Cohesive Subgraph Detection on Large Graphs

by

Bohua Yang

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Australian Artificial Intelligence Institute (AAII)
Faculty of Engineering and Information Technology (FEIT)
University of Technology Sydney (UTS)

April, 2021

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Bohua Yang declare that this thesis, is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 09/04/2021

ACKNOWLEDGEMENTS

First of all, I would like to deliver my sincere gratitude to my principal supervisor A/Prof. Lu Qin. Thanks for his selfless help during my PhD life. He is a good mentor and friend to me. He is very professional; he is a perfect guide who leads me to the beautiful world of the graph. He is very patient; he is always ready to help me whenever I meet problems in research or life. I could not finish my thesis without his illuminating instructions.

Secondly, I would like to thank my co-supervisor Prof. Ying Zhang. He gives me much accurate and valuable advice on my research topics. Besides, he often encourages and guides me to optimize my career development plan.

Thirdly, I would like to thank Dr. Dong Wen. All the research works presented in this thesis are conducted together with him. He patiently guides me in acquiring research skills, such as thinking about problems, writing papers, and preparing presentations. I learn a lot from the collaboration with him.

Fourthly, I would like to thank Prof. Xuemin Lin and Dr. Lijun Chang for supporting the works in this thesis. I would like to thank Prof. Lin for offering an interesting but rigorous research environment. I would like to thank Dr. Chang for his insightful advice on my research works.

Fifthly, I am very grateful to A/Prof. Xin Zhao, my mentor at the Renmin

University of China, for stimulating my research interest and providing helpful advice for my academic career. Furthermore, I would like to thank Prof. Jirong Wen, A/Prof. Hui Sun, Prof. Qing Zhu, and A/Prof. Yan Yu at the Renmin University of China for their kind help.

I would also like to appreciate Prof. Jeffrey Xu Yu, Prof. Wenjie Zhang, Prof. Ronghua Li, A/Prof. Ling Chen, Dr. Yulei Sui, Dr. Xin Cao, A/Prof. Zengfeng Huang, Dr. Shiyu Yang, A/Prof. Yixiang Fang, Prof. Weiwei Liu, and Prof. Xiaoyang Wang for sharing brilliant thoughts and experiences. Thanks to Dr. Dian Ouyang, Dr. Wentao Li, Dr. Conggai Li, Dr. Mingjie Li, Dr. Adi Lin, Dr. Yuan Liang, Mr. Yuxuan Qiu, Mr. Junhua Zhang, Dr. Fan Zhang, Dr. Wanqi Liu, Mr. Hanchen Wang, Mr. Yilun Huang, Dr. Xubo Wang, Dr. Longbin Lai, Dr. Long Yuan, Dr. Xing Feng, Dr. Haida Zhang, Dr. Wei Li, Dr. Chen Zhang, Dr. Yang Yang, Dr. Kai Wang, Dr. You Peng, Dr. Boge Liu, Mr. Yuanhang Yu, Mr. Peilun Yang, Ms. Xiaoshuang Chen, Mr. Xuefeng Chen, Mr. Yuren Mao, Mr. Yixing Yang, Mr. Zhengyi Yang, Mr. Qingyuan Linghu, Mr. Michael Ruisi Yu, Mr. Chenji Huang, Mr. Yu Hao, Mr. Kongzhang Hao, Dr. Peng Zhang, Mr. Xunxiang Yao, Dr. Binbin Huang, Dr. Yarui Chen, Ms. Kun Wang, Mr. Yijun Li, and Ms. Ruilin Liu. The days we spent together bear unforgettable memories.

Finally, I would like to thank my grandfather Mr. Zhencai Yang, and my grandmother Ms. Yuling Zhao, who give me selfless love and endless encouragement. I would like to thank the support from other relatives and friends.

ABSTRACT

Graphs have been widely used to model sophisticated relationships between different entities due to their strong representative properties. Social networks, traffic networks, and biological networks are among the applications that benefit from being expressed as graphs. The cohesive subgraph is an essential structure for understanding the organization of many real-world networks, and cohesive subgraph detection is a crucial problem in network analysis. There are many cohesive subgraph models, such as k-core, strongly connected component, and maximum density subgraph.

Uncertain graph management and analysis have attracted much research attention. Among them, computing k-cores in uncertain graphs (aka, (k, η)-cores) is an important problem and has emerged in many applications. However, the existing algorithms for computing (k, η)-cores heavily depend on the two input parameters k and η. In addition, computing and updating the η-degree for each vertex is the costliest component in the algorithm, and the cost is high.

To overcome these drawbacks, we propose an index-based solution for computing (k, η)-cores. The index size is well-bounded by $O(m)$, where m is the number of edges in the graph. Based on the index, queries for any k and η can be answered in optimal time. We propose an algorithm for index construction with several different optimizations.

We also discuss the (k, η)-core computation when graphs cannot be entirely stored in memory. We adopt the semi-external setting, which allows $O(n)$ mem-

ory usage, where n is the number of vertices in the graph. This assumption is reasonable in practice, and it has been widely adopted in massive graph analysis. We design an index-based solution for I/O efficient (k, η)-core computation.

Given the frequent updates in many real-world graphs, detecting strongly connected components (SCC) in dynamic graphs is a very complicated problem. In the thesis, we study the fully dynamic depth-first search (DFS) problem in directed graphs, which is a crucial basis of dynamic SCC detection. In the literature, most works focus on the dynamic DFS problem in undirected graphs and directed acyclic graphs. However, their methods cannot easily be applied in the case of general directed graphs. Motivated by this, we propose a framework and corresponding algorithms for both edge insertion and deletion in general directed graphs. We further give several optimizations to speed up the algorithms.

We conduct extensive experiments on several large real-world graphs to practically evaluate the performance of all proposed algorithms.

PUBLICATIONS

- ***Bohua Yang***, *Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 64-75. IEEE, 2019.*

- ***Bohua Yang***, *Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. Fully Dynamic Depth-First Search in Directed Graphs. Proceedings of the VLDB Endowment, 13(2):142-154, 2019.*

- *Dong Wen,* ***Bohua Yang***, *Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. Computing K-Cores in Large Uncertain Graphs: An Index-based Optimal Approach. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2020.*

- *Dong Wen,* ***Bohua Yang***, *Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. Span-Reachability Querying in Large Temporal Graphs. The VLDB Journal, Revision Submitted.*

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The graph is a critical data structure that has many applications, such as social networks, transport networks, knowledge bases and biological networks. Specifically, a social network can be modeled as a large graph where users are vertices, and the friend connections are edges. And there are many cohesive subgraphs in a social network because people are always willing to make friends with others who have the similar interest or experience with them. In graph theory, a cohesive structure in a graph is a group of vertices that are similar to each other and different from the rest of the graph, and the cohesive subgraph is the induced subgraph of cohesive structure. Many network applications are developed based on cohesive subgraphs. For example, the cohesive subgraph is useful for friend recommendation because users in a cohesive subgraph are more likely to be friends. Therefore, cohesive subgraph detection is a fundamental and crucial problem in network analysis.

Even though the cohesive subgraph detection problem has been studied for several decades, several specific problems are still insufficiently studied. In the thesis, we select two fundamental and important problems: core computation [58] in large uncertain graphs and strongly connected component detection [21]

in large dynamic graphs. For the first problem, we design index-based algorithms under internal memory and semi-external memory settings separately. For the second problem, we study the fully dynamic depth-first search problem in large directed graphs, an essential basis of the original problem. Several algorithms based on depth-first search can compute strongly connected component in linear time [59, 63, 24]. The algorithm for solving the dynamic strongly connected component detection will emerge from the solution of dynamic depth-first search.

1.1 Core Computation in Large Uncertain Graphs

Many real-world applications contain uncertainty in the form of noise [2], measurement errors [1], the accuracy of predictions [47], privacy concerns [12], and so on. These uncertain relationships are often modeled as an uncertain graph, where the actual existence of each edge is assigned an "existence probability".

A large number of studies on uncertain graph analysis and management have involved combining fundamental graph problems with uncertain graph models. These studies span a range of tasks, such as reliability searches[40], frequent pattern mining [78] and dense subgraph detection [39]. Among the solutions, k-core is a popular and well-studied cohesive subgraph metric [58], and the k-core conception in the uncertain graph model is originally formalized in [13].

k-Cores in Deterministic Graphs. Given a deterministic graph, a k-core is a maximal connected subgraph in which each vertex has a degree of at least k [58]. k-cores are computed by iteratively removing the vertex with the minimum degree and incident edges. This is done in linear time. Computing k-cores has a large number of real-world applications: community detection [22, 32], network visualization [4], network topology analysis [58], system structure analysis [74],

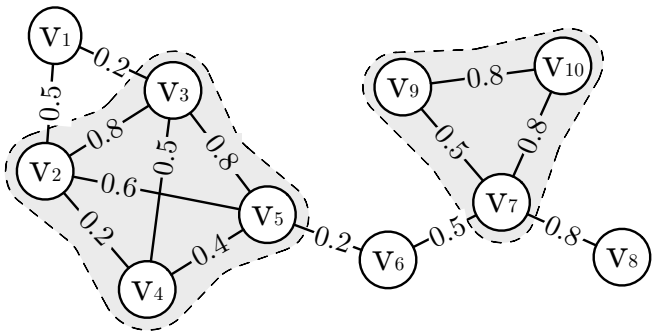


Figure 1.1: The (k, η)-cores of $\mathcal{G}$ for $k = 2$ and $\eta = 0.3$

protein-protein interaction network analysis [6], and so on. It also serves to find an approximation result for densest subgraph [5], betweenness score [34].

$(\mathbf{k}, \eta)$-Cores in Uncertain Graphs. In the context of uncertain graph models, the degree of each vertex is uncertain. A (k, η)-core model in uncertain graphs is formalized in [13]. A (k, η)-core is a maximal subgraph in which each vertex has at least a probability of η that the degree of this vertex is at least k. Note that, in this study, we have imposed a connectivity constraint to ensure the cohesiveness of the resulting subgraph, i.e., a (k, η)-core is connected. Figure 1.1 illustrates an example of the (k, η)-cores. Here, given an integer $k = 2$ and a probability threshold $\eta = 0.3$, the uncertain graph contains two $(2, 0.3)$-cores — $\mathcal{G}[\{v_2, v_3, v_4, v_5\}]$ and $\mathcal{G}[\{v_7, v_9, v_{10}\}]$. Computing the (k, η)-cores can be naturally applied in the aforementioned areas. For example, in DBLP collaboration network, each vertex represents an author, and edges represent co-authorships. The edge probability is an exponential function based on the number of collaborations [52]. A (k, η)-core in this case may be a research group. In addition, [13] introduced some specific applications for (k, η)-cores associated with uncertain graph models, such as influence maximization and task-driven team formation.

Given an uncertain graph $\mathcal{G}$, an integer k and a probability threshold η, this

work explores the problem of efficiently computing all the (k, η)-cores in $\mathcal{G}$. In other words, our aim is to compute a set of vertex sets, and the induced subgraph of each vertex set is a (k, η)-core.

The Online Approach. In [13], (k, η)-cores are derived using an η-core decomposition algorithm. The algorithm computes an η-core number for each vertex u in $\mathcal{G}$, where the η-core number for u is the largest integer k such that a (k, η)-core containing u exists. Let the η-degree of a vertex u be the largest possible degree such that the probability of u to have that degree is no less than η. The algorithm iteratively removes the vertex with the minimum η-degree and updates the η-degrees of the neighbors. With a small modification, this algorithm could compute all the (k, η)-cores in our problem. Specifically, we can iteratively remove all the vertices with η-degrees of less than k and derive a set of resulting vertices. The final result can then be generated by performing a connected component detection procedure.

Motivation. Even though the online approach can successfully compute the (k, η)-cores, several challenges remain:

- *Parameters Tuning.* The results heavily depend on two input parameters, k and η, and these parameter settings usually depend on the topological structure of the input graph along with user's subjective requirements. To arrive at a satisfying result, users may need to run the algorithm several times to properly tune the parameters.

- *Query Efficiency.* Computing and updating the η-degree for each vertex is costly and accounts for the majority of the running time in the algorithm. Even though [13] proposes a dynamic programming approach to partially offset this problem, the algorithm is still time-intensive and is not scalable to large uncertain graphs.

An Index-based Approach. Motivated by these challenges, we have developed an order-based index structure, called $UCO\text{-}Index$. The general idea is to retain the resulting vertices for every possible k and η. Specifically, a probability order for each vertex is maintained. Given an integer k and a probability threshold η, a vertex in the result set is identified by comparing the k-th value in the order for the vertex with η. The final result is then produced by performing a connected component detection on the vertex set. We have imposed a bound on the length of the order for each vertex according to the core number, i.e., the largest integer k such that a k-core exists containing this vertex. Therefore, the space for the $UCO\text{-}Index$ is well-bounded by $O(m)$, where m is the number of edges in the graph. The time complexity for query processing is $O(n + \sum_{u \in C} Deg(u))$ for every possible parameter setting of k and η, where n is the number of vertices and $\sum_{u \in C} Deg(u)$ is the sum of degrees of all the vertices in the result set C.

There is still room to reduce the amount of time it takes for query processing based on the $UCO\text{-}Index$. Hence, we further propose an alternative method for computing the (k, η)-cores based on a forest index structure, called $UCF\text{-}Index$. In this method, rather than maintaining the order of each vertex, $UCF\text{-}Index$ maintains a tree structure for each integer k. Each tree node contains a set of vertices, and a probability value is assigned to the tree node, which means a corresponding (k, η)-core that contains these vertices exists. The size of $UCF\text{-}Index$ is also bounded by $O(m)$. Using the $UCF\text{-}Index$, we make the time complexity of query processing optimal. In other words, let $|C|$ be the number of vertices in the result set. The running time of the query algorithm is bounded by $O(|C|)$.

Further, we have explored two optimizations to speed up construction of the index. The first one is called *core-based reduction*. By computing the core number of each vertex, some unnecessary neighbors of each vertex are pruned to

reduce the running time required to compute and update the probabilities for each vertex. This approach is especially effective in the last few iterations of the index construction algorithm. The second optimization is called *core-based ordering*. This approach avoids the need for repeated computations of each vertex in the dynamic programming schema as each iteration of index construction algorithm proceeds without breaking the correctness. Our experiments demonstrate a significant increase in speed as a result of these two optimizations.

