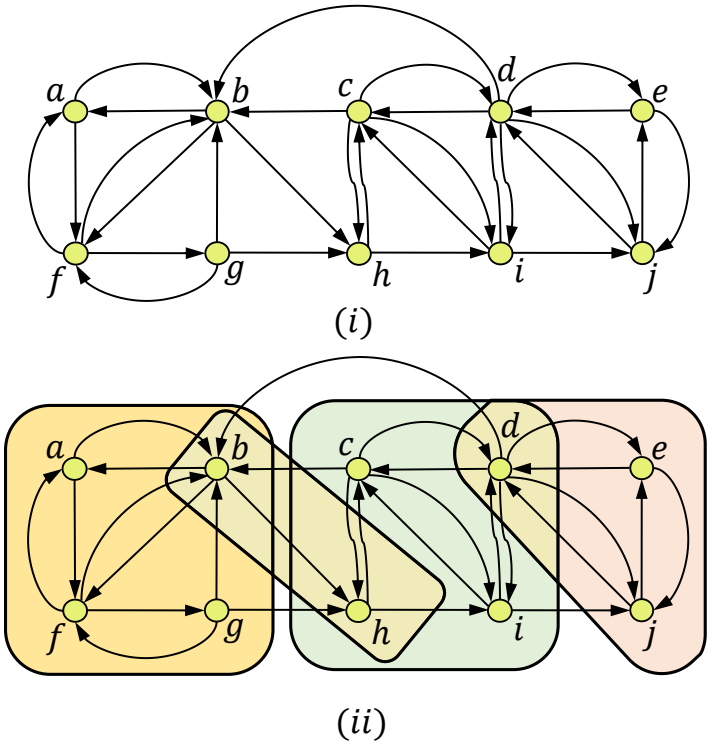


Figure 5.3: (i) Given a digraph $G=(V,E)$, and (ii) *vertex resilent blocks* of G.

VRB to 2VCB When we will have the VRB then we can compute the 2VCB by using Lemma 5.3.3 as follows. We have the VRBs $\mathscr{B}$ and the 2ECBs $\mathscr{S}$ of $G=(V,E)$, then we can simply execute the $refine(\mathscr{B},S)$ to get the 2VCB. According to Leema 5.3.4, this process will take the $O(n)$ time.

5.3.1 Simpler Algorithm

This algorithm is an immediate application of the characterization of the VRB in terms of strong articulation points. The details steps are shown in Algorithm 11, which is taken from [72]. We say that, any vertex $x \in V$ is a strong articulation point if its removal separates the two distinct vertices u and v (i.e., u and v belong to different strongly connected components of $G \setminus x$). Thus, we can compute the vertex-resilient blocks by computing the strongly connected components of $G \setminus x$ for every strong articulation point x. Algorithm 11 defines an operation that refines the currently computed blocks as following. Let $\mathscr{B}$ be a set of blocks, let $\mathscr{S}$ be a partition of a set $U \subseteq V$, and let x be a vertex not in U then

***refine*$(\mathscr{B},\mathscr{S},x)$:** for each block $B \in \mathscr{B}$, substitute B by the sets $B \cap (S \cup \{x\})$ of size at least two, for all $S \in \mathscr{S}$.

Algorithm 11: SIMPLE-VRB

Input: A strongly connected digraph $G = (V,E)$
Output: vertex resilent blocks of G

1 Compute the strong articulation points of G.
2 Initialize the current set of blocks as $\mathscr{B} = \{V\}$. (Start from the trivial set containing only one block.)
3 **foreach** *strong articulation* x **do**
4 Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus x$
5 Execute $refine(\mathscr{B},\mathscr{S},x)$.

Then following two lemmas will explain why and how the Algorithm SIMPLE-VRB takes $O(mn)$ time in worse case scenario, proofs are available in Georgiadis et al. [72].

Lemma 5.3.4. *Let N be the total number of elements in all sets of $\mathscr{B}$ ($N = \sum_{B \in \mathscr{B}} |B|$), and let K be the number of elements in U. Then, the operation $refine(\mathscr{B},S,x)$ can be executed in $O(N+K)$ time (Georgiadis et al. [72]).*

Lemma 5.3.5. *Algorithm 11 runs in $O(mp*)$ time, where $p*$ is the number of strong articulation points of G. This is $O(mn)$ in the worst case (Georgiadis et al. [72]).*

Proof. The strong articulation points of G can be obtained in linear time (Italiano et al. [88]). For each articulation point x, we can compute the strongly connected components of $G \setminus x$ in linear time (Tarjan [154]). When we get the i^{th} strongly connected component (S_i) in $G \setminus x$, then assign label $i(i \in \{1,\ldots,n\})$ to the vertices in S_i. The total number of blocks (including non-trivial as well) cannot decrease during the iteration and $\mathscr{B}$ contains at most $n-1$ blocks (By lemma 5.3.1). It maintains the vertex resilient blocks such that any two distinct blocks in $\mathscr{B}$ have at most one element in common, and that the corresponding block graph is a forest. Therefore, by Lemma 5.3.1, the total number of elements in all blocks is at most $2n-2$. The iteration steps of each strong articulation point takes $O(n)$ time (Lemma 5.3.4). This yields that if there are p strong articulation points, then the desired running time would be $O(mp)$. Since there can be at most n strong articulation points, the algorithm needs to run up to $O(mn)$ time in worst case scenario. □

Lemma 5.3.6. *Let v and w be any vertices of G. Then $v \leftrightarrow_{2v} w$ only if v and w are siblings or one is the parent of the other in both D and D^R (Georgiadis et al. [72]).*

5.3.2 Linear Time Algorithm Through Auxiliary Graph

We adapt this algorithm from [72] and refer as AUXV. It computes the 2-vertex-connected blocks of G in linear time by using Lemma 5.3.6 and auxiliary graphs G_r^{2v}. We note that, AUXV is more complicated than AUXE (which computes the 2ECBs, explained in Chapter 4), due to the fact that: unlike the 2ECBs, 2VCBs do not form a partition of V. In 2ECB computation, auxiliary graph method uses the *canonical decomposition* that maintain the original 2ECB of G and approximate the blocks [71]. Because any vertex in an auxiliary graph G_r is reachable from a vertex outside G_r only through a single *strong bridge*. Whereas in 2VCBs computation [72], it will not have that property because the graph is decomposed according to strong articulation points. Therefore, it needs to

Algorithm 12: AUXV

Input: A strongly connected digraph $G = (V, E)$
Output: vertex resilent blocks of G

// Step 1:
Choose an arbitrary vertex $s \in V$ as a start vertex.
Compute the dominator tree D.
For any vertex v, let $C(v)$ be the set containing v and the children of v in D. For every vertex v that is not a leaf in D, associate block $C(v)$ with every vertex $w \in C(v)$.
// Step 2:
Compute the auxiliary graphs G_r for all vertices r that are not leaves in D.
// Step 3: Process the vertices of D in bottom-up order
foreach auxiliary graph $H = G_r$ with r not a leaf in D **do**
    Compute the dominator tree $T = D_R^H$.
    Compute the set $\mathscr{B}$ of blocks that contain vertices in $C(r)$.
    foreach $B \in \mathscr{B}$ **do**
        execute $split(B, T)$.
    Compute the auxiliary graphs H_q^R for all vertices q that are not leaves in T.
    foreach auxiliary graph H_q^R with q not a leaf **do**
        Compute the set $\mathscr{B}_q$ of blocks that contain at least two ordinary vertices in H_q^R.
        Compute the set $\mathscr{S}$ of the strongly connected components of $H_q^R \setminus q$.
        Refine the blocks in $\mathscr{S}_q$ by executing $refine(\mathscr{B}_q, \mathscr{S}, q)$.

maintain the more complicated forest representation and sophisticated auxiliary graph than 2ECB computation.

For 2-vertex connectivity, we define an auxiliary graph G_r^{2v}, for each vertex r that is not a leaf in D. Let us recall, a vertex $v \neq s$ is a strong articulation point in G if and only if it is not a leaf in $D \cup D^R$ [88]. Let $C^k(r)$ denote the level k descendants of r (i.e., $C^0(r) = \{r\}$, $C^1(r) = C(r)$, etc). We build G_r^{2v} as follows. The vertex set of G_r is $\cup_{k=0}^{3} C^k(r)$ and it is partitioned into a set of *ordinary* vertices $C^1(r) \cup C^2(r)$ and a set of *auxiliary* vertices $C^0(r) \cup C^3(r)$. Then G_r^{2v} results from G by contracting all vertices that are not descendants of r in D into r, and contracting all descendants of each $w \in C^3(r)$ into w.

In AUXV computation, current blocks are refined through its execution, and we

maintain them in a *block forest* data structure $F = (V_F, E_F)$. The node set V_F contains a node for each vertex $u \in V$, and a block node v_B for each block B. The edge set E_F contains an edge (u, v_B) if vertex $u \in V$ is in block B. Algorithm AUXV initializes F by creating one block for each set $C(v) \cup \{v\}$ of cardinality at least two. Next, it computes the first-level auxiliary graphs $J_r = G_r^{2v}$ of G. Then, it uses the dominator tree of each J_r^R in order to construct the second-level auxiliary graphs and refines the blocks in F according to these dominator trees using Lemma 5.3.6. After the removal of a particular vertex in each second level auxiliary graph, and then by refining F according to the block partition induced by those strongly connected components, final 2-vertex-connected blocks will be formed.

***split*(C,T):** Return the set that consists of the blocks $B \cap C_T(v)$ of size at least two, for all $v \in V(T)$.

***Note*:** if $|V(T)| = N$, then split operation executed in $O(N)$ time [72].

5.3.3 Linear Time Algorithm Through Loop Nesting Tree and Dominator Tree

This algorithm uses the loop nesting trees and dominator trees. We take an algorithm from [73] and refer as LNFV. The detailed steps are shown in Algorithm 13.

Algorithm LNFV compute the 2VCBs in linear-time. It uses the similar approach of LNFE, which computes the 2ECBs in linear-time explained in Chapter 4. The main difference is: instead of computing vertex labels as like in LNFE, we maintain the same data structure while refining the maintained blocks as in AUXV. The blocks are refined with respect to the loops in G_s and G_s^R. Recall that, for any vertex v, $C(v)$ (resp., $C^R(v)$) represents the set of children of v in D (resp., D^R). For any pair of vertices u and v we let $C(u,v) = (C(u) \cup \{u\}) \cap (C^R(v) \cup \{v\})$. That is, set $C(u,v)$ contains all vertices in $C(u) \cap C^R(v)$. Also, if $u = v$ or $u \in C^R(v)$ then $u \in C(u,v)$, and if $v \in C(u)$ then $v \in C(u,v)$. We can compute all nonempty $C(u,v)$ sets in $O(n)$ time [44]. The following lemma is an immediate consequence of Lemma 5.3.6.

Algorithm 13: LNFV

Input: A strongly connected digraph $G = (V, E)$
Output: The Vertex resilent blocks of G

/* Initialization */
Compute the reverse digraph G^R. Select an arbitrary start vertex $s \in V$.
Compute the dominator trees D and D^R of the flow graphs G^s and G^R_s, respectively.
Compute the loop nesting trees H and H^R of the flow graphs G^s and G^R_s, respectively.
/* Initialize block forest */
Compute the sets $c(u,v)$ for any pair of vertices u and v.
Initialize the block forest F to contain one block for each set $c(u,v)$ with at least two vertices.
foreach $u \in N \cup \{s\}$ in a *bottom-up order* of D **do** /* Forward direction: */
    Find the set of blocks $\mathscr{B}$ that contain at least two vertices in $c(u) \cup \{u\}$
    Compute the collection of vertex subsets
    $\mathscr{S} = \{H(v) \cap c(u) : h(v) \notin \widetilde{D}(u) \wedge v \in c(u)\}$
    Execute *refine*$(\mathscr{B}, \mathscr{S}, u)$
    if $u \neq s$ **then**
        foreach $B \in \mathscr{B}$ *such that* $u \in B$ **do**
            Choose an arbitrary vertex $v \neq u$ in B
            Compute the nearest common ancestor w of u and v in H
            if $w \notin c(d(u))$ **then**
                Set $B = B \setminus u$
                if $|B| = 1$ **then**
                    delete B from F
foreach $u \in N^R \cup \{s\}$ in a *bottom-up order* of D^R **do** /* Reverse direction */
    Find the set of blocks $\mathscr{B}$ that contain at least two vertices in $c^R(u) \cup \{u\}$
    Compute the collection of vertex subsets
    $\mathscr{S} = \{H^R(v) \cap c^R(u) : h^R(v) \notin \widetilde{D}^R(u) \wedge v \in c^R(u)\}$
    Execute *refine*$(\mathscr{B}, \mathscr{S}, u)$
    if $u \neq s$ **then**
        foreach $B \in \mathscr{B}$ *such that* $u \in B$ **do**
            Choose an arbitrary vertex $v \neq u$ in B
            Compute the nearest common ancestor w^R of u and v in H^R
            if $w^R \notin c^R(d^R(u))$ **then**
                Set $B = B \setminus u$
                if $|B| = 1$ **then**
                    delete B from F

Lemma 5.3.7. *Let $G = (V,E)$ be a strongly connected digraph, and let $s \in V(G)$ be an arbitrary start vertex. Any two vertices x and y are vertex-resilient only if they are located in a common set $C(u,v)$ (Georgiadis et al. [73]).*

According to the Lemma 5.3.7, each vertex-resilient block is contained in a $C(u,v)$ set. Therefore, set $C(u,v)$ defines a "coarse" block forest. The LNFV algorithm refine the $C(u,v)$ by using the loop nesting trees H and H^R with the help of Theorem 5.3.8. Actually, $C(u,v)$ can also be represented by a block forest of size $O(n)$ because as we already know in AUXV algorithm, these sets can be constructed by applying the *split* operation to the sets $C(v) \cup \{v\}$, for each vertex v that is not a leaf in D.

Theorem 5.3.8. ([73, page 38]) *Let u be a strong articulation point of G, and let s be an arbitrary vertex in G. Let C be a strongly connected component of $G \setminus u$ and $\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$) be the set of proper descendants of u in D (resp., D^R). Then one of the following cases holds:*

(a) If u is a nontrivial dominator in G_s but not in G_s^R then either $C \subseteq \widetilde{D}(u)$ or $C = V \setminus D(u)$.

(b) If u is a nontrivial dominator in G_s^R but not in G_s then either $C \subseteq \widetilde{D}^R(u)$ or $C = V \setminus D^R(u)$.

(c) If u is a common nontrivial dominator of G_s and G_s^R then either $C \subseteq \widetilde{D}(u) \setminus \widetilde{D}^R(u)$, or $C \subseteq \widetilde{D}^R(u) \setminus \widetilde{D}(u)$, or $C \subseteq \widetilde{D}(u) \cap \widetilde{D}^R(u)$, or $C = V \setminus (D(u) \cup D^R(u))$.

(d) If $u = s$ then $C \subseteq \widetilde{D}(u)$.

Moreover, if $C \subseteq \widetilde{D}(u)$ (resp., $C \subseteq \widetilde{D}^R(u)$) then $C = H(w)$ (resp., $C = H^R(w)$) where w is a vertex in $\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$) such that $h(w) \notin \widetilde{D}(u)$ (resp., $h^R(w) \notin \widetilde{D}^R(u)$).

The detailed steps of LNFV is in Algorithm 13, which is taken from [73]. After defining the initial blocks, it performs the "forward pass" which processes D and H. During the "forward pass", it visits the non-leaf vertices of D in bottom-up order. For each such vertex u, it computes a partition $\mathscr{S}$ of $C(u)$, such that each set $S \in \mathscr{S}$ contains a subset of children of u in D that are strongly connected in $G \setminus u$. Finally, it need to

find the blocks that may contain vertices that are vertex-resilient with u. After the completion of this forward pass, it executes the "reverse pass" that performs the identical computations in D^R and H^R.

5.3.4 Memory Efficient Version

As in LNFE algorithm of 2ECB computation, we also modify the LNFV algorithm during its *loop nesting tree* computation by using our techniques which we explained in Chapter 3 - Section 3.6.3.3. During the loop nesting tree computation, loops are contracted into a single vertex. Therefore, we can improve the memory usage by excluding the unnecessary or parallel edge insertions in the dynamic lists. We changed the implementation of LNFV by this concept and refer as LNFV-ME.

5.4 Empirical Analysis

We perform the experimental observations between the algorithms AUXV, LNFV and LNFV-ME. With reference to the loop nesting tree base algorithms, as in 2ECB, we implemented the streamline version LNFV and its memory-efficient version LNFV-ME. The development framework, hardware configuration of a testing machine, testing datasets, and experimental paradigms are completely identical with 2ECB computation. In terms of 2VCB, the characteristics of the graphs in our dataset are described in detail in Table 5.1.

Implementation Issues: As in LNFE and LNFE-ME algorithms of 2ECB compuatation, during the LNFV and LNFV-ME computation, *loop nesting tree* and *dominator tree* use the DFS tree. Therefore, we use the single DFS tree for their combine computation that allows us to deallocate the adjacency list of the forward graph after the formation of the dominator tree (since loop nesting tree is formed during the DFS formation and then compute the dominator tree). Furthermore, we reverse the graph by swapping the roles of the forward and the reverse adjacency lists rather than to make a whole new copy. We considered the same real-world graphs that we used in our 2ECB computation. In terms

Graphs				2-vertex-connected blocks		
Name	Type	$n = \|V(G)\|$	$m = \|E(G)\|$	Max-size	Avg-size	Total #
p2p-Gnutella31	P2P	14.1K	50.9K	7.9K	4.3	3.5K
web-NotreDame	WG	54.0K	296.2K	6.2K	3.0	22K
soc-Epinions1	SN	32.2K	443.5K	17.6K	3.4	12.6K
Amazon0302	PCP	241.8K	1.1M	123.6K	4.1	74.8K
WikiTalk	SN	111.9K	1.5M	50.2K	2.9	53.6K
web-Stanford	WG	150.5K	1.6M	26.2K	3.9	40.8K
Amazon0601	PCP	395.2K	3.3M	287.6K	6.0	78.1K
web-Google	WG	434.8K	3.4M	151.4K	3.5	149.3K
web-BerkStan	WG	334.9K	4.5M	64K	3.4	109.1K
SAP-4M	MP	4.1M	11.9M	119.7K	2.6	271.9K
Oracle-6M	MP	6.4M	15.9M	283K	2.5	1471.6K
SAP-11M	MP	11.1M	36.4M	640.9K	3.0	752.3K
USA-USA	RN	23.9M	57.7M	16M	4.24	7.4K
LiveJournal	SN	3.8M	65.3M	2.9M	5.4	862.5K
SAP-32M	MP	32.3M	81.8M	197.4K	3.0	478.4K
SAP-70M	MP	69.8M	214.9M	947.5K	2.3	6.9M

Table 5.1: The characteristics of the real-world graphs that we considered; n and m refers to the number of vertices and the number of edges, respectively.The graph types are encoded as follows: road network (RN), peer to peer (P2P), web graph (WG), social network (SN), production co-purchase (PCP), memory profiling (MP).The graphs are sorted in increasing order according to their number of edges.Additionally, we report the statistics of their 2-vertex-connected blocks, whose size refers to the number of their vertices.

of the 2VCB, their characteristics are summarized in Table 5.1. Furthermore, the same type of random graphs that were created in 2ECB computation are also used here to observe the specific properties and base performance of the algorithms. All the running times reported in our experiments were averaged over ten different executions.

We start the experimental observation of 2VCB by applying the algorithms that we discussed before, AUXV, LNFV and LNFV-ME. In beginning, we used these algorithms

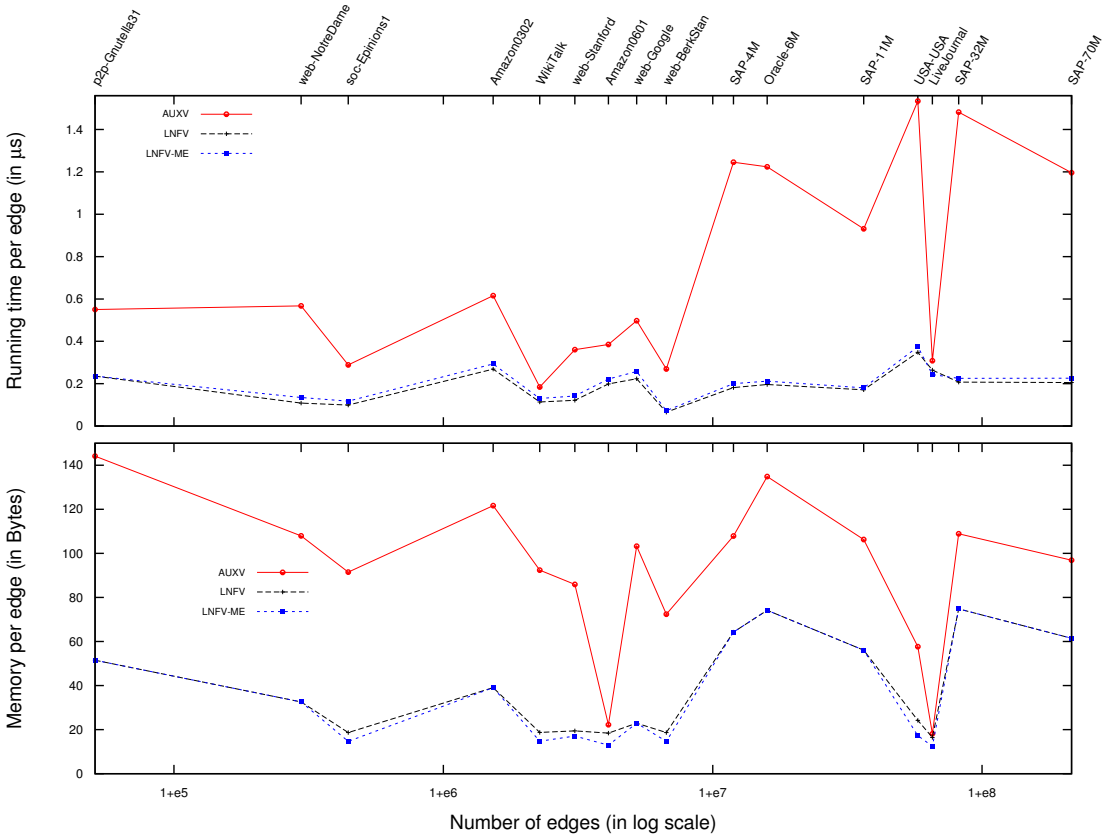


Figure 5.4: Running times per edge in μs (top) and Memory usage per edge in Bytes (bottom) of the algorithms AUXV, LNFV and LNFV-ME on the real world graph datasets presented in Table 5.1. (Better viewed in color.)

over the datasets presented in Table 5.1 and compare their running time. The output of the experiments are reported in Table 5.2 and plotted by the Figure 5.4 (top). We observe that the algorithm LNFV is faster than AUXV by a factor of 5 on average. Also, on average, LNFV is 6.5% faster than LNFV-ME. The result is expected for sparse input graphs, because even if the graph is sparse, LNFV-ME algorithm does not avoid the insertion of many parallel edges in the dynamic lists used by the loop nesting tree computations. To do so, it has to use two more additional arrays which are necessary for filtering all the edges and takes the time.

Moreover, we notice that both LNFV and LNFV-ME are much better than AUXV, in a sense that their running times are less sensitive to the graph structure. On the other hand, the performance of AUXV is more dependent on the graph structure, which affects the

Graphs	Running times in seconds			Memory consumption in MBytes		
	AUXV	LNFV	LNFV-ME	AUXV	LNFV	LNFV-ME
p2p-Gnutella31	0.028	0.012	0.012	7.0	2.5	2.5
web-NotreDame	0.168	0.032	0.040	30.5	9.2	9.2
soc-Epinions1	0.128	0.044	0.052	38.7	7.9	6.2
Amazon0302	0.696	0.304	0.332	131.3	42.1	42.1
WikiTalk	0.272	0.168	0.192	130.2	26.4	20.8
web-Stanford	0.568	0.192	0.224	129.2	29.2	25.7
Amazon0601	1.272	0.656	0.732	70.0	57.9	40.8
web-Google	1.700	0.764	0.880	336.8	74.7	74.7
web-BerkStan	1.220	0.296	0.328	312.2	80.5	63.3
SAP-4M	14.872	2.168	2.388	1228.8	731.6	731.6
Oracle-6M	19.488	3.124	3.372	2048.0	1126.4	1126.4
SAP-11M	33.868	6.212	6.552	3686.4	1945.6	1945.6
USA-USA	88.548	20.076	21.628	3174.4	1331.2	951.0
LiveJournal	20.156	17.264	15.900	1126.4	1024.0	777.1
SAP-32M	121.296	16.956	18.456	8499.2	5836.8	5836.8
SAP-70M	256.992	44.136	48.556	19865.6	12595.2	12595.2

Table 5.2: Running times in seconds and Memory consumption in MBytes respectively of the algorithms for computing the 2-edge-connected blocks executed on the real world graphs of Table 5.1

number and size of the auxiliary graphs. More precisely, the AUXV algorithm is favored in graphs with few 2-vertex-connected blocks because it creates less number auxiliary graphs (no matter how big they are). To scrutinize this behavior of AUXV, we applied the AUXV over the following type of artificial graphs. For a fixed number of vertices ($100K$), we marginally increase the density by 0.5 each time creating three different types of graphs: (i) a graph that is 2-vertex-connected (i.e. itself a 2VCB), (ii) a graph containing 44 unequal 2-vertex-connected blocks (sizes in the range from 2 to $n/5$), and (iii) a graph containing $10K$ blocks of equal size. The result of this experiment

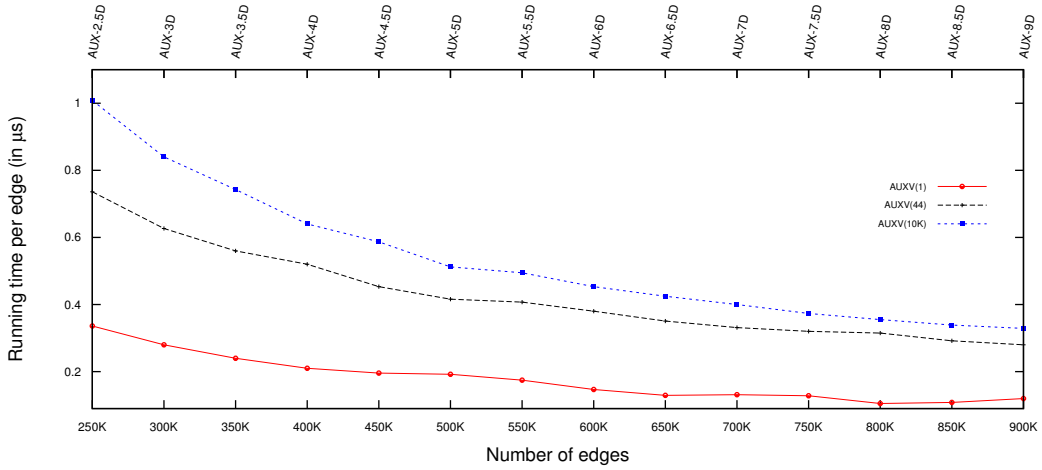


Figure 5.5: Running times in seconds of the AUXV algorithm on the following type of random graphs. The number of vertices is fixed (100K) in all graphs and we marginally increase the density by 0.5, each time creating three graphs: (i) a graph that is 2-vertex-connected (represented by the line AUXV(1)), (ii) a graph containing 44 2-vertex-connected blocks with sizes in the range from 2 to $n/5$ (represented by the line AUXV(44)), and (iii) a graph containing $10K$ 2VCBs of equal size (represented by the line AUXV(10K)). The data are presented in Table 5.3. (Better viewed in color.)

is presented in Table 5.3 and plotted by the Figure 5.5. We can easily notice that the running time of AUXV is consistently larger in the case of type (iii) as compared to type (i) graphs, which verifies that the existence of many 2-vertex-connected blocks (of equal or unequal size) slows down the algorithm. Additionally, on the type (ii) graphs the running time is in between the other two cases and gets closer to type (i) graphs as the density increases.

Next, we analyze the memory consumption of the algorithms AUXV, LNFV and LNFV-ME. Figure 5.4 (bottom) plots the memory usage of the algorithms, reported in Table 5.2. The plots in Figure 5.4 clearly shows that the LNFV and LNFV-ME require significantly less memory for all input graphs than AUXV. On average, LNFV-ME uses about 2.81 times less memory than AUXV, and improves the memory consumption of LNFV by about 9%.

Since all the real-world graphs that we consider are sparse, so we made the additional experiments to compare the performance of the algorithms AUXV, LNFV and

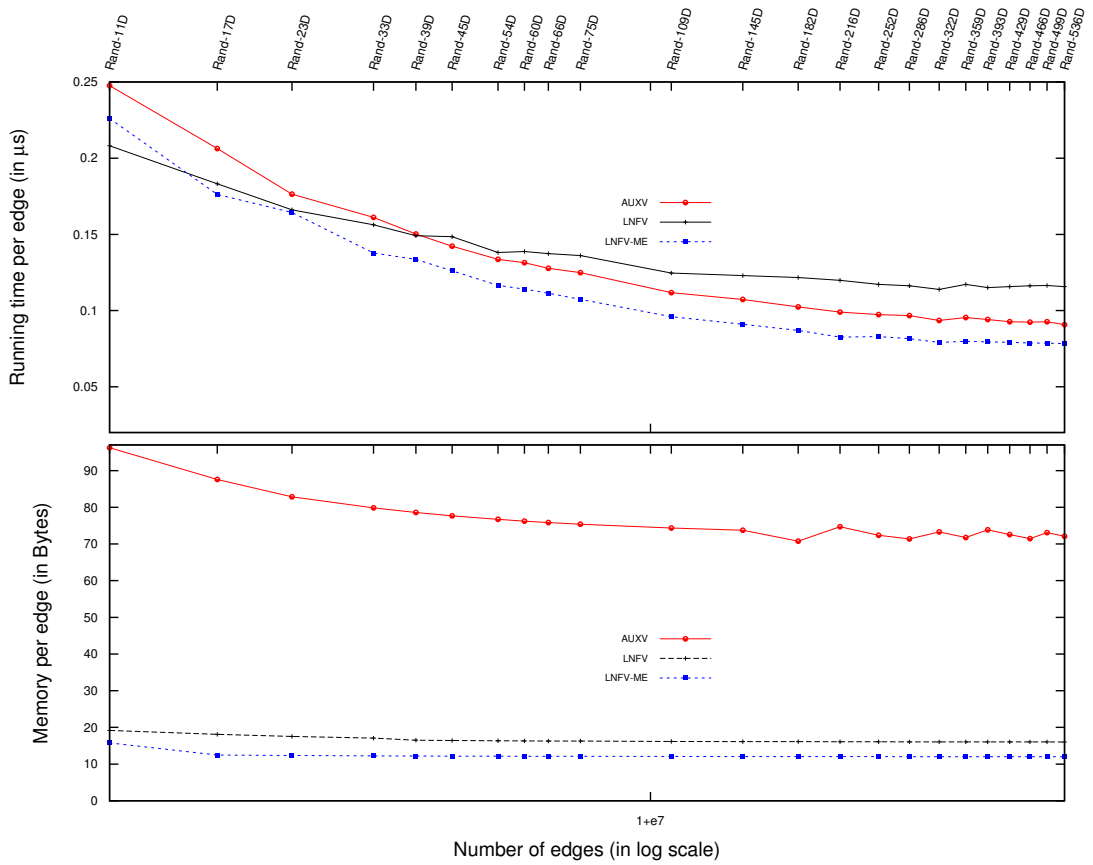


Figure 5.6: Running times per edge in μs (top) and memory usage per edge in Bytes (bottom) of the algorithms AUXV, LNFV and LNFV-ME on the random graphs summarized in Table 5.4. (Better viewed in color.)

LNFV-ME on dense graphs in order to highlight the full potential of the LNFV-ME algorithm. As like in 2ECB computation, we considered the same random graphs with a fixed number of vertices (100K) and density in a range that spans from 11 to 536. The results are plotted in Figure 5.6 (top) (see also the Table 5.4). We observed that LNFV gradually loses the advantage over AUXV, and it even becomes significantly slower for very dense graphs. The bottleneck in the case of dense graphs is the loop nesting tree computation since it introduces many memory writes when it inserts edges into dynamic lists. On the other hand, LNFV-ME was designed to filter many unnecessary insertions in the dynamic lists maintained by the loop nesting tree algorithm, and hence, it performs consistently faster than both AUXV and LNFV.

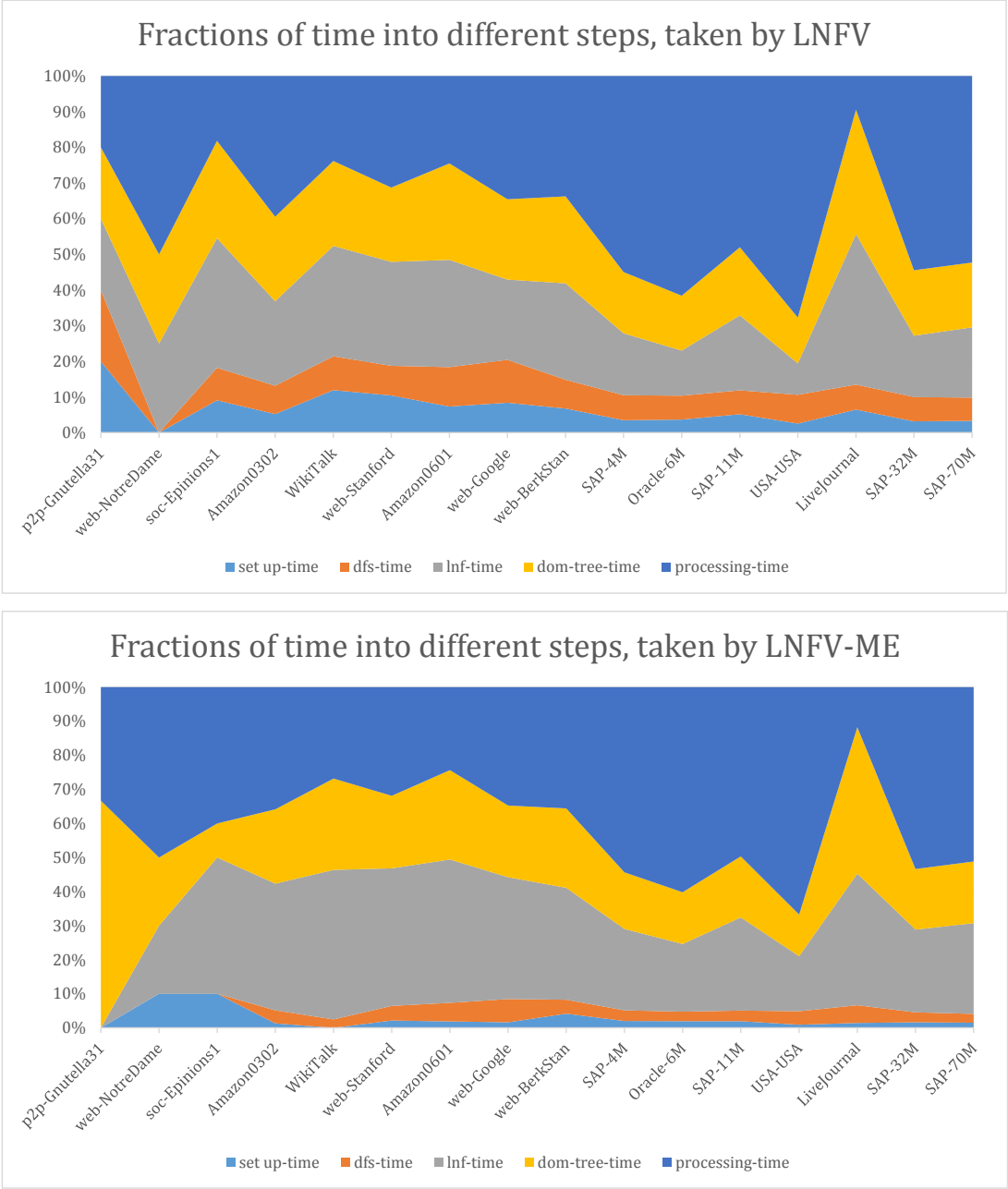


Figure 5.7: Fraction of running time in seconds, that has taken by LNFV(top) and LNFV-ME(bottom) in their different steps during the execution on real world graphs presented in Table 5.1 (Better viewed in color.)

We also analyze the memory consumption of these algorithms for such dense graph computation. Figure 5.6 (bottom) plots the memory usage, reported in Table 5.4. As we can notice that the AUXV requires much more memory than the algorithm that uses the loop nesting tree (in this case by a factor of 3.12). The LNFV-ME algorithm pays off the memory consumption over LNFV by about 30%, on average.

In addition, both the algorithms LNFV and LNFV-ME use the same steps to compute the 2-vertex-connected blocks. We analyzed the time that has taken by these algorithms in different steps as following (*i*) set up time (i.e., time to read the graph and create the adjency list), (*ii*) dfs-time (i.e., time to create the DFS), (*iii*) lnf-time (time to create the loop nesting forest, even though it created with DFS simultaneously, we separately measure its time), (*iv*) dom-tree time (time to create the dominator tree) and (*v*) processing time (to create the 2VCBs). Figure 5.7 represents the graphical representations of these time fractions.

Analysis the result with 2ECB Let take the algorithm LNFE and AUXE from 2ECB computation explain in Chapter 4, and LNFV and AUXV from 2VCB computation. We can heed that the gap between the LNFE and AUXE is smaller as compared to the gap between the LNFV and AUXV. Similarly, as like in 2-edge-connected blocks, in the 2-vertex-connected blocks also the algorithm based on loop nesting trees, LNFV, achieves overall the best performance. This result verified the fact that the auxiliary graphs that are created by AUXE are less complicated than the auxiliary graphs created by AUXV. Moreover, in 2ECB computation, auxiliary graphs need less memory than in 2VCB computation. Furthermore, the time fractions plots of 2ECB and 2VCB computations clearly show that the processing time for 2VCB computation is much higher than then processing time of 2ECB computation for all algorithms (LNFV vs LNFE and LNFV-ME vs LNFE-ME).

AUXV-Nature, running time in seconds				
Graph ($n = 100$K)	Edges	AUXV(1)	AUXV(42)	AUXV(10K)
AUX-2.5D	250K	0.084	0.184	0.252
AUX-3D	300K	0.084	0.188	0.252
AUX-3.5D	350K	0.084	0.196	0.260
AUX-4D	400K	0.084	0.208	0.256
AUX-4.5D	450K	0.088	0.204	0.264
AUX-5D	500K	0.096	0.208	0.256
AUX-5.5D	550K	0.096	0.224	0.272
AUX-6D	600K	0.088	0.228	0.272
AUX-6.5D	650K	0.084	0.228	0.276
AUX-7D	700K	0.092	0.232	0.280
AUX-7.5D	750K	0.096	0.240	0.280
AUX-8D	800K	0.084	0.252	0.284
AUX-8.5D	850K	0.092	0.248	0.288
AUX-9D	900K	0.108	0.252	0.296

Table 5.3: Running times in seconds of the AUXV algorithm on the following type of random graphs. The number of vertices is fixed (100K) in all graphs and we marginally increase the density by 0.5, each time creating three graphs: (i) a graph that is 2-vertex-connected (represented by the line AUXV(1)), (ii) a graph containing 44 2-vertex-connected blocks with sizes in the range from 2 to $n/5$ (represented by the line AUXV(44)), and (iii) a graph containing $10K$ 2VCBs of equal size (represented by the line AUXV(10K)).