I/O Efficient Query Processing. We also study an index-based solution when a graph cannot be entirely loaded in memory. We adopt the semi-external setting [69, 76], which allows $O(n)$ memory usage. The structure of $UCF\text{-}Index$ can be naturally stored in external memory. We use the same strategy of in-memory query processing to derive results from the external index and achieve the optimal I/O complexity $O(|C|/B)$. To construct $UCEF\text{-}Index$ in external memory, straightforwardly using the strategy of in-memory index construction incurs significant I/O cost due to frequent random accesses of external memory. We propose a new framework for external index construction and derive the index by sequentially accessing the external graph in several iterations. Several optimizations are also given to further improve the efficiency.

Contributions. The main contributions of this work are summarized as follows:

- *The first index-based solution for computing (k, η)-cores in uncertain graphs.* This study presents an effective index structure, called $UCF\text{-}Index$, for computing all the (k, η)-cores. The size of $UCF\text{-}Index$ is well-bounded by $O(m)$. To the best of our knowledge, this is the first index-based solution to this problem.

- *Optimal query processing.* We present an efficient query algorithm based on $UCF\text{-}Index$ for any possible k and η. The time complexity is optimal and linear to the number of vertices in the result set.

- *Optimizations for index construction.* We give two optimizations, *core-based reduction* and *core-based ordering*, to improve the efficiency of index construction.

- *Index Construction in External Memory.* We propose a new framework for index construction in external memory. Several optimizations are given to reduce I/O cost and further improve the efficiency.

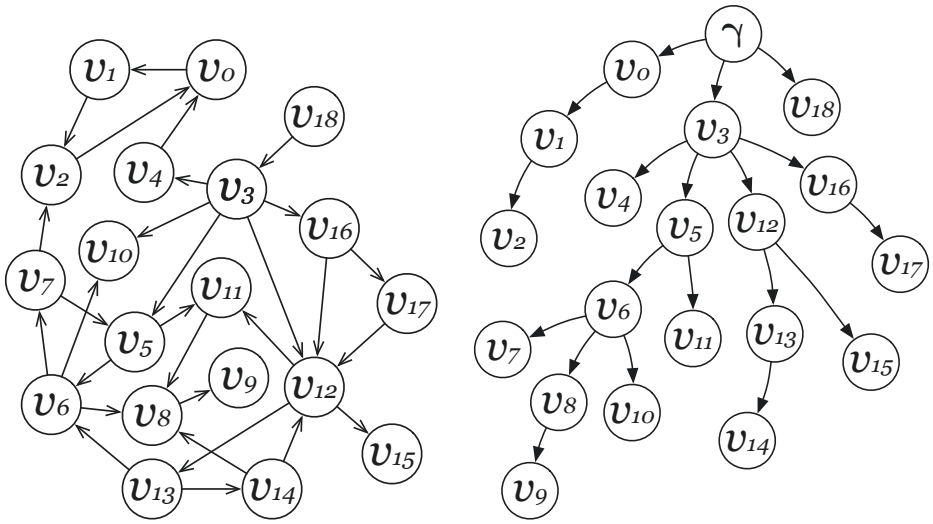
- *Extensive performance studies on both real-world and synthetic datasets.* Extensive experiments were conducted with all the proposed algorithms on eight real-world datasets. The results demonstrate that this index-based approach is several orders of magnitude faster than the online approach.

  The details of this work are presented in Chapter 3 and Chapter 4.

1.2 Fully Dynamic DFS in Large Directed Graphs

Depth-first search (DFS) [21] is an algorithm to traverse a graph. It searches the vertices along a graph as far as possible in each branch before backtracking. The process of a DFS is naturally represented as a search spanning tree following the depth-first order, named the DFS-Tree. Given a graph G in Figure 1.2(a), one possible DFS-Tree $\mathcal{T}$ of G is shown in Figure 1.2(b). The time complexity for performing a DFS traversal and generating a DFS-Tree in a graph $G(V, E)$ is $O(|V| + |E|)$ [63].

DFS is a fundamental algorithm in graph analysis and is the basis for efficiently solving numerous graph problems, such as testing graph reachability [73, 62], detecting strongly connected components[59, 63, 24], detecting biconnected components [35], finding graph bridges [64], finding paths, detecting cycles [67], testing bipartiteness, testing graph planarity [36, 23], and topological



(a) The graph G (b) A DFS-Tree $\mathcal{T}$ of G

Figure 1.2: An example graph G and its DFS-Tree $\mathcal{T}$ (γ is a virtual root connecting all vertices in G)

sorting [65]. These algorithms perform DFS traversal as a subroutine. They require access to vertices in the depth-first order.

In many real-world applications, graphs dynamically update over time. Given the importance of DFS, the DFS-Tree maintenance problem in dynamic directed graphs is insufficiently studied. In this work, we examine this problem, which is to update the DFS-Tree for an inserted or deleted edge. The aforementioned applications of DFS benefit from this study. Specifically, in many graph problems such as computing strongly connected components [63], biconnected components [35], and finding graph bridges [64], a key step is to compute the reachable ancestor with the lowest depth of each vertex in the DFS-Tree. Based on this study, we can simply finish this task by directly tracking the updated DFS-Tree of the graph. For example, in detecting biconnected components, it is required to compute a DFS-Tree of the graph and then traverse the tree to get the shallowest position that each vertex can reach. When the graph updates, we can derive an

updated DFS-Tree instead of performing the DFS traversal from scratch. We can also simultaneously maintain the interval label (discovery time and finish time) of each vertex as a byproduct in the DFS-Tree. The interval label is used in several works [73, 62] as a part of the index to test the graph reachability. These works filter out the queries if two vertices are connected in the tree, and it only takes constant time to check the reachability in the tree using the interval labels. Based on the study in this work, we can immediately derive the updated interval labels when the graph updates instead of rerunning DFS. In addition, in puzzle problems such as mazes, users can check the updated interval labels to efficiently identify the connectivity from the entrance to the goal when the maze updates. They can also directly search the updated DFS-Tree to find a solution.

Existing Works and Challenges. The DFS-Tree maintenance problem for directed acyclic graphs and undirected graphs has been well studied in the literature. However, these techniques cannot be applied to the DFS-Tree maintenance of general directed graphs.

For directed acyclic graphs, [29] and [8] investigate the DFS-Tree maintenance problem under incremental and decremental settings, respectively. Franciosa et al. [29] update the DFS-Tree by locating a range of vertices according to the postorder of the DFS-Tree. They only reconstruct the tree structure for the located range of vertices. Correctness is guaranteed by the property of directed acyclic graphs that there is no backward edge in its DFS-Tree. However, if we follow the same procedure as that in [29] in general directed graphs, a backward edge starting from a vertex in the located range and ending at a vertex outside the range may become a forward-cross edge after the update is finished. Here, an edge (s, t) is a forward-cross edge if s is visited before t in the preorder of the tree, and there is no ancestor-descendant relationship between s and t. A tree with any forward-cross edge is not a valid DFS-Tree. Considering a deleted tree

edge (u, v), the decremental algorithm proposed by Baswana and Choudhary [8] iteratively finds a new position for each vertex in the subtree of v following the topological order. The property of directed acyclic graphs guarantees that the in-neighbors of a vertex do not contain its descendants in the DFS-Tree, whereas in general directed graphs, a vertex may have a descendant as a potential parent which cannot be appended back to the tree.

Baswana et al. [7, 10] and Chen et al. [16, 17] propose fully dynamic algorithms to maintain the DFS-Tree in undirected graphs. They partition the DFS-Tree into disjoint subtrees and paths. The property of undirected graphs guarantees that there is no cross edge between subtrees, and the neighbor of a vertex appears either as its ancestor or its descendant in the DFS-Tree. However, these properties are not applicable to directed graphs since a directed graph may have cross edges in its DFS-Tree, and two adjacent vertices do not always have ancestor-descendant relationships.

Baswana et al. [9] design an incremental algorithm to maintain a DFS-Tree in general directed graphs based on the algorithm presented in [29]. They make use of a structure called stick, which is a long downward path from the root on which there is no branching after a large number of edge insertions [9]. However, the stick structure may be broken due to the edge deletion. Therefore, their algorithm cannot be easily used in the fully dynamic setting. Motivated by the above limitations, we propose efficient, easy-to-implement, and fully dynamic algorithms for DFS-Tree maintenance in general directed graphs.

Our Solution. Given a graph G and its DFS-Tree $\mathcal{T}$, it is necessary to update the DFS-Tree $\mathcal{T}$ if a forward-cross edge has been inserted or a tree edge has been deleted. We use the time intervals to efficiently check the edge type in constant time, where the time interval of a vertex u is an interval starting from the discovery timestamp and ending at the finish timestamp of u in DFS. For

the clarity of presentation, we add a virtual root γ connecting all vertices in the graph, so there always exists a DFS-Tree for any graph. An edge removal operation for edge (s, t) can be transformed into appending the subtree rooted at t to end of the children list of the virtual root γ, and this step may generate several new forward-cross edges due to the back movement of the subtree. Therefore, the tree update is essential to eliminate the forward-cross edges for both edge insertion and deletion. Instead of naively reconstructing the whole DFS-Tree, we first propose a general framework for the tree update based on the concept of time interval. The key step in the framework is to set a range called candidate interval. The candidate interval locates a small set of vertices. We replace the DFS-subtree induced by these vertices by performing DFS only for these vertices in the new graph, whereas the other part of the DFS-Tree remains unchanged. We give the implementations for both edge insertion and deletion. By carefully setting the candidate interval, the computed DFS-Tree is guaranteed to be valid.

To improve the algorithmic efficiency of the basic implementation, we propose several optimizations for both edge insertion and deletion from two perspectives. First, we aim to refine the candidate interval and reduce the number of influenced vertices. Instead of using a fixed candidate interval, we adopt a new strategy that dynamically updates the candidate interval during the process of DFS. We guarantee that the search scope is at most the same as that in the basic algorithm in the worst case. Second, we transform a part of the graph search to the tree search. Recall that in the basic implementation, we scan the out-neighbors of all located vertices in DFS to collect their children in the updated DFS-Tree. We observe that the (tree) children of a set of vertices in the old DFS-Tree can be reused in the updated DFS-Tree, so we avoid scanning the out-neighbors of these vertices in the graph. Our experiments show that the proportion of this kind of vertices is very large, and this optimization greatly speeds up the algorithm

especially in large graphs with many high-degree vertices.

Contributions. We summarize the main contributions in this work as follows.

- *A general and flexible framework.* We design a novel framework for both edge insertion and deletion. To the best of our knowledge, we are the first to study the fully dynamic DFS-Tree maintenance problem in general directed graphs from the perspective of practical implementation.
- *Easy-to-implement algorithms.* We develop algorithms based on the proposed framework for both operations. The algorithms are easy to implement in practice.
- *Two groups of optimizations.* We optimize the algorithms for both operations in two directions. One is to tighten the candidate interval. This reduces the search scope and guarantees that the running time of algorithms only depends on the neighbors of vertices whose visiting time has been changed in the updated DFS-Tree. The other one is to scan the children in the DFS-Tree instead of the out-neighbors in the graph for a large proportion of visited vertices. This optimization further improves the algorithmic efficiency.
- *Extensive experiments.* We conduct experiments on 12 real-world networks to show the performance of our proposed algorithms and the effectiveness of our optimizations.

The details of this work are presented in Chapter 5.

1.3 Graph Model

We consider undirected graphs in Chapter 3 and Chapter 4. Given a deterministic undirected graph $G(V, E)$, V is the set of vertices and E is the set of

edges. Given a vertex u in G, the neighbor set of u is denoted as $N(u, G)$, i.e., $N(u, G) = \{v \in V | (u, v) \in E\}$. The degree of u is denoted as $deg(u, G)$, i.e., $deg(u, G) = |N(u, G)|$. We use the terms $N(u)$ and $deg(u)$ for simplicity when the context is clear. Given a set of vertices V', the induced subgraph of V' is denoted as $G[V']$, i.e., $G[V'] = (V', \{(u, v) \in E | u, v \in V'\})$. Given an uncertain undirected graph $\mathcal{G}(V, E, p)$, p is a function that maps each edge to a probability value in $[0, 1]$ in addition to the vertex set V and the edge set E. The probability of an edge $e \in E$ is denoted by p_e. We denote the neighbor set and the degree of a vertex u in an uncertain graph $\mathcal{G}$ as $\mathcal{N}(u, \mathcal{G})$ and $Deg(u, \mathcal{G})$, respectively.

We consider directed graphs in Chapter 5. Given a directed graph $G(V, E)$, V is the set of vertices and E is the set of edges in G. The numbers of vertices and edges are denoted by n and m, respectively, i.e., $n = |V|$ and $m = |E|$. Given a vertex u, we denote the in-neighbors (resp. out-neighbors) of u by $N_{in}(u)$ (resp. $N_{out}(u)$), and denote the in-degree (resp. out-degree) of u by $d_{in}(u) = |N_{in}(u)|$ (resp. $d_{out}(u) = |N_{out}(u)|$).

Several frequently used notations are summarized in Table 1.1.

1.4 Roadmap

The rest of this thesis is organized as follows. Chapter 2 lists related works. Chapter 3 presents an index-based solution for computing (k, η)-cores. Chapter 4 describes an index-based solution for I/O efficient (k, η)-core computation. Chapter 5 contributes to the fully dynamic DFS in directed graphs. Chapter 6 concludes the whole thesis.

Table 1.1: Notations

Notation	Description
$N(u, G)$	the neighbors of vertex u in deterministic undirected graphs G
$deg(u, G)$	the degree of vertex u in deterministic undirected graphs G
$\mathcal{N}(u, \mathcal{G})$	the neighbors of vertex u in uncertain undirected graphs $\mathcal{G}$
$Deg(u, \mathcal{G})$	the degree of vertex u in uncertain undirected graphs $\mathcal{G}$
$G[V']$	the induced subgraph of vertex set V'
$core(u)$	the core number of vertex u
η-$core(u)$	the η-core number of vertex u
$N_{in}(u)$	the in-neighbors of vertex u in directed graphs
$N_{out}(u)$	the out-neighbors of vertex u in directed graphs
$d_{in}(u)$	the in-degree of vertex u in directed graphs
$d_{out}(u)$	the out-degree of vertex u in directed graphs
$\mathcal{C}(u)$	the children list of vertex u in the DFS-Tree $\mathcal{T}$
$\mathcal{T}(u)$	the vertex set in the subtree rooted at u in $\mathcal{T}$
$\mathcal{I}_{\mathcal{T}}(u)$	the time interval of u in $\mathcal{T}$
$\mathcal{T}[t]$	the visited vertex at the timestamp t in $\mathcal{T}$
$\mathcal{T}[l, r]$	the visited vertex set in the interval $[l, r]$ in $\mathcal{T}$
$\mathcal{T}(\mathcal{I})$	equivalent to $\mathcal{T}[\mathcal{I}.left, \mathcal{I}.right]$
$\mathcal{T}_{new}$	the updated DFS-Tree

Chapter 2

LITERATURE REVIEW

2.1 k-Core and Uncertain Graphs

The k-core is a typical cohesive subgraph model. It is a maximal connected subgraph in which each vertex has at least k neighbors within this subgraph [46]. For example, in a social network, a k-core cohesive subgraph is a maximal group of users in which every user has at least k friends within this group. k-core satisfies the structure cohesiveness. The core decomposition problem, a cohesive subgraph detection problem, is to find k-core of a graph for all k. This problem has been widely studied by researchers. One of the most important reasons is that it can be computed in linear time [11]. Compared with k-core, some similar definitions of cohesive subgraph are inefficient to calculate or even NP-hard.