Graphs ($n = 100$K)		Running times in seconds			Memory in MBytes		
Name	m	AUXV	LNFV	LNFV-ME	AUXV	LNFV	LNFV-ME
Rand-11D	1.1M	0.276	0.232	0.252	102.3	20.4	16.8
Rand-17D	1.7M	0.356	0.316	0.304	144.1	29.8	20.5
Rand-23D	2.3M	0.412	0.388	0.384	184.6	39.1	27.5
Rand-33D	3.3M	0.524	0.508	0.448	247.5	53.0	38.0
Rand-39D	3.9M	0.580	0.576	0.516	289.4	60.8	45.0
Rand-45D	4.5M	0.636	0.664	0.564	331.3	70.1	51.9
Rand-54D	5.4M	0.720	0.744	0.628	394.1	84.1	62.4
Rand-60D	6.0M	0.788	0.832	0.684	436.0	93.4	69.4
Rand-66D	6.6M	0.844	0.908	0.736	478.0	102.7	76.4
Rand-75D	7.5M	0.940	1.024	0.808	540.8	116.7	86.9
Rand-109D	10.9M	1.216	1.356	1.044	771.4	167.9	125.3
Rand-145D	14.5M	1.560	1.788	1.324	1022.8	223.8	167.2
Rand-182D	18.2M	1.864	2.216	1.584	1228.8	279.7	209.1
Rand-216D	21.6M	2.136	2.584	1.780	1536.0	330.9	247.5
Rand-252D	25.2M	2.456	2.956	2.092	1740.8	386.8	289.4
Rand-286D	28.6M	2.764	3.324	2.332	1945.6	438.0	327.8
Rand-322D	32.2M	3.016	3.672	2.552	2252.8	493.9	369.7
Rand-359D	35.9M	3.428	4.208	2.868	2457.6	549.8	411.6
Rand-393D	39.3M	3.696	4.520	3.120	2764.8	601.0	450.1
Rand-429D	42.9M	3.980	4.968	3.400	2969.6	656.9	492.0
Rand-466D	46.6M	4.304	5.416	3.668	3174.4	712.7	533.9
Rand-499D	49.9M	4.628	5.816	3.928	3481.6	764.0	572.3
Rand-536D	53.6M	4.864	6.204	4.200	3686.4	819.0	614.2

Table 5.4: The characteristics of random graphs, where we keep fixed the number of vertices to 100K and increase the edge density.

6 2-Vertex-Connected Components

6.1 Introduction

Let us recall the definition of *2-vertex-connected component* (2VCC) from Chapter 2. Given a directed graph $G = (V, E)$, two distinct vertices $v, w \in V(G)$ are called 2-*vertex-connected*, if there are two internal *vertex-disjoint* paths from v to w and two internal *vertex-disjoint* paths from w to v. But note that, a path from v to w and a path from w to v need not be vertex-disjoint. We let denote the 2-*vertex-connected* relation between two vertices v and w by $v \leftrightarrow_{2v} w$. We already explained in Chapter 5, Menger's Theorem

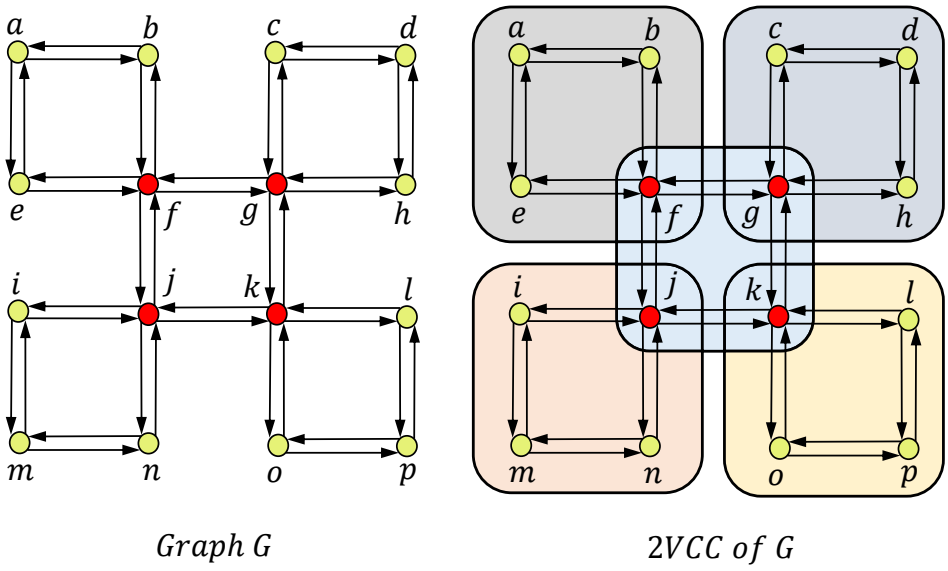


Figure 6.1: Example of 2VCCs of a connected digraph G, SAPs are shown in red color. (Better viewed in color).

[121]* also states the equivalent definition of 2-*vertex-connected*, two different vertices

*To see the statement of Menger's Theorem, please refer the Appendix A.2.1.

$v, w \in G$ are 2-*vertex-connected*, only if the removal of any vertex different from v and w leaves them in the same strongly connected component. But unlike the 2-*edge-connected* relation that is explained in Chapter 4, the converse is not always true. It holds only if v and w are not adjacent to each other. Since two mutually adjacent vertices are left in the same strongly connected component by the removal of any other vertex, but they are not 2-*vertex-connected*. Thus, if $G = (V, E)$ is 2-*vertex-connected*, then it doesn't have any strong articulation points. 2-*vertex-strongly-connected component* (2VCC) of G is its maximal subgraph such that for all two distinct vertices $u, v \in$ 2VCC are 2-*vertex-connected*. Furthermore, unlike in 2VCB, a path for $u \leftrightarrow_{2v} v$ of a 2VCC does not contain a vertex w, if $w \notin$ 2VCC as illustrated in Figure 6.1. Therefore, one vertex may include in multiple 2VCCs but two different 2VCCs cannot have more than one vertex in common that is proved by following lemma 6.1.1.

Lemma 6.1.1. *Two different* 2-*vertex-connected components cannot have a more than one vertex in common.*

Proof. We proceed by contradiction.

Let us consider that $G = (V, E)$ be a digraph, and A and B are the two different 2VCCs in G. We also let a, b be two vertices that are common between A and B as shown in Figure 6.2.

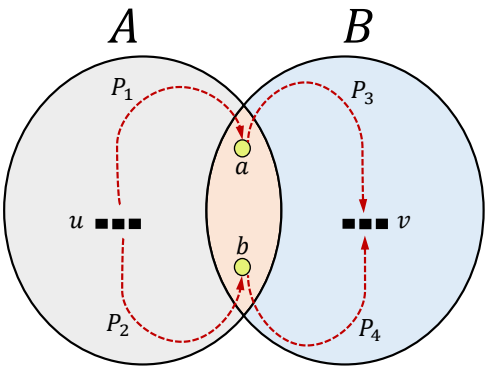


Figure 6.2: Maximum vertices between two different 2VCCs, here we suppose A and B be two different 2VCCs. Also, we let a and b be two different vertices and in common between A and B.

Since $a \in A \implies \forall u \in A \setminus \{a\},\ u \leftrightarrow_{2v} a \implies$ there exist two distinct *vertex-disjoint* paths between u and a, and at least one of them does not contain a vertex b. Let us suppose a path P_1 connects the vertices from u to a without b. Also, $b \in A \implies \forall u \in A \setminus \{b\},\ u \leftrightarrow_{2v} b \implies$ there exist two distinct *vertex-disjoint* paths between u and b, and at least one of them is *vertex-disjoint* to the path P_1. Let us consider a path P_2 connects the vertices from u to b and *vertex-disjoint* to P_1. Again, $a \in B \implies \forall v \in B \setminus \{a\},\ a \leftrightarrow_{2v} v \implies$ there exist two distinct *vertex-disjoint* paths between a and v, and at least one of them does not contain a vertex b. Let say a path P_3 starts at vertex a and ends at v without vertex b. Similarly, $b \in B \implies \forall v \in B \setminus \{b\},\ b \leftrightarrow_{2v} v \implies$ there exist two distinct *vertex-disjoint* paths between b and v, and at least one of them is *vertex-disjoint* to the path P_3. Suppose, a path P_4 connects the vertices from b to v and *vertex-disjoint* to P_3. Now, let us consider the operations given below.

$\mathscr{P}_1 : \underbrace{P_1}_{u \xrightarrow{*} a} \cup \underbrace{P_3}_{a \xrightarrow{*} v}$: a path from u to v.

$\mathscr{P}_2 : \underbrace{P_2}_{\substack{u \xrightarrow{*} b \\ \textit{vertex-disjoint} \\ \text{to } P_1}} \bigcup \underbrace{P_4}_{\substack{b \xrightarrow{*} v \\ \textit{vertex-disjoint} \\ \text{to } P_3}}$: a path from u to v and *vertex-disjoint* to $\mathscr{P}_1$.

Thus, there exist two distinct *vertex-disjoint* paths $\mathscr{P}_1$ and $\mathscr{P}_2$ from every vertex $u \in A$ to each vertex $v \in B$. Analogously, we also have the two *vertex-disjoint* paths from v to u as well. Therefore, all the vertices $u \in A$ and $v \in B$ are 2-*vertex-connected*. It implies that $A \cup B$ is 2-*vertex-connected* subgraph. Since, by definition 2VCC is a maximal 2-*vertex-connected* subgraph and here, A and B are not a maximal 2-*vertex-connected* subgraph. So, they cannot be a 2VCC, it's a contradiction as shown in Figure 6.2.

Note: Lemma 6.1.1 elaborates the concept of a 2VCC and describes the relationships between two different 2VCCs, which will be helpful to explain the architecture of a algorithm in next sections.

□

6.2 Related Work

Erusalimskii and Svetlov [49] first considered the problem of computing the 2-*vertex-connected components* of a digraph. Their algorithm reduces the 2-*vertex-connected components* of an undirected graph without any information on running time complexity bound. The reduction process repeatedly computes the strongly connected components of all subgraphs $G \setminus v$, for every vertex v and removing the edges that connect different strongly connected components. The edge removing process continues until and unless if any of the edges remain, which connect the two different strongly connect component of the current subgraphs of $G \setminus v$. In this process, the 2-*vertex-connected components* of the resulting digraph G are identical to the 2-*vertex-connected components* of the undirected version of G. Later on, Jaberi [90] showed that the algorithm presented on [49] has $O(m^2n)$ running time bound. Moreover, Jaberi [90] also proposed two different algorithms with $O(mn)$ time complexity. The first algorithm decomposes the digraph by repeatedly removing a strong articulation point at a time. And the second algorithm divides the digraph by using a dominator tree [106]. After that, Di Luigi et al. [44] proposed a new $O(mn)$-time algorithm that refines the dominator tree division technique, which is previously applied by an $O(mn)$ time algorithm presented in Jaberi [90]. Very recently, Henzinger et al. [83] propose an $O(n^2)$-time algorithms that apply the hierarchical graph sparsification technique.

To the best of our knowledge, Di Luigi et al. [44] performed the first experimental study on 2-*vertex-connected component* of a directed graph. They find that, on average, their algorithm is faster than the algorithm of Jaberi [90] by a factor of two, while the algorithm by Erusalimskii and Svetlov [49] is not competitive even for graphs of moderate size. Since Henzinger et al. [83] published their algorithm after the experimental observations of Di Luigi et al. [44]. We perform the empirical analysis between the algorithm proposed by Di Luigi et al. [44] that has $O(mn)$ running time, also known as a best in terms of running time from their experimental study, and very recent algorithm proposed by Henzinger et al. [83], which has the $O(n^2)$ time complexity. We also

present the hybrid algorithm that merges the concept of [44] and [83] within $O(n^2)$ time complexity. In our experiments, we evaluated the efficiency of all selected algorithms on large digraphs, which are taken from the different real-world application domains. In the next sections, we will explain the high-level idea of the chosen algorithms and then report our experimental observations later.

6.3 Algorithms

In our assertion, all the algorithms that compute the 2 vertex-connected components of a digraph are generally based on three different approaches. (i) repeatedly removing the strong articulation points (SAP), (ii) using the dominator tree divisions to partition the graphs, and (iii) Hierarchical graph sparsification process. The first approach is trivial, so, we are going to explain the next two techniques in details.

Common Notations. Let us define the common notations for all 2VCC algorithms as following. Let G_s (resp., G_s^R) be the *flow-graph* of a digraph G (resp., G^R). Let D (resp., D^R) be the vertex dominator tree of G_s (resp., G_s^R). Also, for any vertex v, we let $d(v)$ (resp., $d^R(v)$) and $C(v)$ (resp., $C^R(v)$) denote the parent of v ($\neq s$) in D (resp., in D^R) and children of v in D (resp., in D^R) respectively.

6.3.1 Dominator Tree Division

The dominator tree division (DTD) algorithm is proposed by Di Luigi et al. [44]. It is also known as the refinement version of Jaberi [90]. This DTD algorithm is based on the Lemma 6.3.1 that is given below, which is also called the restatement version of the lemma presented in Jaberi [90]. Jaberi's algorithm uses the dominator tree to divide the graph G into several subgraphs that contain all the 2VCCs of G.

Lemma 6.3.1. *Let $G = (V,E)$ be a strongly connected digraph, and let $s \in V$ be an arbitrary start vertex. Any three vertices x, y and z (not necessarily distinct) belong to a common 2-vertex-connected component Σ of G only if they are all siblings in D or one is the immediate dominator of the other two in G (Di Luigi et al. [44]).*

For any pair of vertices u and v of G, we let $C(u,v) = (C(u) \cup \{u\}) \cap (C^R(v) \cup \{v\})$. The set $C(u,v)$ contains all vertices in $C(u) \cap C^R(v)$. Also, if $u = v$ or $u \in C^R(v)$ then $u \in C(u,v)$. Similarly, if $v \in C(u)$ then $v \in C(u,v)$. We can compute all non-empty $C(u,v)$ sets in $O(n)$ time [44]. Let $G(u,v)$ be the *induced subgraph* of G that is induced by the vertices of $C(u,v)$.

Lemma 6.3.2. *Let x and y be any vertices in G such that they are in a 2-vertex-connected component Σ of G. Then x and y are vertices of a subgraph $G(u,v)$ (Di Luigi et al. [44]).*

Proof. We are going to give the proof of this lemma because it explains the concept of DTD algorithm in details.

Observation. Let us apply Lemma 6.3.1 to G_s and G_s^R, $x,y \in \Sigma \implies x$ and y are either siblings in D, or $d(x) = y$, or $d(y) = x$. Also x and y are either siblings in D^R, or $d^R(x) = y$, or $d^R(y) = x$. Now, by considering the relation between x and y in the dominator trees D and D^R, we will have the following cases.

(i) If x and y are siblings in both G_s and G_s^R, then $d(x) = d(y)$ and $d^R(x) = d^R(y)$, so $\{x,y\} \subseteq C(d(x), d^R(x))$.

(ii) If x and y are siblings in G_s and $d^R(x) = y$, then $x \in C(d(x), y)$. But we also have $y \in C(d(x), y)$ because $y \in C(d(x))$.

(iii) If $d(x) = y$ and $d^R(x) = y$, then $x \in C(y,y)$, and by definition, $C(y,y)$ also contains y.

(iv) If $d(x) = y$ and $d^R(y) = x$, then as per our consideration for 2VCC, Σ has at least 3 vertices, suppose a vertex $z \in \{V(\Sigma) \setminus \{x,y\}\}$. By Lemma 6.3.1 vertex z can be neither a sibling of y nor the parent of y in D. So z must be a sibling of x in D. Similarly, we conclude that z is a sibling of y in D^R. Hence $z \in C(y,x)$. But since $y \in C(d^R(x))$ and $x \in C(d(y))$, we also have $x,y \in C(y,x)$.

The remaining cases are analogous (with the role of x and y interchanged), so the lemma follows. □

Algorithm 14: DTD

Input: A strongly connected digraph $G = (V, E)$
Output: 2VCCs of G

// Step 1:
Choose an arbitrary start vertex $s \in V$.
Compute the dominator trees D and D^R.
// Step 2:
If $G \setminus s$ is strongly connected and $d(v) = d^R(v) = s$, for all vertices $v \neq s$, then return G. (G is 2-vertex-connected)
// Step 3:
Compute the subgraphs $G(u, v)$ of G with at least three vertices.
// Step 4:
foreach subgraph $G(u, v)$ with $u \neq v$ **do**
    Compute the strongly connected components of $G(u, v)$.
    Compute recursively the 2-vertex-connected components of each strongly connected component.
// Step 5:
foreach subgraph $G(u, v)$ **do**
    Compute the strongly connected components of $G(u, v) \setminus v$.
    Process each strongly connected component S of $G(v, v) \setminus v$ as follows : If there are two arcs from v to S and two arcs from S to v then compute recursively the 2-vertex-connected components of the subgraph induced by $S \cup \{v\}$. Otherwise, compute recursively the 2-vertex-connected components of the subgraph induced by S.

Now we are going to explain how the DTD algorithm outputs the 2VCCs of the digraphs $G = (V, E)$ in $O(mn)$ time; Its details steps are on Algorithm 14, taken from [44]. At first, it computes the dominator trees D and D^R, then computes all sets $C(u, v)$ such that of $|C(u, v)| \geq 3$ as following. It number the vertices in D (resp., D^R) in pre-order by $pre(v)$ (resp., $pre^R(v)$). Then, for each vertex v, it assign a label with pair $\langle pre(d(v)),\ pre^R(d^R(v))\rangle$, and sort the labels lexicographically in $O(n)$ time by radix sort. Because of the radix sort, the distinct labels $\langle pre(u),\ pre^R(v)\rangle$ are in ascending order. After that, it groups the vertices with identical labels as following. If there is at least one vertex with label $\langle pre(u),\ pre^R(v)\rangle$, then it tests the condition to include the vertices u and v as following. If $d(v) = u$ then it include v in $C(u, v)$, similarly, if $d^R(u) = v$ then it include u in $C(u, v)$. Furthermore, for the two distinct labels of

$\langle pre(u),\ pre^R(v)\rangle$, it numbers the corresponding set $C(u,v)$. Later on, these numbers are used to make the partition of adjacency list for each vertex, which represents the subgraphs of $G(u,v)$. Each recursive call runs linear time $O(V+E)$ [44], where V and E are the sizes of the vertices and edges of the input graph. The depth of the recursion is at most n, and the total size of all subgraphs constructed in Step 3 in each recursion level is $O(m)$ so that full time bound of this algorithm is $O(mn)$.

6.3.2 Hierarchical Graph Sparsification

Henzinger et al. [85] introduced the hierarchical graph sparsification for undirected graphs. Chatterjee et al. [34] and Chatterjee and Henzinger [33] extended this technique for directed graph and game graphs respectively. The sparsification technique allows to replace the 'm' in the $O(mn)$ running time by an 'n', yielding $O(n^2)$. In this sub-section, we are going to discuss the algorithm presented by Henzinger et al. [84] refer as HKL, which follows the sparsification methodology to compute the 2VCC.

2-isolated sets. Henzinger et al. [84] define a 2-*isolated set* (2-IS) of a digraph $G=(V,E)$, where G is not necessarily strongly connected, to be a set of vertices $S \subseteq V$ that (a) cannot be reached by the vertices of $V \setminus S$ or (b) can be reached from $V \setminus S$ only through one vertex v. Every 2-vertex-connected component of G contains either only vertices of $S \cup \{v\}$ or only vertices of $V \setminus S$. Hence, if such a set S is found, we can compute recursively the 2-vertex-connected components in the subgraphs induced by $S \cup \{v\}$ and $V \setminus S$ respectively. Moreover, the set S also called the top strongly connected component tSCC such that either it doesn't have any incoming edges from the vertex $v \in V \setminus S$ or has the incoming edges only through a single vertex $v \notin S$. Furthermore, Henzinger et al. [84] extend the definition of tSCC through the following definitions and lemmas.

Definition 6.3.3. *A set of vertices T induces an almost tSCC in G with respect to a vertex v if $G[T]$ is a tSCC in $G \setminus \{v\}$ but has incoming edges from v in G (Henzinger et al. [84]).*

As like a tSCC, any digraph G may also have contained the bottom strongly connected components bSCC such that the vertex set S either don't have any outgoing edges to the vertex $v \in V \setminus S$ or edges are going out only through a single vertex $v \notin S$. So, if the graph has the bottom strongly connected components, then they would become the tSCCs in reverse graph.

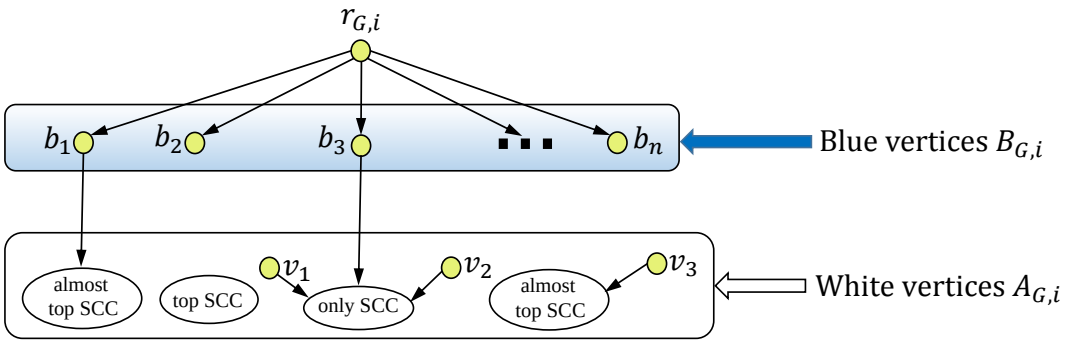


Figure 6.3: Example of *blue* vertices, *white* vertices and (almost) *top* strongly connected component.

Let $G_h = (V_h, E_h)$ be a subgraph of a directed graph $G = (V, E)$, i.e., $V_h \subseteq V$ and $E_h \subseteq G[V_h]$ and h denotes the index of specific subgraphs. Let v be a vertex such that there exist an almost tSCC w.r.t. v in G. Now, our goal is to identify that vertex v in graph G, for that we have to define flow graph created from G_h with an auxiliary root and then identify the v as a vertex-dominator in a flow graph. Let $A_{G,h} \subseteq V(G)$ be the set of *white* vertices for which we have the guarantee that for each vertex in $A_{G,h}$ its incoming edges in G_h are the same as in G. Similarly, Let $B_{G,h} = V_h \setminus A_{G,h}$ be the set of *blue* vertices such that they might miss incoming edges in G_h compared to G. We have to show that as long as the vertices in the almost tSCC are white, i.e., are not missing incoming edges in G_h, an almost tSCC w.r.t. a vertex v in G_h is an almost tSCC w.r.t. v in G and vice versa. Figure 6.3 helps to visualize the set of tSCCs.

Definition 6.3.4. *For a given subgraph $G_h = (V_h, E_h)$ of a directed graph $G = (V, E)$ and a set of* blue *vertices $B_{G,h}$ that contains all vertices that have fewer incoming edges in G_h than in G, we define the flow graph $F_{G,h}(r_G, h)$ as follows. If $|B_{G,h}| \geq 2$, let $F_{G,h}$ be the graph G_h with an additional vertex $r_{G,h}$ and an additional edge from $r_{G,h}$ to*

each vertex in $B_{G,h}$. If $B_{G,h}$ contains a single vertex, we name it $r_{G,h}$ and let $F_{G,h} = G_h$ (Henzinger et al. [84]).

The above definitions and following three lemmas lead us to find the 2VCCs of a directed graph in $O(n^2)$ time. The proofs are available in Henzinger et al. [84].

Lemma 6.3.5. *A set of white vertices $T \subseteq A_{G,h}$ induces a tSCC in G_h and $F_{G,h}$, respectively, if and only if it induces a tSCC in G (Henzinger et al. [84]).*

Lemma 6.3.6. *A set of white vertices $T \subseteq A_{G,h}$ induces an almost tSCC with respect to a vertex $v \in V$ in G_h and $F_{G,h}$, respectively, if and only if it induces an almost tSCC with respect to v in G (Henzinger et al. [84]).*

Lemma 6.3.7. *Assume $B_{G,h} \neq \emptyset$, let $T \subseteq A_{G,h}$ be a set of white vertices, and let $v \in V$ be such that there exists an almost tSCC $G[T]$ with respect to v in G. If v is either not in $B_{G,h}$ and can be reached from a vertex of $B_{G,h}$ or v is in $B_{G,h}$ and $|B_{G,h}| \geq 2$, then v is a dominator in $F_{G,h}(r_G, h)$ (Henzinger et al. [84]).*

Explanation: Let $G = (V, E)$ be a directed graph and $G_i = (V, E_i)$ be the subgraphs of G, where $i \in \mathbb{N}$ and E_i contains first 2^i incoming edges in E for each vertex of $v \in V$. If $i \geq \log(\max_{v \in V} \text{Indeg}_G(v))$, then $G_i = G$. Let $\gamma = \min\ (\max_{v \in V}\ \text{Indeg}_G(v), \max_{v \in V}\ \text{Outdeg}_G(v))$. Following Definition 6.3.4, let a set $B_{G,i}$ contains all vertices that have in-degree more than 2^i in G (i.e., *blue* vertices), and a set W has contains all vertices that have the less than or equal to 2^i incoming edges in G (i.e., *white* vertices). We search a set $S \subseteq W$ such that $G[S]$ be the tSCC or almost tSCC with respect to some vertex v. We can find a set S by searching for SCCs and vertex-dominators in the graphs $F_{G,i}$ constructed from G_i with the artificial root $r_{G,i}$. Similarly, to find bSCCs or almost bSCCs, we have to search for the tSCCs or almost tSCCs in in Rev(G).

Now, we are going to expalin the detail steps of the algorithm, incorporated in the Procedures HKL, 2IsolatedSetLevel, and 2IsolatedSet that are taken from Henzinger et al. [84]. We start the search for (almost) top SCCs at $i = 1$ from the Procedure HKL. If the search is not successful, then we increase i by one, and search again. The process

Procedure HKL(G)

```
1 for i ← 1 to ⌈log γ⌉ − 1 do
2     (S,Z) ← 2IsolatedSetLevel(G,i)
      /* Z contains v if G[S] is almost top or bottom SCC w.r.t. v */
3     if S ≠ ∅ then
4         return HKL(G[S∪Z]) ∪ HKL(G[V \ S])
5 (S,Z) ← 2IsolatedSetLevel(G)
6 if S ≠ ∅ then
7     return HKL(G[S∪Z]) ∪ HKL(G[V \ S])
8 else
9     return {G}
```

Procedure 2IsolatedSetLevel(G, i)

```
1  foreach G ∈ {G, Rev(G)} do
       /* 2^i < max_{v∈V} Indeg_G(v) ⟹ B_{G,i} ≠ ∅ */
2      construct G_i = (V,E_i) with E_i = ∪_{v∈V} { first 2^i edges in In_G(v) }
3      B_{G,i} = { v | Indeg_G(v) > 2^i }
4      S ← TopSCCWithout(G_i, B_{G,i})
5      if S ≠ ∅ then
6          return (S,∅)
7      if |B_{G,i}| = 1 and ∃ tSCC ⊊ V \ {r_{G,i}} in G_i \ {r_{G,i}} then
8          S ← TopSCC (G_i \ {r_{G,i}})
9          return (S, {r_{G,i}})
10     construct flow graph F_{G,i}(r_{G,i})
11     if exists vertex-dominator v ∈ F_{G,i}(r_{G,i}) then
12         S ← TopSCCWithout(G_i \ {v}, B_{G,i})
13         return (S, {v})
14 return (∅,∅)
```

is going to continue until we will get $G_i = G$ or $Rev(G)_i = Rev(G)$. To search the tSCC or bSCC, Procedure 2IsolatedSetLevel will be executed as long as $2^i < \gamma$, i.e., both $B_{G,i}$ and $B_{Rev(G),i}$ are non-empty. If the Procedure 2IsolatedSetLevel can not find any top or bottom SCC, then the Procedure 2IsolatedSet will be executed. By using the known procedures that find the SCCs and articulation points in linear time such as algorithm by Tarjan [154] or by Gabow [60], Procedure 2IsolatedSet identifies the (almost) tSCC

Procedure 2IsolatedSet(G)

```
1 S ← TopSCC(G)
2 if S ⊊ V then
3     return (S, ∅)
4 if exists articulation point v in G then
5     S ← TopSCC(G \ {v})
6     return (S, {v})
7 return (∅, ∅)
```

or bSCC in G. Thus, the Procedure 2IsolatedSet may be used up to i^* time to identify an (almost) top or bottom SCC in G, the identified subgraph contains $\Omega(2^{i^*})$ vertices, where $i^* = \lceil \log \gamma \rceil$ for Procedure 2IsolatedSet. The search in G_i and $Rev(G)_i$ for i up to i^* takes time $O(n.2^{i^*})$ which is $O(n.\min\{|S|, |V \setminus S|\})$. Hence, the algorithm allows us to bound the total running time by $O(n^2)$.

Let $G_i \in \{G_i, Rev(G)_i\}$, Procedure 2IsolatedSetLevel first searches for a tSCC in G_i such that it does not contain any vertex of $B_{G,i}$. It used the Procedure TopSCCWithout (H,B) to denote the search for a tSCC induced by vertices S in a graph H such that S does not contain a vertex of B. If G_i does not have any such tSCC, then according to the size of of $B_{G,i}$, it has two different cases. (i) It consider the special case for $|B_{G,i}| = 1$, and searches for tSCC $\subsetneq V$ in G. $(ii.)$ It constructs a flow graph $F_{G,i}(r_{G,i})$ with artifical root $r_{G,i}$ for the *blue* vertices, and searches for the vertex-dominators. If it finds a vertex-dominator v such that there exist the tSCCs in $G_i \setminus \{v\}$. Then it will stop to find other tSCCs.

If the Procedure 2IsolatedSetLevel does not find any 2-*isolated sets* (i.e. (almost) tSCC or (almost) bSCC) in G, then it searches such sets in G by executing the Procedure 2IsolatedSet. At first, Procedure 2IsolatedSet searches the proper subgraphs which are 2-*isolated* in G. If its find such set then it will stop to search. Otherwise, it collects all the strong articulation point (SAP) of G. If it find any strong articulation point v, then we already knew that the removal of a SAP disjoints the G into two different 2-*isolated sets* (tSCC and bSCC). Therefore, in this case, Procedure 2IsolatedSet returns a tSCC

in $G \setminus \{v\}$. But, if G does not have any articulation points, then by definition, G is itself 2VCC.

6.3.3 Hybrid Algorithm

A simple observation that may help to speed up HKL. When HKL compute strongly connected components in some subgraph constructed by the hierarchical sparsification and call the Procedure 2IsolatedSet then we can search many 2-isolates sets. Therefore, instead of recursing on the partition defined by only one of these sets, we can recurse on all 2-isolated subgraph induced by these sets. Thus, at the point when HKL searches for a 2-isolated set of type (b), we can employ one iteration of DTD in order to refine further the strongly connected subgraphs induced by such 2-isolated sets. We refer to this algorithm as HKL-DTD.

6.4 Experimental Analysis

We perform the empirical observations between the algorithms that we just discussed before, DTD, HKL and HKL-DTD. All the algorithm is implemented in `C++` without using any external graphs library. Moreover, as like in 2ECB (Chapter 4) and 2VCB (Chapter 5) computations, all of the algorithms used the uniform data structures to represent the graphs. Furthermore, the development framework (64−bit Ubuntu `14.04LTS` system, `g++ v.4.8.4` compiler and compiled with full optimization flag (`-O3`), measured `CPU` running time by `getrusage` function, and memory consumption by Valgrind [†] (`v.3.11`)) and the hardware configuration of a testing machine (`3696MHz` Intel `i7-4790` octa-core processor, `16GB` of RAM, `16MB` of L3 cache, and each core has a `2MB` private L2 cache) are also completely identical to the 2ECB and 2VCB computations.

As well as we used the same testing datasets of 2ECB and 2VCB computations, where we choose the graphs from different domains (mostly taken from the *9th DIMACS* implementation challenge [43], and from the Stanford Large Network Dataset Collection [107]). The characteristics of those graphs for 2VCC are summarized in

[†] `http://valgrind.org/`

Graphs				2VCCs		
Name	Type	$n = \|V(G)\|$	$m = \|E(G)\|$	Max-size	Avg-size	Total #
p2p-Gnutella31	P2P	14.1K	50.9K	0.0K	0.0	0
web-NotreDame	WG	54.0K	296.2K	1.5K	20.2	893
soc-Epinions1	SN	32.2K	443.5K	17.1K	84.9	210
Amazon0302	PCP	241.8K	1.1M	55.4K	7.8	19789
WikiTalk	SN	111.9K	1.5M	49.4K	1768.5	28
web-Stanford	WG	150.5K	1.6M	10.9K	16.4	2936
Amazon0601	PCP	395.2K	3.3M	276.0K	35.0	9341
web-Google	WG	434.8K	3.4M	77.5K	12.3	15957
web-BerkStan	WG	334.9K	4.5M	29.1K	15.7	8104
SAP-4M	MP	4.1M	11.9M	2.5K	15.1	1883
Oracle-6M	MP	6.4M	15.9M	3.6K	9.6	47430
SAP-11M	MP	11.1M	36.4M	6.3K	20.1	1479
USA-USA	RN	23.9M	57.7M	16.0M	148.8	112780
LiveJournal	SN	3.8M	65.3M	2.9M	153.7	19202
SAP-32M	MP	32.3M	81.8M	6.6K	10.2	7265
SAP-70M	MP	69.7M	215.7M	7.0K	13.9	10630

Table 6.1: The characteristics of the real-world graphs that we considered; n and m refers to the number of vertices and the number of edges, respectively.Graph types are encoded as follows: road network (RN), peer to peer (P2P), web graph (WG), social network (SN), production co-purchase (PCP), memory profiling (MP). The graphs are sorted in increasing order according to their number of edges. Additionally, we report the statistics of their 2-vertex-connected components, whose size refers to the number of their vertices.

Table 6.1. Also, to analyze the performance of the algorithms in more depth, we had generated the random graphs with specific properties presented in Table 6.3. We averaged the running time of our experiments over ten different runs.

We apply the algorithms DTD, HKL and HKL-DTD over the datasets presented in Table 6.1 to start our experimental analysis. At first, we compared their running time

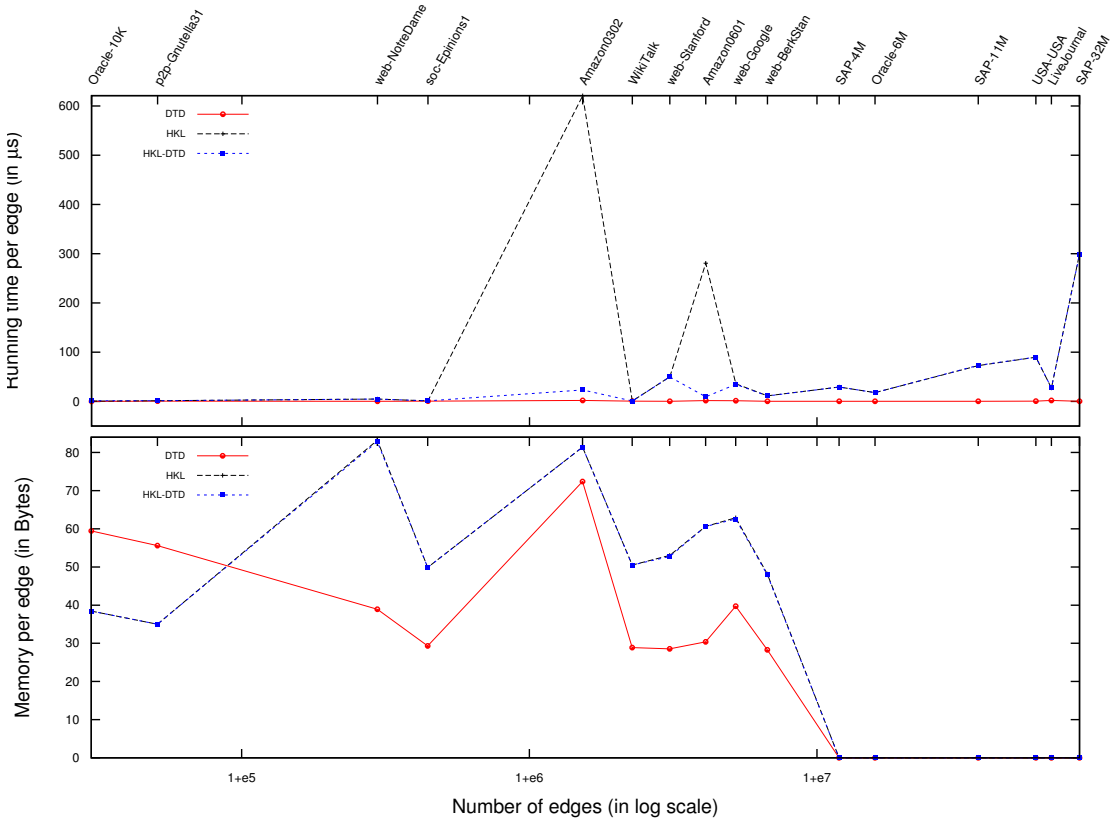


Figure 6.4: Running times per edge in μs (top) and Memory usage per edge in Bytes (bottom) of the algorithms DTD, HKL and HKL-DTD on the real world graph datasets presented in Table 6.1. (Better viewed in color.)

and then the memory consumption. Figure 6.4 plots the report, running time (on top) and memory (on the bottom). The full data of the experiments are reported in Table 6.2. As it can be seen from Figure 6.4 (top), on average DTD was 3 orders of magnitude faster than HKL and HKL-DTD.