Many real-world graphs need to be updated dynamically. For example, in a social network, a new edge is generated when someone adds a new friend. So [55] proposed a traversal algorithm to maintain the core number of each vertex. More detailed, without computing the entire graph, this algorithm locates a small subgraph that is guaranteed to contain all the vertices whose core number should be updated, and efficiently process this subgraph to update the k-core

decomposition [55]. However, [75] found that there are a lot of unnecessary computations in the traversal algorithm given by [55]. Therefore, they proposed an order-based algorithm for core maintenance which can outperform the traversal algorithm up to 3 orders of magnitude.

Most core decomposition algorithms assume that the memory can load the entire graphs. But many real-world graphs are too massive to reside in memory. So, [18] proposed the first external memory algorithm for core decomposition in large-scale graphs. More detailed, they developed a top-down algorithm that computes the k-core from larger values of k to smaller ones, and reduce search space and I/O cost by removing the vertices in each calculated k-core [18]. This algorithm first divides the graph into some small parts. Second, it estimates the upper bound of the core number of vertices in each part. Finally, it uses a top-down strategy to load as many as parts into memory and computes the core number of vertices by an in-memory approach [18]. [69] proposed a semi-external core decomposition algorithm which can bound the memory size. And this semi-external algorithm can support edge insertion and deletion.

A large number of real-world graphs are too massive to compute in a single machine. The graph is saved and calculated separately in more than one computer. We have to consider how many machines work together efficiently. Therefore, [49] proposed a distributed algorithm for core decomposition in the distributed environment. The convergence of continually tightening the upper bound of core number can be proved, and this algorithm can be implemented in many computational models [49].

For a query-dependent variant of the cohesive subgraph detection problem that computes a cohesive subgraph, based on minimal degree, containing all given query vertices, [61] proposed a greedy algorithm to get the optimal answer and analyze the monotonicity of minimal-degree measure and the distance

constraint. For the size constraint, they developed two heuristic algorithms to compute the cohesive subgraphs of size no larger than a specified upper bound [61]. [46] proposed a novel cohesive subgraph model named k-influential cohesive subgraph based on the notion of k-core. And they gave a linear-time online search algorithm to find the top-r k-influential cohesive subgraphs in a graph while the influence of vertex is denoted by PageRank [46].

Some cohesive subgraph search algorithms need to traverse the entire graph and visit all vertices. However, the most intuitive idea is that starting from the query vertex and searching neighbors iteratively until getting a good cohesive subgraph. [22] proposed a local search strategy for the cohesive subgraph search problem. They formulated two cohesive subgraph search problems: the cohesive subgraph search with a threshold constraint (CST) problem and the cohesive subgraph search with a maximality constraint (CSM) problem, then gave several solutions based on local search to solve these two issues.

Most existing research only considers the graphs which only have vertices and edges. But when it comes to some special graphs which not only have vertices and edges but also have other features, such as vertex attributes, edge attributes or edge probabilities, these research may not work. Many real-world graphs are uncertain graphs. For instance, in a social network, some people are friends even if they do not follow each other. So we can predict the existence probability of friend connections between users by using some data mining algorithms. The algorithms run on an uncertain graph with link-predictions may result in better cohesive subgraphs.

Many fundamental graph problems have been studied in uncertain graphs. Jin et al. [40] study the distance-constraint reachability problem on uncertain graphs. Potamias et al. [52] propose a framework to efficiently answer k-nearest neighbor queries over uncertain graphs. Gao et al. [31] study the problem of

reverse k-nearest neighbor search on uncertain graphs. Zou et al. [78] investigate the problem of discovering and mining frequent subgraph patterns in uncertain graphs. Jin et al. [39] consider the problem of discovering highly reliable subgraphs of uncertain graphs. The truss decomposition of uncertain graphs is studied by [38].

The first definition of k-core in uncertain graphs was given by [13]. They proposed a novel notion of (k, η)-core and provided an efficient algorithm based on dynamic programming to compute a (k, η)-core decomposition. The details of this approach are presented in Section 3.3. A variation for the (k, η)-core, denoted by (k, θ)-core, is proposed in [51] to capture the k-core probability of each individual vertex in the uncertain graph.

2.2 SCC and DFS

A strongly connected component (SCC) [21] of a directed graph is a maximal connected subgraph in which every vertex is reachable from every other vertex. Several algorithms based on DFS can compute SCC in linear time [59, 63, 24].

About the dynamic DFS problem, we first show the hardness of the ordered DFS-Tree maintenance problem, and then introduce the related works about the DFS-Tree maintenance problem in directed acyclic graphs, undirected graphs, and general directed graphs separately.

Reif [53] shows that the ordered DFS problem is a P-complete problem. Here, the ordered DFS problem traverses the graph according to the order specified by the adjacency lists, and the ordered DFS-Tree is unique. Reif [54] and Miltersen et al. [48] prove that the P-completeness of a problem implies hardness of the problem in a dynamic environment.

An efficient solution for ordered DFS-Tree maintenance can be transferred

to an efficient solution for some other dynamic problems [54], such as the acceptance of a linear time Turing machine, Boolean circuit evaluation [45], and unit resolution [41]. Miltersen et al. [48] show that the solution to every problem in class P is updatable in $O(poly \log n)$ time if the ordered DFS-Tree is updatable in $O(poly \log n)$ time. In brief, the ordered DFS-Tree maintenance problem is the hardest problem to solve of all the problems in class P.

For directed acyclic graphs, Franciosa et al. [29] propose an incremental algorithm to maintain a DFS-Tree under a sequence of edge insertions in $O(mn)$ total time. Baswana and Choudhary [8] propose a randomized decremental algorithm to maintain a DFS-Tree under a sequence of edge deletions with expected $O(mn \log n)$ total time.

In [29], the algorithm generates the rank of vertices by the postorder numbering of the given DFS-Tree and then updates the DFS-Tree by reconstructing the tree structure for only a set of vertices in a located rank range for each edge insertion. Its correctness is guaranteed by the property of directed acyclic graphs that there is no backward edge in the DFS-Tree. However, if we follow the same procedure in general directed graphs, a backward edge starting from a vertex in the located rank range and ending at a vertex outside the range may become a forward-cross edge after the update is finished. Thus, this algorithm is not applicable to general directed graphs.

In [8], considering a deleted tree edge (u, v), each vertex in the subtree of DFS-Tree rooted at v may have to be hung from a different parent in the new DFS-Tree. Thus, the algorithm finds a new position for each vertex in this subtree by the topological order. A critical property of this algorithm is that when a vertex is processed the positions of all its in-neighbors have already been fixed in the new DFS-Tree due to the fact that an in-neighbor of a vertex cannot be its descendant in the DFS-Tree in directed acyclic graphs. However, this

property is not applicable to general directed graphs, since a vertex may have a descendant in the DFS-Tree as a potential parent.

For undirected graphs, Baswana et al. [7] propose a fully dynamic algorithm for maintaining a DFS-Tree under a sequence of updates with $O(\sqrt{mn}\log^{2.5} n)$ time per update, an incremental algorithm for maintaining a DFS-Tree under a sequence of edge insertions with $O(n\log^3 n)$ time per edge insertion, and a fault-tolerant algorithm for computing a DFS-Tree of graph $G \setminus \mathcal{F}$ with $O(nk\log^4 n)$ time under any set $\mathcal{F}$ of k failed vertices or edges [7]. The time complexities of the above algorithms are further improved by Chen et al. [16, 17] to $O(\sqrt{mn}\log^{1.5} n)$ for the fully dynamic algorithm, $O(n)$ for the incremental algorithm, and $O(nk\log^2 n)$ for the fault-tolerant algorithm. Nakamura and Sadakane [50] optimize the space occupied by the data structure in the above algorithms from $O(m\log^2 n)$ to $O(m\log n)$. Moreover, Baswana et al. [10] improve the above algorithms to achieve $O(\sqrt{mn\log n})$ time for the fully dynamic algorithm and $O(n(k'+\log n)\log n)$ time for the fault-tolerant algorithm, where k' is the maximum number of failed vertices/edges along any root-leaf path of the initial DFS-Tree. Considering the parallel environment, Khan [42] proposed the parallel, semi-streaming and distributed algorithms for maintaining a DFS-Tree in the dynamic setting.

The basic idea of the above algorithms is to use a preprocessed data structure to generate a reduced adjacency list for each vertex, and then perform DFS traversal using these lists. The critical properties of these algorithms are that, in undirected graphs, there is no edge between any subtrees after partitioning the DFS-Tree into disjoint subtrees and paths, and the neighbor of a vertex appears either as its ancestor or its descendant in the DFS-Tree. However, these properties do not apply to directed graphs due to the fact that a directed graph may have cross-edges in its DFS-Tree, and two adjacent vertices do not always

have ancestor-descendant relationships.

Baswana et al. [9] extend the incremental algorithm for directed acyclic graphs presented in [29] to general directed graphs and propose an incremental algorithm to maintain a DFS-Tree in general directed graphs. They also use the rank of vertices given in [29]. For each edge insertion, they mark all the subtrees in a located rank range as unvisited and traverse each unvisited subtree, implicitly maintaining the rank of vertices. A crucial property of this incremental algorithm is the existence of broomstick structure in the incremental setting. In brief, from the root of the DFS-Tree, there exists a long downward path on which there is no branching after a large number of edge insertions [9]. However, in a fully dynamic environment, the broomstick structure does not exist due to the edge deletion. To the best of our knowledge, there does not exist any non-trival fully dynamic algorithm for maintaining a DFS-Tree in general directed graphs.

2.3 Other Cohesive Subgraph Models

Besides k-core and strongly connected component, there are many definitions of the cohesive subgraph. The common idea is that the number of edges inside cohesive subgraphs must be more than the edges linking vertices belong to different cohesive subgraphs [27]. In other words, vertices in a cohesive subgraph must be tightly connected. In this section, we introduce some other cohesive subgraph models.

K-Truss. The k-truss is a typical cohesive subgraph model. It is a maximal subgraph in which each edge is contained in at least $k-2$ triangles within this subgraph [20]. For instance, in a social network, a k-truss cohesive subgraph is a maximal group of users in which any two users have at least $(k-2)$ common friends within this group if the two users are friends. k-truss has a better co-

hesive structure than k-core, but it is harder to compute. Similarly, the truss decomposition problem is to find k-truss of a graph for all k. For this problem, the time complexity of an in-memory algorithm is $O(|E|^{1.5})$, where E is the set of edges [20].

Most truss decomposition algorithms assume that the memory can load the entire graphs. But many real-world graphs are too massive to reside in memory. Therefore, [68] proposed an external-memory truss decomposition algorithm. They first used the bin sort technique to optimize the in-memory algorithm, and then introduce the bottom-up I/O efficient algorithm and the top-down I/O efficient algorithm to handle the huge graphs that cannot reside in memory [68]. The external memory algorithm is an indispensable improvement of the in-memory algorithm.

Considering the k-truss search problem, [37] proposed a novel cohesive subgraph model based on k-truss and the TCP-Index which developed based on the k-truss index. The TCP-Index can search k-truss cohesive subgraphs with a linear cost, and its index size is $O(|E|)$. Also, it supports edge insertion and deletion efficiently. However, the TCP-Index may not efficient enough to support massive graphs. So [3] proposed a truss-equivalence based index which outperforms TCP-Index at least an order of magnitude.

The first definition of k-truss in uncertain graphs was given by [38]. They applied the concept of (k, η)-core was given by [13] into k-truss and proposed a novel notion of local (k, γ)-truss. Also, they pointed out that the concept of (k, η)-core is based on a local definition. In the local definition, we consider that each vertex in k-core and each edge in k-truss is independent. But sometimes, we have to consider the entire graph where vertices and edges are not independent. Therefore, they also proposed a novel notion of global (k, γ)-truss. Like the algorithm of (k, η)-core decomposition given by [13], they proposed an efficient

algorithm for local (k, γ)-truss decomposition based on dynamic programming. For the global (k, γ)-truss decomposition, they proposed a novel approximation algorithm by using Monte Carlo sampling.

Clique. The clique is the most cohesive subgraph model. It is a subgraph in which each vertex is connected to all other vertices. Discovering the maximal clique in a graph is a classical problem in graph theory [14]. The worst-case time complexity of in-memory algorithms for this issue has proved to be optimal [66]. However, a large number of real-world graphs are enormous and grow continuously. Sometimes, we cannot load the whole graph in memory due to the limited memory size. Therefore, [19] proposed a novel notion, H^*-graph, which defines the core of a graph and extends to encompass the neighborhood of the core for the maximal clique computations. This algorithm is the first external memory algorithm for the maximal clique problem. And it can use the H^*-graph to bound the memory usage [19]. This external memory algorithm is an indispensable method to analyze large graphs.

Nuclei. The nuclei is a novel concept of cohesive subgraph [57]. It shows the connections between the definition of k-core and k-truss. Roughly speaking, an (r, s)-nucleus is a maximal subgraph in which each r-clique participates in some s-clique, where r and s are small positive integers and $r < s$ [57]. In this way, the $(1, 2)$-nucleus is k-core, and the $(2, 3)$-nucleus is k-truss. Also, [57] gave a nucleus decomposition algorithm which can be applied in both core decomposition and truss decomposition. Then, [56] proposed an algorithm based on disjoint set to build a hierarchical structure of cohesive subgraphs. For the k-core, k-truss and other cohesive subgraphs, there are many similarities in definitions, decomposition algorithms, and hierarchy constructions. And the hierarchical structure is beneficial for the index construction in cohesive subgraph search.