Also, on average, HKL-DTD is 4.64% faster than HKL. Hence, incorporating the dominator tree division technique in HKL improved its performance, but only slightly. This happened because HKL attempts first to find 2-isolated sets by running a strongly connected components computation, and uses dominators only if this step fails. Therefore, in our experiments, the dominators computation did not frequently occur enough to provide larger speed-ups. Table 6.5 gives the solid evident for such character by summarizing the recursion depth level of each algorithm. Furthermore, we expand the

Graphs	Running times in seconds			Memory consumption in MBytes		
	DTD	HKL	HKL-DTD	DTD	HKL	HKL-DTD
p2p-Gnutella31	0.02	0.05	0.05	1.9	2.8	2.8
web-NotreDame	0.05	1.17	1.14	8.2	11.4	11.4
soc-Epinions1	0.11	0.38	0.38	7.5	11.5	11.5
Amazon0302	1.04	624.81	16.78	35.1	50.5	50.5
WikiTalk	0.42	1.06	1.08	25.2	38.8	38.8
web-Stanford	0.32	59.58	58.83	28.5	37.7	37.7
Amazon0601	2.46	788.46	24.84	69.7	104.3	104.3
web-Google	2.19	96.55	95.73	73.8	100.5	100.5
web-BerkStan	0.98	35.34	34.10	77.0	91.6	91.6
SAP-4M	1.36	296.86	296.05	542.9	764.2	764.2
Oracle-6M	2.06	158.64	155.62	826.1	1100.0	1100.0
SAP-11M	4.12	1681.73	1672.22	1500.0	2100.0	2100.0
USA-USA	17.11	5033.83	4977.68	3300.0	4500.0	4500.0
LiveJournal	89.86	1702.52	1698.14	1100.0	1600.0	1600.0
SAP-32M	10.88	14945.90	14368.10	4100.0	5800.0	5800.0
SAP-70M	29.07	56620.50	55635.10	9100.0	12800.0	12800.0

Table 6.2: Running times in seconds and Memory consumption in MBytes respectively of the algorithms for computing the 2-vertex-connected components executed on the real world graphs of Table 6.1

Table 6.5 to the Tables 6.6 and 6.7. The data presented in Table 6.6 and 6.7 are plotted by Figure 6.5 (top) and (bottom) respectively.

Tables 6.6 and 6.7 show that the HKL and HKL-DTD have many overhead calls because they continuously tried to search the find 2-isolated sets by increasing in incoming edges of the vertices. However, as we can see HKL-DTD has the less overhead calls as compared to HKL in some graphs because it uses the DTD technique when it came to finding the 2-isolated sets in a whole graph. It helps to boost the running time for some graphs for example “Amazon302”, where HKL-DTD has 17860 number of less overhead

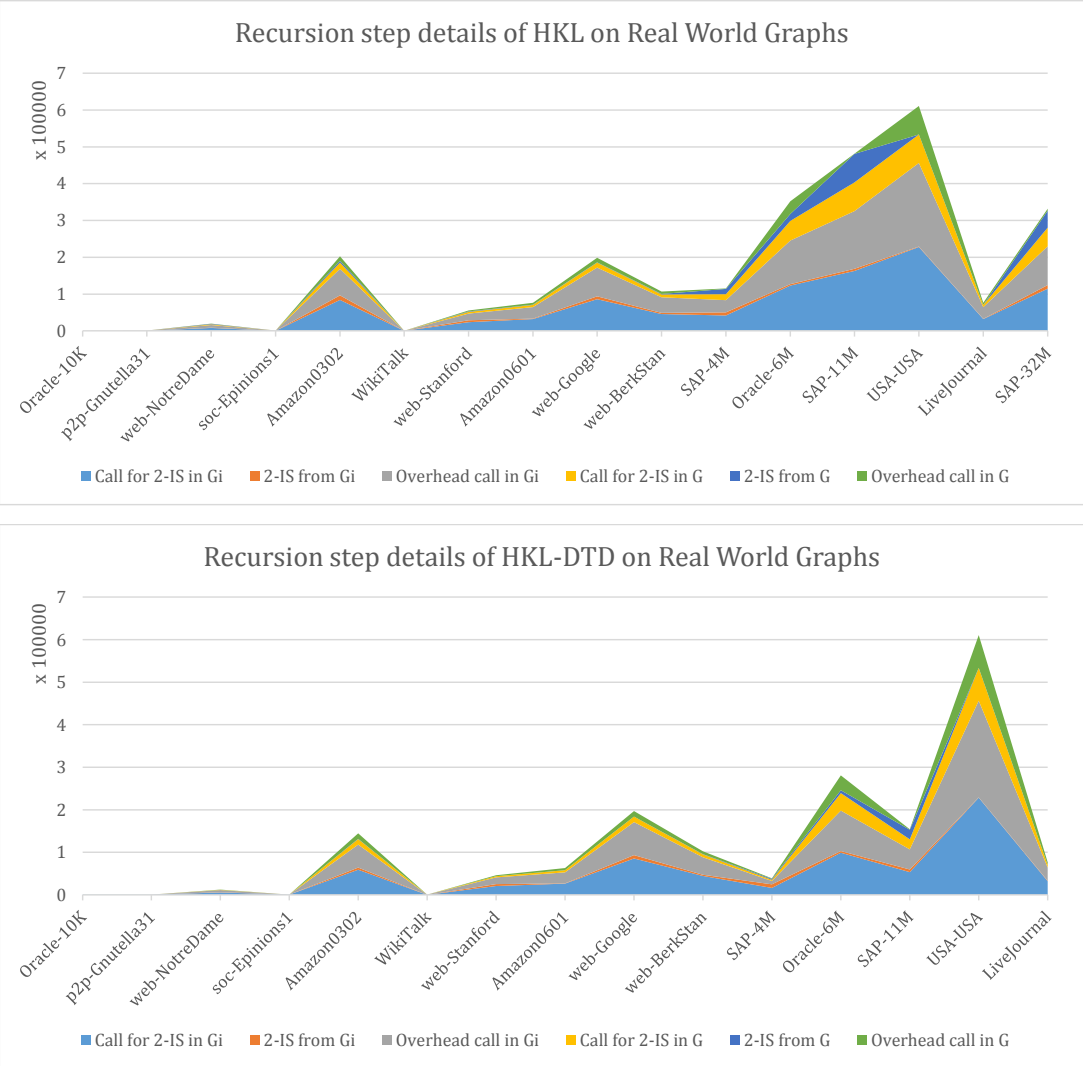


Figure 6.5: Splits of the recursion steps of the algorithms HKL (top) and HKL-DTD (bottom) over the real world graph presented in Table 6.1. (Better viewed in color.)

call then HKL. Therefore, for the graph "Amazon302", HKL-DTD was 2 order of magnitude faster than HKL. We remark that both the algorithms have the equal number of overhead call in G because it happens only when the graph is 2 vertex-connected and all the algorithms output the same number of 2VCCs.

After all, the hierarchical sparsification process is applied to every graph even if the graph is 2VCC. To observe this character of these algorithms, we create the 2-vertex-connected random graphs with constant number vertices and different edges to vertex

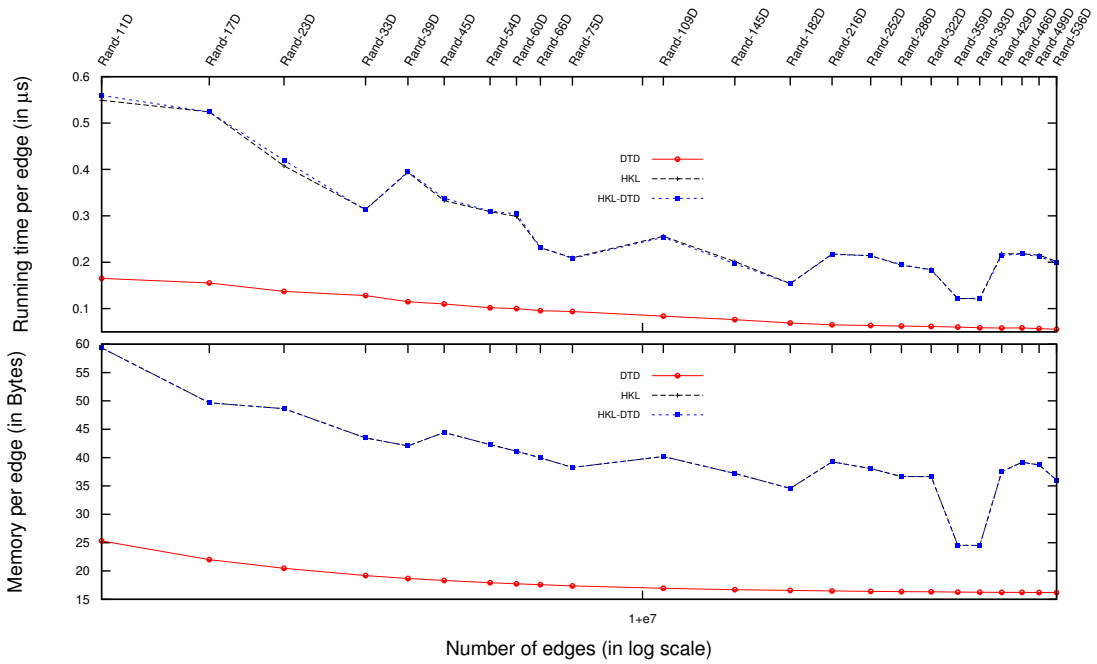


Figure 6.6: Running times per edge in μs (top) and memory usage per edge in Bytes (bottom) of the algorithms DTD, HKL and HKL-DTD on the random graphs summarized in Table 6.3. (Better viewed in color.)

ratio that varies from 11 to 536. The experimental results of these random graphs are provided in Table 6.3. Figure 6.6 (top) and (bottom) plots the performance in running time and the required memory storage respectively, to compute the random graphs. Our experimental observations report show that, on average, DTD is 3 times faster than HKL and HKL-DTD. Theoretically, HKL and HKL-DTD should give the same performance for such 2-vertex-connected random graphs. The experimental observation also reported the same result (with the negligible difference). As we can see that HKL and HKL-DTD have the equal number of unnecessary overhead call given by Table 6.8 and expended by Table 6.9 for HKL and 6.10 for HKL-DTD. Also, Figure 6.7 (top) and (bottom) plots the overhead call of HKL and HKL-DTD respectively. Furhtermore, we noticed that the overhead call is directly proportional to the edge to vertex ratio (which should be in theory as well).

In general, real world graphs are sparse. The experimental report showed that, if the graph is sparse then there are too many strong articulation points exist as well as m is

Graphs ($n = 100$K)		Running times in seconds			Memory in MBytes		
Name	m	DTD	HKL	HKL-DTD	DTD	HKL	HKL-DTD
Rand-11D	1.1M	0.18	0.61	0.62	26.9	63.1	63.1
Rand-17D	1.7M	0.27	0.90	0.90	36.2	81.7	81.7
Rand-23D	2.3M	0.32	0.95	0.98	45.6	108.3	108.3
Rand-33D	3.3M	0.42	1.02	1.02	59.5	134.8	134.8
Rand-39D	3.9M	0.44	1.52	1.53	68.8	154.9	154.9
Rand-45D	4.5M	0.49	1.49	1.51	78.2	189.5	189.5
Rand-54D	5.4M	0.55	1.66	1.67	92.1	217.3	217.3
Rand-60D	6.0M	0.60	1.79	1.82	101.4	235.2	235.2
Rand-66D	6.6M	0.63	1.53	1.54	110.8	251.9	251.9
Rand-75D	7.5M	0.70	1.58	1.56	124.7	274.5	274.5
Rand-109D	10.9M	0.91	2.78	2.76	175.9	417.0	417.0
Rand-145D	14.5M	1.11	2.93	2.86	231.8	516.0	516.0
Rand-182D	18.2M	1.26	2.80	2.80	287.7	600.2	600.2
Rand-216D	21.6M	1.40	4.69	4.68	338.9	807.1	807.1
Rand-252D	25.2M	1.60	5.40	5.39	394.8	915.6	915.6
Rand-286D	28.6M	1.78	5.53	5.57	446.0	999.1	999.1
Rand-322D	32.2M	1.98	5.94	5.89	501.9	1126.4	1126.4
Rand-359D	35.9M	2.17	4.36	4.37	557.8	840.2	840.2
Rand-393D	39.3M	2.31	4.80	4.76	609.0	917.0	917.0
Rand-429D	42.9M	2.49	9.37	9.21	664.9	1536.0	1536.0
Rand-466D	46.6M	2.72	10.17	10.15	720.8	1740.8	1740.8
Rand-499D	49.9M	2.84	10.75	10.58	772.0	1843.2	1843.2
Rand-536D	53.6M	2.97	10.82	10.67	827.8	1843.2	1843.2

Table 6.3: The characteristics of random graphs, where we keep fixed the number of vertices to 100K and increase the edge density.

closer to n. Therefore DTD decomposed the graphs into 2 vertex-connected components very quickly. On the other hand, if the graph is dense (random graphs), then the chances

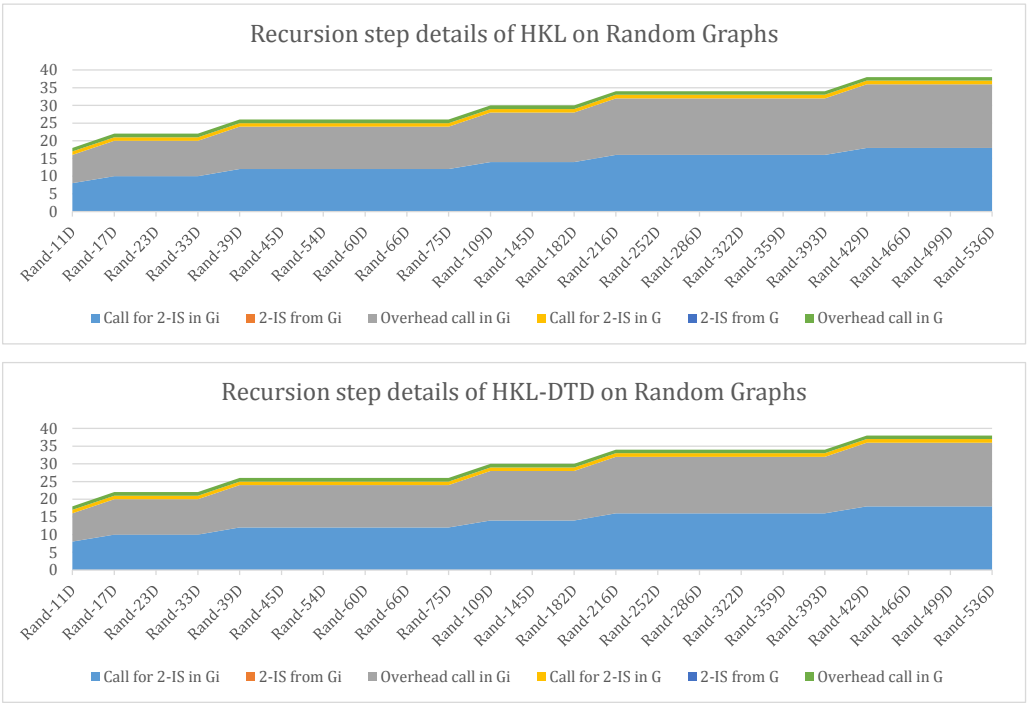


Figure 6.7: Splits of the recursion steps of the algorithms HKL (top) and HKL-DTD (bottom) over the real world graph presented in Table 6.3. (Better viewed in color.)

of a graph to be a 2-*vertex-connected* is very high. In this case, HKL and HKL-DTD have many overhead calls (even if the graph is not a 2-vertex-connected) that cost the algorithms to slow down on their performance. In terms of memory, Figure 6.4 (bottom) shows that, on average, DTD requires almost 29% less memory than HKL and HKL-DTD for the real world graphs. It also shows that both algorithms, HKL and HKL-DTD require the same amount of memory to perform the computation on real world graphs. Similarly, for the 2-*vertex-connected* random graphs, Figure 6.6 (bottom) shows that both HKL and HKL-DTD need the same amount of memory, but need 2.17 times extra memory storage than DTD.

Furthermore, in order to understand the complicated nature of HKL, such that it may have an advantage over DTD, we created a family of dense worst-case instances, shown in Figure 6.8, that we refer to as DTD-BAD. We denote by DTD-BAD(n) the n-

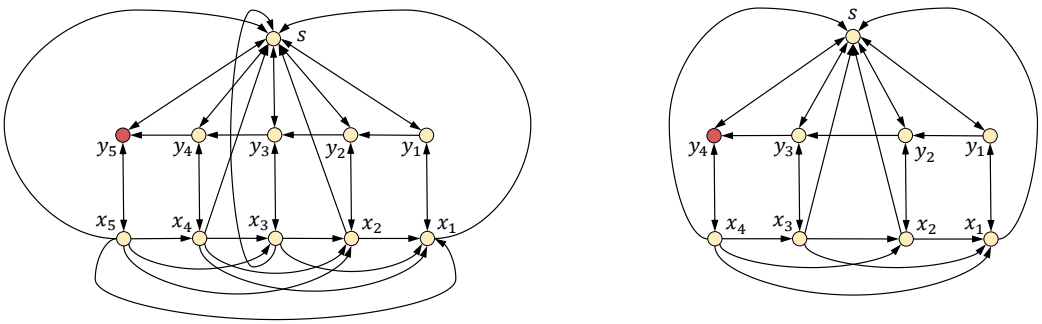


Figure 6.8: A family of digraphs, DTD-BAD, that elicits the worst-case behavior of algorithm DTD. Digraph DTD-BAD(n) has $2n+1$ vertices and $n(n-1)/2+5n$ edges. A double-headed arrow corresponds two parallel but oppositely directed edges. The vertices $x_1, x_2, \ldots, x_n$ are connected by all edges (x_i, x_j) for $i < j$. Vertex y_n (shown in red) is the only strong articulation point of DTD-BAD(n), which when deleted leaves two strongly connected components: an isolated vertex x_n and DTD-BAD($n-1$).

2VCC : DTD-BAD Nature								
Graph details			Running time details			No. of recursive calls		
Name	n	m	α	β	γ	α	β	γ
DTD-BAD(4K)	4K	2M	36.1	7.2	7.1	1997	3998	3998
DTD-BAD(5K)	5K	3.1M	71.1	14.0	14.0	2497	4998	4998
DTD-BAD(6K)	6K	4.5M	124.0	24.0	24.0	2997	5998	5998
DTD-BAD(7K)	7K	6.1M	195.1	37.7	37.9	3497	6998	6998
DTD-BAD(8K)	8K	8M	289.0	56.2	56.3	3997	7998	7998
DTD-BAD(9K)	9K	10.1M	413.4	78.9	79.4	4497	8998	8998
DTD-BAD(10K)	10K	12.5M	561.7	111.4	112.8	4997	9998	9998

Table 6.4: The characteristics of 7 different DTD-BAD graphs, and the running times (in seconds) and the number of recursive calls performed by the algorithms DTD, HKL and HKL-DTD denoted by α, β and γ respectively.

graph in this family, that has $2n+1$ vertices and $n(n-1)/2+5n$ edges. Observe that DTD-BAD(n) has a unique strong articulation point, vertex y_n, which when removed, leaves two strongly connected components: an isolated vertex x_n and DTD-BAD($n-1$). Hence, both HKL and DTD will require $O(n)$ recursive calls to process DTD-BAD(n). For

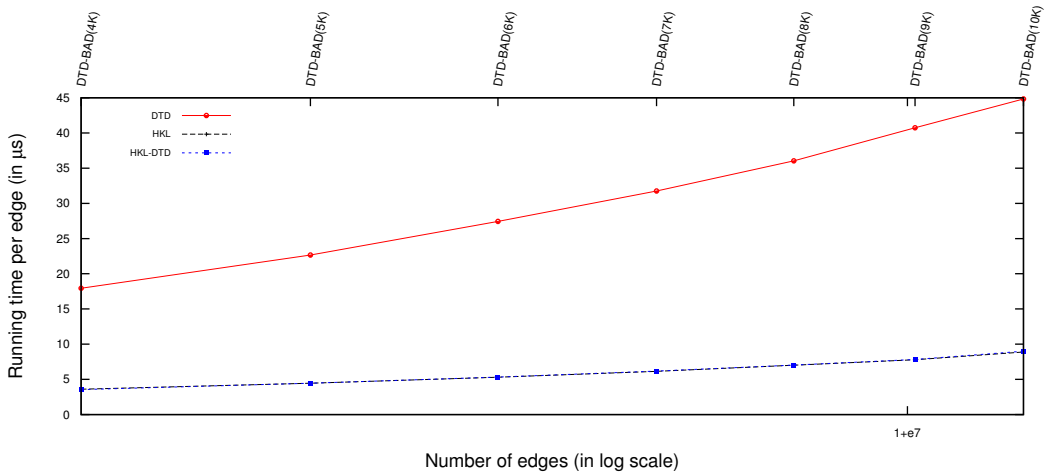


Figure 6.9: DTD-BAD nature graphs.

those graphs, HKL achieves superior performance compared to DTD, since hierarchical sparsification pays off, and it locates a 2-isolated set faster. In this case, the process never reached to the Procedure 2IsolatedSet so that the algorithms HKL and HKL-DTD have the equal running time. Table 6.4 presents the statistics of DTD-BAD graphs and Figure 6.9 plots the corresponding running times of algorithms. It shows that HKL is 5.13 times faster than DTD on average.

Graphs	number of recursive calls		
	DTD	HKL	HKL-DTD
Oracle-10K	8	177	171
p2p-Gnutella31	5	318	285
web-NotreDame	13	9692	6167
soc-Epinions1	4	332	332
Amazon0302	17	101367	72188
WikiTalk	4	78	78
web-Stanford	15	27684	23090
Amazon0601	6	38120	31359
web-Google	11	99357	98441
web-BerkStan	18	53466	51066
SAP-4M	6	57656	19331
Oracle-6M	21	176118	140304
SAP-11M	3	240868	76972
USA-USA	1	305424	305424
LiveJournal	6	38178	38178
SAP-32M	7	166049	0

Table 6.5: Total recursive calls of each algorithm to compute the 2VCCs for the real world graphs presented in Table 6.1. For the algorithms HKL and HKL-DTD, total recursive calls are the sum of total execution of the Procedures 2IsolatedSetLevel and 2IsolatedSet respectively. The details steps for HKL and HKL-DTD are presented in Tables 6.9 and 6.10 respectively.

Graphs	Recursion Details of HKL					
	Call for 2-IS in G_i	2-IS from G_i	Overhead call in G_i	Call for 2-IS in G	2-IS from G	Overhead call in G
Oracle-10K	165	121	44	12	9	3
p2p-Gnutella31	268	161	107	50	50	0
web-NotreDame	7779	615	7164	1913	1268	645
soc-Epinions1	286	47	239	46	1	45
Amazon0302	84381	12198	72183	16986	4378	12605
WikiTalk	72	19	53	6	0	6
web-Stanford	23633	4898	18735	4051	1759	2292
Amazon0601	32195	1350	30845	5925	857	5068
web-Google	86048	7678	78370	13309	561	12748
web-BerkStan	45709	3321	42388	7757	1005	6751
SAP-4M	41795	8604	33191	15861	14593	1268
Oracle-6M	122877	4024	118853	53241	18786	34455
SAP-11M	162755	6069	156686	78113	77193	920
USA-USA	228184	856	227328	77240	0	77240
LiveJournal	32139	833	31306	6039	33	6006
SAP-32M	114821	9334	105487	51228	45611	5617

Table 6.6: Recursion details of the HKL algorithm over the real world graphs presented in Table 6.1. It shows that the total and overhead calls to the Procedures 2IsolatedSetLevel and 2IsolatedSet respectively for 2-IS.

Graphs	Recursion Details of HKL-DTD					
	Call for 2-IS in G_i	2-IS from G_i	Overhead call in G_i	Call for 2-IS in G	2-IS from G	Overhead call in G
Oracle-10K	161	121	40	10	7	3
p2p-Gnutella31	246	161	85	39	39	0
web-NotreDame	5429	614	4815	738	93	645
soc-Epinions1	286	47	239	46	1	45
Amazon0302	59074	4748	54326	13114	509	12605
WikiTalk	72	19	53	6	0	6
web-Stanford	20553	4891	15662	2537	245	2292
Amazon0601	26254	652	25602	5105	37	5068
web-Google	85422	7653	77769	13019	271	12748
web-BerkStan	44042	3270	40772	7024	273	6751
SAP-4M	16245	8604	7641	3086	1818	1268
Oracle-6M	98992	4009	94983	41312	6857	34455
SAP-11M	53491	6069	47422	23481	22561	920
USA-USA	228184	856	227328	77240	0	77240
LiveJournal	32139	833	31306	6039	33	6006
SAP-32M	0	0	0	0	0	0

Table 6.7: Recursion details of the HKL-DTD algorithm over the real world graphs presented in Table 6.1. It shows that the total and overhead calls to the Procedures 2IsolatedSetLevel and 2IsolatedSet respectively for 2-IS.

Graphs	number of recursive calls		
	DTD	HKL	HKL-DTD
Rand-11D	0	9	9
Rand-17D	0	11	11
Rand-23D	0	11	11
Rand-33D	0	11	11
Rand-39D	0	13	13
Rand-45D	0	13	13
Rand-54D	0	13	13
Rand-60D	0	13	13
Rand-66D	0	13	13
Rand-75D	0	13	13
Rand-109D	0	15	15
Rand-145D	0	15	15
Rand-182D	0	15	15
Rand-216D	0	17	17
Rand-252D	0	17	17
Rand-286D	0	17	17
Rand-322D	0	17	17
Rand-359D	0	17	17
Rand-393D	0	17	17
Rand-429D	0	19	19
Rand-466D	0	19	19
Rand-499D	0	19	19
Rand-536D	0	19	19

Table 6.8: Total recursive calls of each algorithm to compute the 2VCCs for random graphs presented in Table 6.3. For the algorithms HKL and HKL-DTD, total recursive calls are the sum of total executions of the Procedures 2IsolatedSetLevel and 2IsolatedSet. The details steps for HKL and HKL-DTD are presented in Tables 6.9 and 6.10 respectively.

Graphs	Recursion Details of HKL					
	Call for 2-IS in G_i	2-IS from G_i	Overhead call in G_i	Call for 2-IS in G	2-IS from G	Overhead call in G
Rand-11D	8	0	8	1	0	1
Rand-17D	10	0	10	1	0	1
Rand-23D	10	0	10	1	0	1
Rand-33D	10	0	10	1	0	1
Rand-39D	12	0	12	1	0	1
Rand-45D	12	0	12	1	0	1
Rand-54D	12	0	12	1	0	1
Rand-60D	12	0	12	1	0	1
Rand-66D	12	0	12	1	0	1
Rand-75D	12	0	12	1	0	1
Rand-109D	14	0	14	1	0	1
Rand-145D	14	0	14	1	0	1
Rand-182D	14	0	14	1	0	1
Rand-216D	16	0	16	1	0	1
Rand-252D	16	0	16	1	0	1
Rand-286D	16	0	16	1	0	1
Rand-322D	16	0	16	1	0	1
Rand-359D	16	0	16	1	0	1
Rand-393D	16	0	16	1	0	1
Rand-429D	18	0	18	1	0	1
Rand-466D	18	0	18	1	0	1
Rand-499D	18	0	18	1	0	1
Rand-536D	18	0	18	1	0	1

Table 6.9: Recursion details of HKL algorithm for random graphs presented in Table 6.3. It shows that the total and overhead calls to the Procedures 2IsolatedSetLevel and 2IsolatedSet respectively for 2-IS.

Graphs	Recursion Details of HKL-DTD					
	Call for 2-IS in G_i	2-IS from G_i	Overhead call in G_i	Call for 2-IS in G	2-IS from G	Overhead call in G
Rand-11D	8	0	8	1	0	1
Rand-17D	10	0	10	1	0	1
Rand-23D	10	0	10	1	0	1
Rand-33D	10	0	10	1	0	1
Rand-39D	12	0	12	1	0	1
Rand-45D	12	0	12	1	0	1
Rand-54D	12	0	12	1	0	1
Rand-60D	12	0	12	1	0	1
Rand-66D	12	0	12	1	0	1
Rand-75D	12	0	12	1	0	1
Rand-109D	14	0	14	1	0	1
Rand-145D	14	0	14	1	0	1
Rand-182D	14	0	14	1	0	1
Rand-216D	16	0	16	1	0	1
Rand-252D	16	0	16	1	0	1
Rand-286D	16	0	16	1	0	1
Rand-322D	16	0	16	1	0	1
Rand-359D	16	0	16	1	0	1
Rand-393D	16	0	16	1	0	1
Rand-429D	18	0	18	1	0	1
Rand-466D	18	0	18	1	0	1
Rand-499D	18	0	18	1	0	1
Rand-536D	18	0	18	1	0	1

Table 6.10: Recursion details of the HKL-DTD algorithm for random graphs presented in Table 6.3. It shows that the total and overhead calls to the Procedures 2IsolatedSetLevel and 2IsolatedSet respectively for 2-IS.

7 Critical Nodes Detection

7.1 Introduction

In many applications in network analysis we wish to identify the nodes of a network that are important for a specific task, where the definition of "importance" varies accordingly to the application at hand. Problems of this type have received a lot of attention recently: for example, one may wish to identify nodes that represent highly influential individuals in a social network [96], or locations in a network that are useful in order to inhibit diffusion of contagions [18, 105], or to assess network vulnerabilities [146]. Motivated by the recent study of Ventresca and Aleman [168], we consider the problem of detecting a set S of critical nodes such that $G \setminus S$ has minimum pairwise strong connectivity. This problem is NP-hard [16, 46], and thus we are interested in practical heuristics. As noted in [16], the critical node detection problem has, in particular, several applications in the field of social network analysis. In the recent years, social networks have been the subject of significant amount of research, aiming to better understand several properties such as cohesion, transitivity, and centrality of specific actors [21]. Other applications of critical nodes include network immunization [37], and the study of covert terrorist networks [103].

Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. Two vertices u and v of G are *strongly connected* if there is a (directed) path from u to v and a (directed) path from v to u. Digraph G is *strongly connected* if every two vertices are strongly connected. The *strongly connected components* of G are its maximal strongly connected subgraphs. Clearly, two vertices u and v are strongly connected if and only if they belong to the same strongly connected component (SCC) of G. The *size* $|C|$ of a strongly connected component C of G is given by its number of vertices. A vertex of G

is a *strong articulation point* if its removal increases the number of strongly connected components. Note that strong articulation points are 1-vertex cuts for digraphs. Let $G \setminus v$ denote the digraph obtained after deleting vertex v together with all its incident edges. Similarly, for a set of vertices S, we let $G \setminus S$ denote the digraph obtained after deleting all vertices in S and their incident edges.

Let G be a directed graph, and let $C_1, C_2, \ldots, C_\ell$ be its strongly connected components. Let the *size* $|C_i|$ of a strongly connected component C_i is given by its number of vertices. We define the *connectivity value of* G as

$$f(G) = \sum_{i=1}^{\ell} \binom{|C_i|}{2}.$$

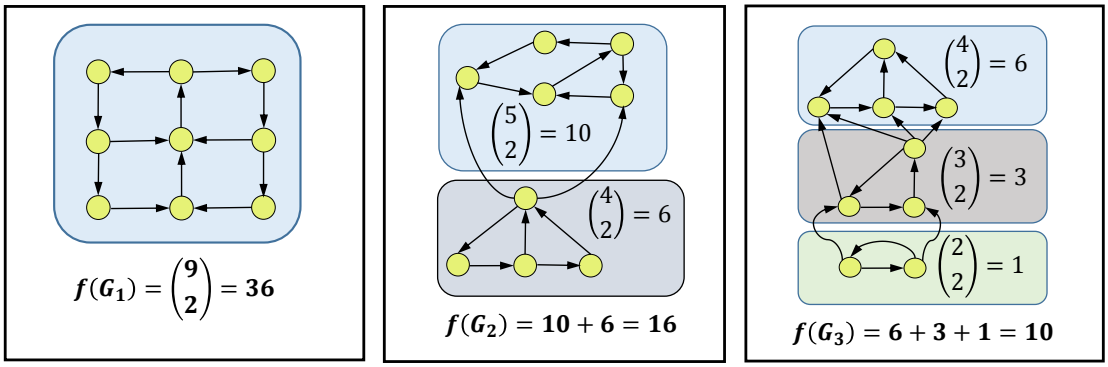


Figure 7.1: Connectivity value of graphs G_1, G_2 and G_3. Even though all of them have the equal number of vertices, their connectivity value are different according to the size and number of SCCs they have.

Note that, $f(G)$ equals the number of vertex pairs in G that are strongly connected. Two vertices are strongly connected if they are mutually reachable from each other. For example, as shown in Figure 7.1, three different graphs G_1, G_2, and G_3 has the equal number of vertices, but their connectivity value $f(G)$ are not equal each other, because $f(G)$ of a graph depends on the numbers and sizes of SCCs. As we can see in Figure 7.1, G_1 has only one SCC of size 9, so, $f(G_1) = \binom{9}{2} = 36$, whereas G_2 has two different SCCs of size 4 and 5, therefore, $f(G_2) = \binom{4}{2} + \binom{5}{2} = 16$. Similarly, G_3 has

three different SCCs of size 2, 3 and 4, implies that, $f(G_3) = \binom{2}{2} + \binom{3}{2} + \binom{4}{2} = 10$.

We wish to compute a set $S \subseteq V$ of vertices such that the connectivity value of the residual graph $G \setminus S$ is minimized, i.e., $S = \min_{S \subseteq V} f(G \setminus S)$. In the special case of $k = 1$, we wish to locate a vertex $x \in V$ such that $f(G \setminus x)$ is minimum. We refer to such a vertex x as the *most critical node* of G. This problem was previously considered in the literature, but only for undirected graphs (see, e.g., [16, 168]). In particular, Ventresca and Aleman [168] presented a linear-time algorithm for the $k = 1$ case in undirected graphs. Their algorithm exploits the relation between *depth-first-search* (DFS) and articulation points and biconnected components of an undirected graph G [158]. Hence, they provide a dfs-based algorithm for locating the most critical node of G.

In this chapter, we present a sophisticated linear-time algorithm for the $k = 1$ case in *directed graphs*. That is, given a directed graph $G = (V, E)$ with n vertices and m edges, we identify the most critical node of G in $O(m+n)$ time. To the best of our knowledge, this is the first non-trivial algorithm for detecting the most critical node of a directed graph, and provides a substantial improvement over the naive solution of computing $f(G \setminus x)$ from scratch, for all vertices x. The preliminary version of the algorithm, heuristics that are proposed in this Chapter and the experimental reports were presented at the "17^{th} International Conference on Algorithm Engineering and Experiments [132]". A journal publication containing all the results is in preparation. As highlighted by several recent results, connectivity-related problems for digraphs are notoriously harder than for undirected graphs, and indeed many notions for undirected connectivity do not translate to the directed case; see, e.g., [74, 83] (Also see in Chapter 2 - Section 2.3). Our algorithm is based on the recent framework of [75] for answering strong connectivity queries in a directed graph under an edge or a vertex failure. A natural extension of this algorithm is to repeatedly remove the most critical node of the current graph G, until we have removed k vertices. This way, we obtain an efficient heuristic for the general case that runs in $O(k(m+n))$ time. We assess the performance of our algorithms experimentally. We show that the linear-time algorithm performs very well in practice, while the naive approach of computing $f(G \setminus v)$ for all vertices v is not

competitive even for very small graphs. Also, our heuristic is shown to achieve much better fragmentation of the input graph compared to selecting nodes by other popular heuristics, such as Betweenness Centrality [24], Page Rank [130], and Maximum Degree.

7.2 Algorithms

In this section, we present our linear-time algorithm for computing the most critical node in a directed graph G. This algorithm can be extended, in a straightforward manner, to provide an efficient heuristic for the general case of the critical node detection problem. We first review some fundamental concepts used by our algorithms. Before going through the algorithm, we suggest that the reader review the notation from Chapters 2 and 3, in particular the notions of flow graphs, dominators, strong articulation points and loop nesting forests.

Notation. Let G_s (resp., G_s^R) be the flow-graph of G (resp., G^R). We denoted the vertex dominator trees and loop nesting tree of the flow-graph G_s (resp., G_s^R) by D (resp., D^R) and H (resp., H^R) respectively. Moreover, let $d(v)$ (resp., $d^R(v)$) denote the parent of $v \neq s$ in D (resp., D^R) and let $C(v)$ (resp., $C^R(v)$) denote the set of children of a vertex v in D (resp., D^R). Similarly, let $\widetilde{D}(u)$ represents the set of proper descendants of a vertex u in D. Let T be a tree rooted at s. If vertex v is an ancestor of vertex w, then $T[v,w]$ denotes the path from v to w in T; we let $T(v,w)$ denote the part of $T[v,w]$ from the child of v that is an ancestor of w to the parent of w. Also, we let $T(v)$ denote the set of descendants of a vertex v in T, and $\widetilde{T}(v)$ the set of proper descendants of v in T, i.e., $\widetilde{T}(v) = T(v) \setminus v$. For a set of vertices $S \subseteq V$, we let $G[S]$ be the subgraph of G that is induced by S. For simplicity we denote $f(S) = f(G[S])$.

7.2.1 Linear-Time Algorithm for Most Critical Node.

Here we present our linear-time algorithm, to compute the most critical node of a digraph G. Throughout, we refer to this algorithm as MCN (Most Critical Node). We

assume that G is strongly connected. If this is not the case, then we can execute our algorithm on each strongly connected component of G separately and pick the vertex v that minimizes $f(G \setminus v)$, as follows: If v is the most critical node of a strongly connected component C of G, then $f(G \setminus v) = f(G) - f(C) + f(C \setminus v)$ illustrated in Figure 7.2. Hence, given the most critical nodes of all $O(n)$ strongly connected components of G, we can choose the most critical node of G in $O(n)$ time.

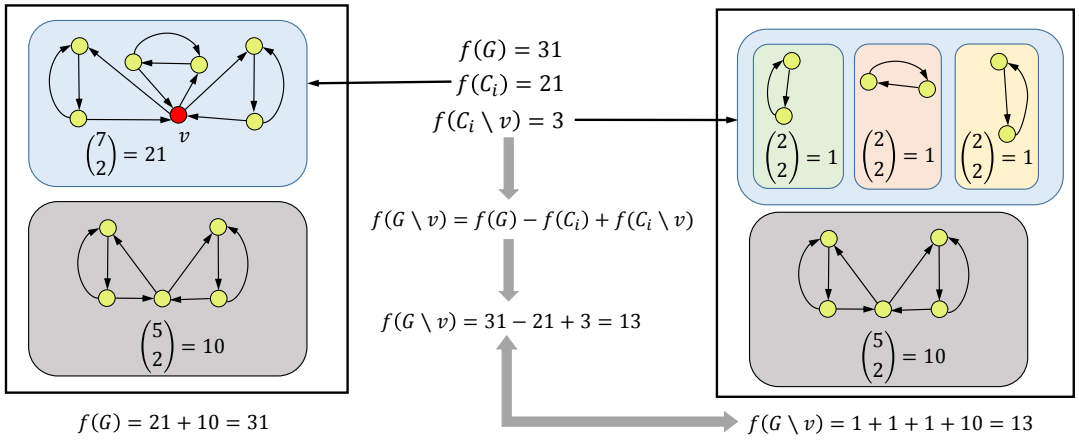


Figure 7.2: Compute the connectivity value of a graph after the removal of a vertex.

Our algorithm hinges upon the following key result from [75]:

Theorem 7.2.1. ([75]) *Let u be a strong articulation point of G, and let s be an arbitrary vertex in G. Let C be a strongly connected component of $G \setminus u$. Then one of the following cases holds:*

(a) If u is a nontrivial dominator in G_s but not in G_s^R then either $C \subseteq \widetilde{D}(u)$ or $C = V \setminus D(u)$.

(b) If u is a nontrivial dominator in G_s^R but not in G_s then either $C \subseteq \widetilde{D}^R(u)$ or $C = V \setminus D^R(u)$.