Maximum Density Subgraph. The maximum density subgraph model is an

intuitive cohesive subgraph model. Intuitively, the density of a graph $G(V, E)$ is $|E|/|V|$, where the V is the set of vertices and the E is the set of edges. Based on the definition of density, the maximum density subgraph problem is to find a subgraph with the maximal density. For this problem, [33] proposed an algorithm based on the max-flow problem. This algorithm can find the maximum density subgraph in polynomial time by invoking $O(log(n))$ max-flow computations. But due to the high time complexity of max-flow computations, the applications of this model is limited.

K-Edge Connected Component. The k-edge connected component model is also a fundamental cohesive subgraph model. A graph is k-edge connected if it is still connected after removing any $k-1$ edges [77]. The k-edge connected component problem is to find the maximal subgraph that is k-edge connected. For this problem, [15] proposed a novel and efficient threshold-based graph decomposition algorithm. This algorithm iteratively decomposes a graph for computing its k-edge connected components. Intuitively, the k-edge connected component avoids some weaknesses of k-truss. It has a better cohesive structure, but it is more complicated to compute.

Chapter 3

INTERNAL MEMORY CORE COMPUTATION IN LARGE UNCERTAIN GRAPHS

3.1 Overview

In this chapter, we introduce our proposed index-based solution for computing all the (k, η)-cores in uncertain graphs under internal memory setting. The work is published in [71] and the rest of this chapter is organized as follows. Section 3.2 provides some preliminary concepts and formally defines the problem. In Section 3.3, we review an existing solution and explain the online approach in detail. Section 3.4 describes the basic structure of the index. Section 3.5 presents the optimized forest-based index structure. Section 3.6 practically evaluates the proposed algorithms in practical terms. Section 3.7 summarizes the chapter.

3.2 Preliminary

Before stating the problem, we first introduce the concept of k-cores in deterministic graphs, followed by the definition of (k, η)-core in uncertain graphs.

K-Core in Deterministic Graphs. The definitions for k-core and core number are presented as follows.

Definition 1. (k-Core) *Given a graph $G(V, E)$ and an integer k, the k-core is a maximal connected induced subgraph $G'[V']$ in which every vertex has a degree of at least k, i.e., $\forall u \in V', deg(u, G') \geq k$. [58]*

Definition 2. (Core Number) *Given a graph $G(V, E)$, the core number for a vertex u, denoted as $core(u)$, is the largest integer of k such that a k-core containing u exists.*

Given a deterministic graph G, the core numbers for all vertices can be computed by iteratively removing the vertex with the minimum degree. The pseudocode is given in Algorithm 1.

Algorithm 1: Core Decomposition

Input: A graph $G(V, E)$
Output: The core numbers for all vertices in G

```
1 while V ≠ ∅ do
2     k ← min_{u∈V} deg(u, G);
3     while ∃u ∈ V s.t. deg(u, G) ≤ k do
4         core(u) ← k;
5         foreach v ∈ N(u, G) do
6             remove edge (u, v) from G;
7             deg(v, G) ← deg(v, G) − 1;
8         V ← V \ {u};
9 return core(u) for all vertices u;
```

K-Core in Uncertain Graphs. In line with existing works, we assume that the probability of each edge actually existing is independent, and adopt the well-

known possible-world semantics for uncertain graph analysis. There exist $2^{|E|}$ possible graph instances under this assumption. The probability of observing a graph instance $G(V, E')$, denoted by $Pr(G)$, is:

$$Pr(G) = \prod_{e \in E'} p_e \prod_{e \in E \setminus E'} (1 - p_e). \tag{3.1}$$

The concept of (k, η)-cores, originally defined in [13], is based on possible-world semantics.

Definition 3. ((k, η)-CORE) *Given an uncertain graph $\mathcal{G}(V, E, p)$, an integer k and a probabilistic threshold $\eta \in [0, 1]$, the (k, η)-core of $\mathcal{G}$ is a maximal connected induced subgraph $\mathcal{G}'[V']$ such that the probability that each vertex $u \in V'$ has a degree of at least k in $\mathcal{G}'$ is not less than η, i.e., $\forall u \in V', Pr[deg(u, \mathcal{G}') \geq k] \geq \eta$.*

Note that we have slightly revised this definition by adding a connectivity constraint to the (k, η)-cores. An example of (k, η)-cores is given as follows.

Example 1. *Consider the uncertain graph $\mathcal{G}$ in Figure 1.1. Given an integer $k = 2$ and a probability threshold $\eta = 0.3$, we identify two $(2, 0.3)$-cores, as marked in the figure. One is $\mathcal{G}[\{v_2, v_3, v_4, v_5\}]$, and the other one is $\mathcal{G}[\{v_7, v_9, v_{10}\}]$. We denote $\mathcal{G}[\{v_2, v_3, v_4, v_5\}]$ as $\mathcal{G}_1$ for simplicity. Consider the vertex v_2 in $\mathcal{G}_1$. There are three edges connected to v_2 in $\mathcal{G}_1$, and we have $Pr[deg(v_2, \mathcal{G}_1) \geq 2] = 0.568$. Similarly, we have $Pr[deg(v_3, \mathcal{G}_1) \geq 2] = 0.8$, $Pr[deg(v_4, \mathcal{G}_1) \geq 2] = 0.3$ and $Pr[deg(v_5, \mathcal{G}_1) \geq 2] = 0.656$. $\mathcal{G}_1$ is maximal. Assume that we add v_1 in to $\mathcal{G}_1$. We have $Pr[deg(v_1, \mathcal{G}_1) \geq 2] = 0.1 < 0.3$. Therefore, v_1 cannot be in the $(2, 0.3)$-core.*

Based on Definition 3, the problem is defined as follows.

Problem Definition. Given an uncertain graph $\mathcal{G}(V, E, p)$, an integer k and a probabilistic threshold $\eta \in [0, 1]$, we examine the problem of efficiently computing all the (k, η)-cores of $\mathcal{G}$.

Specifically, let C be the vertex set such that the induced subgraph $\mathcal{G}[C]$ of C is a (k, η)-core. The aim is to compute a set $\mathcal{R}$ containing all such vertex sets C without any duplication. In the case of $k = 2, \eta = 0.3$ in Figure 1.1, we return $\{\{v_2, v_3, v_4, v_5\}, \{v_7, v_9, v_{10}\}\}$.

3.3 Online (k, η)-Cores Computation

In this section, we first review an existing solution [13] for the problem of η-core decomposition, as several key concepts and ideas intuitively fit our problem. Then, we provide our online solution for computing (k, η)-cores.

3.3.1 An Existing Solution for η-Core Decomposition

Given an uncertain graph $\mathcal{G}$, let $\mathcal{G}_u^{\geq k}$ be the set of all possible graph instances where u has a degree of at least k, i.e., $\mathcal{G}_u^{\geq k} = \{G \sqsubseteq \mathcal{G} | deg(u, G) \geq k\}$. We have the following equation [13]:

$$Pr[deg(u, \mathcal{G}) \geq k] = \sum_{G \in \mathcal{G}_u^{\geq k}} Pr(G). \tag{3.2}$$

Based on Equation 3.2, the definition for the η-degree for each vertex follows.

Definition 4. (η-Degree) *Given an uncertain graph $\mathcal{G}(V, E, p)$ and a probabilistic threshold $\eta \in [0, 1]$, the η-degree of a vertex $u \in V$, denoted by η-$deg(u, \mathcal{G})$, is the largest integer of k that satisfies $Pr[deg(u, \mathcal{G}) \geq k] \geq \eta$. [13]*

Given a vertex u, $Pr[deg(u) \geq k]$ monotonously decreases when k increases. Next, we define the η-core number.

Definition 5. (η-Core Number) *Given an uncertain graph $\mathcal{G}(V, E, p)$ and a probabilistic threshold $\eta \in [0, 1]$, the η-core number for a vertex u, denoted as η-$core(u)$, is the largest integer of k such that a (k, η)-core containing u exists.*

Based on Definition 3 and Definition 5, we have the following lemma.

Lemma 1. *Given an uncertain graph $\mathcal{G}$ and a probability threshold $\eta \in [0, 1]$, a vertex u is in a (k, η)-core if and only if η-core$(u) \geq k$.*

The problem of computing the η-core numbers for all vertices in the uncertain graph $\mathcal{G}$ is called η-core decomposition. The solution proposed in [13] is provided in Algorithm 2.

Algorithm 2: η-CORE DECOMPOSITION

Input: An uncertain graph $\mathcal{G}(V, E, p)$ and a probability threshold η
Output: η-core numbers for all vertices in $\mathcal{G}$

```
1  compute η-deg(u, G) for all u ∈ V;
2  while G is not empty do
3      k ← min_{u∈V} η-deg(u, G);
4      while ∃u ∈ V s.t. η-deg(u, G) ≤ k do
5          η-core(u) ← k;
6          foreach v ∈ N(u, G) do
7              remove edge (u, v) from G;
8              update η-deg(v, G);
9          V ← V \ {u};
10 return η-core(u) for all vertices u;
```

Algorithm 2 shares the similar idea with Algorithm 1, and the pseudocode is self-explanatory. The key steps in the algorithm are computing (line 1) and updating (line 8) the η-degrees of the vertices. We introduce their details below.

η-Degree Computation. To compute the η-degree, we first present the following equation:

$$Pr[deg(u) \geq k] = \sum_{i=k}^{Deg(u)} Pr[deg(u) = i] = 1 - \sum_{i=0}^{k-1} Pr[deg(u) = i]. \tag{3.3}$$

Based on Equation 3.3, we can start with $Pr[deg(u) \geq 0] = 1$. Iteratively, we increase i by one and compute $Pr[deg(u) = i]$ for a vertex u. We calculate

$Pr[deg(u) \geq i+1]$ as $Pr[deg(u) \geq i] - Pr[deg(u) = i]$. We repeat this step and terminate once $Pr[deg(u) \geq i+1] < \eta$. Then we have $\eta\text{-}deg(u) = i$.

To compute $Pr[deg(u) = i]$ for a vertex u, we use the dynamic-programming method given in [13]. Assume that $E(u) = \{e_1, e_2, ..., e_{Deg(u)}\}$ is the set of all the edges connected to u in some order. The intuitive idea of dynamic programming is that, if a vertex u has a degree of i, one of the following two cases applies: either (i) $i-1$ edges exist in $\{e_1, e_2, ..., e_{Deg(u)-1}\}$ and $e_{Deg(u)}$ exists; or (ii) i edges exist in $\{e_1, e_2, ..., e_{Deg(u)-1}\}$ and $e_{Deg(u)}$ does not exist.

Given a subset $E'(u) \subseteq E(u)$, let $deg(u|E'(u))$ be the degree of u in the subgraph $\mathcal{G}'(V, E\backslash(E(u)\backslash E'(u)), p)$, and $X(h, j) = Pr[deg(u|\{e_1, e_2, ..., e_h\}) = j]$. We have the following dynamic-programming recursive function [13] for all $h \in [1, Deg(u)], j \in [0, h]$:

$$X(h, j) = p_{e_h} X(h-1, j-1) + (1 - p_{e_h}) X(h-1, j). \tag{3.4}$$

Several initialization cases are also given as follows:

$$\begin{cases} X(0,0) = 1, \\ X(h,-1) = 0, & \text{for all } h \in [0, Deg(u)], \\ X(h,j) = 0, & \text{for all } h \in [0, Deg(u)], j \in [h+1, i]. \end{cases} \tag{3.5}$$

The time complexity to compute the η-degree of a vertex is given as follows.

Lemma 2. *The time complexity to compute the η-degree of a vertex u is $O(\eta\text{-}deg(u) \cdot Deg(u))$. [13]*

η-Degree Update. Given the incident edge set $E(u)$ of a vertex u, assume that an edge e is removed from $E(u)$. To compute the updated probability $Pr[deg(u|E(u)\backslash\{e\}) = i]$, we introduce the following equation [13]:

$$\begin{aligned} &Pr[deg(u|E(u)\backslash\{e\}) = i)] = \\ &\quad \frac{Pr[deg(u) = i] - p_e Pr[deg(u|E(u)\backslash\{e\}) = i-1]}{1 - p_e}. \end{aligned} \tag{3.6}$$

Based on Equation 3.6, we compute $Pr[deg(u|E(u)\backslash\{e\}) = i]$ for each $i \in [1, \eta\text{-}deg(u)]$ in constant time, given that $Pr[deg(u|E(u)\backslash\{e\}) = 0] = \frac{1}{1-p_e} Pr[deg(u) = 0]$. The time complexity to update the η-degree is given as follows.

Lemma 3. *Given an uncertain graph $\mathcal{G}$ and a removed incident edge e to a vertex u, the time complexity to update the η-degree of u is $O(\eta\text{-}deg(u))$. [13]*

We also provide the time complexity of Algorithm 2 below.

Lemma 4. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the time complexity of Algorithm 2 is $O(\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u))$. [13]*

3.3.2 Our Approach to Compute (k, η)-Cores

Based on several concepts introduced in the previous section, we turn to the online approach for computing all the (k, η)-cores. Our approach is similar to Algorithm 2, which iteratively removes the vertex that does not belong to the result set. Before presenting the algorithm, we make the following observation about optimization.

Observation 1. *Given an uncertain graph $\mathcal{G}$ and a (k, η)-core $\mathcal{G}[C]$ for any parameter settings for k and η, there exists a k-core $G[C']$ containing $\mathcal{G}[C]$, i.e., $C \subseteq C'$.*

Based on Observation 1, we can first recursively remove the vertices with degrees of less than k, since these vertices cannot be in the result set for any (k, η)-core. We provide the pseudocode for our approach in Algorithm 3.

Lines 1–5 compute the k-cores. Lines 6–11 recursively remove the vertices with η-degrees of less than k and generate a subgraph containing all result vertices. Lines 12–14 determine the connected components in the result. The time

Algorithm 3: (k, η)-CORES COMPUTATION

Input: An uncertain graph $\mathcal{G}(V, E, p)$, an integer k and a probability threshold η

Output: All (k, η)-*cores* in $\mathcal{G}$

```
1  while ∃u ∈ V s.t. Deg(u) < k do
2      foreach v ∈ 𝒩(u, 𝒢) do
3          remove the edge (u, v) from 𝒢;
4          Deg(v) ← Deg(v) − 1;
5      V ← V \ {u};
6  compute η-deg(u) for all u ∈ V;
7  while ∃u ∈ V s.t. η-deg(u) < k do
8      foreach v ∈ 𝒩(u, 𝒢) do
9          remove the edge (u, v) from 𝒢;
10         update η-deg(v);
11     V ← V \ {u};
12 ℛ ← ∅;
13 foreach connected component 𝒢[C] ∈ 𝒢 do
14     ℛ ← ℛ ∪ {C};
15 return ℛ;
```

complexity of Algorithm 3 is $O(\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u))$, which is the same as that of Algorithm 2.