(c) If u is a common nontrivial dominator of G_s and G_s^R then either $C \subseteq \widetilde{D}(u) \setminus \widetilde{D}^R(u)$, or $C \subseteq \widetilde{D}^R(u) \setminus \widetilde{D}(u)$, or $C \subseteq \widetilde{D}(u) \cap \widetilde{D}^R(u)$, or $C = V \setminus (D(u) \cup D^R(u))$.

(d) If $u = s$ then $C \subseteq \widetilde{D}(u)$.

Moreover, if $C \subseteq \widetilde{D}(u)$ *(resp.,* $C \subseteq \widetilde{D}^R(u)$*) then* $C = H(w)$ *(resp.,* $C = H^R(w)$*) where* w *is a vertex in* $\widetilde{D}(u)$ *(resp.,* $\widetilde{D}^R(u)$*) such that* $h(w) \notin \widetilde{D}(u)$ *(resp.,* $h^R(w) \notin \widetilde{D}^R(u)$*).*

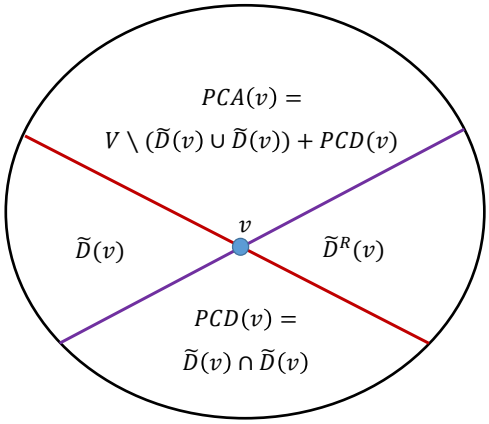


Figure 7.3: Overview of the number of SCCs in $G \setminus v$. $\widetilde{D}(v)$ (resp., $\widetilde{D}^R(v)$) denotes the proper descendants of v in D (resp., in D^R). Similarly, the proper common descendants (resp., ancestors) of v are represented by $PCD(v)$ (resp., $PCA(v)$).

The above theorem provides the means to detect SCCs after the deletion of a vertex. Our goal is to use this information in order to compute $f(G \setminus v)$ for each strong articulation point v of G. To do this efficiently, we find a way to compute this function for all strong articulation points simultaneously.

Let $PCD(u) = \widetilde{D}^R(u) \cap \widetilde{D}(u)$ and $PCA(u) = V \setminus (D(u) \cup D^R(u))$, respectively, be the set of proper common descendants and the set of proper common ancestors of u in D and D^R. The digrammatic representation of these partitions are presented in Figure 7.3. We divide the computation of $f(G \setminus v)$ in four parts:

$$f(\widetilde{D}(v)) = \sum_{w \in \widetilde{D}(v), h(w) \notin \widetilde{D}(v)} \binom{|H(w)|}{2} \tag{7.1}$$

$$f(\widetilde{D}^R(v)) = \sum_{w \in \widetilde{D}^R(v), h^R(w) \notin \widetilde{D}^R(v)} \binom{|H^R(w)|}{2} \tag{7.2}$$

$$f(PCD(v)) = \sum_{w \in PCD(v), h(w) \notin \widetilde{D}(v)} \binom{|H(w)|}{2} \tag{7.3}$$

$$f(PCA(v)) = \binom{|PCA(v)|}{2} \tag{7.4}$$

The above equalities follow from Theorem 7.2.1. Equations (7.1) and (7.2) calculate the connectivity values of the subgraphs induced by $\widetilde{D}(v)$ and $\widetilde{D}^R(v)$, respectively. Similarly, equations (7.3) and (7.4) calculate the connectivity values of the subgraphs induced by $PCD(v) = \widetilde{D}(v) \cap \widetilde{D}^R(v)$ and $PCA(v) = V \setminus (D(v) \cup D^R(v))$, respectively. Then, $f(G \setminus v) = f(\widetilde{D}(v)) + f(\widetilde{D}^R(v)) - f(PCD(v)) + f(PCA(v))$. (See Algorithm 15.)

Algorithm 15: MostCriticalNode

Input: A strongly connected digraph G
Output: Most critical node v of G

Compute the reverse digraph G^R.
Select an arbitrary start vertex $s \in V$.
Compute the dominator trees D and D^R of the flow graphs G_s and G_s^R, respectively.
Compute the sets of nontrivial dominators N and N^R of the flow graphs G_s and G_s^R, respectively.
Compute the loop nesting trees H and H^R of the flow graphs G_s and G_s^R, respectively.
$cnode \leftarrow 0$, $cvalue \leftarrow f(G)$, $value \leftarrow 0$
foreach *strong articulation point v of G* **do**
    /* calculate the connectivity value for v */
    Compute $f(\widetilde{D}(v))$, $f(\widetilde{D}^R(v))$, $f(PCD(v))$, $f(PCA(v))$
    $value \leftarrow f(\widetilde{D}(v)) + f(\widetilde{D}^R(v)) - f(PCD(v)) + f(PCA(v))$
    /* vertex with minimum *value* is the most critical node */
    if $cvalue > value$ **then**
        $cnode \leftarrow v$
        $cvalue \leftarrow value$
return *cnode*

Next we describe how to perform the computations (7.1)–(7.4). To that end we apply the framework of [75], which provides an efficient way to compute several functions defined on the decompositions induced by the 1-connectivity cuts (strong articulation points or strong bridges) of a strongly connected digraph. The pairwise strong connectivity function $f(G)$ that we consider fits within this framework. We remark, however,

that the computations for vertex-cuts in [75] use a reduction to edge-cuts via vertex-splitting. Here we provide a direct algorithm that avoids this reduction, and is thus expected to be faster in practice.

In the following, we assume that we have precomputed the dominator trees D and D^R, and the loop nesting trees H and H^R of flow graphs G_s and G_s^R. These can be done in $O(m)$ time [30]. As we show next, given these trees, we can perform the computations (7.1)–(7.4) for all strong articulation points in $O(n)$ total time. Thus, we obtain the following result:

Theorem 7.2.2. *We can compute the most critical node of a directed graph G with n vertices and m edges in $O(m+n)$ time.*

Computing $f(\widetilde{D}(v))$ and $f(\widetilde{D}^R(v))$. Here we describe how to compute $f(\widetilde{D}(v))$ for all strong articulation points v of G. The computation of $f(\widetilde{D}^R(v))$ is analogous. The main idea in our algorithm is the following. Suppose C is a SCC in the induced subgraph $G[\widetilde{D}(v)]$ for some vertex v. Our goal is to add the corresponding value $\binom{|C|}{2}$ to all ancestors u of v such that C is also a SCC in $G[\widetilde{D}(u)]$. Next we introduce the notion of a bundle that identifies the appropriate set of these ancestors.

Bundle of vertex v in D: Let u be the lowest ancestor of v in D such that $h(v)$ is a proper descendant of u (i.e., $h(v) \in \widetilde{D}(u)$). If u exists, then we define the bundle of v to be the vertices in the path $D[u,v]$. See Figure 7.4. Otherwise, if u does not exist, then $h(v) = s$ and we let the bundle of v to be $D[s',v]$, where s' is a dummy vertex that we think of as the parent of s in D and in D^R.

We can locate u with the help of the next lemmata.

Lemma 7.2.3. *For any vertex $v \neq s$, if $h(v) \neq s$ then $d(h(v))$ dominates v.*

Proof. Assume, for contradiction, that the lemma is false. Then, there is a path π_{sv} from s to v that avoids $d(h(v))$. By the definition of $h(v)$ function, we have a path $\pi_{v,h(v)}$ from v to $h(v)$ that contains only descendants of $h(v)$ in the corresponding dfs tree. Hence, $d(h(v)) \notin \pi_{v,h(v)}$. But then, $\pi_{sv} \cdot \pi_{v,h(v)}$ is a path from s to $h(v)$ that avoids $d(h(v))$, a

contradiction. □

Lemma 7.2.4. *For any vertex $v \neq s$, if $h(v) \neq s$ then the bundle of v is $D[d(h(v)), v]$.*

Proof. Let u be the lowest ancestor of v in D such that $h(v)$ is a proper descendant of u. Let z be the nearest common ancestor of $h(v)$ and v in D. If $h(v)$ is an ancestor of v in D then $z = h(v)$ and $u = d(z)$. Otherwise, $u = z$ and by Lemma 7.2.3 we have that $u = d(h(v))$. □

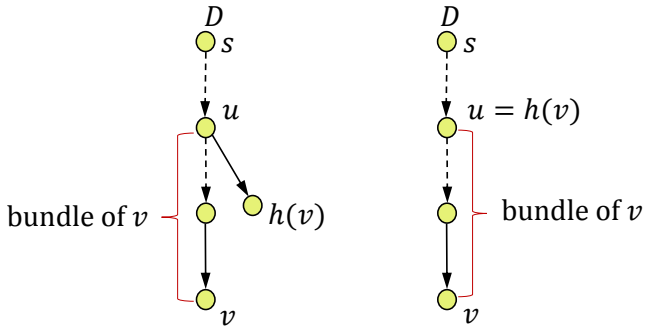


Figure 7.4: Bundle of vertex v in D. In the first case (left), $h(v)$ is not an ancestor of v in D, while in the second case (right), $h(v)$ is an ancestor of v in D.

Procedure DescendantValues(D, H)

1 Initialize $DSum(v) \leftarrow 0$ for each vertex $v \in V$
2 **foreach** *vertex* $v \in V \setminus s$ **do**
3 Let $D[u,v]$ be the bundle of v in D, where $u = d(h(v))$ // $u = s'$ if $h(v) = s$
4 $DSum(d(v)) \leftarrow DSum(d(v)) + \binom{|H(v)|}{2}$
5 $DSum(u) \leftarrow DSum(u) - \binom{|H(v)|}{2}$
6 **foreach** *vertex* v *in* D *in bottom-up fashion* **do**
7 **foreach** *child* c *of* v *in* D **do**
8 $DSum(v) \leftarrow DSum(v) + DSum(c)$

By Lemma 7.2.4 we can locate the bundle of any vertex in $O(1)$ time. The meaning of the bundle of v is that $H(v)$ is a SCC of $G \setminus x$ for all $x \in D(u,v)$. Hence, $H(v)$

contributes the value $f(H(v)) = \binom{|H(v)|}{2}$ in $f(G \setminus x)$ for all $x \in D(u,v)$. We can accumulate these values by processing D in a bottom-up fashion, as shown in Procedure DescendantValues. The accumulated values are stored in variables $DSum(v)$ for each vertex v. To compute the desired sums of these values, we first add the value $f(H(v))$ to $DSum(d(v))$ and subtract it from $DSum(u)$. This has the effect of cancelling the value of $f(H(v))$ at all ancestors u of v in D such that $h(v) \in D(u)$. Hence, Procedure DescendantValues computes $DSum(v) = f(\widetilde{D}(v))$ for all vertices v.

It is easy to verify that, given the dominator tree D and the loop nesting tree H, Procedure DescendantValues runs in $O(n)$ time.

Computing $f(PCD(v))$. Now our goal is to locate the vertices in the bundle of v that are also ancestors of v in D^R. We can do this fast with the help of our next lemma.

Lemma 7.2.5. *Let $D[u,v] = \langle u = w_0, w_1, \ldots, w_{l-1}, w_l = v \rangle$ be the bundle of v in D. If w_1 is not a dominator of v in D^R then no w_j, $1 < j \leq l-1$, is. Otherwise, let j be the largest index $1 \leq j \leq l-1$ such that vertex v is a common descendant of w_j in D and D^R. Then $\langle w_{l-1}, w_{j-2}, \ldots w_1 \rangle$ is path in D^R and v is a common descendant of every vertex w_i for $1 \leq i \leq l-1$.*

To prove Lemma 7.2.5, we use the following two results:

Lemma 7.2.6. *Let x, y and z be distinct vertices such that x is an ancestor of z in D and $y \in D(x,z)$. If z is an ancestor of x in D^R then $y \in D^R(z,x)$.*

Proof. Suppose, for contradiction, that $y \in D(x,z)$ but $y \notin D^R(z,x)$. The fact that $y \in D(x,z)$ implies that all paths from x to z in G contain y. But $y \notin D^R(z,x)$ implies that there is a path in G^R from z to x that avoids y, a contradiction. □

Lemma 7.2.7 (Path Lemma [157])**.** *Let T be a dfs tree of a digraph G, and let $pre(v)$ denote the preorder number of vertex v in T. If v and w are vertices such that $pre(v) < pre(w)$, then any path from v to w must contain a common ancestor of v and w in T.*

Proof. We already stated and proved the Path Lemma in Chapter 3, section 3.3 as Lemma 3.3.1. □

Now we are ready to state the proof of Lemma 7.2.5.

Proof. Let $u = w_0, w_1, \ldots, w_{l-1}, w_l = v$ be the vertices in the bundle of v in D, in the order they appear on $D[u,v]$. From the definition of the bundle we have $h(v)$ is not a descendant of w_i in D for $1 < i \leq l$. It suffices to show that if w_i, $1 < i < l$, is an ancestor of v in D^R then $w_{i-1} \in D(w_i, v)$. Assume by contradiction that the above statement is not true. Then, there is a path π from v to w_i that avoids w_{i-1}. Path π does not contain any vertex $x \notin D(w_{i-1})$, since otherwise the part of π from x to w_i would have to include w_{i-1}. Therefore, all vertices in π are descendants of w_{i-1} in D. Let T be the dfs tree that generated the loop nesting tree H of G_s, and let *pre* be the preorder numbering in T. We claim that there is a vertex z in π such that all vertices in π are descendants of z in T. The claim implies that $v \in H(z)$, so $h(v)$ is a descendant of w_i in D. But this contradicts the fact that $h(v)$ is not a descendant of w_i in D. Hence, the lemma will follow.

To prove the claim, choose z to be the vertex in π such that $pre(z)$ is minimum. Then Lemma 7.2.7 implies that z is an ancestor of w_i in T. Let y be any vertex in π. We argue that $pre(z) \leq pre(y) < pre(z) + |T(z)|$, hence y is a descendant of z in π. By the choice of z we have $pre(z) \leq pre(y)$, so it remains to prove the second inequality. Suppose $pre(y) \geq pre(z) + |T(z)|$. Since v is descendant of w_i in D it is also a descendant of w_i in T. So $pre(v) < pre(y)$. By Lemma 7.2.7, path π contains a common ancestor q of v and y in T. Vertex q is an ancestor of z in T, since $v \in T(z)$ and $y \notin T(z)$. But then $pre(q) < pre(z)$, which contradicts the choice of z.

Therefore, any path π from v to w_i must contain w_{i-1}. We conclude that $w_{i-1} \in D[w_i, v]$. Now the fact that $\langle w_{l-1}, w_{j-2}, \ldots w_1 \rangle$ is path in D^R follows from Lemma 7.2.6. □

Lemma 7.2.5 implies that we can compute the contribution of the SCCs of $G[PCD(v)]$ in $f(G \setminus v)$ by applying a similar approach as for $G[\widetilde{D}(v)]$. To facilitate the corresponding computations, we use an auxiliary data structure that we refer to as the *common dominator forest* Q of D and D^R.

Common dominator forest $\mathcal{Q}$: Forest $\mathcal{Q}$ is the resulting subgraph of D after deleting

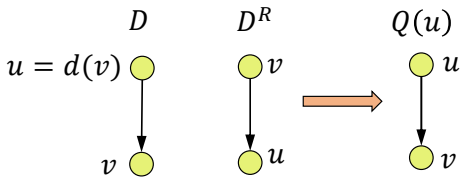


Figure 7.5: Illustration of edges $(d(v), v)$ included in the common dominator forest $\mathcal{Q}$.

all dominator tree edges $(d(v), v)$ such that $d(v)$ is not a child of v in D^R. That is, $(d(v), v)$ is an edge of $\mathcal{Q}$ if and only if $d(v) \in D^R(v)$. See Figure 7.5.

Procedure CommonDominatorForest(D,D^R)

```
1 foreach vertices v ∈ V \ s do
2     d_v ← d(v)
3     if v = d^R(d_v) then /* i.e., d_v is a child of v in D^R */
4         q(v) ← d_v
5     else
6         q(v) ← null
7 return 𝒬 /* q(v) is the parent of v in 𝒬 */
```

Note that $\mathcal{Q}$ is a forest that consists of subtrees of D. We use the notation $Q(v)$ to denote the tree in the common dominator forest $\mathcal{Q}$ that contains vertex v.

Procedure CommonDescendantValues implements the computation of $f(PCD(v))$ for all v. First, we construct a list L of vertex pairs $\langle h_v, v\rangle$ such that $h_v = h(v)$ is not a sibling of v in D. This list contains the vertices v for which we wish to locate their common nontrivial dominators in the bundle $D[u, v]$ of v in D. Note that L contains at most n such pairs.

We process the pairs $\langle h_v, v\rangle$ in L using a similar approach with Procedure DescendantValues, but here we operate on the common dominator forest $\mathcal{Q}$ instead of D. By Lemma 7.2.5, the common nontrival dominators of v that are in the bundle $D[u, v]$ are located in the tree $Q(w)$, where w is the child of $d(h_v)$ that is an ancestor of v in D. Let z be the nearest ancestor of v in D such that $z \in Q(w)$. Then, the common nontrivial dominators of v that we wish to locate are in the path $Q[w, v]$.

It is straightforward to implement Procedure CommonDescendantValues so that it

Procedure CommonDescendantValues(D, D^R, H)

```
1  forall vertices v ∈ V do
2      cdSize(v) ← startSize(v) ← endSize(v) ← 0
3      cdSum(v) ← startSum(v) ← endSum(v) ← 0
4  Initialize an empty list of pairs L.
5  foreach vertex v do /* construct list of pairs                          */
6      h_v ← h(v)
7      if d(h_v) ≠ d(v) then // h_v is not a sibling of v in D
8          L ← L ∪ ⟨h_v, v⟩
   /* process L to find the vertex W(v) = w that is a child of d(h_v) and an
      ancestor of v in D and D^R, for each vertex v with ⟨h_v, v⟩ ∈ L       */
9  W ← FindW(L) // computes all vertices W(v)
10 𝒬 ← CommonDominatorForest (D, D^R)
   /* process W and 𝒬 to find the vertex z that is the nearest ancestor of v
      in D with z ∈ Q(w) and W(v) = w ≠ null                               */
11 List Z ← FindZ (W, 𝒬)
   /* Z contains the tuples ⟨v, w, z⟩ such that W(v) ≠ null                */
12 foreach tuple ⟨v, w, z⟩ ∈ Z do
       /* w is the child of d(h_v) and an ancestor of v in D and D^R       */
       /* z is nearest ancestor of v in D with z ∈ Q(w)                    */
13     endSize(z) ← endSize(z) + |H(v)|
14     startSize(w) ← startSize(w) + |H(v)|
15     endSum(z) ← endSum(z) + (|H(v)| choose 2)
16     startSum(w) ← startSum(w) + (|H(v)| choose 2)
17 foreach tree Q in 𝒬 do
18     foreach vertex v ∈ Q, in a bottom-up fashion do
19         cdSize(v) ← endSize(v)
20         cdSum(v) ← endSum(v)
21         foreach child c of v in Q do
22             cdSize(v) ← cdSize(v) + cdSize(c) − startSize(c)
23             cdSum(v) ← cdSum(v) + cdSum(c) − startSum(c)
```

runs in $O(n)$ time (given the dominator and loop nesting trees), except for the computation of w and z. We can compute these vertices in $O(n)$ time for all pairs in L by bucket sort. We perform a preorder traversal of D and store the preorder number $pre(v)$ of each

vertex v. We also store the reverse mapping pre_to_vertex(p_v), which gives the vertex with preorder number p_v. That is pre_to_vertex($pre(v)$) $= v$. In addition, we use two abstract functions defined next. Let C be a list of tuples, and let t be a tuple in C. Then, C.getTuple(i) returns the i^{th} tuple in C, and t.getElement(k) returns the k^{th} element in tuple t.

For each pair $\langle h_v, v\rangle$ in L we create a tuple $\langle pre(d(h_v)), pre(v), 1\rangle$. Also, for each vertex v we create the tuple $\langle pre(d(v)), pre(v), 0\rangle$. We sort these tuples and process them in increasing order. For each tuple $\langle pre(d(h_v)), pre(v), 1\rangle$ we locate its predecessor of the form $\langle pre(d(v)), pre(v), 0\rangle$ in the sorted list. Then, we find the vertex w which is a child of $d(h_v)$ and an ancestor of v in D and D^R. The final result is stored in $W(v)$, which gives the desired vertex w of v. If w is not an ancestor of v in D^R, then $W(v)$ is null. (For the details, please refer to Procedure FindW).

We compute the common dominator forest $\mathcal{Q}$ from D and D^R. If $d(v)$ is a child of v in D^R, then we store this relation as $q(v) = d(v)$, otherwise the value of $q(v)$ is null. See Procedure CommonDominatorForest .

The calculation of z is similar to the computation of w. For all vertices v such that $W(v) = w \neq null$, we create a tuple $\langle TreeID(W(v)), pre(v), 1\rangle$ where $TreeID(w)$ is an integer id that specifies the tree $Q(w)$ of $\mathcal{Q}$ containing w. Notice that the last element of each tuple has value 1, which indicates that we have to find the nearest ancestor z of v in D with $z \in Q(w)$. We also create a tuple $\langle TreeID(u), pre(u), 0\rangle$ for each vertex $u \in V$. We sort all these tuples in ascending order by bucket sort. Then, for each tuple $\langle TreeID(W(v)), pre(v), 1\rangle$ we locate its predecessor of the form $\langle TreeID(z), pre(z), 0\rangle$ in the sorted list of tuples, and extract the corresponding vertex z. Finally, we create a tuple $\langle v, w, z\rangle$ for each v such that $W(v) \neq null$, and store it in the list Z. See Procedure FindZ for the details.

Computing $f(PCA(v))$. In order to compute (7.4) we need to specify the size of the set $PCA(v)$. Since $|PCA(v)| = |V| - |D(v) \cup D^R(v)| = |V| - |D(v)| - |D^R(v)| + |\widetilde{D}(v) \cap \widetilde{D}^R(v)| + 1 = |V| - |D(v)| - |D^R(v)| + |PCD(v)| + 1$, it suffices to compute $|PCD(v)|$. This

can be done with the same procedure as in the computation of $f(PCD(v))$, where we substitute $\binom{|H(w)|}{2}$ with $|H(w)|$ in the corresponding calculations, see Procedure CommonDescendantValues. We are going to give a complete example in next section.

Procedure FindW(List L of pairs $\langle h_v, v\rangle$)

```
1  Initialize lists A, B, C
   /* pre(v) is the preorder number p_v of a vertex v in the dominator tree D,
      and pre_to_vertex(p_v) is the vertex v with preorder number p_v in D      */
2  forall vertices v ∈ V do
3      W(v) ← null
4  foreach ⟨h_v, v⟩ ∈ L do
5      l_id ← 1 // l_id = 1 indicates list A
6      A ← A ∪ ⟨pre(d(h_v)), pre(v), l_id⟩
7  foreach v ∈ V do
8      l_id ← 0 // l_id = 0 indicates list B
9      B ← B ∪ ⟨pre(d(v)), pre(v), l_id⟩
10 C ← A ∪ B
11 Bucket sort list C
12 csize ← size of List C
13 for i ← csize to 1 do
       /* C.getTuple(i) gives the i^th tuple in the sorted list C              */
14     tuple ← C.getTuple(i)
       /* tuple.getElement(k) gives the k^th element in tuple                  */
15     if tuple.getElement(2) = 0 then // i.e., l_id = 0 so tuple ∈ B
16         continue
       /* since the second element of tuple is pre(v), extract v from it      */
17     v ← pre_to_vertex(tuple.getElement(2))
       /* Find the largest index j < i of list C such that
          C.getTuple(j).getElement(2) = 0                                      */
18     temp_tuple ← C.getTuple(j)
       /* i.e., temp_tuple ∈ B and is the nearest such predecessor of tuple in
          list C                                                               */
       /* since the second element of temp_tuple is pre(w), extract w from it */
19     w ← pre_to_vertex(temp_tuple.getElement(2))
       /* test if w is an ancestor of v in D^R using the ancestor-descendant
          test of [158]                                                        */
20     p_v ← preorder number of v in D^R
21     p_w ← preorder number of w in D^R
22     size_w ← |D^R(w)|
23     if p_w ≤ p_y and p_v < (p_w + size_w) then
           /* w is an ancestor of v in D^R                                     */
24         W(v) ← w
25 return W
```

Procedure FindZ(W, $\mathcal{Q}$)

```
1  Initialize lists A, B, C, Z
   /* pre(v) is the preorder number p_v of a vertex v in the dominator tree D,
      and pre_to_vertex(p_v) is the vertex v with preorder number p_v in D      */
   /* TreeID(w) is an integer id that specifies the tree Q(w) of 𝒬 containing
      w                                                                        */
2  l_id ← 1 // l_id = 1 indicates list A
3  foreach v ∈ V do
4      if W(v) ≠ null then
5          w ← W(v)
6          A ← A ∪ ⟨TreeID(w), pre(v), l_id⟩
7  l_id ← 0 // l_id = 0 indicates list B
8  foreach v ∈ V do
9      B ← B ∪ ⟨TreeID(v), pre(v), l_id⟩
10 C ← A ∪ B
11 Bucket sort list C
12 csize ← size of list C
13 for i ← csize to 1 do
       /* C.getTuple(i) gives the i^th tuple in the sorted list C             */
14     tuple ← C.getTuple(i)
       /* tuple.getElement(k) gives the k^th element in tuple                 */
15     if tuple.getElement(2) = 0 then
           // i.e., l_id = 0 so tuple ∈ B
16         continue
17     Find the largest index j < i of list C such that C.getTuple(j).getElement(2) =
        0
18     temp_tuple ← C.getTuple(j)
       /* i.e., temp_tuple ∈ B and is the nearest such predecessor of tuple in
          list C                                                              */
       /* z is the nearest ancestor of v in D with z ∈ Q(w); the second element
          of temp_tuple contains pre(z), so extract z from it                 */
19     z ← pre_to_vertex(temp_tuple.getElement(2))
       /* second element of tuple contains the pre(v), extract v from it      */
20     v ← pre_to_vertex(tuple.getElement(2))
       /* extract w of v                                                      */
21     w ← W(v)
22     Z ← Z ∪ ⟨v, w, z⟩
   /* return a list Z with tuples ⟨v, w, z⟩                                   */
23 return Z
```

7.3 Complete Example

Let us consider the flow graph G_1 (resp., reverse flow graph G_1^R) of a input graph G (resp., G^R, reverse graph of G) with start vertex 1 as shown in Figure 7.6.

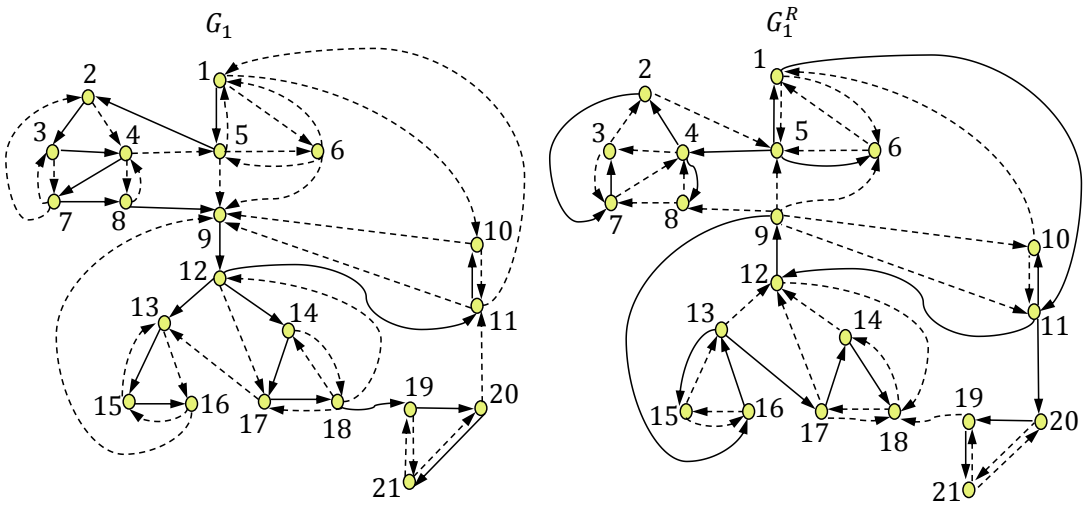


Figure 7.6: Flow graph G_1 (resp., reverse flow graph G_1^R) of strongly connected graph G. Solid edges represent the dfs tree edges with root vertex 1.

Initialization. Let us compute the dominator tree D (resp., D^R) and loop nesting tree H (resp., H^R) of the flow graph G_1 (resp ., G_1^R) as shown in Figure 7.7 and 7.8 respectively. After that, we will assign a preorder number to each vertex of every tree in a DFS visit.

Computation. We already explained that the Procedure DescendantValues outputs the accumulated values of $\binom{|H(v)|}{2}$ in $f(G \setminus x)$ for all $x \in D(u,v)$. The resulted values of SCCs in $\widetilde{D}(v)$ and in $\widetilde{D}^R(v)$ are presented in the Tables 7.1 and 7.2 respectively. Table 7.3 contains the accumulated values of $\binom{|H(v)|}{2}$ for each $x \in D(u,v)$.

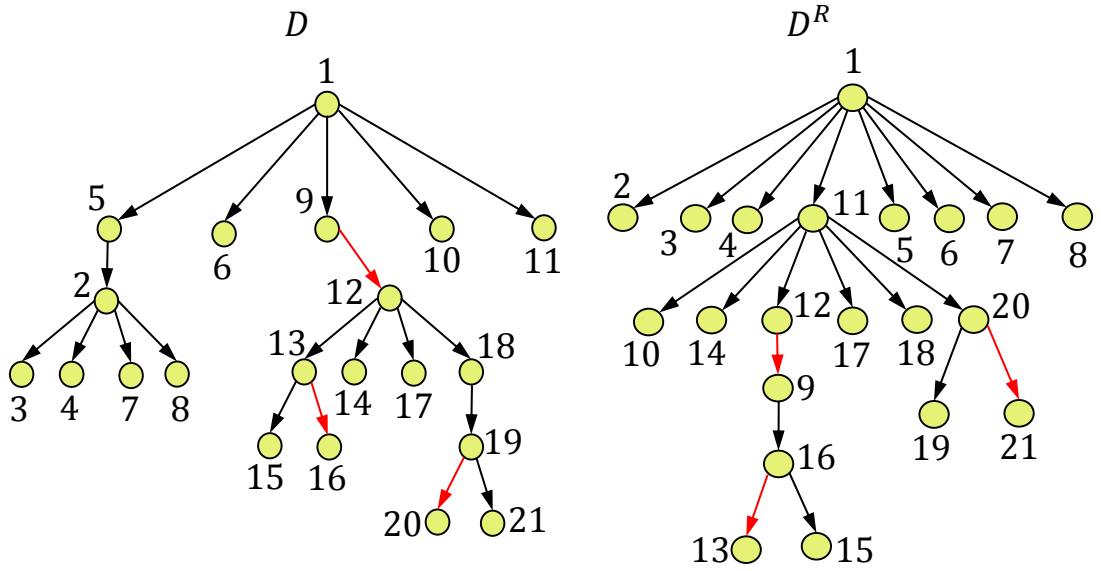


Figure 7.7: Dominator Trees D and D^R of the flow graphs G_1 and G_1^R respectively. Edges $(d(v), v)$ are shown in red color in D, if $d(v)$ is the child of v in D^R and vice versa.

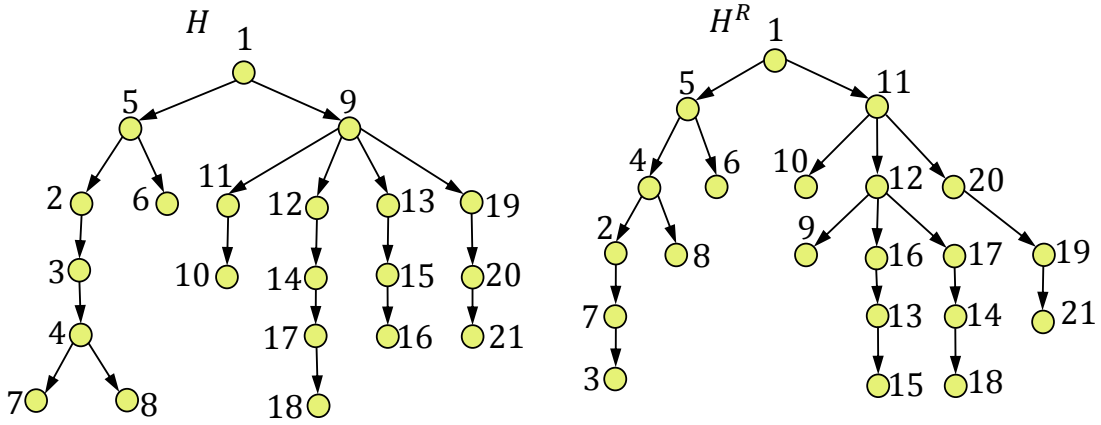


Figure 7.8: Loop Nesting Trees of G and G^R with start vertex 1

v	$d(v)$	$h(v)$	α	$Bundle[v,\alpha]$	$\lvert H(v)\rvert$	$DSum(v)$	$DSum(\alpha)$
2	5	5	1	[2,1]	5	10	-10
3	2	2	5	[3,5]	4	6	-6
5	1	1	0	[5,0]	7	21	-21
9	1	1	0	[9,0]	13	78	-78
12	9	9	1	[12,1]	4	6	-6
13	12	9	1	[13,1]	3	3	-3
14	12	12	9	[14,9]	3	3	-3
15	13	13	12	[15,12]	2	1	-1
19	18	9	1	[19,1]	3	3	-3
20	19	19	18	[20,18]	2	1	-1

Table 7.1: Calculation of SCCs in $\widetilde{D}(v)$, dummy vertex($s' = 0$), Table only contains those vertices v such that $d(v) \neq d(h(v))$, α denotes the value of $d(h(v))$

v	$d^R(v)$	$h^R(v)$	α	$Bundle[v,\alpha]$	$\lvert H^R(v)\rvert$	$DSum^R(v)$	$DSum^R(\alpha)$
5	1	1	0	[5,0]	7	21	-21
9	12	12	11	[9,11]	1	1	-1
10	11	11	1	[10,1]	1	1	-1
11	1	1	0	[11,0]	13	78	-78
12	11	11	1	[12,1]	8	28	-28
13	16	16	9	[13,9]	2	1	-1
16	9	12	11	[16,11]	3	3	-3
19	20	20	11	[19,11]	2	1	-1
20	11	11	1	[20,1]	3	3	-3

Table 7.2: Calculation of SCCs in $\widetilde{D}^R(v)$, dummy vertex($s' = 0$), Table only contains those vertices v, such that $d^R(v) \neq d^R(h^R(v))$, α denotes the value of $d^R(h^R(v))$.

Vertices from 1 to 11											
v	1	2	3	4	5	6	7	8	9	10	11
In D	99	6	0	0	10	0	0	0	12	0	0
In D^R	99	0	0	0	0	0	0	0	3	0	32
Vertices from 12 to 21											
In v	12	13	14	15	16	17	18	19	20	21	–
In D	9	1	0	0	0	0	3	1	0	0	–
In D^R	4	0	0	0	1	0	0	0	1	0	–

Table 7.3: Final Calculation of SCCs in $\widetilde{D}(v)$ and $\widetilde{D}^R(v)$ in bottom up fashion.

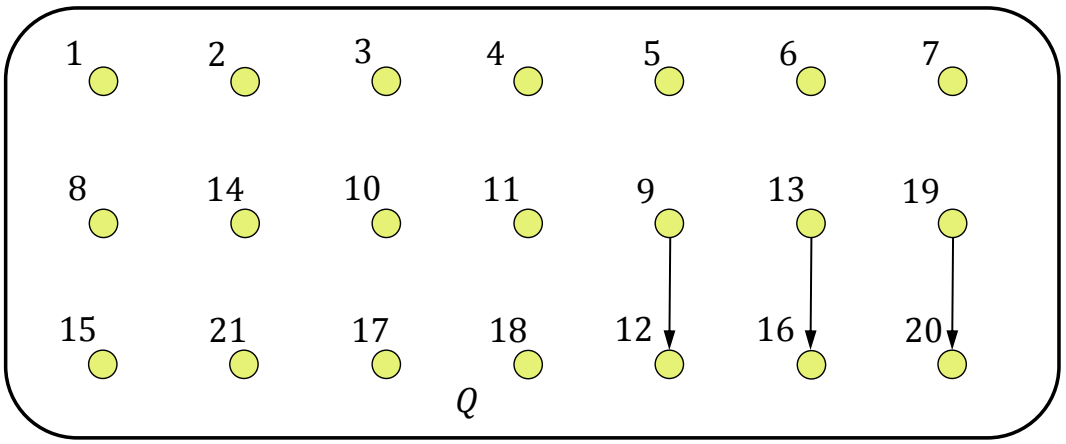


Figure 7.9: Common dominator forest obtain from the D and D^R.

Now, our goal is to compute the SCCs in the common descendants of each vertex $v \in V$. Hence, at first, by executing the Procedure CommonDominatorForest , we create a common dominator forest, which is shown in Figure 7.9. The tree-id of each vertex in the dominator forest is presented in Table 7.4. Then, we compute the SCCs in $\widetilde{D}(v) \cap \widetilde{D}^R(v)$ for each vertex $v \in V$ by using the Procedure CommonDescendantValues along with Procedures FindW and FindZ . The calculated value of y, w, and z during the

execution of Procedure CommonDescendantValues are presented in Table 7.5 and end results are available in Table 7.6.

Still, we have to calculate the $C_A(v)$ and SCCs value in $C_A(v)$ for each vertex $v \in V$. Table 7.7 contains the result of $C_A(v)$ calculations.

The final accumulated calculation is shown in Table 7.8, which shows that node 12 is the most critical node.

Tree-ID	Elements	Remarks
1	{11}	
2	{10}	
3	{21}	
4	{19,20}	19–>20
5	{18}	
6	{9,12}	9–>12
7	{17}	
8	{14}	
9	{13,16}	13–>16
10	{15}	
11	{6}	
12	{8}	
13	{2}	
14	{7}	
15	{4}	
16	{3}	
17	{5}	
18	{1}	

Table 7.4: Generating the Trees in common forest $\mathscr{Q}$

y	w	y in $\widetilde{D}^R(w)$	z	$\lvert H(y)\rvert$	$startSize$ (w)	$endSize$ (z)	$startSum$ (w)	$endSum$ (z)
20	19	No	–	–	–	–	–	–
15	13	No	–	–	–	–	–	–
14	12	No	–	–	–	–	–	–
3	2	No	–	–	–	–	–	–
19	9	No	–	–	–	–	–	–
13	9	Yes	12	3	3	3	3	3
12	9	No	–	–	–	–	–	–
2	5	No	–	–	–	–	–	–
9	1	Yes	1	13	13	13	78	78
5	1	Yes	1	7	20	20	99	99

Table 7.5: Finding the y,w and z

Vertices from 1 to 11											
v	1	2	3	4	5	6	7	8	9	10	11
$cdSize(v)$	20	0	0	0	0	0	0	0	3	0	0
$cdSum(v)$	99	0	0	0	0	0	0	0	3	0	0
Vertices from 12 to 21											
In v	12	13	14	15	16	17	18	19	20	21	–
$cdSize(v)$	3	0	0	0	0	0	0	0	0	0	–
$cdSum(v)$	3	0	0	0	0	0	0	0	0	0	–

Table 7.6: Final Calculation of SCCs in $C_D(v) = \widetilde{D}(v) \cap \widetilde{D}^R(v)$ in bottom up fashion.

v	$\lvert\widetilde{D}(v)\rvert$	$\lvert\widetilde{D}^R(v)\rvert$	$\lvert C_D(v)\rvert$	$\lvert C_A(v)\rvert$	$SCCValue(C_A(v))$
1	20	20	20	0	0
2	4	0	0	16	120
3	0	0	0	20	190
4	0	0	0	20	190
5	5	0	0	15	105
6	0	0	0	20	190
7	0	0	0	20	190
8	0	0	0	20	190
9	10	3	3	10	45
10	0	0	0	20	190
11	0	12	0	8	28
12	9	4	3	10	45
13	2	0	0	18	153
14	0	0	0	20	190
15	0	0	0	20	190
16	0	2	0	18	153
17	0	0	0	20	190
18	3	0	0	17	136
19	2	0	0	18	153
20	0	2	0	18	153
21	0	0	0	20	190

Table 7.7: SCC value in C_A (in our case $|V|= 21$) and $|C_A|= |V|-|\widetilde{D}(v)| -|\widetilde{D}^R(v)|+C_D-1$

v	$SCCValue$ $(\widetilde{D}(v))$	$SCCValue$ $(\widetilde{D}^R(v))$	$SCCValue$ $(C_D(v))$	$SCCValue$ $(C_A(v))$	$criticalValue$ (v)
1	99	99	99	0	99
2	6	0	0	120	126
3	0	0	0	190	190
4	0	0	0	190	190
5	10	0	0	105	115
6	0	0	0	190	190
7	0	0	0	190	190
8	0	0	0	190	190
9	12	3	3	45	57
10	0	0	0	190	190
11	0	32	0	28	60
12	9	4	3	45	55
13	1	0	0	153	154
14	0	0	0	190	190
15	0	0	0	190	190
16	0	1	0	153	154
17	0	0	0	190	190
18	3	0	0	136	139
19	1	0	0	153	154
20	0	1	0	153	154
21	0	0	0	190	190

Table 7.8: Final calculation of critical value of the vertices

7.4 Experimental Analysis

In this section we report experimental results of various algorithms for the critical node detection problem. Recall that we are given a directed graph $G = (V, E)$ and a parameter k, and our goal is to decrease the pairwise connectivity $f(G)$ of G as much as possible by deleting up to k vertices. For $k = 1$ (finding the most critical node), the algorithm of section 7.2.1 gives an exact linear-time solution. We compare the actual running time of this algorithm against the naive algorithm that tests $f(G \setminus v)$ for all strong articulation points v of G. Then, we turn to the general case, $k > 1$, where we compare the performance of various heuristics, both in terms of solution quality and of running time.

We wrote our codes in `C++` without using any external graph library. We used a uniform framework for the development and the same data structures for representing graphs. The source codes were compiled in `g++ v.4.9.3` with full optimization (flag `-O3`). We conducted the experiments on a 64-bit GNU/Linux machine running on Ubuntu 14.04LTS with Intel i5-3210*M* quard-core processor, first core has 2300 MHz and other three cores have 1200 MHz frequency. The machine has 4GB of *SODIMM* DDR_3 *synchronous* 1600 *MHz* of RAM, 32KB, 256KB, and 3MB of L_1, L_2 and L_3 caches respectively. All experiments were executed on a single core without using any parallelization and the CPU running time was measured with the `getrusage` function.

7.4.1 Algorithms and Heuristics

We compare the performance of two algorithms for computing the most critical node of a directed graph.

Naive (NAIVE). This is the straightforward algorithm to compute the most critical node of a given graph. For each strong articulation point v, it calculates the value of $f(G \setminus v)$ and chooses a vertex v that minimizes $f(G \setminus v)$.

Most Critical Node (MCN). This refers to the linear-time algorithm that we presented in Section 7.2.1.

For the general case of CNDP, we consider the heuristic that repeatedly removes the most critical node (computed by Algorithm MCN of Section 7.2.1) in the current graph. We refer to this heuristic also as MCN since there is no risk of confusion. For example, if we apply our MCN algorithm to the graph shown in Figure 7.10(*i*), then it will choose vertex a as the most critical node. The removal of vertex a will decompose the graph into 5 strongly connected components $\{d\},\{e\},\{f,g\},\{b,c\}$. (See Figure 7.10 (*ix*).) Therefore, the connectivity value of $G \setminus a$ is $f(G \setminus a) = \binom{2}{2} + \binom{2}{2} = 2$, which is indeed the highest possible fragmentation of the graph after the removal of any single vertex. In addition to that, we considered the following three heuristics which were among the top performers in the experiments on undirected graphs by Ventresca and Aleman [168]:

Maximum degree (MAX-DEG). This heuristic repeatedly removes a vertex of maximum degree. Since we deal with directed graphs, the degree of a vertex is the total number of incoming and outgoing edges. Clearly, we expect this to be a fast heuristic that produces low-quality solutions. For example, if we apply this heuristic to the graph presented in Figure 7.10 (*i*), then it will choose vertex c, which has the maximum degree in the graph. The resulting graph $G \setminus c$ has three SCCs $\{s,a,d,e\},\{f,g\},\{b\}$ (see Figure 7.10(*x*)) and $f(G \setminus c) = \binom{4}{2} + \binom{2}{2} = 7$.

PageRank (PR). PageRank is a well-known algorithm used to rank websites in search engines. This heuristic repeatedly removes the vertex with highest PageRank [130].

Betweenness Centrality (BC). Betweenness Centrality is a measure of a vertex centrality in a graph. It is equal to the number of shortest paths (between any pair of vertices) that pass through that vertex. In our experiments we used the algorithm of Brandes [24] which has $O(mn+n^2)$ running time.

We also designed the next two new heuristics:

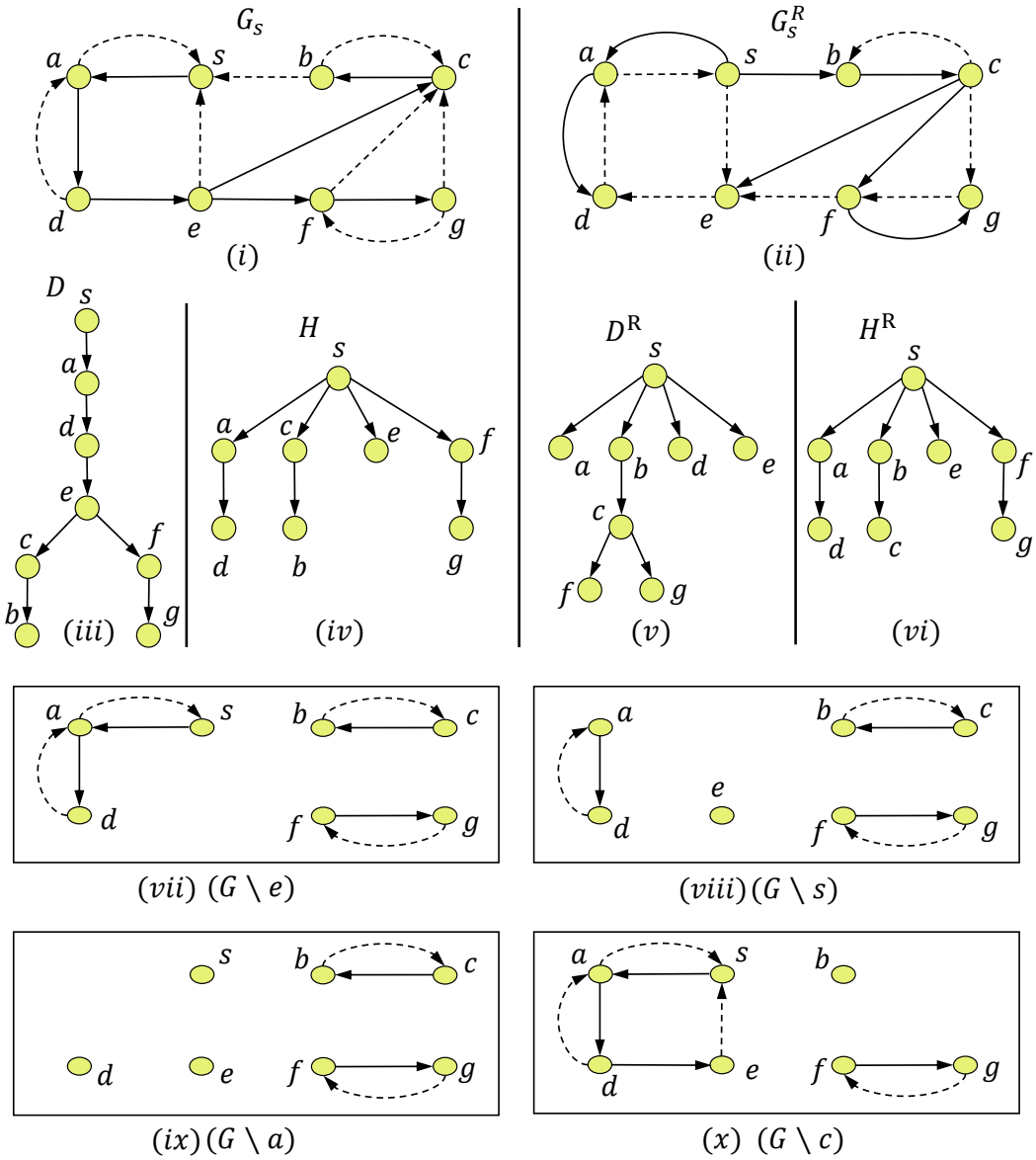


Figure 7.10: A flow graph G_s (i) and its reverse G_s^R (ii), their dominator trees D (iii) and D^R (v) and loop nesting trees H (iv) and H^R (vi). The corresponding digraph G is strongly connected. Solid edges in G_s and G_s^R represent the dfs trees edges with root s that generate H and H^R, respectively. During the dfs of G_s and G_s^R, the edges are examined in lexicographic order. The strongly connected components of $G \setminus v$, for $v \in \{e, s, a, c\}$, are shown in figures (vii), $(viii)$, (ix) and (x), respectively.

Maximum number of children in Loop Nesting Tree (LNT). This heuristic repeatedly removes a vertex with maximum number of children in the loop nesting tree.

The intuition for this heuristic is the following. Let v be any non-leaf vertex in H,

and let $v_1, v_2, \ldots, v_l$ be its children. If v has many children, then it is likely that several of their loops may induce a different SCC in $G \setminus v$. (Note, however, that this is not guaranteed, as $loop(v_i)$ and $loop(v_j)$ of two distinct children v_i and v_j of v, may still belong in the same SCC of $G \setminus v$.) For example, if we apply this heuristic to the loop nesting tree H (Figure 7.10(*iv*)) of G_s (Figure 7.10(*i*)), s has the maximum number of children. The removal of vertex s causes the graph to decompose into four SCCs (see Figure 7.10(*viii*)) and $f(G \setminus s) = 3$. As we shall see in the experimental results, this heuristic performed reasonably well in most of the tested instances.

Maximum number of children in Dominator Tree (DT). The main idea behind this heuristic is similar to LNT. The only difference is that it chooses a vertex with maximum number of children in the dominator tree instead of the loop nesting tree. For example, if we apply this heuristic to the dominator tree D (Figure 7.10(*iii*)) of G_s (Figure 7.10(*i*)) then it will choose vertex e. The resulting graph $G \setminus e$ has three SCCs (Figure 7.10(*vii*)) and $f(G \setminus e) = 5$. In our experiments, the performance of the DT heuristic turned out to be inferior of LNT in most instances.

Remark. Note that in all of the above heuristics, we may have several candidate vertices to choose for the next vertex to be removed. (It seems that we are giving an advantage to MCN.) In this case, we break ties by choosing a vertex uniformly at random among these candidates.

Datasets and Experiments performed. For our experiments, we used the same dataset that were used in the experiments in Chapters 4, 5 and 6. Most of them are taken from the 9^{th} *DIMACS* implementation challenge [43] and from the Stanford Large Network Dataset Collection [107]. The characteristics of those graphs in terms of CNDP are described in Table 7.9. We performed several experiments in order to highlight the characteristics of the different methods considered in our study. First, we compared the running times of MCN and NAIVE in order to assess the practical efficiency of MCN. Next, we performed an experiment where each heuristic removed 5% of the vertices in

the input graph G, according to its own strategy. We measured the running times of the different heuristics and the quality of the solutions they provided in this experiment, i.e., their effectiveness in decreasing the connectivity function $f(G)$. We finally measured how many vertices each heuristic needs to remove in order to decrease the connectivity function $f(G)$ of the input graph G by 50%. In all our experiments, we took the average of 10 different runs for each heuristic. Moreover, any execution running longer than 24 hours was terminated.

Graph	Type	n	m	Final $f(G)$	MCN	NAIVE
rome99	RN	3353	8859	5596195	0.003	0.107
p2p-Gnutella25	P2P	5153	17695	13212370	0.010	0.563
p2p-Gnutella31	P2P	14149	50916	99454356	0.023	3.868
web-NotreDame	WG	53968	296228	1243042947	0.079	30.362
soc-Epinions1	SN	32223	443506	512720305	0.143	40.505
USA-road-NY	RN	264346	730100	34635284926	0.289	1014.150
USA-road-BAY	RN	321270	794830	51526469796	0.378	2259.050
Amazon-302	PCP	241761	1131217	28983800958	0.948	4530.410
WikiTalk	SN	111881	1477893	6197128792	0.667	209.248
web-Stanford	WG	150532	1576314	8719604712	1.381	1206.280
Amazon-601	PCP	395234	3301092	77978742454	2.188	9890.940
web-Google	WG	434818	3419124	94192566065	3.905	17184.800
web-BerkStan	WG	334857	4523232	42556873058	1.230	2437.020

Table 7.9: The characteristics of the real-world graphs that we consider; n and m refers to the number of vertices and the number of edges, respectively. The graph types are encoded as: road network (RN), peer to peer (P2P), web graph (WG), social network (SN), production co-purchase (PCP). The graphs are sorted in increasing order according to their number of edges. Also, we reported the running time in seconds for MCN and NAIVE algorithms to find the most critical node. Initial $f(G) = \binom{n}{2}$ and final $f(G)$ is obtained after deleting a most critical node of G.

NAIVE vs MCN. In our first experiment, we aim at assessing the practicality of MCN, by measuring the time it takes MCN and NAIVE to remove the most critical node in a graph.

As it can be seen from Figure 7.11, it pays off to use a sophisticated algorithm, as MCN was consistently 2 to 4 orders of magnitude faster than NAIVE for all the graphs considered in our dataset. The full data for this experiment are reported in Table 7.9.

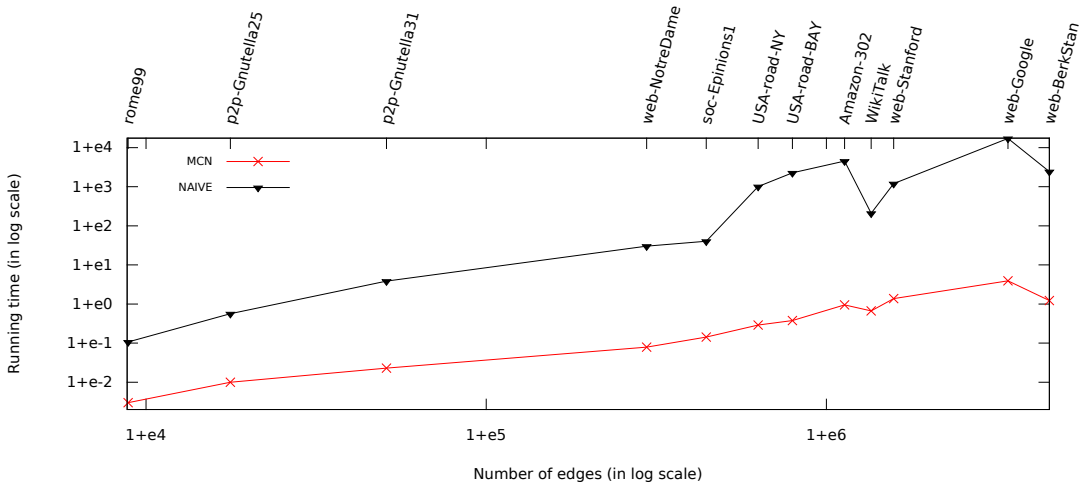


Figure 7.11: Comparison of running times to find the most critical node, between the NAIVE and MCN algorithms. Running times are in log scale of $\mu s/edge$. The exact values are presented in Table 7.9.

Final $f(G)$ values after removing the 5% of total vertices. In our second experiment we tried to assess the effectiveness of the heuristics, by measuring the decrease in $f(G)$ that resulted after removing the top 5% vertices according to each heuristic. Figure 7.12 (top) plots the final $f(G)$ achieved by the heuristics after the removal of those 5% most critical nodes. The plot does not include the PR and BC heuristics because either $f(G)$ decreased marginally or their execution was taking too long ($>$ 24 hours). The details of this experiments are reported in Tables 7.10 and 7.11. The analytical observation showed that, on average, the value of $f(G)$ was decreased by 85.04%, 75.18%, 64.88%, 60.92%, 12.03%, and 35.66% by MCN, LNT, MAX-DEG, DT, PR, and BC respectively. Therefore, on average, the MCN algorithm provides 13.11%, 31.08%, 39.58%, 6.81

times, and 2.04% times better solution quality than the heuristics LNT, MAX-DEG, DT, PR, and BC respectively. Among these other five heuristics, LNT produced the best results. Specifically, it achieves 23.40%, 15.89%, 4.7 times, and 1.9 times better results than DT MAX-DEG, PR, and BC respectively. Furthermore, we notice that MAX-DEG is 5.94%, 4.9 times and 48.71% better than DT, PR, and BC respectively, which may seem rather surprising. Among the worst three performers, DT generates the best results, as it decreases $f(G)$ by a factor of 4.7 and 1.7 with respect to PR and BC respectively.

In addition, we noticed that every heuristic produce a better results for some graphs. In particular, we observed this phenomena for web graphs ("Web Norte Dame", "Web Google", "web Stanford" and "Web Berkastan") of our datasets. It seems that those web graphs have a weak connectivity structure so that they can be easily destroyed after the removal of few number of vertices. (For the details report, please refer the Tables 7.10 and 7.11).

We also measured the total running time required to remove the specified 5% number of critical nodes. The results are reported in Tables 7.10 and 7.11 and plotted in Figure 7.12 (bottom). The experimental results showed that, on average, LNT has better performance in terms of running time than the other heuristics. It is 3.18 times faster than MCN, and 34.12%, 22.72% faster than DT and MAX-DEG respectively. We also note that MAX-DEG is 14% faster than DT. We conclude that MCN was able to achieve a very good fragmentation on our datasets, at the price of being about three times slower than our faster heuristic LNT.

Total number of critical nodes that need to be deleted to decrease $f(G)$ by 50%. In our third experiment paradigm, we evaluated all the heuristics in terms of the total number of critical nodes they need to remove from the graph to decrease the value of $f(G)$ by 50%. The results of this experiment are presented in Tables 7.12 and 7.13, and plotted in Figure 7.13 (top). As in our previous experiment, PR and BC perform poorly. They remove an excessive number of critical nodes, and for many instances were terminated as they took longer than 24 hours to complete the task of decreasing

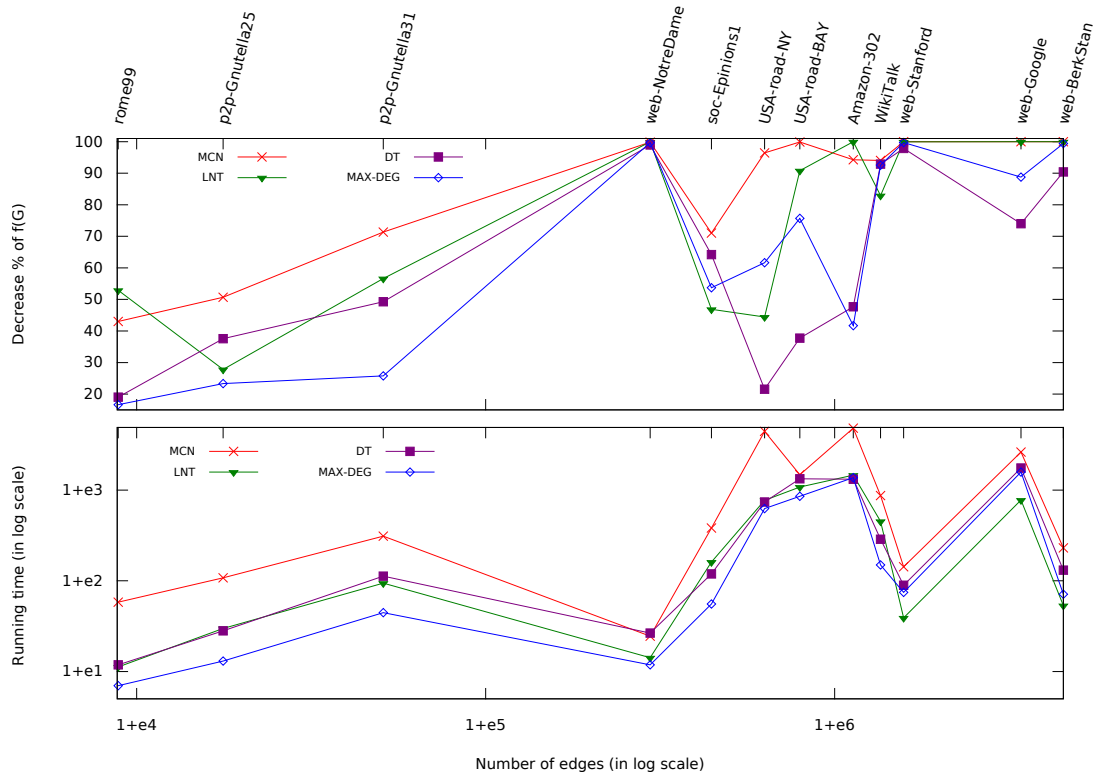


Figure 7.12: Decrease % of $f(G)$ values after the removal of 5% critical nodes of the total number vertices (top), and the total running time to delete those critical nodes (bottom).

$f(G)$ by 50%. Therefore, the plot in Figure 7.13 does not include these two heuristics. The best results, overall, were again achieved by MCN: On average, it removed a number of critical nodes that is less than the corresponding number of nodes removed by LNT, MAX-DEG and DT, by a factor of 1.74, 2.52 and 3.17, respectively. See Figure 7.13 (top). As like in 5% critical node removal case, we again notice that every heuristic give a good result in some graphs for this case as well. More precisely, this happened to most of the web graphs (“Web Norte Dame", “Web Google", “web Stanford" and “Web Berkastan") of our datasets. Therefore, again it shows that the web graphs of our datasets can be easily destroyed after the removal of few vertices (For the details report, see in Table 7.12).

Finally, we measured the running time required by each heuristic to decrease $f(G)$ by 50%. The experimental data are reported in Tables 7.12 and 7.13 and plotted in

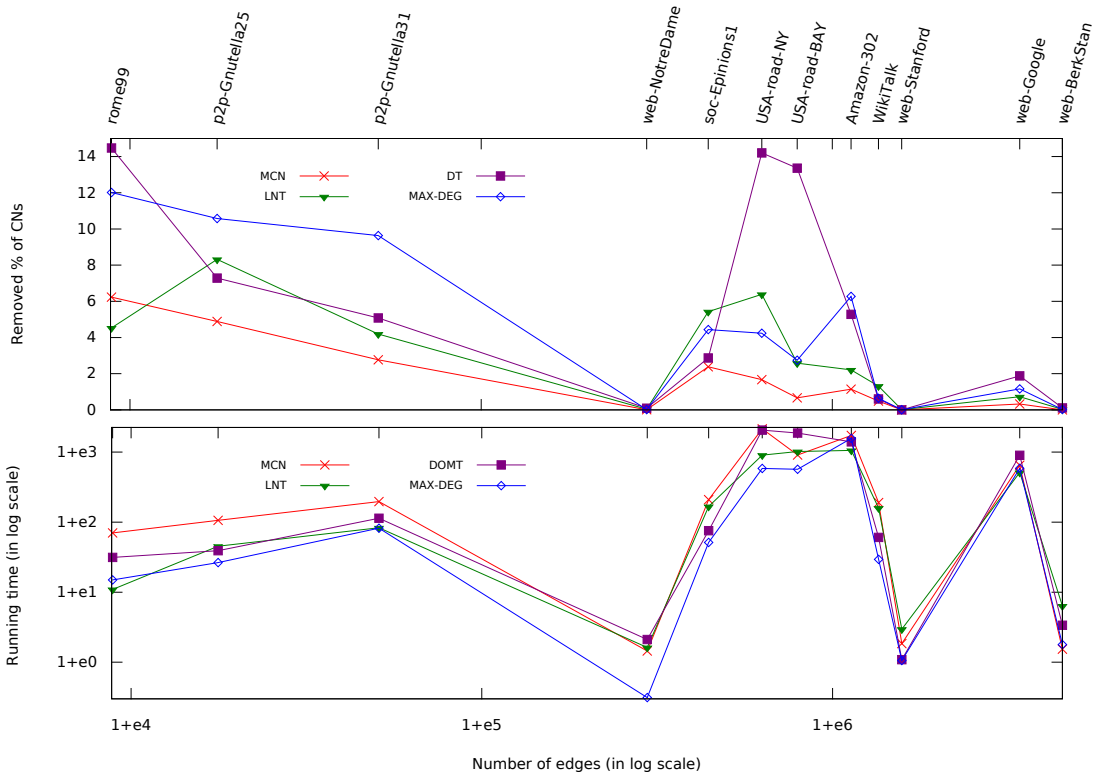


Figure 7.13: Comparison of removed % of critical nodes that need to be delete to decrease the value of $f(G)$ by 50% (top), Comparison of running times to delete those critical nodes, running time is showed in log scale of $\mu s/edge$ presented in Tables 7.12 and 7.13.

Figure 7.13 (bottom). We observe that, on average, LNT and MAX-DEG are the fastest, and they completed this task almost within the same time. Specifically, LNT is 30% and 39% faster than MCN and DT, respectively. We can also see that MCN pays off 12% over DT.

We conclude that on our datasets MCN was able to produce the desired fragmentation by removing a much smaller number of critical nodes than its competitors. Moreover, MCN achieved the required fragmentation without incurring a significant penalty on the running time, since on average it was only about 30% slower than the fastest heuristic.

Graphs	MCN		MAX-DEG	
	Time	Final $f(G)$	Time	Final $f(G)$
rome99	0.512	3202817	0.062	4683393.0
p2p-Gnutella25	1.910	6543153	0.230	10176816.0
p2p-Gnutella31	15.837	28694190	2.266	74291955.0
web-NotreDame	7.269	2472820	3.504	4757339.0
soc-Epinions1	169.322	150424430	24.614	240432817.0
USA-road-NY	3234.930	1244058481	456.339	13406602004.7
USA-road-BAY	1171.528	61195614	679.580	12542084237.0
Amazon-302	5445.483	1677930792	1551.250	17037852002.0
WikiTalk	1283.187	372200268	220.511	466731449.0
web-Stanford	225.311	1269730.1	117.499	34659085.7
web-Google	8964.042	1795860.8	5390.427	10631052114.0
web-BerkStan	1041.739	15226342	321.578	331876257.7
Graphs	**LNT**		**DT**	
	Time	Final $f(G)$	Time	Final $f(G)$
rome99	0.099	2645775.3	0.1050841	4550311.8
p2p-Gnutella25	0.527	9582277.7	0.4974243	8284945.4
p2p-Gnutella31	4.804	43402402.9	5.719832	50773962.5
web-NotreDame	4.206	1162775.8	7.852506	15036096
soc-Epinions1	71.333	275818164.1	52.73888	185733546.5
USA-road-NY	562.951	19417764438	539.8804	27408935605
USA-road-BAY	858.288	4739852019	1057.7161	32136610238
Amazon-302	1651.789	5312281	1488.914	15298675869
WikiTalk	668.444	1069698408	424.2926	444865777.9
web-Stanford	61.542	5346952.1	140.8944	243992686.4
web-Google	2649.884	2430948.4	5973.944	24586821491
web-BerkStan	241.218	15972686	591.7983	5388369331

Table 7.10: Running time and efficiency details of the heuristics MCN, MAX-DEG, LNT and DT to decrease the value of $f(G)$ by removing the 5% critical nodes of total vertices, execution running longer than 24 hours (86400 seconds) were terminated.

Graphs	PR		BC	
	Time	Final $f(G)$	Time	Final $f(G)$
rome99	0.573	4884376	25.145	3532319
p2p-Gnutella25	1.958	11695866	624.720	9748320
p2p-Gnutella31	11.768	89171335	14107.600	43389814
web-NotreDame	40.757	1187213578	5988.960	1187242406
soc-Epinions1	333.320	466574893	$>$ 24h	N/A
USA-road-NY	4921.500	31279714089	$>$ 24h	N/A
USA-road-BAY	9090.710	46331398366	$>$ 24h	N/A
Amazon-302	10960.800	26328129005	$>$ 24h	N/A
WikiTalk	4787.530	5626218004	$>$ 24h	N/A
web-Stanford	8198.780	9608941068	60304.000	6859065625
Amazon-601	45517.300	70465771155	$>$ 24h	N/A
web-Google	$>$ 24h	N/A	$>$ 24h	N/A
web-BerkStan	$>$ 24h	N/A	$>$ 24h	N/A

Table 7.11: Running time and efficiency details of the heuristics PR, BC to decrease the value of $f(G)$ by removing the 5% critical nodes of total vertices, execution running longer than 24 hours (86400 seconds) were terminated.

Graphs	MCN			MAX-DEG		
	Time	CNs	CNs%	Time	CNs	CNs%
rome99	0.622	209	6.233	0.133	403	12.019
p2p-Gnutella25	1.879	252	4.890	0.468	545	10.576
p2p-Gnutella31	9.996	392	2.771	4.175	1363	9.633
web-NotreDame	0.429	5	0.009	0.093	8	0.015
soc-Epinions1	92.851	768	2.383	22.806	1430	4.438
USA-road-NY	1571.749	4413	1.669	427.597	11202.1	4.238
USA-road-BAY	722.032	2137	0.665	452.241	8824	2.747
Amazon-302	1962.022	2771	1.146	1777.947	15160	6.271
WikiTalk	280.789	547	0.489	43.538	719	0.643
web-Stanford	2.91	3	0.002	1.67	14	0.009
web-Google	2223.312	1416	0.326	2006.759	5063	1.164
web-BerkStan	6.936	5	0.001	8.082	79	0.024
Graphs	**LNT**			**DT**		
	Time	CNs	CNs%	Time	CNs	CNs%
rome99	0.097	152	4.533	0.278	485.3	14.474
p2p-Gnutella25	0.802	428.7	8.319	0.693	375.4	7.285
p2p-Gnutella31	4.229	593.5	4.195	5.795	719.2	5.083
web-NotreDame	0.481	23	0.043	0.624	51.8	0.096
soc-Epinions1	74.302	1748	5.425	33.481	925.7	2.873
USA-road-NY	662.274	16886.8	6.388	1508.227	37553.1	14.206
USA-road-BAY	809.933	8310.4	2.587	1488.333	42929.6	13.362
Amazon-302	1197.401	5351.5	2.214	1586.308	12767.8	5.281
WikiTalk	235.568	1474.7	1.318	89.458	675	0.603
web-Stanford	4.665	23	0.015	1.713	4	0.003
web-Google	1748.406	3149.1	0.724	3074.473	8169.4	1.879
web-BerkStan	28.821	105	0.031	15.219	356	0.106

Table 7.12: Running time and efficiency details of MCN algorithm and the heuristics MAX-DEG, LNT and DT to decrease the value of $f(G)$ by 50%, computation running longer than 24 hours (86400 seconds) were terminated.

Graphs	PR			BC		
	Time	CNs	CNs%	Time	CNs	CNs%
rome99	2.009	698	20.817	27.855	338	10.081
p2p-Gnutella25	8.401	1228	23.831	850.125	403	7.821
p2p-Gnutella31	58.176	3733	26.383	13071.000	596	4.212
web-NotreDame	1269.170	12446	23.062	1901.440	392	0.726
soc-Epinions1	1738.610	9250	28.706	$>$ 24h	$>$ 24h	$>$ 24h
USA-road-NY	17381.000	67585	25.567	$>$ 24h	$>$ 24h	$>$ 24h
USA-road-BAY	35393.700	71333	22.203	$>$ 24h	$>$ 24h	$>$ 24h
Amazon-302	57202.600	69881	28.905	$>$ 24h	$>$ 24h	$>$ 24h
WikiTalk	26322.900	32423	28.980	$>$ 24h	$>$ 24h	$>$ 24h
web-Stanford	46097.200	36357	24.152	60305.200	9166	6.089
web-Google	$>$ 24h	N/A	N/A	$>$ 24h	N/A	N/A
web-BerkStan	$>$ 24h	N/A	N/A	$>$ 24h	N/A	N/A

Table 7.13: Running time and efficiency details of MCN algorithm and the heuristics PR and BC to decrease the value of $f(G)$ by 50%, computation running longer than 24 hours (86400 seconds) were terminated.

8

Handwritten Signature Verification

8.1 Introduction

Biometrics examine the physical or behavioral traits that can be used to determine a person's identity. Biometric recognition allows for the automatic recognition of an individual based on one or more of these traits. This method of authentication ensures that the person is physically present at the point-of-identification and makes unnecessary to remember a password or to carry a token. The most popular biometric traits used for authentication are face, voice, fingerprint, iris and handwritten signature.

In our study, we focus on handwritten signature verification (HSV), which is a most common, natural, and trusted method for user identity verification. HSV can be classified into two main classes, based on the device used and on the method used to acquire data related to the signature: online and offline signature verification. Offline methods process handwritten signatures taken from scanned documents, which are, therefore, represented as images. This means that offline HSV systems only process the 2D spatial representation of the handwritten signature (i.e., its shape). Conversely, online systems use specific hardware, such as pen tablets, to register pen movements during the act of signing. For this reason, online HSV systems are able to process dynamic features of signatures, such as the time series of the pen's position and pressure.

Online HSV has been shown to achieve higher accuracy than offline HSV [93, 95, 137] but unfortunately it suffers from several limitations. In fact, handwritten signatures are usually acquired by means of digitizing tablets connected to a computer, because common low-end mobile devices (such as mobile phones) may not be able to support the verification algorithms (due to their hardware configuration capacity to compute the algorithm) or may be too slow to run the verification algorithm (due to limited

computational power). As a result, the range of possible usages of the verification process is strongly limited by the hardware needed. To overcome this limitation, one needs techniques capable of verifying handwritten signatures acquired by smartphones and tablets in mobile scenarios with very high accuracy. Online HSV systems (such

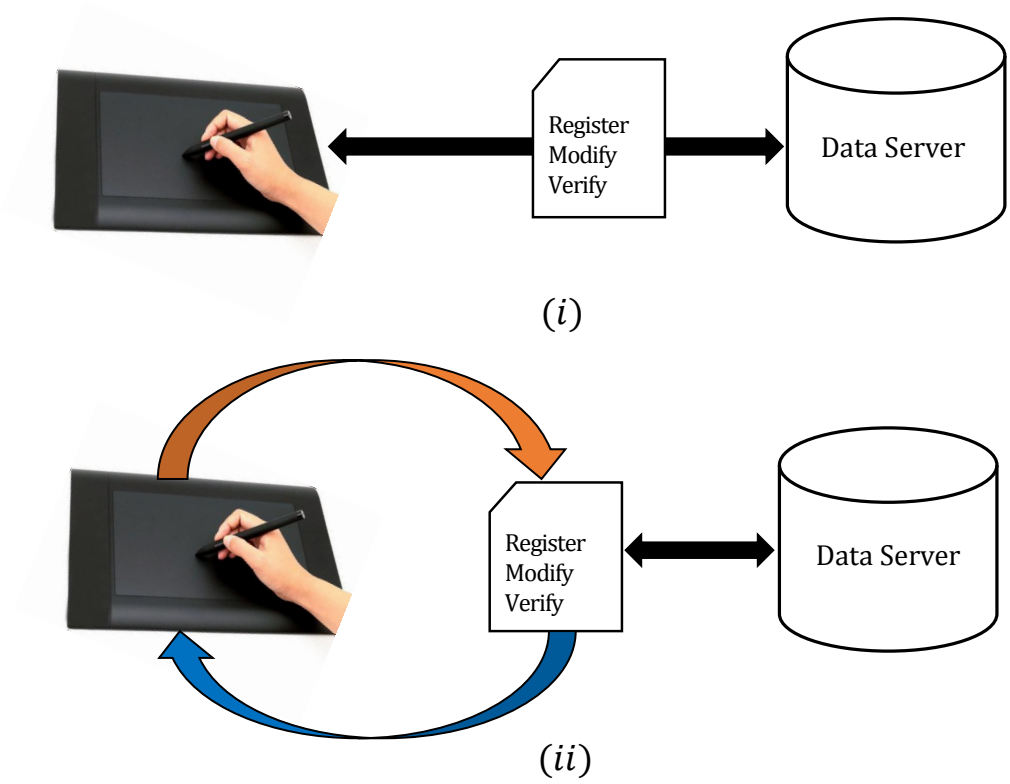


Figure 8.1: Overview of the Handwritten Signature Verification

as [41, 111, 151, 165, 176]) are able to address only partially these issues: they are supported by mobile devices, but they are not inherently designed for common low-end mobile devices such as mobile phones; several approaches make use of pen pads (special purpose hardware for handwriting), signature tablets (special purpose desktop and mobile hardware for signing), interactive pen displays (complete instruments for working in digital applications), Kiosk systems and PC Tablets.

As for the online HSV systems described in [23, 104, 120], even if experiments related to online HSV were carried out on low-end devices in order to evaluate the verification accuracy, no analysis addressing the computational time is used in the algo-

rithm design (which is particularly important, due to the limited computational power of mobile devices).

Normally, low-end devices take an input stream of thandwritten signature and then send it to the server for verification, which has powerful hardware configuration. After that, server analyzed the input stream and send the response flag to the corresponding devices, which determines the permission for the respective user as shown in Figure 8.1-(i). In our security model, the client, i.e. low end device, itself checks the user identity through an application and hence not require to send the data to server for the verification as shown in 8.1-(ii). Moreover, if the user modifies his/her signature, then it will immediately notify the changes to server. The main goal of our work is to address the above challenges by designing a new online HSV system that can be run on low-end devices too. The novelties of our approach lie mainly in the following aspects. First, we propose a method for the verification of signature dynamics which is compatible to a wide range of low-end mobile devices (in terms of computational overhead and verification accuracy) so that no special hardware is needed. Secondly, our new method makes use of several technical features that, to the best of our knowledge, have not been previously used for handwritten signature recognition. Finally, in order to assess the verification accuracy of our HSV system, along with the average computational time, we conduct an experimental study whose results are reported for different data sets of signatures. A preliminary version of this Chapter was presented at the 2^{nd} International Conference on Information Systems Security and Privacy [131]. Moreover, the presented algorithm is published as a book Chapter in "Communications in Computer and Information Science Series" by Springer Publications [133].