Theorem 1. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the time complexity of Algorithm 3 is $O(\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u))$.*

Proof. The time complexity of line 1–5 is $O(|E|)$. The major cost in Algorithm 3 appears in line 6, which is $\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u)$ according to Lemma 2. The time complexity of lines 7–11 is also $\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u)$ in the worst case, and the time complexity of lines 12–14 is $O(|E|)$. Therefore, the total complexity is $O(\sum_{u \in V} \eta\text{-}deg(u) \cdot Deg(u))$. □

Drawbacks of the Online Approach. Even though Algorithm 3 successfully computes all the (k, η)-cores, several drawbacks still exist. First, changing the input parameters may heavily influence the resulting (k, η)-cores, especially in large graphs. We consider the case in Figure 1.1. If we change the input parameter η from 0.3 to 0.4 and keep $k = 2$, vertex v_4 will be removed and the result will change to $\{\{v_2, v_3, v_5\}, \{v_7, v_9, v_{10}\}\}$. Additionally, we find that the major cost in Algorithm 3 is computing and updating the η-degrees of the vertices. This is extremely time-consuming and means the algorithm cannot be scaled to big graphs.

Motivated by the above challenges, we propose an index-based approach. Based on the proposed index, we can answer a query for any given k and η with a time complexity that is only proportional to the size of the results. To make our solution scalable to big graphs, the index size is well-bounded, with an acceptable time cost for constructing the index.

We propose a basic index approach in Section 3.4, and in Section 3.5, we optimize both the index structure and the query processing procedure.

3.4 An Index-based Approach

3.4.1 The Index Structure

In this section, we introduce an index structure, called the *uncertain core η-orders index* (*UCO-Index*). The general idea of this index is to maintain the result vertices for every possible k and η. In other words, given an integer k and a probability threshold η, we aim to efficiently compute all the result vertices based on the index structure. To complete this task, we start by computing all result vertices from any given probability threshold η under a specific fixed integer k, as there is only a limited number of possible k. Based on Observation 1, we provide the range of integer k as follows.

Observation 2. *Given an uncertain graph $\mathcal{G}$, we only need to consider the parameter $1 \leq k \leq k_{max}$, where $k_{max} = \max_{u \in V} core(u)$.*

If $k > k_{max}$, the probability that a (k, η)-core exists is 0. We also provide the largest possible integer for k of each vertex in the following observation.

Observation 3. *Given an uncertain graph $\mathcal{G}$ and an integer k, a vertex u cannot be in the (k, η)-core if $core(u) < k$.*

Based on Observation 3, we derive a candidate set of result vertices by only considering the parameter k. That is $\{u \in V | core(u) \geq k\}$.

Now, given the candidate set for each integer k, we consider computing the exact result set by the probability threshold η. Recall that a vertex u is in the (k, η)-core if the η-degree of u is at least k. We have the following lemma.

Lemma 5. *Given an uncertain graph $\mathcal{G}$, a parameter k and two probability threshold $0 \leq \eta \leq \eta' \leq 1$, a vertex u is in (k, η)-core if it is in (k, η')-core.*

According to the monotonicity in Lemma 5, we only need to save the largest probability value η for each vertex u that will be in the (k, η)-core. We call such value the *η-threshold*, which is formally defined as follows.

Definition 6. (η-THRESHOLD) *Given an uncertain graph $\mathcal{G}(V, E, p)$ and an integer k, the η-threshold of a vertex u, denoted by η-$threshold_k(u)$, is the largest η such that a (k, η)-core containing u exists.*

Based on Observation 3 and Definition 6, we have η-$threshold_k(u) = 0$ for any vertex u if $core(u) < k$, and we give a necessary and sufficient condition that a vertex will be in the (k, η)-core as follows.

Lemma 6. *Given an uncertain graph $\mathcal{G}$, an integer k and a probability threshold η, a vertex u is in the (k, η)-core if and only if η-$threshold_k(u) \geq \eta$.*

To efficiently compute all result vertices, we save all η-thresholds of each vertex u in an order, which is formally defined as follows.

Definition 7. (η-ORDER) *Given an uncertain graph $\mathcal{G}$ and a vertex u, the η-order of u, denoted by η-$order(u)$, is a probability order such that (i) the i-th value in η-$order(u)$ is η-$threshold_i(u)$, and (ii) the length of η-$order(u)$ is $core(u)$.*

Example 2. *The η-orders for all vertices in the uncertain graph $\mathcal{G}$ in Figure 1.1 are given in Figure 3.1. We consider the example of vertex v_4. Given $k = 2$, we have η-$threshold_2(v_4) = 0.3$. That means v_4 is in a $(2, 0.3)$-core, but not in any $(2, \eta)$-core if $\eta > 0.3$.*

Given the η-order of a vertex u and an integer k, we use a constant time complexity to compute the η-$threshold_k(u)$. We save the η-orders for all vertices as our *UCO-Index*. The size of the *UCO-Index* is well-bounded.

k	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	0.6	0.92	0.92	0.76	0.92	0.6	0.9	0.8	0.9	0.9
2	0.1	0.48	0.48	0.3	0.48	0.1	0.4		0.4	0.4
3		0.04	0.04	0.04	0.04					

Figure 3.1: The *UCO-Index* of $\mathcal{G}$

Theorem 2. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the space complexity of the UCO-Index is $O(\sum_{u \in V} core(u))$.*

Since $core(u) \leq Deg(u)$ for each vertex u, the size of the *UCO-Index* is also roughly bounded by $O(|E|)$.

3.4.2 Query Processing

Before presenting the query processing algorithm, we first give an alternative definition for the (k, η)-core based on Definition 6.

Lemma 7. *Given a set of vertices C in an uncertain graph $\mathcal{G}$, the induced subgraph $\mathcal{G}[C]$ is a (k, η)-core if and only if (i) $\forall u \in C, \eta$-threshold$_k(u) \geq \eta$; (ii) $\mathcal{G}[C]$ is connected; and (iii) C is maximal.*

Based on the above lemma, we present the pseudocode for the query processing in Algorithm 4. It first identifies all vertices whose η-threshold is not less than η in line 1. Given the input integer k, the η-threshold of a vertex u can be computed by checking the k-th item in the η-order of u according to Definition 7. The algorithm then computes each (k, η)-core in lines 3–4. The correctness of Algorithm 4 can be guaranteed according to Lemma 7. The running time of Algorithm 4 is analyzed as follows.

Algorithm 4: UCO-BASED QUERY

Input: An uncertain graph $\mathcal{G}(V, E, p)$, an integer k, a probability threshold η and $UCO\text{-}Index$

Output: All (k, η)-*cores* in $\mathcal{G}$

```
1 V' ← {u ∈ V | η-threshold_k(u) ≥ η};
2 R ← ∅;
3 foreach connected component G[C] ∈ G[V'] do
4     R ← R ∪ {C};
5 return R;
```

Example 3. *We use the uncertain graph $\mathcal{G}$ in Figure 1.1 as an example for Algorithm 4. Given $k = 2$ and $\eta = 0.3$, we first compute all result vertices according to their η-orders in Figure 3.1, which are $v_2, v_3, v_4, v_5, v_7, v_9, v_{10}$. Then we check the connectivity of the vertices and derive the result $\{\{v_2, v_3, v_4, v_5\}, \{v_7, v_9, v_{10}\}\}$.*

Theorem 3. *Given an uncertain graph $\mathcal{G}(V, E, p)$, an integer k and a probability threshold η, the time complexity of Algorithm 4 is $O(|V| + \sum_{u \in C} Deg(u))$, where C is the set of all result vertices, i.e., $C = \{u \in V | \eta\text{-}threshold_k(u) \geq \eta\}$.*

3.4.3 Index Construction

Before introducing the algorithm used to construct the $UCO\text{-}Index$, we first give the following definition for the ease of presentation.

Definition 8. (k-PROBABILITY) *Given an uncertain graph $\mathcal{G}$ and an integer k, the k-probability of a vertex u, denoted by $k\text{-}prob(u, \mathcal{G})$, is the probability that $Pr[deg(u, \mathcal{G}) \geq k]$.*

Based on Definition 8, the general idea to construct the $UCO\text{-}Index$ is iteratively removing the vertex with the minimum k-probability. The detailed pseudocode is given in Algorithm 5.

In Algorithm 5, an empty order is initialized for each vertex in line 2. The η-thresholds for all vertices under a specific k are computed from line 4 to line 18.

Algorithm 5: UCO-Index Construction

Input: An uncertain graph $\mathcal{G}(V, E, p)$
Output: *UCO-Index* of $\mathcal{G}$

```
1  k ← 0;
2  foreach u ∈ V do η-order(u) ← ∅;
3  repeat
4      k ← k + 1;
5      isEnd ← true;
6      G' ← G;
7      curThres ← 0;
8      compute k-prob(u) for each u ∈ V';
9      while G' is not empty do
10         u ← arg min_{v∈V'} k-prob(v, G');
11         curThres ← max(k-prob(u, G'), curThres);
12         if curThres > 0 then
13             isEnd ← false;
14             η-order(u).push(curThres);
15         V' ← V' \ {u};
16         foreach v ∈ N(u, G') do
17             remove the edge (u, v) from G';
18             update k-prob(v, G');
19 until isEnd = true;
20 return η-order(u) for all vertices u;
```

Specifically, a vertex u with the minimum k-probability is selected in line 10. In line 11, the variable $curThres$ is used to save the η-threshold for the selected vertex u. We know that the $(k, curThres)$-core exists if $curThres > 0$ in line 12. The variable $isEnd$ is used to identify whether the iterations should be terminated, and we assign $isEnd$ with $false$ if $curThre > 0$ in line 13. We push the η-threshold of u into the η-order of u in line 14. We remove the vertex u and update the k-probability for each neighbor v of u in lines 15–18. The η-orders are derived when the algorithm is terminated. We prove this as follows.

Theorem 4. *Given an uncertain graph $\mathcal{G}$, Algorithm 5 correctly computes the η-orders for all vertices in $\mathcal{G}$.*

Proof. We start by proving that the value of $curThres$ in line 14 is equal to the η-$threshold_k(u)$ of the vertex u. This is obvious since we remove vertices in a non-decreasing order of k-probability, and the variable $curThres$ always maintains the largest probability that a $(k, curThres)$-core exists for u.

Given that $curThres = \eta$-$threshold_k(u)$, we correctly compute the η-$threshold_k(u)$ for all vertices u in $\mathcal{G}$ in each iteration (lines 4–18). The iteration is terminated if the k-probability for every vertex is 0 (line 19), which means the corresponding (k, η)-core does not exist. Therefore, we obtain the complete η-order for each vertex, and the proof is finished. □

Theorem 5. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the time complexity of Algorithm 5 is $O(k_{max}|V| \log |V| + k_{max}^2|E|)$, where $k_{max} = \max_{u \in V} core(u)$.*

Proof. Based on Lemma 2 and Lemma 3, given a vertex u, computing (line 8) and updating (line 18) the k-probability cost $O(k \cdot Deg(u))$ time and $O(k)$ time, respectively.

In each iteration of lines 3–19, the time complexity of line 8 is $O(\sum_{u \in V} k \cdot Deg(u))$. We use a minimum priority queue (implemented by a Fibonacci heap)

to maintain all vertices where the keys are their k-probabilities. The Fibonacci heap takes a constant amount of time for insertion, and linear amount of time to build the priority queue, plus a constant amount of time for updating if the involved key is decreased, and a logarithmic amount of time for delete-min. Thus, in each iteration of lines 3–19, line 10 and line 15 totally take $O(|V| \log |V|)$ time to remove the vertex with the minimum k-probability, and lines 16–18 totally take $O(\sum_{u \in V} k \cdot Deg(u))$ time for each k, since the k-probability of each vertex u can be updated at most $Deg(u)$ times.

Therefore each iteration in lines 3–19 takes $O(|V| \log |V| + \sum_{u \in V} k \cdot Deg(u))$ time, which can be arranged as $O(|V| \log |V| + k|E|)$. The number of iterations is bounded by $O(k_{max})$, where $k_{max} = \max_{u \in V} core(u)$. The time complexity of Algorithm 5 is $O(k_{max}|V| \log |V| + k_{max}^2|E|)$. □

3.5 Making Query Processing Optimal

We proposed a $UCO\text{-}Index$ based approach in the previous section. Even though computing the η-degree is avoided and the used space can be well-bounded, the $UCO\text{-}Index$ still needs to detect all vertices to compute the results in the query processing, and this may be hard to tolerate in big graphs. To address this issue, we propose a forest-based index structure, namely *uncertain core η-forest index* ($UCF\text{-}Index$). Based on the $UCF\text{-}Index$, we compute the result set in optimal time.

The index structure is introduced in Subsection 3.5.1. We provide the query processing algorithm in Subsection 3.5.2. We propose the algorithm to construct the $UCF\text{-}Index$ with several optimization techniques in Subsection 3.5.3.

3.5.1 Forest-based Index Structure

According to Lemma 7, the key to computing all result (k, η)-cores is computing all vertices of u such that $\eta\text{-}threshold_k(u) \geq \eta$. This costs $O(|V|)$ time in Algorithm 4. A straightforward idea to improve the query's efficiency is to sort the vertices in a non-increasing order of their η-threshold for each integer k. Based on this structure, we can compute all result vertices in optimal time, and the total size of this structure can still be bounded by $O(\sum_{u \in V} core(u))$. However, given that there is no topological information between vertices, we still use $O(\sum_{u \in C} Deg(u))$ time to identify the connected components, where $C = \{u \in V | \eta\text{-}threshold_k(u) \geq \eta\}$.

Motivated by this, we propose a novel index, called the $UCF\text{-}Index$, which organizes the vertices and their η-thresholds into a tree structure, for each possible integer k. The tree is built based on Lemma 5, where vertices with smaller η-thresholds are on the upper side, and larger η-thresholds are on the lower side. We name the tree structure η-tree, which is denoted by $\eta\text{-}tree_k$. Specifically, let C_k be the set of vertices whose core numbers are not less than k, i.e., $C_k = \{u \in V | core(u) \geq k\}$. We divide all vertices in C_k into different tree nodes in $\eta\text{-}tree_k$. Considering a tree node $\mathbb{X}$ in the $\eta\text{-}tree_k$, the attributes of $\mathbb{X}$ are summarized as follows:

- $\mathbb{X}.vertices$: return a set of vertices.
- $\mathbb{X}.\eta\text{-}threshold$: return $\eta\text{-}threshold_k(u)$ for any vertex $u \in \mathbb{X}.vertices$.
- $\mathbb{X}.parent$: return the parent node of $\mathbb{X}$.
- $\mathbb{X}.children$: return the children nodes of $\mathbb{X}$.