8.2 Features of the Online Signatures

8.2.1 Dynamics

An online handwritten signature on a digital device is a series of points, and each point is represented by a vector in four dimensions, X, Y, Pressure and Time. We define these

series of points as dynamics of the signature. When the user writes the signature, s/he might do pen-up and pen-down moves rather than moving the pen tip continuously. We define a stroke (ST) as the trajectory of a pen tip between a pen-down and a pen-up. A signature can be can be partitioned into multiple strokes as shown in Figure 8.2 and in Figure 8.3.

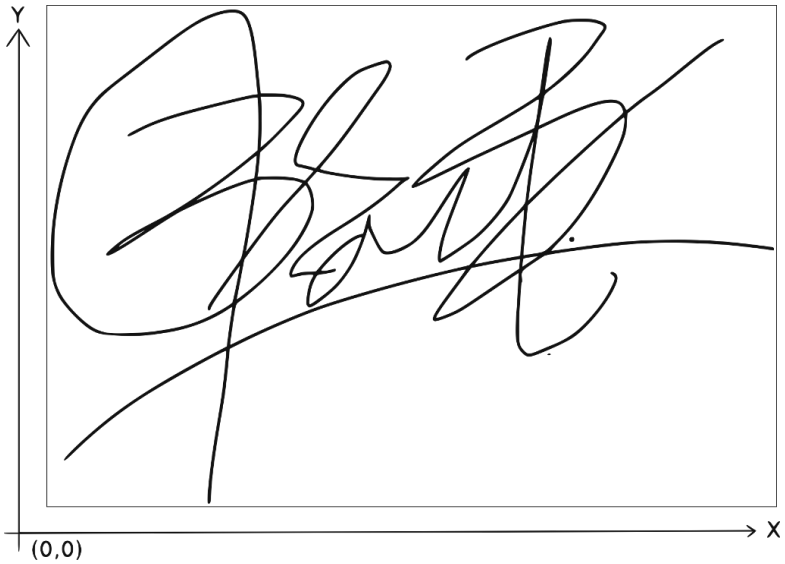


Figure 8.2: Handwritten Signature

- **X,Y:** The x and y coordinates of each sampled point that is captured from the device screen. Since the user may put his/her signature on any region of the screen, a translated mean origin point is computed and all the X-Y coordinates are translated into the new coordinates with the reference of that new origin point.
- **Pressure (P):** The pressure with which the screen is pressed. When the pen is down, or when the user draws the line continuously, then the pressure value becomes 1 (maximum value) for that points. Similarly, when the pen is released from the screen, then the pressure value becomes 0 (minimum value) for that specific point.
- **Time Series (TS)**: The sequence of equispaced sampling time instants. The sampling period, i.e., the time difference between two consecutive samples, is constant and exactly equal to the inverse of the device sampling frequency.

Figure 8.3: Strokes of the signature shown in Figure 8.2, blocks are in left to right and top to bottom order.

8.2.2 Features

We use the features to study the structure of the signature and of its strokes from the various perspectives. Each feature is important for both the registration and verification steps. Sections 8.3.1 and 8.3.2 explain why they are important and how they do the work for the signature registration and verification steps. Features are computed over the dynamics by means of mathematical tools as explained in the following subsection.

8.2.2.1 Features of the Signature

(i) **Pen-Up number**: Total number of pen-ups done by the user while writing his/her full signature.

(ii) **Path Length(PL)**: The total path length travelled by the user pen tip during the signature creation. The device sampling frequency gives the value of all dynamics in equal interval of time, and the Euclidean distance formula calculates the distance between two consecutive points of each interval. So, the total path length of the signature (PL) is defined by the equation

$$\sum_{i=2}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}, \text{ where } x_i \in \text{X and } y_i \in \text{Y}. \quad (8.1)$$

(iii) **Diagonal Length(DL)**: We take the maximum(x_{max}, y_{max}) and minimum(x_{min}, y_{min}) points in X, Y and then by using the Euclidean distance formula for two-dimensional plane, the equation that defines the diagonal length (DL) is

$$\sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2} \quad (8.2)$$

(iv) **Time Length(TL)**: The total time in milliseconds that has taken by the user to write his/her complete signature (the time duration between the first pen down and last pen up).

(v) **Mean Speed(MS)**: The average speed of the signature. We have four different dynamics sets (X, Y, TS and P) of equal size. All the points in these sets are sequential and tracked on the same time interval from the device's screen. We calculate the velocity between two consecutive points and then make a sum. After that, we divide the total sum of the velocities by the total number of points. The mean speed (MS) is defined by the equation

$$\frac{1}{n} \sum_{i=2}^{n} \frac{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}}{(t_i - t_{i-1})}, \text{ where } x_i \in \text{X}, y_i \in \text{Y and } t_i \in \text{TS}. \quad (8.3)$$

(vi) **Covariance-XY(CXY)**: In order to measure the scatteredness of the points in signature path, we calculate the Covariance-XY(CXY) by using the statistical variance equation

$$\frac{1}{n} \sum_{i=1}^{n} \sqrt{(x_i)^2 + (y_i)^2}, \text{ where } x_i \in \text{X and } y_i \in \text{Y}. \quad (8.4)$$

(vii) **Vector Length Ratio (VLR)**: Each point of the signature captured by the acquiring device has a 4 dimensional representation (X, Y, TS and P), but for the Vector length ratio (VLR), we only focus on x-axis and y-axis and calculate the sum of

the length of all the vectors drawn from the origin to each point of (X,Y). Finally, the sum is divided by (PL). So, the VLR is given by the equation

$$\frac{1}{\text{PL}}\sum_{i=1}^{n}\sqrt{(x_i - x_{origin})^2 + (y_i - y_{origin})^2}, \text{ where } x_i \in \text{X and } y_i \in \text{Y}. \tag{8.5}$$

8.2.2.2 Features of the Strokes

As previously mentioned, a stroke is a part of a signature. So, it is the subsequence of a signature sequence and has the same features and dynamics that the signature has (except the pen-up number because it is a trajectory between the pen down and pen up). Our goal is to find the ratio between each stroke's feature to the corresponding signature's feature. That gives us an idea how much amount (regarding feature's unit) does a single stroke takes to form the full signature.

(i) **Path Length Ratio (PLR):** The ratio between the total path length of the stroke over the total path length of the signature that is given by the equation:

$$\frac{\text{PL of the stroke}}{\text{PL of the signature}} \tag{8.6}$$

(ii) **Time Length Ratio (TLR):** The ratio between the total time taken by the user to write the stroke (part of the signature) over the total time length of the signature. It is given by the equation:

$$\frac{\text{TL of the stroke}}{\text{TL of the signature}} \tag{8.7}$$

(iii) **Diagonal Length Ratio (DLR):** We find a single block for the whole signature after having the maximum(x_{max}, y_{max}) and minimum(x_{min}, y_{min}) points in X, Y. Similarly, we find the same kind of block for a stroke. After that, the ratio between the diagonal length of a stroke block over the signature block is calculated by the equation

$$\frac{\text{DL of the stroke}}{\text{DL of the signature}} \tag{8.8}$$

(iv) **Mean Speed Ratio (MSR):** The user may move his pen tip with different speed to write the signature. Most of the users write the signature with different starting and ending speed. So, the mean speed is different for each stroke. Our target is to calculate the ratio between the mean speed of a stroke and the mean speed of the full signature, that is given by the following equation:

$$\frac{\text{MS of the stroke}}{\text{MS of the signature}} \tag{8.9}$$

(v) **Covariance XY Ratio (CXYR):** It gives the scatteredness of the point within a block. So, in some stroke the points may be close to each other and dense as well. Whereas in some block may not be. So, we calculate the ratio of the scatteredness of the points in each block over the full signature by using the following equation:

$$\frac{\text{CXY of the stroke}}{\text{CXY of the signature}} \tag{8.10}$$

(vi) **Stroke Vector Length Ratio (STVLR):** It gives the ratio between the vector length ratio of a stroke over the vector length ratio of a full signature by means of the following expression:

$$\frac{\text{VLR of the stroke}}{\text{VLR of the signature}} \tag{8.11}$$

8.3 The Signature Verification Algorithm

We describe next the registration and verification process from a technical perspective.

8.3.1 Signature Registration Phase

In this phase, the system takes the user's genuine signatures as input and generates the biometric template of the features with the following steps.

8.3.1.1 Acquisition and Pre-processing

In the acquisition phase, the user has to write the signatures with the same number of pen ups for three rounds as input. In each round, whenever the signature is captured from the screen, the pre-processing starts immediately. Then, the system eliminates the noise, normalizes the path and all kind of features are calculated and then checked with the features of existing signatures. In the checking process, the signature should have exactly the same number of pen ups. The area covered by the signature and its length depend on the screen sizes. Since various devices may have different screen sizes, the feature values PL, DL, TL, MS, CXY, VLR, PLR, DLR, TLR, MSR, CXYR, STVLR depend on the screen pixel density.

In addition, it is almost impossible to write the signature with the same dynamics and features as before. But it is possible to write a signature, that is similar to the previous signature up to a certain percentage. So, for the very first time, the user is totally free to write the signature as he/she wants. But at the second time, the signature has to match the first signature up to a certain level. Similarly, the third signature has to match the first and the second signature up to the certain tolerance factor. For example, during the first signature, the user may write a vertical line and at the second time, instead of writing a vertical line he/she may write a horizontal line with same speed, length and time. Topologically both of those lines are similar, but in practice, they are different. The X-length, Y-length and diagonal length take the control and reject the second signature. For this reason, each feature should be similar to the features of existing signatures and their corresponding strokes up to a certain tolerance factor. Otherwise, the user has to write the signature again for that round.

8.3.1.2 Template Generation and Store

Once the pre-processing is completed, then the system has the features and dynamics of three different signature samples. So in this step, we calculate the average of each feature as follows: $\frac{1}{3}\sum_{i=1}^{3} Feature_{(i)}$ and create an interval for each feature with its average value up to a certain threshold factor.

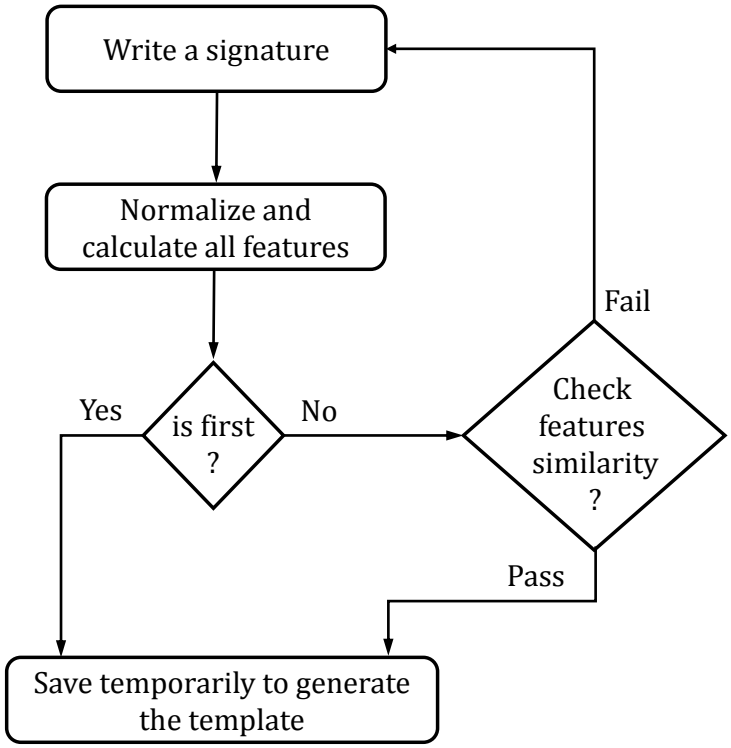


Figure 8.4: Flowchart for signature registration.

We also use the dynamic time warping (DTW) for the template generation and signature verification process. In time series analysis, dynamic time warping [124] is an algorithm for measuring similarity between two temporal sequences which may vary in time or speed. In addition it has also been used for partial shape matching applications. Moreover, it has been successfully used in literature for both on-line and off-line HSV [55, 122, 134] (For the details of DTW algorithm, please refer to Appendix Section A.4).

There are different kinds of algorithms to check the similarity between the sequences like Frèchet distance but we use DTW. This is because of its high accuracy and efficiency (in terms of computational time) which is well suited for our algorithm that is specially designed for mobile devices.

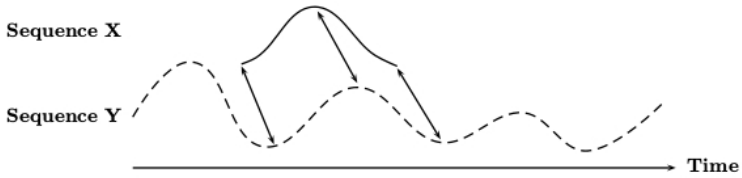


Figure 8.5: Maximum match between two different time series by using DTW, source [124]

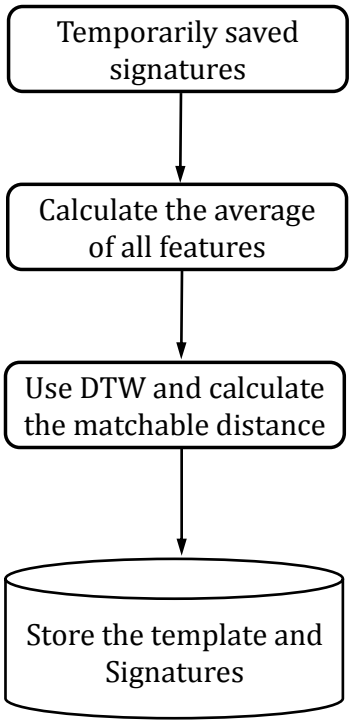


Figure 8.6: Flowchart for Template generation phase.

8.3.2 Signature Verification Phase

In the verification phase, the system makes the decision on whether the claimed signature is genuine or forged. We already calculated the accepted interval for each feature in the template generation phase. The steps for the verification process are as follows:

8.3.2.1 Check with Global Features of the Signature

(i) Check with Pen up Number: If the claimed signature has a different number of pen-ups, then it will be rejected.

(ii) Check with all features of the signature, PL, DL, TL, MS, CXY and VLR respectively:

If each feature of the claimed signature does not fall in its corresponding interval generated by template generation step, then it will be rejected.

8.3.2.2 Check with Features of the Strokes

The claimed signature may have more than a single stroke. For every stroke, the system checks all the features, TLR, DLR, MSR, CXYR and STVLR. Each feature should lie in the corresponding interval that was generated at the template generation phase. The system counts how many strokes pass the test. If this percentage is lower than a certain threshold then the signature is rejected.

8.3.2.3 Check with DTW

If the claimed signature passes all the above verification steps, then DTW is applied on it as follows.

Let m be the total number of strokes in a single signature. Then by using the feature of each signature, the following m-dimensional vector is computed. Let the i^{th} stroke (related to feature f of signature) of the j^{th} signature in a 1D time series be denoted as S^i_j. $\mathsf{DTW}(S^i_j, S^i_k)$ denotes the 1D DTW method applied to the i^{th} segments of the j^{th} and k^{th} signatures.

$$\left\| \begin{matrix} f^1 \\ f^2 \\ \dots \\ f^m \end{matrix} \right\| = \left\| \begin{matrix} \frac{\mathsf{DTW}(S^1_1, S^1_2) + \mathsf{DTW}(S^1_1, S^1_3) + \mathsf{DTW}(S^1_2, S^1_3)}{3} \\ \frac{\mathsf{DTW}(S^2_1, S^2_2) + \mathsf{DTW}(S^2_1, S^2_3) + \mathsf{DTW}(S^2_2, S^2_3)}{3} \\ \dots \\ \frac{\mathsf{DTW}(S^m_1, S^m_2) + \mathsf{DTW}(S^m_1, S^m_3) + \mathsf{DTW}(S^m_2, S^m_3)}{3} \end{matrix} \right\|$$

When $\left\|f\right\|$ vector is computed for each feature f, we get a $\left\|X\right\|$ vector (x coordinates), a $\left\|Y\right\|$ vector (y coordinates), a $\left\|P\right\|$ vector (P coordinates), and a $\left\|T\right\|$ vector (TS coordinates).

Finally, we combine the metrics with the following sums,

$$\left\|\begin{matrix} d^1 \\ d^2 \\ ... \\ d^m \end{matrix}\right\| = \left\|\begin{matrix} X^1+Y^1+P^1+...+T^1 \\ X^2+Y^2+P^2+...+T^2 \\ ... \\ X^m+Y^m+P^m+...+T^m \end{matrix}\right\|$$

The output distance vector $\left\|d\right\|$ represents the "distance" among the three signatures. The whole process is repeated twice; the first time between the genuine registered signatures ($\left\|d_g\right\|$ as output, which is already calculated during the template generation phase) and the second time between the claimed signature and registered signatures ($\left\|d_v\right\|$ as output). In the template generation phase, we also calculated the interval by using the threshold factor in $\left\|d_g\right\|$. So if $\left\|d_v\right\|$ does not lie in that interval, then the claimed signature is rejected, otherwise accepted.

Now, we describe our algorithm. We present several samples of genuine and forges signatures by Figure 8.8. First, the total pen up number is considered. If the signature to be verified has a different number of pen-ups, then the signature is assumed to be a forgery. If the forger writes the signature very fast then he/she produces the better line quality with less accuracy. Similarly, if he/she writes very slowly then the signature may be more accurate but the line quality is poor, and the time length is unnaturally high. So in either case, our algorithm works because of **TL**.

During the template generation phase, the user is totally free to write the genuine signature on the device screen. So, we calculate the **Features** and **DL** for his/her signature from the device perspective. Now, if the forger writes the signature on all the available area then it has a very high value in **Features** and **DL**. Similarly, if he/she writes in a small area then it will have very low **Features** and **DL**. In either case the algorithm works to rejects it.

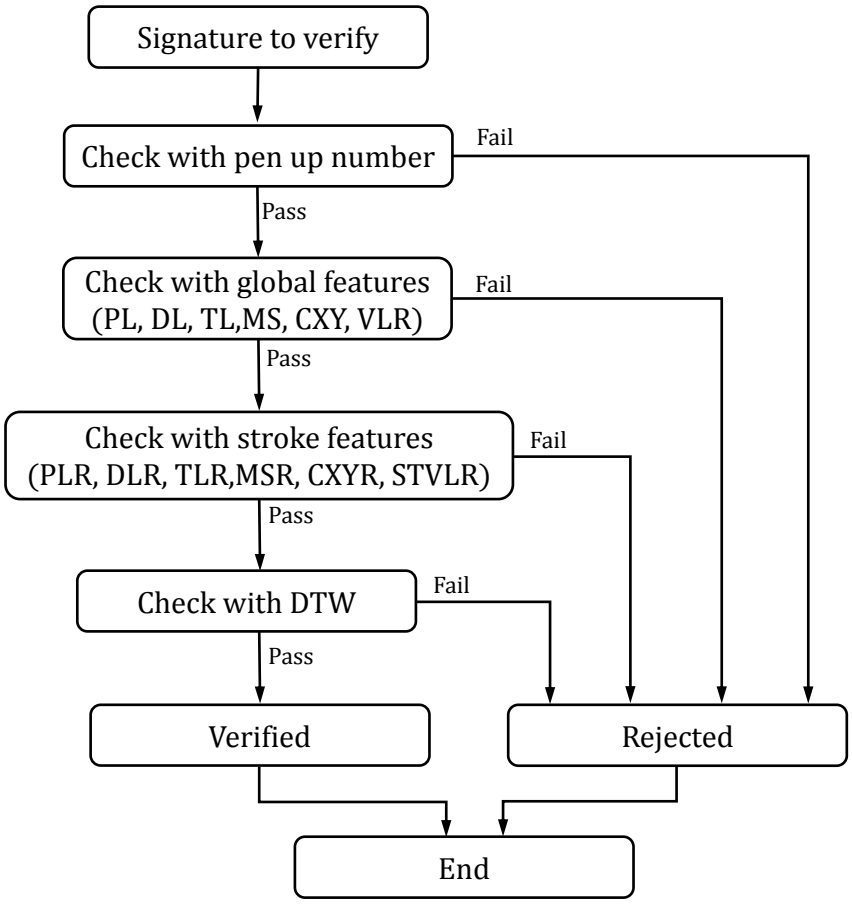


Figure 8.7: Flow chart for the verification process.

Even if the forger writes the signature within a given area with expected length and time. Still, it is tough to write the signature with tolerable **MS**. Whereas the real user can write his/her signature within the acceptable interval of **MS**. So our algorithm can easily recognize the forger's speed and rejects his/her attempt.

CXY measures the scatter value of all points in a signature that are distributed on the device screen. So, even if the forger writes a signature matching **PL, TL, MS**, it is unlikely to match the distribution of the points with the genuine signatures. So, whenever his/her signature does not match the **CXY** then our algorithm detects that it is a forgery. The **VLR** tides the signature points with its path length and measures the trend of the points and its quality. If the signature to be checked has a different trend as

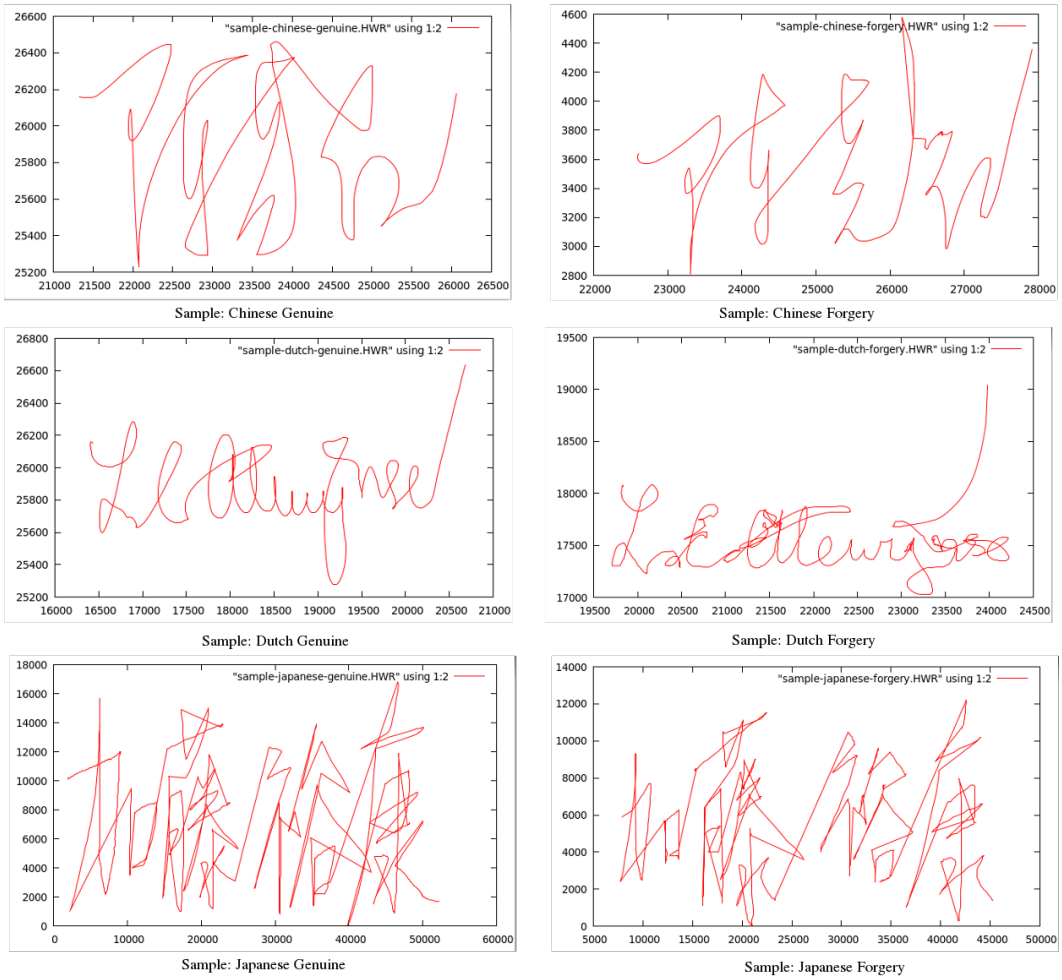


Figure 8.8: Samples of genuine and forgery signatures

compared to the registered template, then it is rejected.

A signature may have multiple strokes, and each stroke has the features (**PL, TL, DL, MS, CXY and VLR**), because it is just a subsequence of the signature sequence. The features of each stroke are different from each other. So, our algorithm calculates all the features of each stroke and then finds out its ratio with respect to the whole signature. So, even if the forger is able to write a signature that passes the all global features test successfully, still, if it does not pass the stroke ratio verification process, that includes the (**PLR, TLR, DLR, MSR, CXYR and STVLR**) then the signature is rejected.

Finally, if the forgery passes all the global and stroke feature tests, then, the signa-

ture undergoes DTW testing. DTW compares the similarity between two sequences. We find out two distance vectors: $\|d_g\|$ represents the "distance" among the three genuine signatures, while $\|d_v\|$ represents the "distance" among three genuine with claimed signature. If $\|d_v\|$ does not lie in the interval which is calculated on the basis of $\|d_g\|$ by a certain threshold at the template generation phase, then it is rejected as a forgery.

8.4 Experiment

In this section, we present the implementation prototype and the experimental results concerning identity verification with our system.We implemented our algorithms in Java and tested on Android version ≥ 4.0. Figure 8.9 represent the class diagram for implementation prototype. The accuracy of a recognition algorithm is generally measured in terms of two potential types of errors: false negatives (*fn*) and false positives (*fp*). *fp* are cases where a claimed identity is accepted, but it should not be, while *fn* are cases where a claimed identity is not accepted, while it should be. The frequency at which false acceptance errors occur is denoted as False Acceptance Rate **(FAR)**, while the frequency at which false rejection errors occur is denoted as False Rejection Rate **(FRR)**. Two metrics building on true/false positives/negatives (*tp,fp,tn,fn*) are widely adopted: precision and recall. Recal (**RCL**) = $tp/(tp+fn)$ is the probability that a valid identity is accepted by the system (i.e., true positive rate) while precision (**PCR**) = $tp/(tp+fp)$ is the probability that a claimed identity which is accepted by the system is valid. F-measure **(FMR)** = $(2 \times prec \times recall)/(prec+recall)$, which is the harmonic mean of precision and recall, that combines both metrics into a global measure.

TF	PCR	RCL	FMR	FRR	FAR
34%	0.983	0.919	0.943	0.008	0.008
35%	0.969	0.934	0.945	0.014	0.065
36%	0.953	0.936	0.936	0.021	0.063

Table 8.1: PCR, RCL, FMR, FAR and FRR as a functions of a tolerance factor (**TF**).

A threshold on the similarity score must be identified for determining whether two

signatures are similar (accept the identity) or significantly different (reject the identity). The higher the threshold, the higher the precision (i.e., the lower the risk of accepting invalid identities). However, a high threshold also decreases the recall of the system (i.e., the higher the risk to reject valid identities).

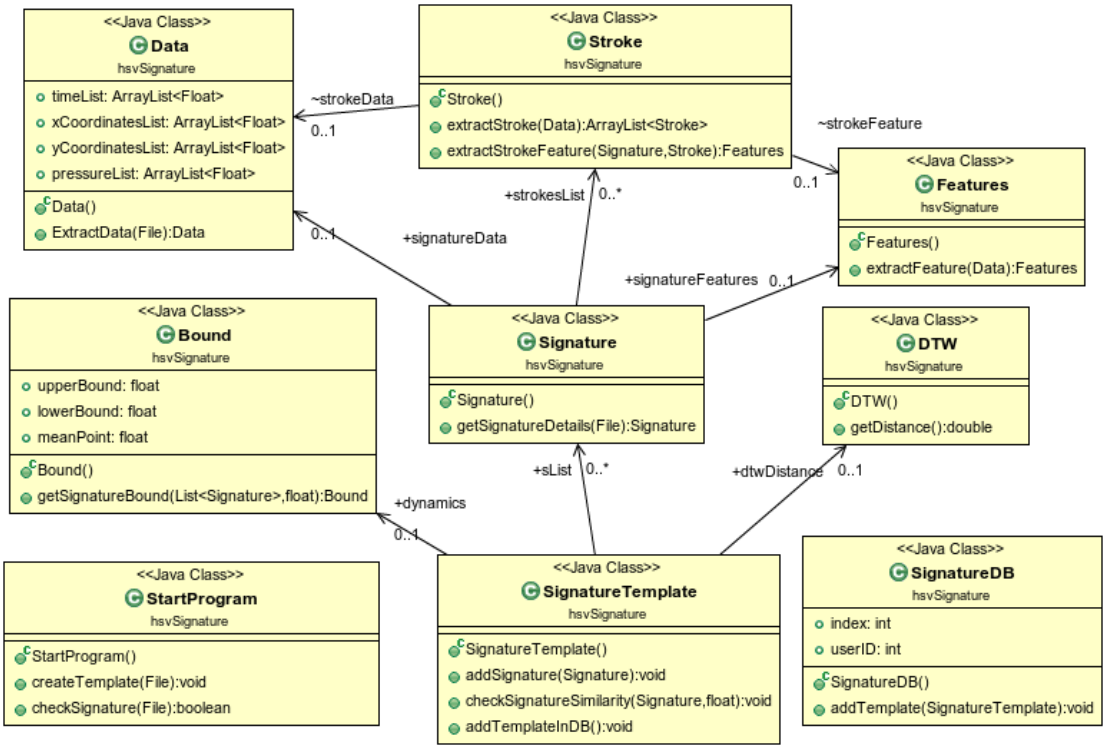


Figure 8.9: Implementation prototype of our algorithm

The performance of the proposed scheme has been assessed in terms of PCR, RCL, FAR, FRR and FMR on three different datasets: on the SigComp2011 Dutch and Chinese datasets [110]; on the SigComp2013 Japanese dataset [113]. We start by describing the experimental set-up. Several mobile devices have been involved in our experiments (i.e., Google Nexus 5, GalaxyS2, XperiaZ2 and ZTE Blade A430), along with several standard datasets. The specification of the datasets involved are as follows:

- The SigComp2011 [110] competition involved (online) dutch and chinese data. The purpose of using these two data sets was to evaluate the validity of the participating systems on both Western and Chinese signatures. Signature data were

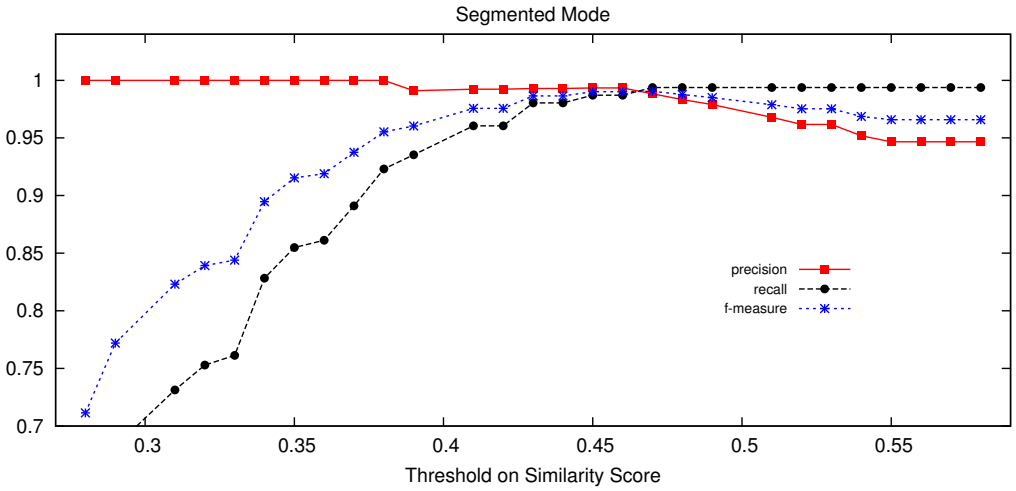


Figure 8.10: Average value for Chinese Signature

acquired using a WACOM Intuos3 A3 Wide USB Pen Tablet and collection software, i.e., MovAlyzer.

  - **Dutch Dataset**. The dataset is divided in two non-overlapping parts, a training set (comprised of 10 authors with 330 genuine signatures and 119 forgeries) and a test set (comprised of 10 authors with 648 genuine signatures and 611 corresponding forgeries).
  - **Chinese Dataset**. The dataset is divided in two non-overlapping parts, a training set (comprised of 10 authors with 230 genuine signatures and 430 forgeries) and a test set (comprised of 10 authors with 120 genuine signatures and 461 corresponding forgeries).

- The SigComp2013 [113] competition involved (online) data collected by PRresearchers at the Human Interface Laboratory, Mie University Japan.

  - **Japanese Dataset**. The signature data were acquired using a HP EliteBook 2730p tablet PC and self-made collection software built with Microsoft INK SDK. The whole dataset consists of 1260 genuine signatures (42 specimens/individual) and 1080 skilled forgeries (36 specimens/forgery). The dataset is divided in two non-overlapping parts, a training set (comprised of 11 au-

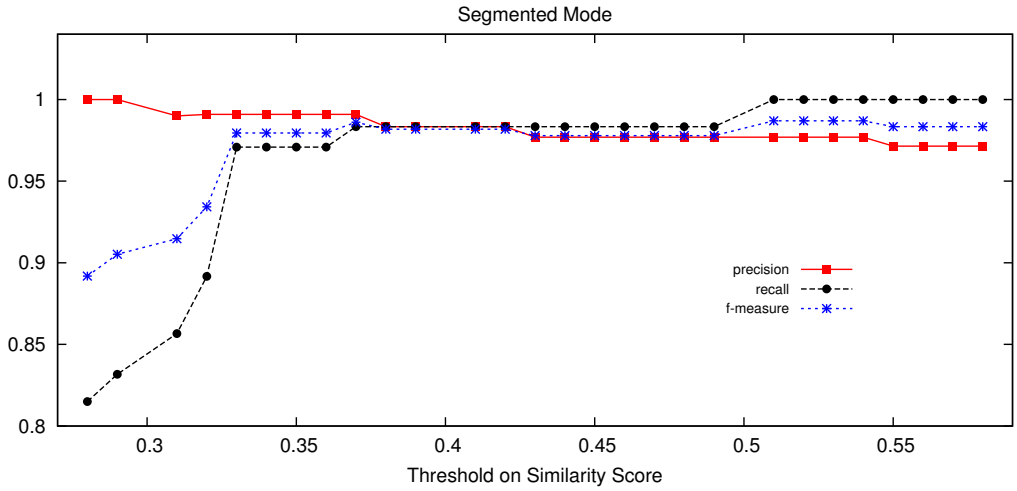


Figure 8.11: Average value for Dutch Signature

thors with 42 genuine signatures of each author and 36 forgeries per author) and a test set (comprised of 20 authors with 42 genuine signatures each and 36 corresponding forgeries per author).

The experimental results in terms of PCR, RCL and FMR (that vary according to the chosen thresholds) have been used for tuning the thresholds in order to get better performance. We did the experiment from 5% to 150% threshold to find the best solution. In the datasets, single users have genuine signatures with a different Pen-Up numbers. We grouped the genuine signature by the number of Pen-Ups and then generate a template. Later on, when we perform the signature testing operation, we identify the corresponding group for that user by Signature Pen-Up number.The main results of our findings are discussed in the remainder of this section.

Figure 8.8 is the samples of genuine and forgery signatures for different datasets. The algorithm is based on the signature pattern. We observed that, in general, signatures from the same language have similar patterns. So, the average value for a dataset from one language may differ to datasets from other languages. Figures 8.10, 8.11 and 8.12 plot the average of Chinese, Dutch and Japanese datasets respectively. As it can be seen from those figures, the best tolerance factor for Chinese dataset is 47%. Similarly, 37%

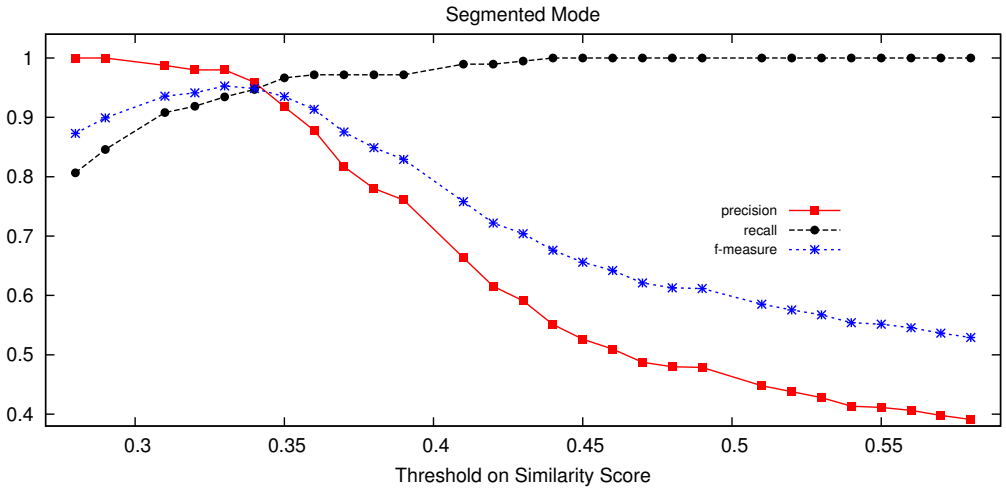


Figure 8.12: Average value for Japanese Signature

Datasets	Experimental details				
	Samsung Galaxy S2 Plus	Sony-Xperia Z2	ZTE Blade A430	LG-Nexus 5	Average
Chinese	0.47	0.08	0.10	0.04	0.17
Dutch	0.98	0.22	0.27	0.11	0.40
Japanese	4.36	1.00	1.94	0.80	2.03
Average	1.94	0.43	0.77	0.32	0.87

Table 8.2: Computational time of datasets in different mobile devices, time is in seconds

and 33% for Dutch and Japanese datasets respectively. The average for each dataset has calculated from the sample of both genuine and forgery signatures of users. After that, we calculated the best threshold for overall datasets.

Figure 8.13 plots the PCR, RCL and FMR as a function of the chosen tolerance factor, i.e., the threshold reported in Table 8.1. That shows the results related to precision, recall, f-measure, FAR, and FRR for values which maximize the f-measure. The best results for average were achieved using a 35% tolerance factor.Claimed identities are accepted whenever the score is above the threshold, rejected otherwise. The higher the

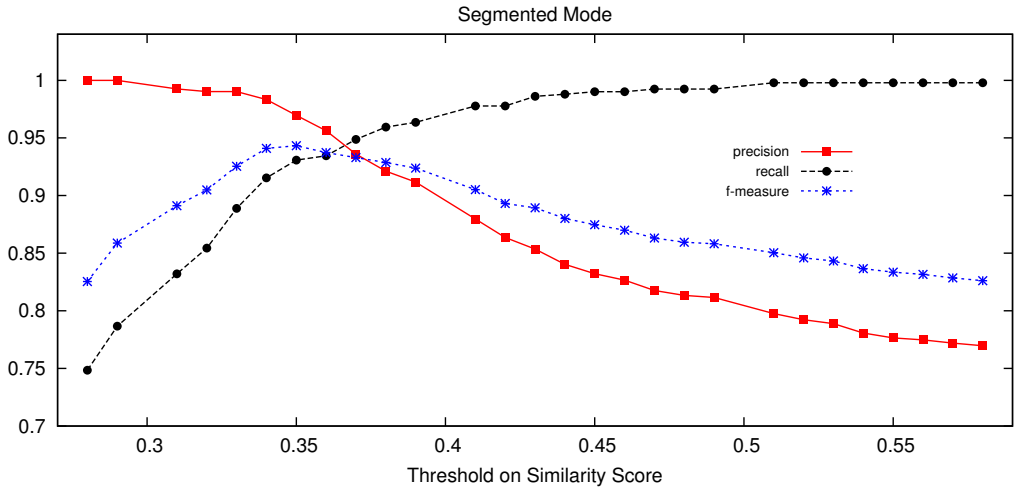


Figure 8.13: PCR, RCL, FMR, FAR and FRR as a functions of a tolerance factor (**TF**).

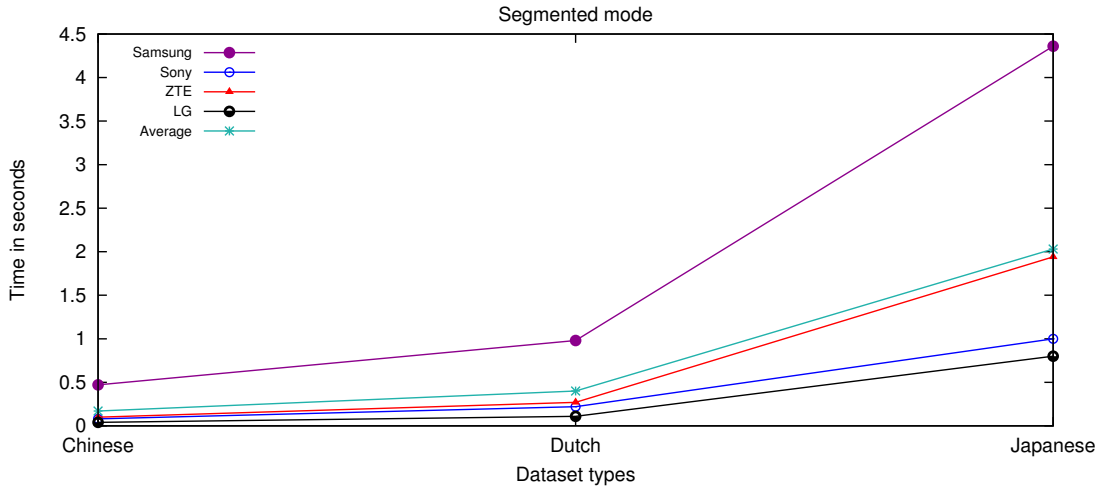


Figure 8.14: Computational time of datasets in different mobile devices, time is in seconds.

threshold, the higher the precision, but the lower the recall.

Finally, we address the computational overhead. We stress that the overall running time is important, since in many applications handwritten signatures could be decoded on low-end devices, such as mobile phones or tablets. Figure 8.14 plots the average of computational time taken by different devices for all datasets reported in Table 8.2.

Similarly,The plots of Figure 8.15 represent the scatterplot matrix of the computational time for different mobile devices, box indicates the lower quartile, median, upper quartile, and whisker represents the smallest and largest observation (Graph is generated by the Statgraphics Software). It shows that even low-end devices (such as Samsung Galaxy S2) are able to verify the signature quickly (i.e., in a few seconds), while devices with high performance (such as Google Nexus 5) are really fast in verifying signatures (i.e., in a few hundreds of milliseconds).

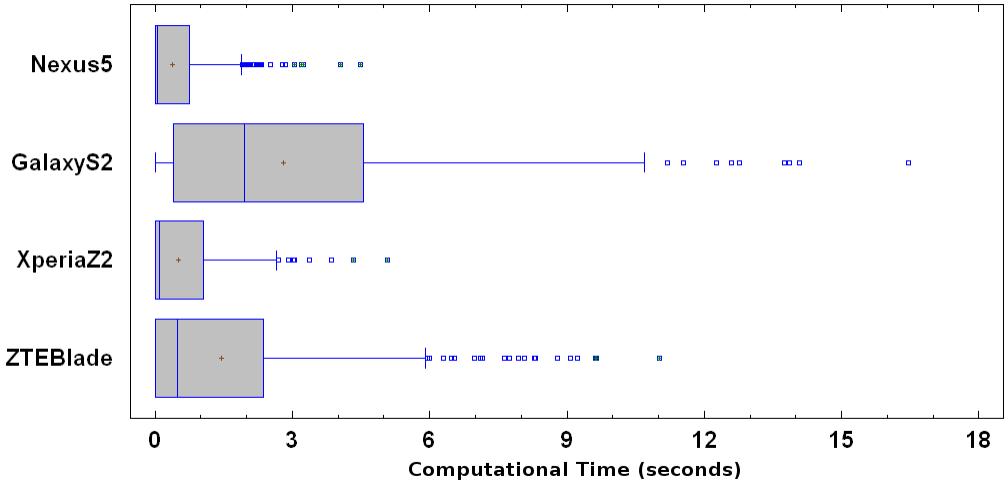


Figure 8.15: Computational time for different mobile devices

9 Conclusion

In this thesis, we analyzed the design space of recent algorithms that perform well in practice in the field of "Graph Connectivity" and "Authentication System". In addition, we also composed the techniques and presented some novel algorithms to solve the few problems efficiently in those areas.

We implemented and then explored the merits and weaknesses of the algorithms that evaluate the structure of Graph Connectivity in practice by conducting a thorough empirical analysis. We first presented an algorithm to compute the loop-nesting tree of a dense graph based on the algorithm available in [30]. We compared the algorithms that compute the 2-*edge-connected blocks*, 2-*vertex-connected blocks*, and 2-*vertex connected components* respectively. In addition, we presented a new algorithm to compute the most critical node of a *directed graph* in a linear time. With respect to the Authentication Systems, we presented a new authentication system, which is especially suitable for the mobile devices with low hardware configuration. We are going to summarize the main results achieved as following.

- We presented a new memory efficient version of an algorithm to compute the loop nesting forest of a directed graph, which is derived from the single pass Tarjan's Streamline version [30]. The experimental reports proved that it worked well for the dense graph.
- We performed a thorough experimental study of the linear time $O(m+n)$ (m and n are the size of edges and vertices respectively) algorithms to compute the 2-*edge connected blocks* of a digraph, presented in [71], and in [73]. Moreover, we also designed a memory efficient version, which derived from the algorithm available in [73]. Analytical reports show that the algorithm presented in [73] does not

depend on the graph structure so that it pays off over the algorithm presented in [71]. Our memory efficient algorithm also produced the better result for the dense graph and did not have any relation with the Graph structure.

- We have implemented the algorithm that compute the 2-*vertex connected blocks* of a digraph in linear time, available in [72] and [73]. We also presented a memory efficient version of an algorithm proposed in [73]. The experimental evaluation showed that the graph connectivity structure is directly proportional to the algorithm presented in [72], and does not have any relation to the algorithm given in [73]. Therefore the algorithm available in [73] produce consistent result compared to the algorithm available in [72]. Our memory efficient version has the consistent and better results for the dense graphs. We believe that our memory efficient version will alleviate the loop nesting computation to the consistent level even if the graph density increases.
- We analyzed the recent algorithms that compute the 2-*vertex connected component* of a directed graph. In particular, we implemented the algorithms available in [83], and in [44]. We also presented a new hybrid algorithm. After a thourough empirical study, our experimental reports showed that the algorithm presented in [44] performed better than the algorithm available in [83] for the real-world graph. However, the algorithm available in [83] has much better performance than the algorithm proposed in [44] for some special type of artificial graphs. Our hybrid algorithm produces the in-between results. We believe that, this comparative observation will help to choose the suitable algorithms for the real-life application according to the target area.
- We presented the first linear time algorithm to compute the most critical node of a directed graph. In addition, we also designed the two more heuristics. We implemented our algorithm and heuristics and other available famous heuristics such as *maximum-degree, Page Rank, Betweenness Centrality*, then performed an experimental evaluation. The empirical investigation proved that our linear time algorithm produces the better result than the other heuristics, both regarding

solution quality, and running time. We hope that our pioneer algorithm will be the milestone to solve the most critical node problem in directed graph.

- We presented a new Authentication System by online handwritten signature, whose novelties lie mainly in the following aspects. First, we proposed a method for the verification of signature features which is compatible with a broad range of low-end mobile devices (concerning computational overhead and verification accuracy), so that no special hardware is needed. Secondly, our new method makes use of several technical features that, to the best of our knowledge, have not been previously used for handwritten signature recognition. We implemented and tested the signatures dataset from various languages in the mobile devices with small hardware configuration. The experimental observation report confirms that our method achieved 95% efficiency in terms of accurate recognition for the Chinese, Japanese, and Dutch signatures in less than a second. The result is impressive, especially when the limited computational power of mobile devices is considered. We belive the work presented will be helpful in advancing the authentication system for low-end devices.

9.1 Open Problems

We are going to leave few open questions, which are listed as following.

- As we see in Chapter 4, and 5, the best current bound to compute the 2-vertex-connected (resp., 2-edge-connected) blocks of a directed graph is linear time $O(m+n)$. In addition, we explained in Chapter 6 that the best current bound for 2-vertex-connected (resp., 2-edge-connected) components of a digraph is $O(n^2)$. Therefore, we leave as an open question whether one can compute also the 2-vertex-connected components and 2-edge-connected-components in linear time.
- We presented the first non-trivial linear-time algorithm to calculate a most critical node for unweighted directed graphs. We wonder whether is it possible to extend the same algorithm within the same time bound (i.e., linear) for a weighted directed graph.

A
Appendix

A.1 Asymptotic Notations

Several *algorithms* are available to solve a single problem. *Asymptotic notation* help to identify the growth order of running time of an algorithm, which analyzes the algorithm's *efficiency*.

A.1.1 Big O Notation

Big O notation is the formal way to express the *upper bound* of an algorithm's running time. It is the efficiency notion of an algorithm on the basis of algorithm's *worst case* instance (i.e., maximum amount of time can be taken by an algorithm to solve a problem). Let us consider that $f(n)$ and $g(n)$ be two non-negative functions, then the Big O notation between $f(n)$ and $g(n)$ is defined as following.

$$f(n) = O(g(n)) \iff \{\exists c, n_0 \in \mathbb{N} \text{ such that } 0 \leq f(n) \leq c.g(n), \forall n \geq n_0\}$$

In general, most of the algorithms efficiency are compared on the basis of Big O notation.

In typical usage, the Big O notation for a function f is derived by the following simplified rules rather than using the above formal definition.

(i) If $f(x)$ is a sum of several terms and there exists a term T with largest growth rate, then T is what determine the growth rate of $f(x)$.

(ii) If T is a product of several factors, any constants in T that do not depend on x can be omitted.

Example: Let us consider a function $f(x) = 3x^4 - 2x^3 + x^2 + 4$, and suppose we wish to simplify the $f(x)$ to describe its growth rate as $x \to \infty$ by using Big O notation.

According to rule (i): $f(x)$ is the sum of four terms, $3x^4$, $-2x^3$, x^2 and 4. Thus, we select the term which is largest growth rate of x (i.e. which has the largest exponent of x) and others are ommitted. Therefore, the selected term is $3x^4$.
Again, by using rule (ii): $3x^4$ is a product of two factors 3 and x^4, where the first factor 3 does not depend on x. Hence, by omitting this constant factor, the simplified form of the result is x^4.
Thus, we say that $f(x)$ is a $O(x^4)$, or $f(x) = O(x^4)$.
We can confirm this calculation by using the formal definition as following:
Let $f(x) = 3x^4 - 2x^3 + x^2 + 4$ and $g(x) = x^4$.
Now, we have to find out the c and $x_0 \in \mathbb{Z}^+$ such that $\forall x \geq x_0$, $0 \leq f(x) \leq c.g(x)$ holds. There can be many c and x_0, for example, few of them are as follows. $x_0 = 1$ and $c = 6$, $x_0 = 2$ and $c = 3$, $x_0 = 3$ and $c = 3$, and so on.

A.1.2 Big Ω Notation

Big Ω represents the *best case* scenario of an algorithm or the *lower bound* of the growth rate of an algorithm's running time. Let us consider the two non-negative functions $f(n)$ and $g(n)$, then the Big Ω between $f(n)$ and $g(n)$ is defined by

$$f(n) = \Omega(g(n)) \iff \{\exists c, n_0 \in \mathbb{Z}^+ \text{ such that } 0 \leq c.g(n) \leq f(n), \forall n \geq n_0\}.$$

A.1.3 Big Θ Notation

Big Θ denotes the asymptotically *tight bound (lower and upper)* on the growth rate of the running time of an algorithm. So it defines the *exact asymptotic behavior*. For example, let us consider the two non-negative functions $f(n)$ and $g(n)$, Then we can define the Big Θ notation between $f(n)$ and $g(n)$ as following

$$f(n) = \Theta(g(n)) \iff \{\exists c_1, c_2, n_0 \in \mathbb{Z}^+ \text{ such that } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0\}.$$

A.1.4 Small *o* Notation

Small *o* notation denotes the *upper bound (that is not asymptotically tight)* of the growth rate of the running time of an algorithm. If $f(n)$ and $g(n)$ are two non-negative functions

and $f(n) = o(g(n))$, then $g(n)$ is the *upper bound* for $f(n)$ but $f(n)$ can never equal to $g(n)$.

$$f(n) = o(g(n)) \iff \{\exists c, n_0 \in \mathbb{Z}^+ \text{ such that } f(n) < c.g(n), \forall n \geq n_0\}.$$

The main difference between the *Big O-notation* and *small o-notation* is explained below.

In *Big O-notation*, if $f(n) = O(g(n))$, then the bound $f(n) \leq c.g(n)$ holds for *some constant* $c > 0$.

But in *small o-notation*, if $f(n) = o(g(n))$, then the bound $f(n) < c.g(n)$ holds for *all constants* $c > 0$.

A.1.5 Small ω Notation

Small ω notation is used to denote the lower bound (that is not asymptotically tight) of the growth rate of runtime of an algorithm. Let us consider that $f(n)$ and $g(n)$ be two non-negative functions and $f(n) = \omega(g(n))$, then $g(n)$ is the *lower bound* for $f(n)$ but $g(n)$ never equal to $f(n)$.

$$f(n) = \omega(g(n)) \iff \{\exists c, n_0 \in \mathbb{Z}^+ \text{ such that } f(n) > c.g(n), \forall n \geq n_0\}.$$

We are going to give the major difference between *Big* Ω*-notation* and *small* ω*-notation* as following:

In *Big* Ω*-notation*, if $f(n) = \Omega(g(n))$, then the bound $f(n) \geq c.g(n)$ holds for *some constant* $c > 0$.

Whereas, in *small* ω*-notation*, if $f(n) = \omega(g(n))$, then the bound $f(n) > c.g(n)$ holds for *all constants* $c > 0$.

A.1.6 Summary

For the two non-negative functions $f(x)$ and $g(x)$, we summarized the asymptotic relations between them in Table A.1 given below.

Asymptotic notation	Bound definition	Limit definition
$f \in O(g)$	$f \leq g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Omega(g)$	$f \geq g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \Theta(g)$	$f = g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R} > 0$
$f \in o(g)$	$f < g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in \omega(g)$	$f > g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Table A.1: Summary of the asymptotic notations.

A.2 Missing Functions and Equations

A.2.1 Menger's Theorem

In 1927, the Austrian Mathematician Karl Menger state a theorem to characterize the connectivity of graphs in terms of the minimum number of disjoint paths that can be found between any pair of vertices. Menger theorem[121] shows one of the most fundamental results in graph theory stated as following.

Theorem A.2.1. ***(Menger's Theorem [121])*** *Let G=(V,E) be a directed multigraph and let $u, v \in V(G)$ be a pair of distinct vertices. Then the following holds:*

(a) The maximum number of arc-disjoint (u,v)-paths equals the minimum number of arcs covering all (u,v)-paths and this minimum is attained for some (u,v)-cut $(X,\bar{X})$.

(b) If the arc uv is not in $E(G)$, then the maximum number of internally disjoint (u,v)-paths equals the minimum number of vertices in a (u,v)- separator.

Proof. Please refer to [121] (in German) or to [20, pg.353] in English. □

A.2.2 Ackermann Functions

In the field of Computational theory, *Ackermann* functions are well known simple examples of computable (implementable using a combination of *while/for-loops*) but not *primitive* recursive (implementable using only a *FINITE number* of *do-while/for-loops*) functions [99]. It is an equation discovered by **Wilhelm Ackermann** in 1928, state as following. *"All primitive recursive functions are total and computable but all total computable functions need not to be primitive recursive"*. The original function is published by Ackermann [1] (in German Language), which requires three nonnegative integers for its arguments as following:

$$
\begin{aligned}
&\varphi(0,m,n) = m+n \\
&\varphi(1,m,0) = 0 \\
&\varphi(2,m,0) = 1 \qquad \text{A.1} \\
&\varphi(p,m,0) = m \qquad \text{if } p > 2 \\
&\varphi(p,m,n) = \varphi(p-1,m,\varphi(p,m,n-1)) \qquad \text{in general}
\end{aligned}
$$

Equation A.1 is extended through the *Hyperoperation* as following.

Hyperoperation. Hyperoperation is a function $H_n(x,y)$, and defined recursively with two integers arguments as following:

$H_0(x,y) = y+1$

$H_1(x,0) = x$

$H_2(x,0) = 0$

$H_n(x,0) = 1$ for $n > 2$

$H_n(x,y) = H_{n-1}(x,H_n(x,y-1))$ for integers $n > 0$ and $y > 0$.

Consequently:

$H_0(x,y) = y+1$ is the successor function on y.

$H_1(x,y) = x+y$ is addition.

$H_2(x,y) = x \times y$ is multiplication.

$H_3(x,y) = x^y$ is exponentiation.

$H_4(x,y) =$ is tetration (a height-y exponential tower $x^{x^{x^{\cdots}}}$, we shall denote by Knuth's up-arrow notation)

$\vdots$

and so on.

Knuth's up-arrow notation This is a method to denote the large integers, introduced by Donald Knuth in 1976, defined as following:

i. $a \uparrow b = a^b = \underbrace{a \times a \times \ldots \times a}_{b \text{ times}}$

ii. $a \uparrow\uparrow b = {}^b a = \underbrace{a^{a^{\cdots}}}_{b \text{ times}} = a \uparrow (a \uparrow (\ldots \uparrow a)) = \underbrace{a \uparrow (a \uparrow (\ldots \uparrow a))}_{b \text{ times}}$

For example:

i. $2 \uparrow 3 = \underbrace{2 \times 2 \times 2}_{3 \text{ times}} = 8$

ii. $2 \uparrow\uparrow 3 = {}^3 2 = \underbrace{2^{2^2}}_{3 \text{ times}} = \underbrace{2 \uparrow (2 \uparrow 2)}_{3 \text{ times}} = 16.$

Let go back to the hyperoperation, if we extend the hyperoperation for negative-order by recursive formula, then

$H_0(x,y) = H_{-1}(x, H_0(x, y-1)) = H_{-1}(x,y).$

Therefore, $H_{-n}(x,y) = H_0(x,y)$ for every *non-negative n*.

Similarly, if we compare the equation A.1 with the **hyperoperation**, then we will get the following:

$$H(0,m,n) = m+1$$

$$H(1,m,0) = m$$

$$H(2,m,0) = 0$$

$$H(p,m,0) = 1$$

Then, we have the Ackermann function variants are 3-argument functions, and the equation A.1 satisfies the following recurrence relation.

$$\varphi(1,m,n) = m * n$$

$$\varphi(2,m,n) = m^n$$

As we see, the family of Ackermann functions can be simplified by omitting the m variable of the 3-argument function by making them into two arguments. The 2-argument Ackermann function would be then a function satisfying the recurrence relation:

$$f(p,n) = f(p-1, f(p,n-1)) \tag{A.2}$$

Many authors modified the original Ackerman's equation A.1 to fit it for different propose as like in equation A.2. Nowadays, "the Ackermann function" may refer to any of numerous variants of the equation A.1, the most common version is the two-argument Ackermann-Péter function which exactly follows the equation A.2 as following.

$$A(m,n) = \begin{cases} n+1 & \text{if } \mathrm{m} = 0 \\ A(m-1,1) & \text{if } \mathrm{m} > 0 \text{ and } \mathrm{n} = 0 \\ A(m-1, A(m,n-1)) & \text{if } \mathrm{m} > 0 \text{ and } \mathrm{n} > 0. \end{cases} \tag{A.3}$$

Example:

$$\begin{aligned} A(1,2) &= A(0,A(1,1)) \\ &= A(0,A(0,A(1,0))) \\ &= A(0,A(0,A(0,1))) \\ &= A(0,A(0,2)) \ \{ \text{ since } m = 0, \therefore \text{ output} = n+1 \ \} \\ &= A(0,3) \text{ similar to above step} \\ &= 4 \ \{ \text{ same to the previous step } \} \end{aligned}$$

Similarly, we can get

i. $A(4,3) \approx 2^{2^{65536}}$

ii. $A(4,4) = 16 \uparrow\uparrow 16$

It may not be directly obvious that the evaluation of $A(m,n)$ always terminates. However, in each recursion either m remains the same and n decreases or m decreases. Each time that n reaches zero, m decreases, so m eventually reaches zero as well. Therefore, we can say that the recursion has bounded. The equation A.3 give an idea that how it is recursive and bounded.

A.2.3 Inverse Ackermann Function.

Since the Ackermann function $f(n) = A(n,n)$ grows very quickly, its inverse function, f^{-1} (denoted by α), grows very slowly. In fact, $\alpha(n) < 5$ for most of the input size n, because as we see from equation A.3 with increasing input size m, n, the function $\alpha(n)$ decreases. This inverse appears in the time complexity of some algorithms, such as *disjoint-set* data structure, Lengauer and Tarjan's algorithm to compute the *dominator tree* in *flow graph*, Tarjan's algorithm to compute the *loop nesting forest* in a *flow graph*, Chazelle's algorithm for *minimum spanning trees*, and etc. The *two-parameter* variation of inverse Ackermann function can be defined as following:

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}, \text{ where } \lfloor x \rfloor \text{ is the } \textit{floor} \text{ function.}$$

This function gives the refined time bound for the precise analyses of the algorithms. In the *disjoint-set* data structure, m represents the number of operations while n represents the number of elements; in the minimum spanning tree algorithm, m represents the number of edges and n represents the number of vertices. Because of its definition in terms of extremely deep recursion, it can be used as a benchmark of a compiler's ability to optimize recursion [150]. For more information, please refer in [94, p.60], [54, p.255].

A.3 Missing Algorithms

A.3.1 Random Access Model

Random Access Machine (RAM) is a tool to measure the efficiency of an algorithm in a machine-independent way. It allows us to compare the algorithms by their performance. The smallest unit of the memory is a *register*, and an array of *cells*, each of them has a unique integer address. Furthermore, both the *register* and a *cell* have the capacity to store a single integer value or a real number of sized bounded by the world length, called parameter of the model. In RAM, memory consists of an unbounded sequence of registers, each of which is capable of holding an integer. The arithmetic operations are allowed to compute the address of a memory *register*. RAM consists of a fixed program, where each instruction will be executed one after the other, i.e., no concurrent operations. Usually, It is assumed that the size of a *register* is bounded by $O(\log n)$ bits, where n is the input size of the problem. The instruction contains either arithmetic or logical operation. During the execution of an instruction, it needs to use the contents of the *registers* or *cells*, therefore, either it loads the contents of a single memory *cell* into a *register*, or store the contents of a *register* to a memory *cell*. The load and store operations can recognize the respective memory by the address. We determine the running time of an algorithm by the number of time steps needed to execute to complete the given instruction. In this model, each simple operation requires a *unit* time step and each memory access also needs a *unit* time step. In general, we already assure that there is no shortage of memory. Nevertheless, Subroutines and Loops are not simple operations. (See also 76, pg.133 and [38].)

A.3.2 Pointer Access Model

The pointer machine model (PAM) [22, 100, 102, 143, 164] differs from RAM in memory organization; In particular, PAM consists of an unbounded collection of *registers*, which are connected by *pointers*. Each *register* can contain an arbitrary amount of additional information but we cannot perform the arithmetic operations to compute the

address of *register*, because the only way to access a *register* is by following *pointers*. Nodes manage the *pointers* in PAM, and hence memory of a PAM consists of an extendable collection of nodes. Each node can store the finite number of fields, and each field can store either a number, which will use in computation or pointer to a node. Concerning the running time complexity, creating a memory node takes a one-time step, and accessing a memory node given to that pointer also takes the one-time step. Normally, among the pointer-based algorithms, two different classes were defined, specifically for set union problems: separable pointer algorithms [164] and non-separable pointer algorithms [119]. (See also [19, chap. 5] and [76, pg.133].)

A.3.3 Tree Traversal

Tree Traversal methods are the fundamental strategies in most of the graph algorithms. It can be stipulated by the ordering of three different objects, (i) current node, (ii) left subtree and (iii) right subtree. We assume that the left subtree always comes before the right subtree. Then there are three different ways to travel, *pre-order*, *in-order*, and *post-order*. All of these ways are referred to as *depth-first-search*, where we have to search the tree as deep as possible on each child before going to the next sibling. For each node, N, the general recursive traveling technique in any non-empty tree is processed as following:

(a) (L) Traverse its left subtree recursively, when the step is finished then back to N again.

(b) (R) Traverse its right subtree recursively, when the step is finished then back to N again.

(c) (N) Process then the node N itself.

Then the traveling methods can be defined as following:

i. *Preorder*: The order sequence is: the current node, the left subtree, the right subtree (NLR).

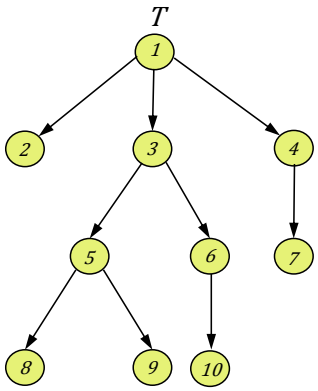


Figure A.1: Tree traversal example.

ii. *Inorder*: The order sequence is: the left subtree, the current node, the right subtree (*LNR*).

iii. *Postorder*: The order sequence is: the left subtree, the right subtree, the current node (*LRN*).

Example, Let us consider a tree shown in Figure A.1 and start to traverse by all methods that defined before. We get the following sequences.

i. *Preorder*: $1,2,3,5,8,9,6,10,4,7$.

ii. *Inorder* $2,1,8,5,9,3,10,6,7,4$.

iii. *Postorder*: $2,8,9,5,10,6,3,7,4,1$.

A.4 Dynamic Time Warping

Dynamic time warping (DTW) is a well-known technique to find an optimal alignment between two given (time-dependent) sequences as shown in Figure A.2. It compares two (time-dependent) sequences $X := (x_1, x_2, \ldots, x_N)$ of length $N \in \mathbb{N}$ and $Y := (y_1, y_2, \ldots, y_M)$ of length $M \in \mathbb{N}$. These sequences are sampled at equidistant points in time. Let $\mathcal{F}$ be a feature space. Then $x_n, y_m \in \mathcal{F}$ for $n \in [1:N]$ and $m \in [1:M]$. To compare two different features $x, y \in \mathcal{F}$, one needs a *local cost measure* (also referred to as *local distance measure*), which is defined to be a function

$$c : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geq 0} \tag{A.4}$$

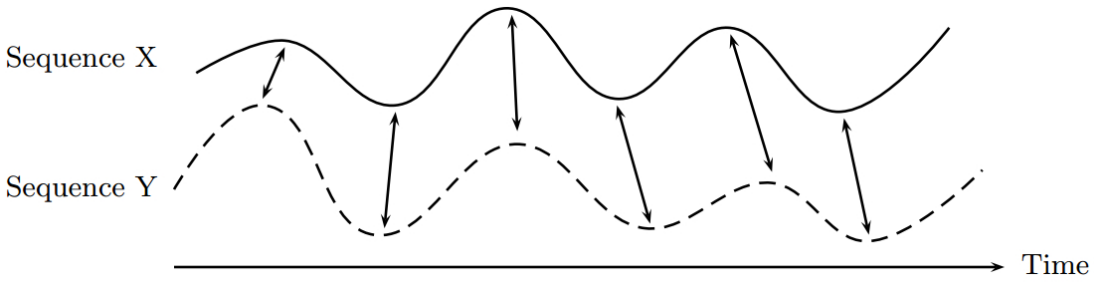


Figure A.2: Time alignment of two time-dependent sequences Aligned points are indicated by the arrows, source [124]

Typically, $c(x, y)$ is directly proportional to the similarity between x and y. It means that $c(x, y)$ is small (low cost) if x and y are similar to each other, and otherwise $c(x, y)$ is large (high cost). Evaluating the local cost measure for each pair of element of the sequences X and Y, one obtains the cost matrix $C \in \mathbb{R}^{N \times M}$ defined by $C(n, m) := c(x_n, y_m)$. Hence, the goal is to find an alignment between X and Y having minimal overall cost. We can find many scholarly articles about dynamic time warping. Here, we are presenting that how it works in the application scenario, which is taken from Müller [124] and also used in our Handwritten Online Signature Algorithm.

Definition A.4.1. *An (N,M)-warping path (or simply referred to as warping path if N and M are clear from the context) is a sequence $p = (p_1,\ldots,p_L)$ with $p_l = (n_l,m_l) \in [1{:}N] \times [1:M]$ for $l \in [1{:}L]$ satisfying the following three conditions.*

(i) Boundary condition: $p_1 = (1,1)$ and $p_L = (N,M)$.

(ii) Monotonicity condition: $n_1 \leq n_2 \leq \ldots \leq n_L$ and $m_1 \leq m_2 \leq \ldots \leq m_L$.

(iii) Step size condition: $p_{l+1} - p_l \in \{(1,0),(0,1),(1,1)\}$ for $l \in [1{:}L-1]$.

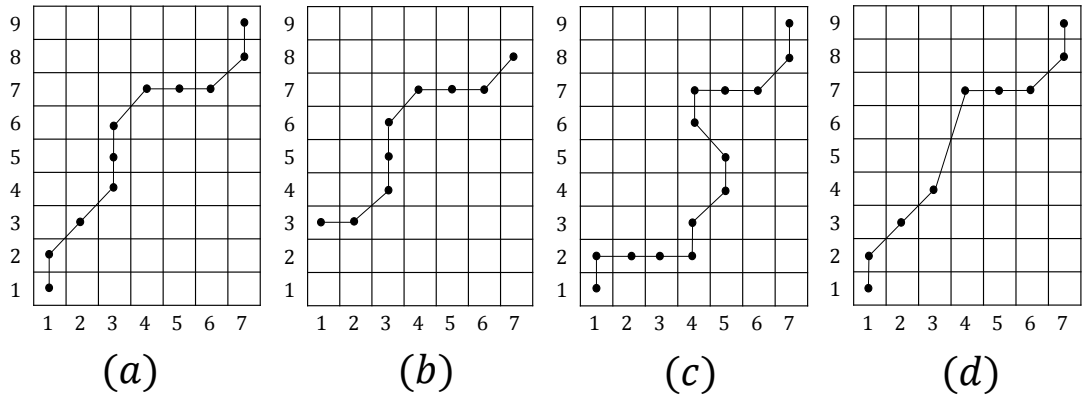


Figure A.3: Illustration of paths of index pairs for some sequence X of length $N = 9$ and some sequence Y of length $M = 7$. (a) Admissible warping path satisfying the conditions (i), (ii), and (iii) of Definition A.4.1. (b) Boundary condition (i) is violated. (c) Monotonicity condition (ii) is violated. (d) Step size condition (iii) is violated, source [124]

Note that, the step size condition (iii) implies the monotonicity condition (ii). An (N,M)-warping path $p = (p_1,\ldots,p_L)$ defines an alignment between two sequences $X = (x_1,x_2,\ldots,x_N)$ and $Y = (y_1,y_2,\ldots,y_M)$ by assigning the element x_n of X to the element y_m of Y. The boundary condition (i) enforces that the first elements of X and Y as well as the last elements of X and Y are aligned to each other. In other words, the alignment refers to the entire sequences X and Y. The monotonicity condition (ii) reflects the requirement of faithful timing: if an element in X precedes a second one this should also hold for the corresponding elements in Y, and vice versa. Finally, the step size condition (iii) expresses a kind of continuity condition: no element in X and Y can be

omitted and there are no replications in the alignment (in the sense that all index pairs contained in a warping path p are pairwise distinct). Figure A.3 illustrates the three conditions.

The total cost $c_p(X,Y)$ of a warping path p between X and Y with respect to the local cost measure c is defined by following equation.

$$c_p(X,Y) := \sum_{l=1}^{L} c\big(x_{nl}, y_{ml}\big). \tag{A.5}$$

Furthermore, an optimal warping path between X and Y is a warping path p^*, which has a total cost among all possible warping paths. The DTW distance $\mathsf{DTW}(X,Y)$ between X and Y is then defined as the total cost of p^*:

$$\begin{aligned} \mathsf{DTW}(X,Y) :=\ & c_{p^*}(X,Y) \\ & = \min\{c_p(X,Y) \mid p \text{ is an } (N,M)\text{-warping path}\} \end{aligned} \tag{A.6}$$

To determine an optimal path p^*, we need to calculate every possible warping path between X and Y. But this procedure would lead to a computational complexity that is exponential in the lengths N and M. However, there is an $O(NM)$ algorithm based on *dynamic programming* to calculate an optimal path p^* as following. It defines the prefix sequences $X(1:n) := (x_1,\ldots,x_n)$ for $n \in [1:N]$ and $Y(1:m) := (y_1,\ldots y_m)$ for $m \in [1:M]$ and set

$$D(n,m) :=\ \mathsf{DTW}(X(1:n),Y(1:m)). \tag{A.7}$$

The values $D(n,m)$ define an $N \times M$ matrix D, which also referred as the accumulated cost matrix. Obviously, one has $D(N,M) = \mathsf{DTW}(X,Y)$. In the following, a tuple (n,m) representing a matrix entry of the cost matrix C or of the accumulated cost matrix D will be referred to as a cell. The next theorem shows how D can be computed efficiently.

Theorem A.4.2 (Müller [124]). *The accumulated cost matrix D satisfies the following identities:*

$$D(n,1) = \sum_{k=1}^{n} c(x_k, y_1) \text{ for } n \in [1{:}N],$$

$$D(1,m) = \sum_{k=1}^{m} c(x_1, y_k) \text{ for } m \in [1:M], \text{ and}$$

$$D(n,m) = min\{D(n-1,m-1), D(n-1,m), D(n,m-1)\} + c(x_n, y_m) \quad \text{(A.8)}$$

for $1 < n \leq N$ and $1 < m \leq M$. In particular, $\mathsf{DTW}(X,Y) = D(N,M)$ can be computed with $O(NM)$ operations.

Algorithm 16: Compute Optimal Warping Path

Input: Accumulated cost matrix D.
Output: Optimal warping path p^*.
Procedure: The optimal path $p^* = (p_1, \ldots, p_L)$ is computed in reverse order of the indices starting with $p_L = (N,M)$. Suppose $p_L = (n,m)$ has been computed. In case $(n,m) = (1,1)$, one must have $L = 1$ and we are finished. Otherwise,

$$p_{l-1} := \begin{cases} (1, m-1), & \text{if } n = 1 \\ (n-1, 1), & \text{if } m = 1 \\ \text{argmin}\{D(n-1,m-1), & \\ D(n-1,m), D(n,m-1)\}, & \text{otherwise,} \end{cases} \quad \text{(A.9)}$$

where we take the lexicographically smallest pair in case "argmin" is not unique.

Theorem A.4.2 facilitates a recursive computation of the matrix D. The initialization can be simplified by extending the matrix D with an additional row and column and formally setting $D(n,0) = \infty$ for $n \in [1{:}N]$, $D(0,m) = \infty$ for $m \in [1{:}M]$, and $D(0,0) = 0$. The recursion of (A.8) holds for $n \in [1:N]$ and $m \in [1:M]$. Moreover, note that D can be computed in a column-wise fashion, where the computation of the m^{th} column only requires the values of the $(m-1)^{th}$ column. This implies that if we only need the value $\mathsf{DTW}(X,Y) = D(N,M)$, then the required storage space will be $O(N)$. Similarly, we can also proceed in a row-wise fashion, leading to $O(M)$. In either case, the running time is $O(NM)$. Furthermore, to compute an optimal warping path p^*, the entire $(N \times M)$-matrix D is needed. The Algorithm 16 (adapted from [124]) fulfills this task.

A.4.1 Variations of DTW

Several types of modifications are proposed to better control the possible routes of the warping paths as well as to speed up DTW computations. Modify in step size calculation and local weights are two of them and we are going to discuss them in this section.

A.4.1.1 Step Size Condition

We already explained that the step size condition (iii) represents a kind of local continuity condition, which ensures that each element of the sequence $X = (x_1, x_2, \ldots x_N)$ is assigned to an element of $Y = (y_1, y_2, \ldots, y_M)$ and vice versa. Nevertheless, one drawback of this condition is that a single element of one sequence may be assigned to many consecutive elements of the another sequence, leading to vertical and horizontal segments in the warping path, see Figure A.5-(a). Therefore, the warping path can be stuck at some position with respect to another sequence.

To avoid such degenerations, we can modify the step size condition to constrain the slope of the admissible warping paths. As a first example, we can replace the step size condition (iii) of Definition A.4.1 by the condition $p_{l+1} - p_l \in \{(2,1),(1,2),(1,1)\}$ for $l \in [1{:}L]$, as shown in Figure A.5-(b). This leads to warping paths having a local slope within the bounds $\frac{1}{2}$ and 2. Then for $n \in [2{:}N]$ and $m \in [2{:}N]$, the accumulated cost matrix D can be computed by the recursion as following.

$$D(n,m) = \min\{D(n-1,m-1), D(n-2,m-1), D(n-1,m-2)\} + c(x_n, y_m) \quad \text{A.10}$$

Here, we set the initial values as following.

$D(0,0) = 0.$

$D(1,1) = c(x_1, y_1).$

$D(n,0) = \infty$ for $n \in [1{:}N]$.

$D(n,1) = \infty$ for $n \in [2{:}N]$.

$D(0,m) = \infty$ for $m \in [1{:}M]$.

$D(1,m) = \infty$ for $m \in [2{:}M]$.