The details to implement these attributes are presented below. Formally, the vertex set for each tree node is computed using the following rule.

Lemma 8. *Given an uncertain graph $\mathcal{G}$ and an integer k, we group a vertex set S into a tree node $\mathbb{X}$, i.e., $\mathbb{X}.vertices = S$ if (i) $\forall u, v \in S, \eta\text{-}threshold_k(u) = \eta\text{-}threshold_k(v)$; (ii) let $\eta = \eta\text{-}threshold_k(u)$ for any $u \in S$, there is a (k, η)-core $\mathcal{G}[C]$, such that $S \subseteq C$; and (iii) S is maximal.*

Then we give the rules for the parent-children relationship of tree nodes. Let $\mathcal{G}[V_{\mathbb{X}}]$ be the $(k, \mathbb{X}.\eta\text{-}threshold)$-core containing $\mathbb{X}.vertices$, and $N(\mathcal{G}_{\mathbb{X}})$ be the set of tree nodes in which each tree node $\mathbb{Y}$ satisfies $\exists u \in V_{\mathbb{X}}, v \in \mathbb{Y}.vertices : (u, v) \in E \wedge v \notin V_{\mathbb{X}}$. Note that there does not exist a tree node $\mathbb{Y} \in N(\mathcal{G}_{\mathbb{X}})$ such that $\mathbb{Y}.\eta\text{-}threshold \geq \mathbb{X}.\eta\text{-}threshold$. Otherwise, the vertices in $\mathbb{Y}$ also belong to $V_{\mathbb{X}}$. The parent for each tree node is defined as follows.

Lemma 9. *Given an uncertain graph $\mathcal{G}$ and an integer k, a tree node $\mathbb{Y}$ is the parent of the tree node $\mathbb{X}$ in $\eta\text{-}tree_k$, if $\mathbb{Y}$ is the tree node in $N(\mathcal{G}_{\mathbb{X}})$ with the largest η-threshold, i.e., $\mathbb{Y} = \arg\max_{\mathbb{Y} \in N(\mathcal{G}_{\mathbb{X}})} \mathbb{Y}.\eta\text{-}threshold$.*

In the case of $N(\mathcal{G}_{\mathbb{X}}) = \emptyset$, the tree node $\mathbb{X}$ is the root node, and there may exist more than one trees for each integer k. We give an example of the tree node and the η-tree as follows.

Example 4. *Still considering the uncertain graph $\mathcal{G}$ in Figure 1.1, we give the η-tree of $\mathcal{G}$ for $k = 2$ in Figure 3.2. The η-threshold for each tree node is listed on the right side. For the tree node $\{v_2, v_3, v_5\}$, the corresponding $(2, 0.48)$-core is the induced subgraph of the same vertex set. There are two neighbor tree nodes — $\{v_1, v_6\}$ and $\{v_4\}$. The η-threshold of $\{v_4\}$ is larger, and we set $\{v_4\}$ as the parent of $\{v_2, v_3, v_5\}$.*

Theorem 6. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the space complexity of the UCF-Index is $O(\sum_{u \in V} core(u))$.*

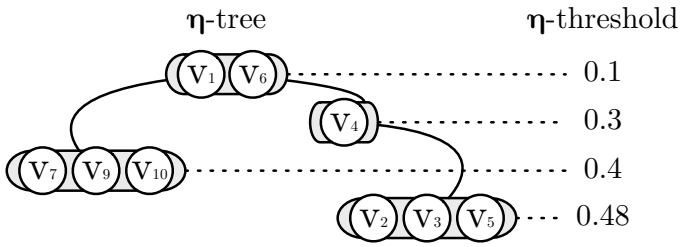


Figure 3.2: The η-tree of $\mathcal{G}$ for $k = 2$

Proof. The number of tree nodes is not larger than the number of vertices, and for each vertex u, it appears in $core(u)$ η-trees. Therefore, similar to Theorem 2, the total space complexity is also $O(\sum_{u \in V} core(u))$ or $O(|E|)$. □

3.5.2 Optimal Query Processing

We give an alternative definition for (k, η)-core based on the proposed $UCF\text{-}Index$.

Lemma 10. *Given an uncertain graph $\mathcal{G}$, an integer k, and a probability threshold η, let $\mathbb{R}$ be a tree node in η-tree$_k$ such that (i) $\mathbb{R}.\eta$-threshold $\geq \eta$; and (ii) there does not exist a parent $\mathbb{R}'$ of $\mathbb{R}$ such that $\mathbb{R}'.\eta$-threshold $\geq \eta$. Let C be the set of all vertices in the subtree rooted by $\mathbb{R}$. The induced subgraph $\mathcal{G}[C]$ is a (k, η)-core.*

According to Lemma 10, the key to the query processing is collecting all tree nodes in the subtree rooted by the tree node $\mathbb{R}$ mentioned in the lemma. Following this general idea, we provide the pseudocode for query processing in Algorithm 6.

We first collect all result tree nodes in lines 1–7. We can use constant time to derive the tree node with the largest η-threshold in line 4, if the tree nodes are

Algorithm 6: UCF-based Query

Input: An uncertain graph $\mathcal{G}(V, E, p)$, an integer k, a probability threshold η and $UCF\text{-}Index$ index

Output: All (k, η)-*cores* in $\mathcal{G}$

$\mathcal{T} \leftarrow$ the set of all tree nodes in η-$tree_k$;
$\mathcal{S} \leftarrow$ initialize an empty stack;
while $\mathcal{T}$ *is not empty* **do**
  $\mathbb{X} \leftarrow \arg\max_{\mathbb{X} \in \mathcal{T}} \mathbb{X}.\eta\text{-}threshold$;
  if $\mathbb{X}.\eta\text{-}threshold \geq \eta$ **then** $\mathcal{S}.push(\mathbb{X})$;
  else break;
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \{\mathbb{X}\}$;
$\mathcal{R} \leftarrow \emptyset$;
while $\mathcal{S}$ *is not empty* **do**
  $\mathbb{X} \leftarrow \mathcal{S}.pop()$;
  if $\mathbb{X}$ *is visited* **then continue**;
  $C \leftarrow \emptyset$;
  $\mathcal{Q} \leftarrow$ initialize an empty queue;
  $\mathcal{Q}.insert(\mathbb{X})$;
  while $\mathcal{Q}$ *is not empty* **do**
    $\mathbb{Y} \leftarrow \mathcal{Q}.pop()$;
    mark $\mathbb{Y}$ as visited;
    $C \leftarrow C \cup \mathbb{Y}.vertices$;
    foreach $\mathbb{Z} \in \mathbb{Y}.children$ **do** $\mathcal{Q}.insert(\mathbb{Z})$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{C\}$;
return $\mathcal{R}$;

sorted in a non-increasing order of their η-thresholds. The order can be precomputed in the index construction phase, which will be detailed introduced in the next subsection. Note that the number of tree nodes does not change in the order, and the total space complexity of the $UCF\text{-}Index$ is still $O(\sum_{u \in V} core(u))$.

We iteratively process each tree node in the stack in lines 9–20. Once we find an unvisited tree node in line 11, we find a root node satisfying the conditions in Lemma 10. We use a queue to compute all tree nodes rooted by $\mathbb{X}$, and collect all vertices in the tree nodes into C in lines 12–19. We add C into the result set in line 20.

Example 5. *Given an example for computing the $(k = 2, \eta = 0.3)$-core in $\mathcal{G}$ of Figure 1.1 based on the $UCF\text{-}Index$. The η-tree for $k = 2$ is given in Figure 3.2. We first locate the tree nodes $\mathbb{R}$ in Lemma 10, which are $\{v_4\}$ and $\{v_7, v_9, v_{10}\}$. Then we get two result cores, $\{v_4, v_2, v_3, v_5\}$ and $\{v_7, v_9, v_{10}\}$.*

Lemma 10 guarantees the correctness of Algorithm 6. The time complexity of Algorithm 6 is summarized as follows.

Theorem 7. *Given an uncertain graph $\mathcal{G}(V, E, p)$, an integer k and a probability threshold η, the time complexity of Algorithm 6 is $O(|C|)$, where C is the set of all result vertices, i.e., $C = \{u \in V | \eta\text{-}threshold_k(u) \geq \eta\}$.*

Proof. Given the sorted tree nodes for each k, the time cost of lines 1–7 is $O(|\mathcal{S}|)$. Line 10 is performed $O(|\mathcal{S}|)$ times. The total cost of scanning the subtree and collecting vertices in line 12–19 is $O(|\mathcal{S}| + |C|)$. Since $|\mathcal{S}| \leq |C|$, the total time complexity of Algorithm 6 is $O(|C|)$. □

Based on the above theorem, we claim that the time complexity of our query processing algorithm is optimal, since it is bounded by the result size.

3.5.3 Optimizations for the Index Construction Algorithm

The algorithm to construct the $UCF\text{-}Index$ is given in this section. We first introduce the general idea. For each integer, we compute the η-threshold for each vertex using the idea in Algorithm 5. Given the η-thresholds, we can build the η-tree using the disjoint-set data structure [30]. A similar idea can be found in [26, 56]. There are two main operations for the disjoint-set: $\mathsf{Union}(\mathbb{X}, \mathbb{Y})$ merges the dynamic sets $\mathbb{X}$ and $\mathbb{Y}$. $\mathsf{Find}(\mathbb{X})$ returns the set containing $\mathbb{X}$. With the two optimization techniques, *union by rank* and *path compression*, the amortized time per operation of the disjoint-set is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function and is normally less than 5 [30].

The dominating cost in constructing the $UCF\text{-}Index$ is still computing the η-thresholds for the vertices. Specifically, we review the Algorithm 5. First, computing and updating the k-probability of the vertices in each iteration is time-consuming as the degree of each vertex may be very large. Second, the number of iterations reaches k_{max}, and the k-probability of the vertices are computed from scratch in each iteration.

To speed up computing the η-threshold, we propose two optimizations in response to the above two challenges.

Core-based Reduction. Given any vertex u in each iteration of lines 3–19 of Algorithm 5, the k-probability of u monotonically decreases due to the removal of its neighbors v with $\eta\text{-}threshold_k(v) \leq \eta\text{-}threshold_k(u)$, and $\eta\text{-}threshold_k(u) = k\text{-}prob(u, \mathcal{G}[\{v \in V | \eta\text{-}threshold_k(v) \geq \eta\text{-}threshold_k(u)\}])$. Without breaking the correctness, we do not need to consider the edge (u, v) when computing $k\text{-}prob(u)$, if $\eta\text{-}threshold_k(v) < \eta\text{-}threshold_k(u)$. According to Observation 1, the η-threshold of a vertex u for an integer k is 0 if the core number for u is less than k. Therefore, for each integer k, we can compute and update the k-probability of each vertex in the subgraph $\mathcal{G}[\{u \in V | core(u) \geq k\}]$.

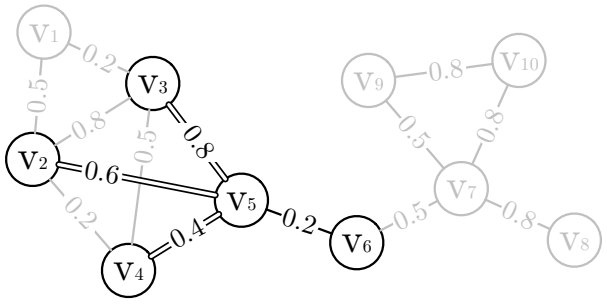


Figure 3.3: The optimization of core-based ordering

Core-based Ordering. To conquer the second challenge, we propose a top-down strategy. For each vertex u, we arrange the neighbors of u in a non-increasing order of their core numbers. Specifically, we denote the set of neighbors v of u such that $core(v) \geq k$ by $N_k(u)$. Without losing generality, let $\{e_1, e_2, ..., e_i\}$ be the set of edges between u and $N_k(u)$, and $\{e_{i+1}, e_{i+2}, ..., e_j\}$ be the set of edges between u and $N_{k-1}(u) \setminus N_k(u)$.

According to the aforementioned core-based reduction, the k-probability of u, is initially computed based on the edge set $\{e_1, e_2, ..., e_i\}$, i.e., $k\text{-}prob(u) = Pr[deg(u|\{e_1, e_2, ..., e_i\}) \geq k]$. Let $X(l, r, k) = Pr[deg(u|\{e_l, e_{l+1}, ..., e_r\}) = k]$. Given the integer $k-1$, we aim to compute $(k-1)\text{-}prob(u)$ based on $\{e_1, e_2, ..., e_j\}$, which is equivalent to computing $X(1, j, h)$ for each $h \in [0, k-1]$. We give the following equation for computing $X(1, j, h)$:

$$X(1, j, h) = \sum_{l=0}^{h} X(1, i, l) \cdot X(i+1, j, h-l). \tag{3.7}$$

Note that the $X(1, i, l)$ of u for each $l \in [0, k]$ has been computed when computing the k-probability of u in $\{e_1, e_2, ..., e_i\}$. Given the integer $k-1$, we reduce the computation from $X(1, j, h)$ to $X(i+1, j, h)$ for each $h \in [0, k-1]$.

Example 6. *An example of core-based ordering is given in Figure 3.3. Consider*

vertex v_5. *Given an integer* $k = 2$, *we need to compute the* k*-probability of* v_5, *which is* $Pr[deg(v_5|\{(v_2, v_5), (v_3, v_5), (v_4, v_5), (v_6, v_5)\}) \geq 2]$. *Given that the* $Pr[deg(v_5|\{(v_2, v_5), (v_3, v_5), (v_4, v_5)\}) = i]$ *for* $i \in [0, 3]$ *has been computed in the previous iteration of* $k = 3$, *we only compute* $Pr[deg(v_5|\{(v_6, v_5)\}) = i]$ *for* $i \in [0, 2]$.

The Overall Algorithm. Based on the two proposed optimizations, we give the overall algorithm to construct the $UCF\text{-}Index$. The pseudocode is presented in Algorithm 7. For each vertex, we compute the core number and sort the neighbors by their core numbers in lines 1 and 2. We decrease k from k_{max} to 1 in each iteration of lines 4–19. We initialize the k-probability for each vertex based on Equation 3.7. Then similar to Algorithm 5, we iteratively remove the vertex with the minimum k-probability from the graph and push it into a stack $\mathcal{S}$ in line 14. With this strategy, the vertex with the smallest η-threshold is at the bottom of $\mathcal{S}$, while the vertex with the largest η-threshold is at the top of $\mathcal{S}$.

Based on the stack $\mathcal{S}$, we construct the η-tree in Algorithm 8. Given that the vertex is sorted by the η-threshold in $\mathcal{S}$. We compute the set of vertices with the largest η-threshold in lines 2–5. According to Lemma 8, any two vertices u and v belong to the same η-tree node if $\eta\text{-}threshold_k(u) = \eta\text{-}threshold_k(v)$ and (u, v) exists. Therefore, we safely create a tree node $\mathbb{X}$ for the vertex set of each connected component (lines 7–8). We connect or merge $\mathbb{X}$ to existing tree nodes in lines 9–15. In line 9, we locate the neighbors of connected component $\mathcal{G}[C]$, i.e., $\mathcal{N}(\mathcal{G}[C]) = \{v \in V | u \in C, v \notin C, (u, v) \in E\}$. Note that $\eta\text{-}threshold_k(v) \neq ct$ for each neighbor v in line 9. Otherwise, v will be contained in the connected component $\mathcal{G}[C]$. If $\eta\text{-}threshold_k(v) < ct$, that means the tree node containing v has not been created, and that vertex is skipped in line 10. We locate the tree node $\mathbb{Y}$ containing v in line 11, and the root node $\mathbb{Z}$ of $\mathbb{Y}$ in line 12. We assign the parent of $\mathbb{Z}$ as $\mathbb{X}$ in line 14 if $\mathbb{Z}.\eta\text{-}threshold$ is larger. Otherwise, we have

Algorithm 7: UCF-Index Construction

Input: An uncertain graph $\mathcal{G}(V, E, p)$
Output: $UCF\text{-}Index$ index of $\mathcal{G}$

invoke Algorithm 1 to compute $core(u)$ for all $u \in V$;
sort the neighbors of each $u \in V$ by their core numbers;
$k_{max} \leftarrow \max_{u \in V} core(u)$;
for $k \leftarrow k_{max}$ **to** 1 **do**
    $\mathcal{G}'(V', E', p) \leftarrow \mathcal{G}[\{u \in V | core(u) \geq k\}]$;
    foreach $u \in V'$ **do**
        compute $k\text{-}prob(u, \mathcal{G}')$ according to Equation 3.7;
    $curThres \leftarrow 0$;
    $\mathcal{S} \leftarrow$ initialize an empty stack;
    while $\mathcal{G}'$ *is not empty* **do**
        $u \leftarrow \arg\min_{v \in V'} k\text{-}prob(v, \mathcal{G}')$;
        $curThres \leftarrow \max(k\text{-}prob(u, \mathcal{G}'), curThres)$;
        $\eta\text{-}threshold_k(u) \leftarrow curThres$;
        $\mathcal{S}.push(u)$;
        foreach $v \in \mathcal{N}(u, \mathcal{G}')$ **do**
            remove the edge (u, v) from $\mathcal{G}'$;
            update $k\text{-}prob(v, \mathcal{G}')$;
        $V' \leftarrow V' \setminus \{u\}$;
    construct $\eta\text{-}tree_k$ by invoking Algorithm 8;
return $\eta\text{-}tree_k$ *for all* $1 \leq k \leq k_{max}$;

$\mathbb{X}.\eta\text{-}threshold = \mathbb{Z}.\eta\text{-}threshold$ and merge them into one tree node (line 15) even though $\mathbb{X}$ and $\mathbb{Z}$ are not directly connected. We analyze the running time of Algorithm 8 as follows.

Algorithm 8: η-TREE CONSTRUCTION

Input: An uncertain graph $\mathcal{G}$, an integer k and a vertex stack $\mathcal{S}$
Output: The $\eta\text{-}tree_k$ of $\mathcal{G}$

```
1  while S is not empty do
2      ct ← η-threshold_k(S.top());
3      H ← ∅;
4      while S ≠ ∅ ∧ η-threshold_k(S.top()) = ct do
5          H ← H ∪ {S.pop()};
6      foreach connected component G[C] ∈ G[H] do
7          X ← create a tree node;
8          X.vertices ← C;
9          foreach v ∈ N(G[C]) do
10             if η-threshold_k(v) < ct then continue;
11             Y ←get the node containing v;
12             Z ←get the root of Y;
13             if Z.η-threshold > X.η-threshold then
14                 Z.parent ← X;
15             else merge the tree nodes X and Z;
16 return η-trees_k;
```

Lemma 11. *Given an uncertain graph $\mathcal{G}(V, E, p)$ and the vertex stack $\mathcal{S}$, the time complexity of Algorithm 8 is $O(|E_k|)$, where E_k is the set of edges in the subgraph induced by $\{u \in V | core(u) \geq k\}$.*

Proof. Given the input stack $\mathcal{S}$, the total time complexity of line 5 is $O(|\mathcal{S}|)$. To compute the connected components in line 6, we need to detect the neighbors of each vertex. Note that we only need to detect the neighbors whose core number is not less than k, and the neighbors of each vertex are sorted by their core numbers in Algorithm 7. The total time complexity of both line 6 and line 9

is $O(|E_k|)$. Since the operations of the disjoint-set structure cost constant time, the overall time complexity of Algorithm 8 is $O(|E_k|)$. □

Based on Lemma 11 and Theorem 5, we summarize the time complexity of Algorithm 7 as follows.

Theorem 8. *Given an uncertain graph $\mathcal{G}(V, E, p)$, the time complexity of Algorithm 7 is $O(k_{max}|V|\log|V| + k_{max}^2|E|)$, where $k_{max} = \max_{u\in V} core(u)$.*

Proof. The time complexity of lines 1–2 is $O(|E|)$ using a bin sort method. Similar to Theorem 5, lines 5–18 takes $O(|V|\log|V| + k|E|)$ time. Line 19 takes $O(|E_k|)$ time. The overall time complexity is $O(k_{max}|V|\log|V| + k_{max}^2|E|)$. □

3.6 Experiments

We conducted extensive experiments to evaluate the performance of our proposed solutions. The algorithms evaluated in our experiments are summarized as follows:

- UC-Online: Algorithm 3.
- UCO-Query: Algorithm 4.
- UCF-Query: Algorithm 6.
- UCO-Construct: Algorithm 5.
- UCF-Construct: the naive algorithm to construct the $UCF\text{-}Index$, which invokes UCO-Construct to compute the η-threshold for each vertex.
- UCF-Construct-R: the algorithm to construct the $UCF\text{-}Index$, which applies the *core-based reduction* optimization from Section 3.5.

- UCF-Construct* (Algorithm 7): the algorithm to construct the $UCF\text{-}Index$, which applies both *core-based order* and *core-based ordering* optimizations from Section 3.5.

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with Intel Xeon 3.46GHz CPU, 96GB 1866MHz ECC DDR3-RAM, and 2TB 7200 RPM SATA III Hard Drive.

Datasets. We used eight publicly-available real-world graphs to evaluate the algorithms. The edge probabilities in the first four datasets came from real-world applications, while the probabilities in the last four datasets were randomly assigned.

Krogan is a protein-protein interaction (PPI) network [44]. The vertices represent proteins, and the edges represent the interactions between pairs of proteins. The edge probability represents the possibility of an interaction between the pair of proteins connected by this edge [28]. Flickr (`https://www.flickr.com`) is an online community for sharing photos. The edge probability is the Jaccard coefficient of interest groups two users share [52, 13]. DBLP (`https://dblp.uni-trier.de`) is a computer science bibliography website. Each vertex corresponds to an author, and edges represent co-authorships. The edge probability is an exponential function based on the number of collaborations [52, 13]. BioMine (`https://www.cs.helsinki.fi/group/biomine/`) is a snapshot of the database of the BioMine project [25] containing biological interactions. The edge probability is based on the confidence that the interaction actually exists [52, 13].

Web-Google is a web network. Cit-Patents is a citation network. LiveJournal and Orkut are social networks. A detailed description of these four networks can be found in SNAP (`http://snap.stanford.edu/index.html`) with edge

Table 3.1: Network statistics

Datasets	$\|V\|$	$\|E\|$	deg_{max}	k_{max}
Krogan	2,559	7,031	141	15
Flickr	24,125	300,836	546	225
DBLP	684,911	2,284,991	611	114
BioMine	1,008,201	6,722,503	139,624	448
Web-Google	875,713	4,322,051	6,332	44
Cit-Patents	3,774,768	16,518,947	793	64
LiveJournal	3,997,962	34,681,189	14,815	360
Orkut	3,072,441	117,185,083	33,313	253

probabilities assigned uniformly and at random using the interval $[0, 1]$.

Detailed statistics of these datasets are summarized in Table 3.1. The maximum degree (deg_{max}) and the maximum core number (k_{max}) are shown in the last two columns.

3.6.1 Performance of Query Processing

Our first set of experiments evaluate the performance of query processing by varying k and η. We adopt similar settings used in [13] for η, and choose 0.1, 0.3, 0.5, 0.7, and 0.9, with 0.5 as the default. We choose 5, 10, 15, 20, and 25 for k, with 15 as the default. Due to the space limitations, we only report the figures for DBLP, BioMine, LiveJournal and Orkut. The results on other datasets show the similar trends. The results for all datasets using the default parameter settings follow.

Evaluation-I: Varying k. The running time for UCF-Query, UCO-Query, and UC-Online when varying k is shown in Figure 3.4. UCF-Query is faster than UCO-Query for every k on all datasets, and the gap between them gradually increases as k grows. For example, in LiveJournal, UCF-Query takes about 14ms while UCO-Query takes around 435ms when $k = 5$. When k reaches

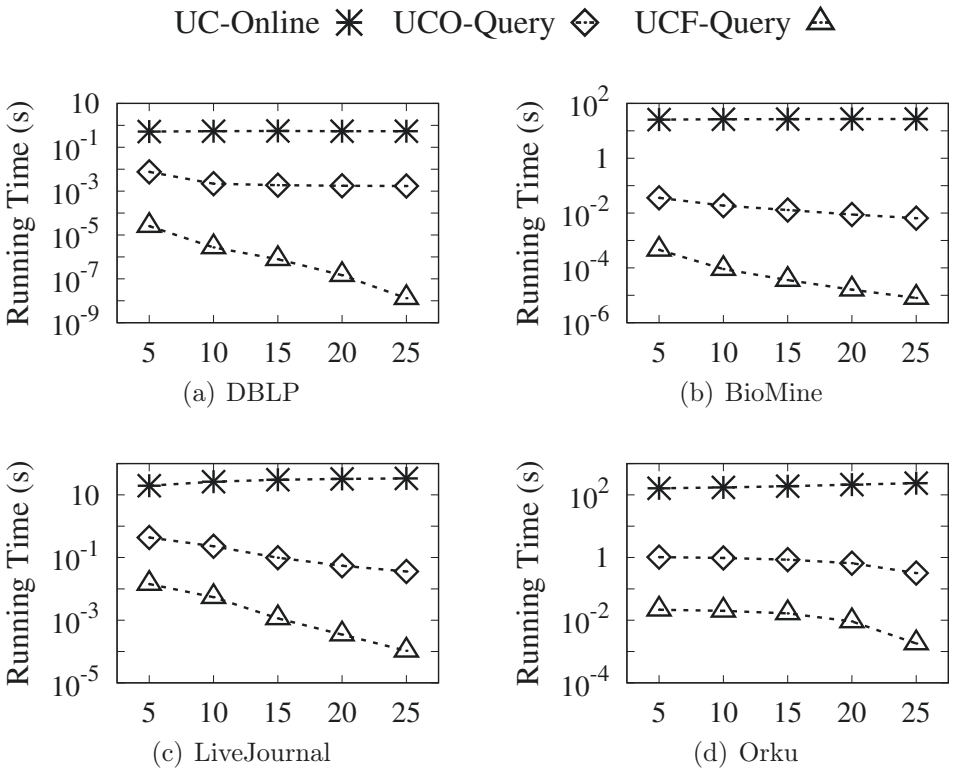


Figure 3.4: Query time for different k ($\eta = 0.5$)

25, UCF-Query takes only 105μs (1μs = 10^{-6}s), while UCO-Query still takes approximately 36ms. Additionally, as k grows, UCF-Query shows a significant downward trend on all datasets, because the time UCF-Query takes to process is strictly dependent on the size of results, and the size of results becomes smaller as k increases. The time for UCO-Query is relatively steadier compared to UCF-Query. However, on some datasets, UC-Online shows slightly upward trends, since it takes more time to initialize and update η-degree for a large k. Overall, UCF-Query is significantly faster than UC-Online, particularly for a large k.

Evaluation-II: Varying η. Figure 3.5 shows the running time for

UCF-Query, UCO-Query, and UC-Online when varying η. Compared to Figure 3.4, the changes as η increases for all algorithms are not as obvious. In fact, for some datasets, the trends are almost stable. For example, with BioMine, the running time for UCF-Query drops from 67μs with $\eta = 0.1$ to 17μs with $\eta = 0.9$. Meanwhile, UCO-Query drops from about 16ms to 10ms. Additionally, unlike Figure 3.4, the running time of UC-Online demonstrates a slight downward trend with some datasets. The dominating factor, in this case, is that the η-degree becomes smaller as η becomes larger. We can see that the running time for *UCO-Index* and *UCF-Index* is more sensitive to k than η. This is mainly because the size of k-core will be largely reduced when k increases, and the (k, η)-core is contained in a k-core. Overall, the UCF-Query significantly outperforms both UCO-Query and UC-Online.

Evaluation-III: Query Performance on Different Datasets. The running time for UCF-Query, UCO-Query, and UC-Online with the default parameters $k = 15$ and $\eta = 0.5$ on all datasets are shown in Figure 3.6. We can see that UCF-Query is not only more efficient than UCO-Query but is also several orders of magnitude faster than UC-Online on all datasets. The running time for UCF-Query on Krogan is about 0.012μs, which is the smallest value of all the results. Meanwhile, the costs for UCO-Query and UC-Online are about 8μs and 2ms, respectively. On the Orkut dataset with over 100 million edges, UCF-Query only takes about 17ms, while UCO-Query and UC-Online take approximately 857ms and 190s, respectively.