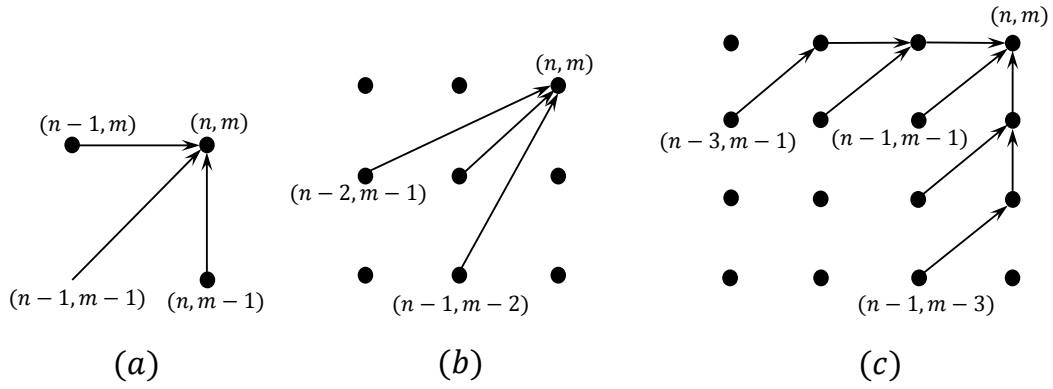


Figure A.4: Illustration of three different step size conditions, which express different local constraints on the admissible warping paths. (a) corresponds to the step size condition (iii) of Definition A.4.1, source [124]

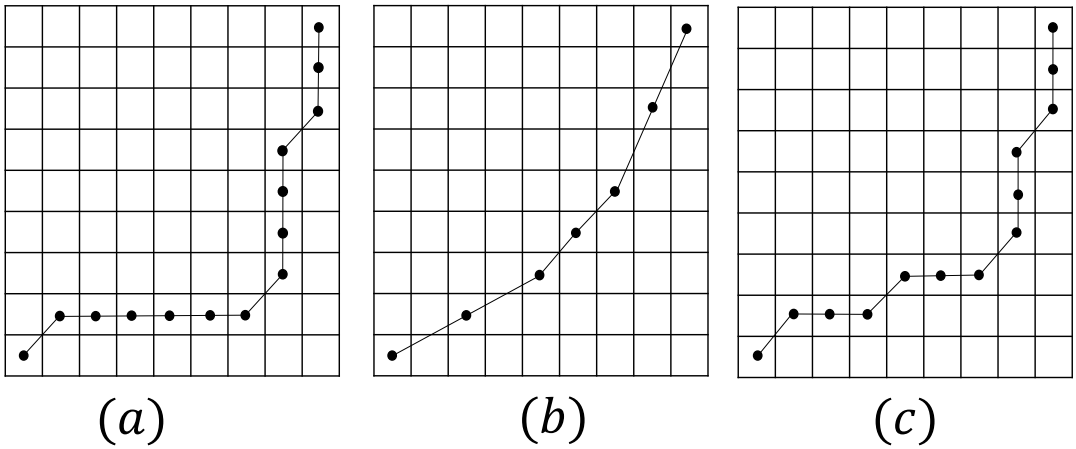


Figure A.5: Three warping paths with respect to the different step size conditions indicated by Figure A.4. (a) Step size condition of Figure-A.4-(a) may result in degenerations of the warping path. (b) Step size condition of Figure A.4-(b) may result in the omission of elements in the alignment of X and Y. (c) Warping path with respect to the step size condition of Figure A.4-(c), source [124]

Note that, with respect to the modified step size condition, there is a warping path between two sequences X and Y if and only if the difference between the lengths N and M is allowed to at most by a factor of two. Moreover, all elements of X need not be assigned to some element of Y and vice versa as shown in Figure A.5-(b), where x_1 is assigned to y_1, x_3 is assigned to y_2, but x_2 is not assigned to any element of Y (i.e., x_2 is

omitted and does not cause any cost at all).

Similarly, we can avoid such omission while imposing constraints on the slope of the warping path as shown in Figure-A.5-(c), where the recursion of the resulting accumulated cost matrix D is given by

$$D(n,m) = \min \begin{cases} D(n-1,m-1)+c(x_n,y_m) \\ D(n-2,m-1)+c(x_n-1,y_m)+c(x_n,y_m) \\ D(n-1,m-2)+c(x_n,y_m-1)+c(x_n,y_m) \\ D(n-3,m-1)+c(x_n-2,y_m)+c(x_n-1,y_m)+c(x_n,y_m) \\ D(n-1,m-3)+c(x_n,y_m-2)+c(x_n,y_m-1)+c(x_n,y_m) \end{cases} \quad \text{A.11}$$

for $(n,m) \in [1{:}N] \times [1:M] \setminus \{(1,1)\}$, and we set the initial values as following.

$$D(1,1) = c(x_1,y_1).$$
$$D(n,-2) = D(n,-1) = D(n,0) = \infty \text{ for } n \in [-2{:}N].$$
$$D(-2,m) = D(-1,m) = D(0,m) = \infty \text{ for } m \in [-2{:}M].$$

In this case, the slopes of the resulting warping paths are lie between the values $\frac{1}{3}$ and 3. Note that, this step size conditions enforce that all elements of X are aligned to some element of Y and vice versa. In other words, in the recursion (A.11) all elements of X and Y generate some cost in the accumulated cost matrix D - opposed to the recursion (A.10). Figure A.5 illustrates the differences of the resulting optimal warping paths computed with respect to different step size conditions.

A.4.1.2 Local Weights

We can add an additional weight vector $(w_d, w_h, w_v) \in \mathbb{R}^3$ that improve the vertical, horizontal, or diagonal direction in the alignment, yielding the recursion as following.

$$D(n,m) = \min \begin{cases} D(n-1,m-1)+w_d.c(x_n,y_m) \\ D(n-1,m)+w_d.c(x_n,y_m) \\ D(n,m-1)+w_d.c(x_n,y_m) \end{cases} \quad \text{A.12}$$

for $n \in [2{:}N]$ and $m \in [2{:}M]$.

Moreover, the other initial values are to be set as following.

$$D(n,1) = \sum_{k=1}^{n} (w_h.c(x_k,y_1)) \text{ for } n > 1$$
$$D(1,m) = \sum_{k=1}^{m} (w_v.c(x_1,y_k)) \text{ for } m > 1$$
$$D(1,1) = c(x_1,y_1).$$

The equally weighted case $(w_d, w_h, w_v) = (1,1,1)$ reduces to default (original) DTW (equation A.8). Also note that for $(w_d, w_h, w_v) = (1,1,1)$, if we have a preference of the diagonal alignment direction, then one diagonal step (cost of one cell) corresponds to the combination of one horizontal and one vertical step (cost of two cells). To counterbalance this preference, in general, it would better to choose $(w_d, w_h, w_v) = (2,1,1)$. Similarly, we can also apply other weighting factors for other step size conditions.

A.4.2 Subsequence DTW

If we need to find a subsequence within the longer sequence that optimally matches with the shorter sequence as shown in Figure A.4.1. Then the problem of finding optimal subsequences can be solved by a variant of dynamic time warping, which is going to describe in this section.

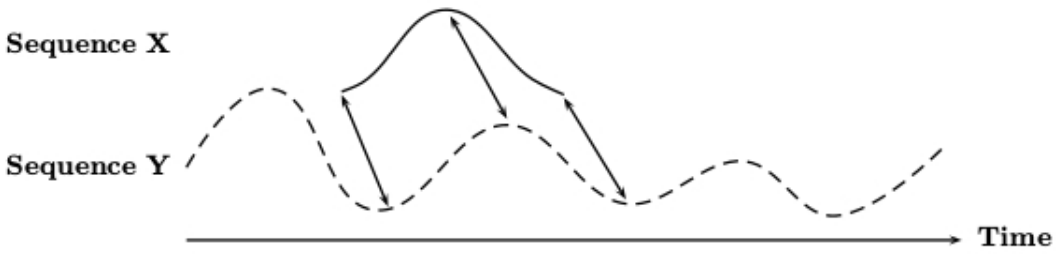


Figure A.6: Optimal time alignment of the sequence X with a subsequence of Y. Aligned points are indicated by the arrows, source [124]

Let us suppose $X = (x_1, x_2, \ldots, x_N)$, and $Y = (y_1, y_2, \ldots, y_M)$ are the feature sequences such that the length of a sequence Y (i.e., M) is much larger than the length of a

sequence X (i.e., N). Then, our goal is to find a subsequence $Y(a^*:b^*) = (y_{a^*}, y_{a^*+1}, \ldots, y_{b^*})$ with $1 \leq a^* \leq b^* \leq M$ that minimizes the DTW distance to X over all possible subsequences of Y as following.

$$(a^*, b^*) = \underset{(a,b)\,:\,1\leq a\leq b\leq M}{\operatorname{argmin}} (\mathsf{DTW}(X, Y(a:b))). \tag{A.13}$$

After the small modification in the initialization of the DTW algorithm described in Theorem A.4.2, an optimal alignment between X and the subsequence $Y(a^*:b^*)$ as well as the indices a^* and b^* can be computed. Let modify the definition of the accumulated cost matrix D by setting $D(n,1) = \sum_{k=1}^{n} c(x_k, y_1)$ for $n \in [1:N]$ and $D(1,m) = c(x_1, y_m)$ (opposed to $D(1,m) = \sum_{k=1}^{m} c(x_1, y_k)$ for $m \in [1:M]$). Then the remaining values of D can be defined recursively as explained by equation A.8 for $n \in [2:N]$ and $m \in [2:N]$. An extended accumulated cost matrix can also be defined by setting $D(n,0) = \infty$ for $n \in [0:N]$ and $D(0,m) = 0$ (opposed to $D(0,m) = \infty$) for $m \in [0:M]$. The index b^* can be determined from D as following.

$$b^* = \underset{b\in[1:M]}{\operatorname{argmin}} D(N, b). \tag{A.14}$$

To determine a^* and the optimal warping path between X and the subsequence $Y(a^*:b^*)$, we have to apply the Algorithm 16, but in this time, we need to start with $p_L = (N, b^*)$ as following. Let $p^* = (p_1, \ldots, p_L)$ be the resulting path, then $a^* \in [1:M]$ be the maximal index such that $p = (a^*, 1)$ for some $l \in [1:L]$. That is all elements of Y to the left of y_{a^*} and to the right of y_{b^*} are left unconsidered in the alignment and do not cause any additional costs. The computational complexity of the subsequence DTW algorithm is $O(NM)$. Note that, in general, the optimal alignment of the subsequence $Y(a^*:b^*)$ is not uniquely defined because there may be several choices for b^* in equation A.14, and in the construction of a^*.

Let define a distance function by an equation A.15 to explain how the accumulated cost matrix D can be used to derive an entire list of subsequences of Y that are optimally close to X with respect to the DTW distance.

$$\Delta : [1:M] \to \mathbb{R} \qquad \Delta(b) = D(N,b) \tag{A.15}$$

The equation A.15 assigns to each index $b \in [1:M]$ and the minimal DTW distance $\Delta(b)$ can be achieved between X and a subsequence $Y(a:b)$ of Y ending in y_b. For each $b \in [1:M]$, the DTW-minimizing $a \in [1:M]$ can be computed analogously to a^* by using the Algorithm 16, which is start with $p_L = (N,b)$. Note that if $\Delta(b)$ is small for some $b \in [1:M]$ and if $a \in [1:M]$ be the corresponding DTW-minimizing index, then the subsequence $Y(a:b)$ is close to X. Thus, this observation suggests the Algorithm 17 (adapted from Müller [124]) to compute all (up to some specified overlap) subsequences of Y that is similar to X.

Algorithm 17: Compute Similar Subsequences

Input: $X = (x_1, \ldots, x_N)$ query sequence
$Y = (y_1, \ldots, y_M)$ database sequence
Output: Ranked list of all (essential distinct) subsequences of Y that have a DTW distance to X below the threshold τ.

(1) Initialize the ranked list to be the empty list.
(2) Compute the accumulated cost matrix D w.r.t. X and Y.
(3) Determine the distance function Δ by an equation A.15
(4) Determine the minimum $b^* \in [1:M]$ of Δ.
(5) If $\Delta(b^*) > \tau$, then terminate the procedure.
(6) Compute the corresponding DTW-minimizing index $a^* \in [1:M]$.
(7) Extend the ranked list by the subsequence $Y(a^*:b^*)$.
(8) Set $\Delta(b) = \infty$ for all b within a suitable neighborhood of b^*.
(9) Continue with Step (4).

We can notice that in the Step (8) of Algorithm 17, it exclude an entire neighborhood of b^* from further consideration. Therefore, it avoids the ranked output list, which contains many subsequences that only differ by a slight shift. For example, if $Y(a:b)$ is in the list, then we can prevent that $Y(a:b+1)$ is in the list as well. Hence, depending on the application, we may choose a fixed size of the neighborhood around b^* or adjust the size according to the local property of Δ around b^*.

Bibliography

[1] W. Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928. URL http://eudml.org/doc/159248.

[2] Bernardetta Addis, Marco Di Summa, and Andrea Grosso. Identifying critical nodes in undirected graphs: Complexity results and polynomial algorithms for the case of bounded treewidth. *Discrete Appl. Math.*, 161(16-17):2349–2360, November 2013. ISSN 0166-218X. doi: 10.1016/j.dam.2013.03.021. URL http://dx.doi.org/10.1016/j.dam.2013.03.021.

[3] Kiarash Ahi and Mehdi Anwar. Advanced terahertz techniques for quality control and counterfeit detection. *Proc. SPIE*, 9856:98560G–98560G–14, 2016. doi: 10.1117/12.2228684. URL http://dx.doi.org/10.1117/12.2228684.

[4] A. V. Aho and J. D. Ullman. *Principles of Compilers Design*. Addison-Wesley, 1977.

[5] A. V. Aho, J. E. Hopcroft, and J. D.Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing, MA, 1974.

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–32, 1976.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[8] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[9] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL http://doi.acm.org/10.1145/390013.808479.

[10] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3): 351–358, 2004.

[11] Stefano Allesina, Antonio Bodini, and Cristina Bondavalli. Secondary extinctions in ecological networks: Bottlenecks unveiled. *Ecological Modelling*, 194(1-3):150 – 161, 2006. ISSN 0304-3800. doi: http://dx.doi.org/10.1016/j.ecolmodel.2005.10.016. URL http://www.sciencedirect.com/science/article/pii/S0304380005005132.

[12] S. Alstrup and M. Thorup. Optimal pointer algorithms for finding nearest common ancestors in dynamic trees. *Journal of Algorithms*, 35:169–88, 2000.

[13] S. Alstrup, P. W. Lauridsen, and M. Thorup. Dominators in linear time. Technical Report DIKU TOPPS D-320, Dept. of Computer Science, U. Copenhagen, 1996.

[14] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[15] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.

[16] A. Arulselvan, C. W. Commander, L. Elefteriadou, and P. M. Pardalos. Detecting critical nodes in sparse graphs. *Comput. Oper. Res.*, 36(7):2193–2200, July 2009. ISSN 0305-0548. doi: 10.1016/j.cor.2008.08.016. URL http://dx.doi.org/10.1016/j.cor.2008.08.016.

[17] Ashwin Arulselvan, Clayton W. Commander, Lily Elefteriadou, and Panos M. Pardalos. Detecting critical nodes in sparse graphs. *Comput. Oper. Res.*, 36(7):2193–2200, July 2009. ISSN 0305-0548. doi: 10.1016/j.cor.2008.08.016. URL http://dx.doi.org/10.1016/j.cor.2008.08.016.

[18] J. Aspnes, K. Chang, and A. Yampolskiy. Inoculation strategies for victims of viruses and the sum-of-squares partition problem. *J. Comput. Syst. Sci.*, 72(6):1077–1093, September 2006. ISSN 0022-0000. doi: 10.1016/j.jcss.2006.02.003. URL http://dx.doi.org/10.1016/j.jcss.2006.02.003.

[19] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998. ISBN 0849326494.

[20] Jrgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications (Springer Monographs in Mathematics)*. Springer, 2^{nd} edition, 2008. ISBN 1848009976, 9781848009974.

[21] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[22] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *J. ACM*, 39 (3):617–648, July 1992. ISSN 0004-5411. doi: 10.1145/146637.146666. URL http://doi.acm.org/10.1145/146637.146666.

[23] Ramon Blanco-Gonzalo, Raul Sanchez-Reillo, Oscar Miguel-Hurtado, and Judith Liu-Jimenez. Performance evaluation of handwritten signature recognition in mobile environments. *IET Biometrics*, 3(3):139–146, 2013.

[24] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi: 10.1080/0022250X.2001.9990249. URL http://dx.doi.org/10.1080/0022250X.2001.9990249.

[25] Gilles Brassard and 1940 Bratley, Paul. *Fundamentals of algorithmics*. Englewood, N.J. : Prentice Hall, 1996. ISBN 0133350681.

[26] Gilles Brassard and Paul Bratley. *Algorithmics: Theory &Amp; Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-023243-2.

[27] S. Brin and L. Page. Proceedings of the seventh international world wide web conference the anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107 – 117, 1998. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/S0169-7552(98)00110-X. URL http://www.sciencedirect.com/science/article/pii/S016975529800110X.

[28] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum in 27(3):383-7, 2005.

[29] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time pointer-machine algorithms for path-evaluation problems in trees and graphs. Submitted. Preliminary version available online at http://arxiv.org/abs/cs.DS/0207061., 2006.

[30] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[31] A.L. Cauchy. Recherche sur les polyèdres - premier mémoire. *Journal de l'ÃL'cole Polytechnique*, 9(Cahier 16):66–86, 1813.

[32] E. Cayley. Ueber die analytischen figuren, welche in der mathematik bäume genannt werden und ihre anwendung auf die theorie chemischer verbindungen. *Berichte der deutschen chemischen Gesellschaft*, 8(2):1056–1059, 1875. ISSN 1099-0682. doi: 10.1002/cber.18750080252. URL http://dx.doi.org/10.1002/cber.18750080252.

[33] K. Chatterjee and M. Henzinger. Efficient and dynamic algorithms for alternating büchi games and maximal end-component decomposition. *J. ACM*, 61(3):15:1–15:40, June 2014. ISSN 0004-5411. doi: 10.1145/2597631. URL http://doi.acm.org/10.1145/2597631.

[34] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved algorithms for one-pair and k-pair streett objectives. In *Logic in Computer Science (LICS), 2015 30th Annual ACM/IEEE Symposium on*, pages 269–280, July 2015. doi: 10.1109/LICS.2015.34.

[35] Krishnendu Chatterjee, Monika Henzinger, and Veronika Loitzenbauer. Improved algorithms for one-pair and k-pair streett objectives. *CoRR*, abs/1410.0833, 2014. URL http://arxiv.org/abs/1410.0833.

[36] 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, 2017. doi: 10.1137/1.9781611974782.124. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611974782.124.

[37] R. Cohen, S. Havlin, and D. ben-Avraham. Efficient immunization strategies for computer networks and populations. *Phys. Rev. Lett.*, 91:247901, Dec 2003. doi: 10.1103/PhysRevLett.91.247901. URL http://link.aps.org/doi/10.1103/PhysRevLett.91.247901.

[38] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, August 1973. ISSN 0022-0000. doi: 10.1016/S0022-0000(73)80029-7. URL http://dx.doi.org/10.1016/S0022-0000(73)80029-7.

[39] K. D. Cooper, 2003. Personal communication.

[40] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-38870, Rice Computer Science, 2006.

[41] Andxor Corporation. View2sign. http://www.view2sign.com/supported-signatures.html, 2013. [Online; accessed 01-April-2014].

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320.

[43] C. Demetrescu, A.V. Goldberg, and D.S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http:// www.dis.uniroma1.it/~challenge9/, 2007.

[44] W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-connectivity in directed graphs: An experimental study. In *Proc. 17th Workshop on Algorithm Engineering and Experiments*, pages 173–187, 2015. doi: 10.1137/1.9781611973754.15.

[45] Marco Di Summa, Andrea Grosso, and Marco Locatelli. Branch and cut algorithms for detecting critical nodes in undirected graphs. *Computa-*

tional Optimization and Applications, 53(3):649–680, 2012. ISSN 1573-2894. doi: 10.1007/s10589-012-9458-y. URL http://dx.doi.org/10.1007/s10589-012-9458-y.

[46] T. N. Dinh, Y. Xuan, M. T. Thai, P. M. Pardalos, and T. Znati. On new approaches of assessing network vulnerability: Hardness and approximation. *IEEE/ACM Trans. Netw.*, 20(2):609–619, April 2012. ISSN 1063-6692. doi: 10.1109/TNET.2011.2170849. URL http://dx.doi.org/10.1109/TNET.2011.2170849.

[47] Thang N. Dinh, Ying Xuan, My T. Thai, Panos M. Pardalos, and Taieb Znati. On new approaches of assessing network vulnerability: Hardness and approximation. *IEEE/ACM Trans. Netw.*, 20(2):609–619, April 2012. ISSN 1063-6692. doi: 10.1109/TNET.2011.2170849. URL http://dx.doi.org/10.1109/TNET.2011.2170849.

[48] Richard Duncan. An overview of different authentication methods and protocols. Technical report, SANS Institute, October 2016. URL https://www.sans.org/reading-room/whitepapers/authentication/overview-authentication-methods-protocols-118.

[49] Ya. M. Erusalimskii and G. G. Svetlov. Bijoin points, bibridges, and biblocks of directed graphs. *Cybernetics*, 16(1):41–44, 1980. doi: 10.1007/BF01099359.

[50] A. Cayley Esq. On the theory of the analytical forms called trees. *Philosophical Magazine Series 4*, 13(85):172–176, 1857. doi: 10.1080/14786445708642275. URL http://www.tandfonline.com/doi/abs/10.1080/14786445708642275.

[51] EUROSYSTEM European Central Bank. Recommendations for the security of internet payments, final version after public consultation. https://www.ecb.europa.eu/pub/pdf/other/recommendationssecurityinternetpaymentsoutcomeofpc201301en.pdf?9dcf429ac9158818771d9bbfacc62215, 31 January 2013.

[52] Shimon Even. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716780445.

[53] Simon Even. *Graph Algorithms*. Computer Science Press Inc, 1979.

[54] Graham Everest, Alf van der Poorten, Igor Shparlinski, and Thomas Ward. *Recurrence Sequences*, volume 104. American Mathematical Society, 2003. ISBN 978-0-8218-3387-2.

[55] Marcos Faundez-Zanuy. On-line signature recognition based on VQ-DTW. *Pattern Recognition*, 40(3):981–992, 2007. doi: 10.1016/j.patcog.2006.06.007. URL http://dx.doi.org/10.1016/j.patcog.2006.06.007.

[56] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[57] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. ISSN 1570-8667. doi: http://dx.doi.org/10.1016/j.jda.2013.10.003.

[58] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 434–43, 1990.

[59] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32th IEEE Symp. on Foundations of Computer Science*, pages 812–821, 1991. doi: 10.1109/SFCS.1991.185453. Full version: CU-CS-545-91, Dept. of Computer Science, University of Colorado at Boulder, 1991.

[60] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000.

[61] H. N. Gabow. Graph theory definitions. Lecture note, 2008. URL https://www.cs.colorado.edu/~hal/Papers/DFS/alldfs.pdf.

[62] H. N. Gabow. A poset approach to dominator computation. Unpublished manuscript 2010, revised unpublished manuscript, 2013.

[63] H. N. Gabow. The minset-poset approach to representations of graph connectivity. Unpublished manuscript, 2013.

[64] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.

[65] L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint *s-t* paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

[66] L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

[67] L. Georgiadis and R. E. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.

[68] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms*, pages 433–442, 2005.

[69] L. Georgiadis and R. E. Tarjan. Dominator tree certification and independent spanning trees. *CoRR*, abs/1210.8303, 2012.

[70] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

[71] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms*, pages 1988–2005, 2015.

[72] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015.

[73] L. Georgiadis, G. F. Italiano, and N. Parotsidis. A New Framework for Strong Connectivity and 2-Connectivity in Directed Graphs. *ArXiv e-prints*, November 2015.

[74] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016. ISSN 1549-6325. doi: 10.1145/2968448. URL http://doi.acm.org/10.1145/2968448.

[75] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures. In *Proc. 28th ACM-SIAM Symp. on Discrete Algorithms*, 2017. To appear.

[76] Loukas Georgiadis. *Linear-Time Algorithms for Dominators and Related Problems*. PhD thesis, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08540-5233, November 2005. URL ftp://ftp.cs.princeton.edu/reports/2005/737.pdf.

[77] Loukas Georgiadis, Luigi Laura, Nikos Parotsidis, and Robert E. Tarjan. *Loop Nesting Forests, Dominators, and Applications*, pages 174–186. Springer International Publishing, Cham, 2014. ISBN 978-3-319-07959-2. doi: 10.1007/978-3-319-07959-2_15. URL http://dx.doi.org/10.1007/978-3-319-07959-2_15.

[78] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, October 1965. ISSN 0004-5411. doi: 10.1145/321296.321300. URL http://doi.acm.org/10.1145/321296.321300.

[79] Paul A. Grassi and James L. Fenton. Draft nist special publication 800-63-3: Digital authentication guideline. Technical Report 800-63-3, National Institute of Standards and Technology, U.S. Department of Commerce, TBD 2016. URL https://pages.nist.gov/800-63-3/sp800-63-3.html.

[80] Frank Harary. *Graph Theory*. Addison-Wesley series in mathematics. Addison-Wesley Publishing Company, 1969. ISBN 9780201410334. URL https://books.google.com/books?id=9nOljWrLzAAC.

[81] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997. doi: 10.1145/262004.262005.

[82] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[83] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015)*, 2015.

[84] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015.

[85] R. M. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999. ISSN 1432-0541. doi: 10.1007/PL00009268. URL http://dx.doi.org/10.1007/PL00009268.

[86] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973. doi: 10.1137/0202024. URL http://dx.doi.org/10.1137/0202024.

[87] W. Issel. Aho, a.v., j. e. hopcroft, j. d. ullman: Data structures and algorithms. addison-wesley amsterdam 1983. 436 s. *Biometrical Journal*, 26(4):390–390, 1984. ISSN 1521-4036. doi: 10.1002/bimj.4710260406. URL http://dx.doi.org/10.1002/bimj.4710260406.

[88] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. ISSN 0304-3975. doi: 10.1016/j.tcs.2011.11.011. URL http://www.sciencedirect.com/science/article/pii/S0304397511009303.

[89] R. Jaberi. Computing the 2-blocks of directed graphs. *CoRR*, abs/1407.6178, 2014.

[90] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *CoRR*, abs/1401.6000, 2014.

[91] R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015. doi: 10.1051/ita/2015001. URL http://dx.doi.org/10.1051/ita/2015001.

[92] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/j.dam.2015.10.001. URL http://www.sciencedirect.com/science/article/pii/S0166218X15004886.

[93] Anil K Jain, Friederike D Griess, and Scott D Connell. On-line signature verification. *Pattern recognition*, 35(12):2963–2972, 2002.

[94] Richard K. Guy John H. Conway. *The Book of Numbers*. Springer New York, 1996. ISBN 978-1-4612-4072-3.

[95] Meenakshi K Kalera, Sargur Srihari, and Aihua Xu. Offline signature verification and identification using distance statistics. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(07):1339–1360, 2004. doi: 10.1142/S0218001404003630. URL http://dx.doi.org/10.1142/S0218001404003630.

[96] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM. ISBN 1-58113-737-0. doi: 10.1145/956750.956769. URL http://doi.acm.org/10.1145/956750.956769.

[97] G. Kirchhoff. Ueber die auflösung der gleichungen, auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird. *Annalen der Physik*, 148(12):497–508, 1847. ISSN 1521-3889. doi: 10.1002/andp.18471481202. URL http://dx.doi.org/10.1002/andp.18471481202.

[98] Thomas P. Kirkman. On the representation of polyedra. *Philosophical Transactions of the Royal Society of London*, 146:413–418, 1856. ISSN 02610523. URL http://www.jstor.org/stable/108593.

[99] Don Knuth. The on-line encyclopedia of integer sequences. https://oeis.org/A046859, Sequence A046859. Simplified Ackermann function (main diagonal of Ackermann-Péter function).

[100] Donald E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Massachusetts, 1968.

[101] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM, New York, NY, USA, 1993. ISBN 0-201-54275-7.

[102] A.N. Kolmogorov and V.A. Uspenskij. On the definition of an algorithm. *Transl., Ser. 2, Am. Math. Soc.*, 29:217–245, 1963. ISSN 0065-9290.

[103] V. Krebs. Uncloaking terrorist networks. *First Monday*, 7(4), 2002. ISSN 13960466. URL http://firstmonday.org/ojs/index.php/fm/article/view/941.

[104] Ram P Krish, Julian Fierrez, Javier Galbally, and Marcos Martinez-Diaz. Dynamic signature verification on smart phones. In *Highlights on Practical Applications of Agents and Multi-Agent Systems*, pages 213–222. Springer, 2013.

[105] C. J. Kuhlman, V. S. Anil Kumar, M. V. Marathe, S. S. Ravi, and D. J. Rosenkrantz. Finding critical nodes for inhibiting diffusion of complex contagions in social networks. In *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part II*, ECML PKDD'10, pages 111–127, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15882-X, 978-3-642-15882-7. URL http://dl.acm.org/citation.cfm?id=1888305.1888314.

[106] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

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

[108] S.-A.-J. L'Huillier. Mémoire sur la polyèdrométrie. *Annales de Mathématiques*, 3:169–189, 1812.

[109] J.B. Listing. *Vorstudien zur Topologie*. Vandenhoeck und Ruprecht, 1848. URL https://books.google.it/books?id=12cLAAAAYAAJ.

[110] Marcus Liwicki, Muhammad Imran Malik, C Elisa van den Heuvel, Xiaohong Chen, Charles Berger, Reinoud Stoel, Michael Blumenstein, and Bryan Found. Signature verification competition for online and offline skilled forgeries (sigcomp2011). In *International Conference on Document Analysis and Recognition (ICDAR)*, pages 1480–1484. IEEE, 2011.

[111] Musa Mailah and Boon Han Lim. Biometric signature verification using pen position, time, velocity and pressure parameters. *Jurnal Teknologi*, 48(1):35–54, 2012. doi: 10.11113/jt.v48.218. URL http://dx.doi.org/10.11113/jt.v48.218.

[112] S. Makino. An algorithm for finding all the k-components of a digraph. *International Journal of Computer Mathematics*, 24(3–4):213–221, 1988.

[113] Muhammad Imran Malik, Marcus Liwicki, Linda Alewijnse, Wataru Ohyama, Michael Blumenstein, and Bryan Found. Icdar 2013 competitions on signature verification and writer identification for on-and offline skilled forgeries (sigwicomp 2013). In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 1477–1483. IEEE, 2013.

[114] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201120372.

[115] D.W. Matula and R.V. Vohra. Calculating the connectivity of a directed graph. Technical Report 386, Institute for Mathematics and Application, University of Minnesota, 1988.

[116] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 115–124, 2010.

[117] Erin McTigue, Elaine Thornton, and Patricia Wiese. Authentication projects for historical fiction: Do you believe it? *The Reading Teacher*, 66(6):495–505, 2013. ISSN 1936-2714. doi: 10.1002/TRTR.1132. URL http://dx.doi.org/10.1002/TRTR.1132.

[118] Kurt Mehlhorn. *Graph Algorithms and NP-completeness*. Springer-Verlag New York, Inc., New York, NY, USA, 1984. ISBN 0-387-13641-X.

[119] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM J. Comput.*, 17(6):1093–1102, December 1988. ISSN 0097-5397. doi: 10.1137/0217070. URL http://dx.doi.org/10.1137/0217070.

[120] Aitor Mendaza-Ormaza, Oscar Miguel-Hurtado, Ramon Blanco-Gonzalo, and Francisco-Jose Diez-Jimeno. Analysis of handwritten signature performances using mobile devices. In *Security Technology (ICCST), 2011 IEEE International Carnahan Conference on*, pages 1–6. IEEE, 2011.

[121] Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10 (1):96–115, 1927. URL http://eudml.org/doc/211191.

[122] Oscar Miguel-Hurtado, Luis Mengibar-Pozo, Michael G Lorenz, and Judith Liu-Jimenez. On-line signature verification by dynamic time warping and Gaussian mixture models. In *International Carnahan Conference on Security Technology*, pages 23–29. IEEE, 2007. doi: 10.1109/CCST.2007.4373463. URL http://dx.doi.org/10.1109/CCST.2007.4373463.

[123] S. S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[124] Meinard Müller. *Information retrieval for music and motion*, volume 2. Springer, 2007.

[125] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008. 1st edition.

[126] H. Nagamochi and T. 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(4):513–517, apr 1993. ISSN 09168508. URL http://ci.nii.ac.jp/naid/110003215497/en/.

[127] H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76–A.4:513–517, 1993.

[128] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co., 1971. ISBN 0070465738.

[129] Committee on National Security Systems. National information assurance (ia) glossary. https://www.ncsc.gov/nittf/docs/CNSSI-4009_National_Information_Assurance.pdf, 26 April 2010.

[130] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/.

[131] Nilakantha Paudel, Marco Querini, and Giuseppe F. Italiano. Handwritten signature verification for mobile phones. In *2nd International Conference on Information Systems Security and Privacy*, pages 46–52, 2016.

[132] Nilakantha Paudel, Loukas Georgiadis, and Giuseppe F. Italiano. Computing critical nodes in directed graphs. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 43–57, 2017. doi: 10.1137/1.9781611974768.4. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611974768.4.

[133] Nilakantha Paudel, Marco Querini, and Giuseppe F. Italiano. *Online Handwritten Signature Verification for Low-End Devices*, pages 25–43. Springer International Publishing, Cham, 2017. ISBN 978-3-319-54433-5. doi: 10.1007/978-3-319-54433-5_3. URL http://dx.doi.org/10.1007/978-3-319-54433-5_3.

[134] A Piyush Shanker and AN Rajagopalan. Off-line signature verification using DTW. *Pattern Recognition Letters*, 28(12):1407–1414, 2007.

[135] Pólya Prize. https://www.siam.org/prizes/sponsored/polya.php, established in 1969.

[136] Paul W. Purdom, Jr. and Edward F. Moore. Immediate predominators in a directed graph [h]. *Commun. ACM*, 15(8):777–778, August 1972. ISSN 0001-0782. doi: 10.1145/361532.361566. URL http://doi.acm.org/10.1145/361532.361566.

[137] Yu Qiao, Jianzhuang Liu, and Xiaoou Tang. Offline signature verification using online handwriting registration. In *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07*, pages 1–8. IEEE, 2007. doi: 10.1109/CVPR.2007.383263. URL http://dx.doi.org/10.1109/CVPR.2007.383263.

[138] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

[139] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, 1999. doi: 10.1145/316686.316687.

[140] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002. doi: 10.1145/570886.570887.

[141] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.

[142] J. H. Reif and P. G. Spirakis. Strong k-connectivity in digraphs and random digraphs. Technical Report TR-25-81, Harvard University, 1981.

[143] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980. doi: 10.1137/0209036. URL http://dx.doi.org/10.1137/0209036.

[144] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51425-7.

[145] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3):141–153, 1980.

[146] Y. Shen, N. P. Nguyen, Y. Xuan, and M. T. Thai. On the discovery of critical links and nodes for assessing network vulnerability. *IEEE/ACM Trans. Netw.*, 21 (3):963–973, June 2013. ISSN 1063-6692. doi: 10.1109/TNET.2012.2215882. URL http://dx.doi.org/10.1109/TNET.2012.2215882.

[147] Deb Shinder. Understanding and selecting authentication methods. http://www.techrepublic.com/article/understanding-and-selecting-authentication-methods/, 28 August 2001.

[148] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996. doi: 10.1145/236114.236115.

[149] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, 1993.

[150] Yngve Sundblad. The ackermann function. a theoretical, computational, and formula manipulative study. *BIT Numerical Mathematics*, 11(1):107–119, 1971. ISSN 1572-9125. doi: 10.1007/BF01935330. URL http://dx.doi.org/10.1007/BF01935330.

[151] SutiDSignature. Sutidsignature. http://www.sutisoft.com/sutidsignature, 2013. [Online; accessed 01-April-2014].

[152] P. H. Sweany and S. J. Beaty. Dominator-path scheduling: A global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 260–263, 1992.

[153] J. J. SYLVESTER. Chemistry and algebra. *Nature*, 17(432):284–284, feb 1878. doi: 10.1038/017284a0. URL http://dx.doi.org/10.1038/017284a0.

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

[155] R. E. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 96–107, 1973.

[156] R. E. Tarjan. Edge-disjoint spanning trees, dominators, and depth-first search. Technical report, Stanford University, Stanford, CA, USA, 1974.

[157] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[158] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[159] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[160] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. ISSN 0004-5411.

[161] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

[162] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[163] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–27, 1979.

[164] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110 – 127, 1979. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(79)90042-4. URL http://www.sciencedirect.com/science/article/pii/0022000079900424.

[165] Jarrod Trevathan and Alan McCabe. Remote handwritten signature authentication. In *ICETE*, pages 335–339. Citeseer, 2005.

[166] Dawn M. Turner. Digital authentication – the basics. https://www.cryptomathic.com/news-events/blog/digital-authentication-the-basics, Retrieved 9 August 2016.

[167] M. Ventresca and D. Aleman. A randomized algorithm with local search for containment of pandemic disease spread. *Computers and Operations Research*, 48:11 – 19, 2014. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2014.02.003. URL http://www.sciencedirect.com/science/article/pii/S030505481400029X.

[168] M. Ventresca and D. Aleman. Efficiently identifying critical nodes in large complex networks. *Computational Social Networks*, 2(1):1–16, 2015. ISSN 2197-4314. doi: 10.1186/s40649-015-0010-y. URL http://dx.doi.org/10.1186/s40649-015-0010-y.

[169] Mario Ventresca. Global search algorithms using a combinatorial unranking-based problem representation for the critical node detection problem. *Comput. Oper. Res.*, 39(11):2763–2775, November 2012. ISSN 0305-0548. doi: 10.1016/j.cor.2012.02.008. URL http://dx.doi.org/10.1016/j.cor.2012.02.008.

[170] Mario Ventresca and Dionne Aleman. A derandomized approximation algorithm for the critical node detection problem. *Comput. Oper. Res.*, 43:261–270, March 2014. ISSN 0305-0548. doi: 10.1016/j.cor.2013.09.012. URL http://dx.doi.org/10.1016/j.cor.2013.09.012.

[171] Alexander Veremyev, Vladimir Boginski, and Eduardo L. Pasiliao. Exact identification of critical nodes in sparse networks via new compact formulations. *Optimization Letters*, 8(4):1245–1259, 2014. ISSN 1862-4480. doi: 10.1007/s11590-013-0666-x. URL http://dx.doi.org/10.1007/s11590-013-0666-x.

[172] Alexander Veremyev, Oleg A. Prokopyev, and Eduardo L. Pasiliao. An integer programming framework for critical elements detection in graphs.

Journal of Combinatorial Optimization, 28(1):233–273, 2014. ISSN 1573-2886. doi: 10.1007/s10878-014-9730-4. URL http://dx.doi.org/10.1007/s10878-014-9730-4.

[173] Mitsuko Wate-Mizuno. Mathematical recreations of dénes könig and his work on graph theory. *Historia Mathematica*, 41(4):377 – 399, 2014. ISSN 0315-0860. doi: http://dx.doi.org/10.1016/j.hm.2014.06.001. URL http://www.sciencedirect.com/science/article/pii/S0315086014000573.

[174] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998. ISBN 0201361221.

[175] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.

[176] Xyzmo. Xyzmo signature solution. http://www.xyzmo.com/en/products/Pages/Signature-Verification.aspx, 2013. [Online; accessed 01-April-2014].