3.6.2 Performance of Index Construction

Evaluation-IV: Index Size with Different Datasets. The size of *UCF-Index* for different datasets is reported in Figure 3.7 with *UCO-Index* added as a comparison. The size of *UCF-Index* gradually grows as the number

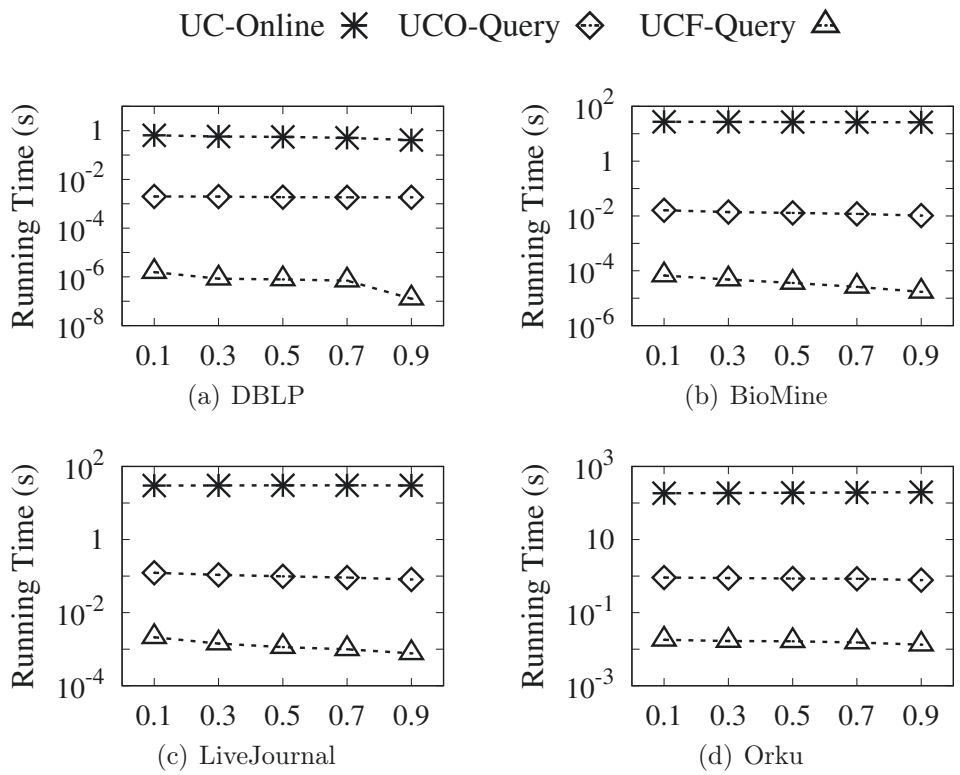


Figure 3.5: Query time for different η ($k = 15$)

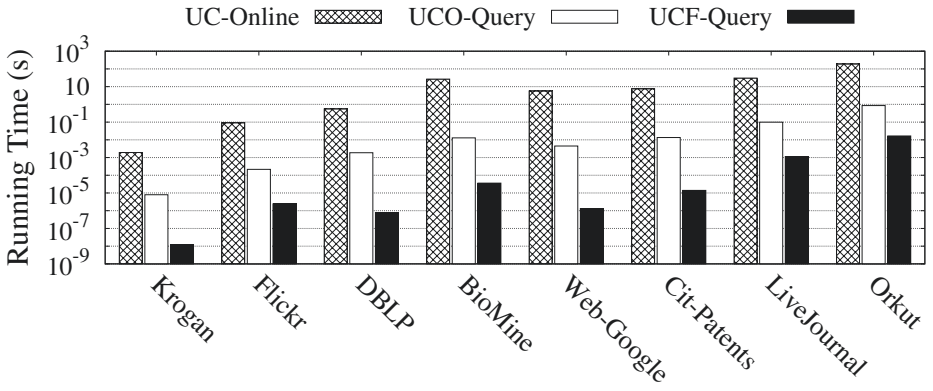


Figure 3.6: Query time on different datasets

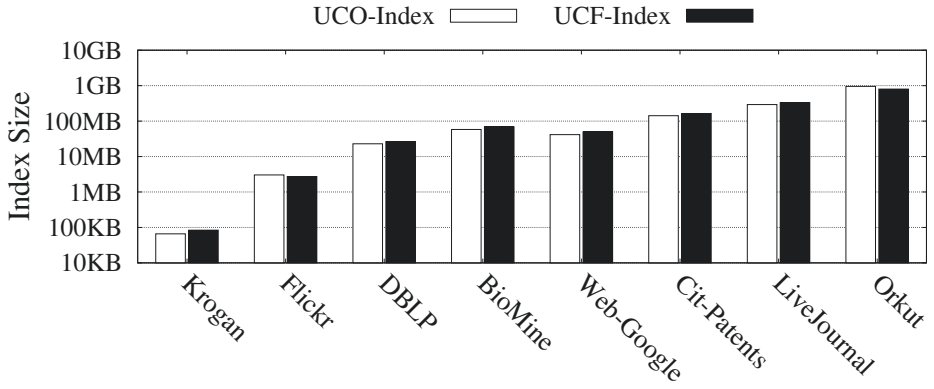


Figure 3.7: Index size for different datasets

of edges increases, and with most datasets, $UCF\text{-}Index$ needs more space due to the maintenance of the parent-children relationship in the η-tree. For example, in Cit-Patents, $UCO\text{-}Index$ takes up about 140MB, while $UCF\text{-}Index$ consumes about 160MB. However, we find that, with some datasets, the size of $UCF\text{-}Index$ is smaller than $UCO\text{-}Index$. For example, in Orkut, $UCO\text{-}Index$ takes up about 950MB, while $UCF\text{-}Index$ only needs 800MB. This is because several vertices were contracted into each tree node, and the η-threshold is only maintained for the tree node.

Evaluation-V: Construction Time on Different Datasets. This experiment is designed to evaluate our optimization techniques for index construction. UCF-Construct denotes the naive algorithm to construct the $UCF\text{-}Index$, with Algorithm 5 to compute the η-thresholds for all vertices and all integers of k. UCF-Construct-R denotes the *core-based reduction* technique described in Subsection 3.5.3. UCF-Construct* denotes the combined *core-based reduction* and *core-based ordering* optimizations (Algorithm 7). We have also included the running time for UCO-Construct, the algorithm that constructs the $UCO\text{-}Index$, as a comparison. The results are reported in Figure 3.8. Compared to UCO-Construct,

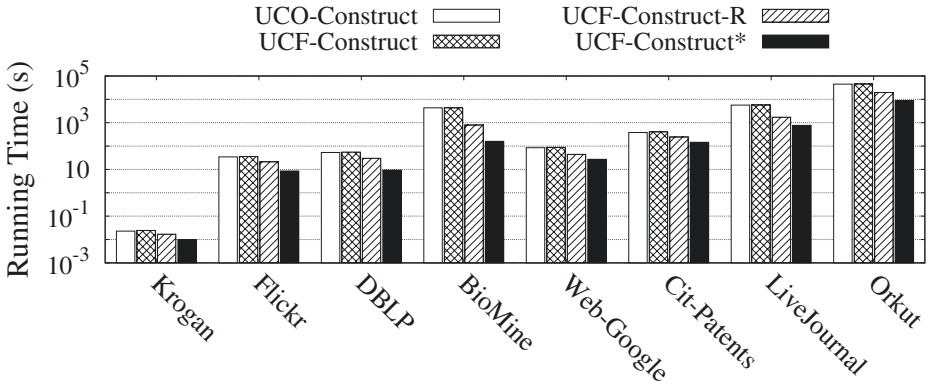


Figure 3.8: Time cost for index construction

UCF-Construct originally requires additional time to construct the η-tree; however, our two optimizations significantly reduce this time. On the largest dataset Orkut, UCO-Construct, UCF-Construct, UCF-Construct-R, and UCF-Construct* take approximately 12.5 hours, 12.6 hours, 5.5 hours, and 2.4 hours, respectively.

Evaluation-VI: Scalability Testing. This experiment tests the scalability of our proposed algorithms. Due to the space limitations, we have only included the results for two real-world graph datasets as representatives — BioMine and Orkut. The results using the other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling nodes and edges from 20% to 100%. When sampling nodes, we derive the induced subgraph of the sampled nodes, and when sampling edges, we select the incident nodes of the edges as the vertex set. The time cost of UCF-Construct* at different percentages are reported in Figure 3.9, with UCO-Construct, UCF-Construct and UCF-Construct-R as comparisons.

As we can see as $|V|$ or $|E|$ increases, the processing time to construct all the indexes grows. However, the gap between UCF-Construct and UCO-Construct

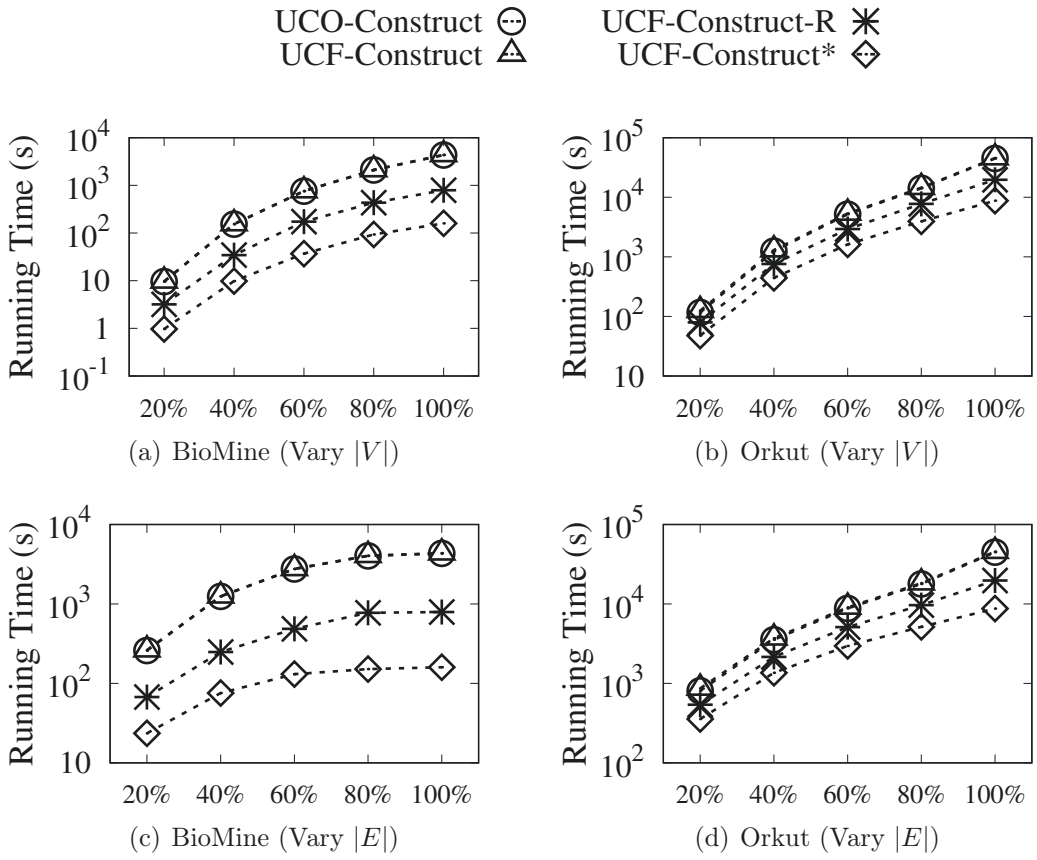


Figure 3.9: Scalability of index construction

is not obvious, because the major cost in UCF-Construct is computing the η-thresholds. UCF-Construct* performs better than the other algorithms in all cases, and the gaps between UCF-Construct* and the other algorithms increase as the sampling ratio increases. Overall, these results imply that our optimization techniques are effective, especially with big graphs.

3.7 Chapter Summary

This chapter presents an index-based solution for computing all the (k, η)-cores in uncertain graphs. Our proposed index, called $UCF\text{-}Index$, maintains a tree structure for each integer k. The size of $UCF\text{-}Index$ is well-bounded by $O(m)$. Based on $UCF\text{-}Index$, queries for any input parameter k and η can be answered in optimal time. Two optimizations for $UCF\text{-}Index$ are also presented to speed up the index construction. The results of extensive performance studies demonstrate the effectiveness of this index-based approach and the efficiency of the query algorithm.

Chapter 4

SEMI-EXTERNAL MEMORY CORE COMPUTATION IN LARGE UNCERTAIN GRAPHS

4.1 Overview

In this chapter, we introduce our proposed index-based solution for computing all the (k, η)-cores in uncertain graphs under semi-external memory setting. The work is published in [70] and the rest of this chapter is organized as follows. We introduce the data structure to store $UCF\text{-}Index$ in external memory and give the corresponding query processing algorithm in Section 4.2. Section 4.3 proposes a new strategy to locally compute η-thresholds, and Section 4.4 presents the corresponding algorithm for index construction. Section 4.5 proposes several optimizations to further reduce I/Os and improve efficiency. Section 4.6 practically evaluates the proposed algorithms in practical terms. Section 4.7 summarizes the chapter.

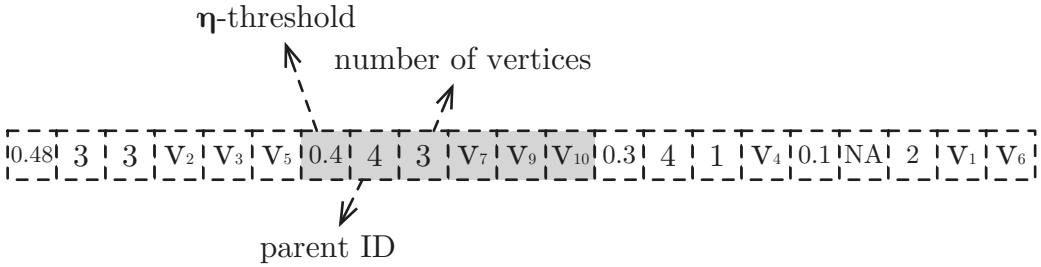


Figure 4.1: The $UCEF\text{-}Index$ of $\mathcal{G}$ for $k = 2$

4.2 UCF-Index in External Memory

We can naturally extend the structure of $UCF\text{-}Index$ for the external memory setting, which is called $UCEF\text{-}Index$. Specifically, for each integer k, all tree nodes are arranged in a non-increasing order of their η-thresholds. For each tree node, we store the following three elements, (1) the node's η-threshold, (2) the node's parent ID, and (3) the containing vertices. Note that the nodes' IDs are assigned in the index construction by the order they arranged in the hard disk. Consequently, we can derive the current vertex ID in query processing accordingly and avoid the ID storage in the index. For the containing vertices of each node, we store an integer in the front to indicate the number of vertices. We give an example of the index as follows.

Example 7. *We show the $UCEF\text{-}Index$ of $\mathcal{G}$ (Figure 1.1) for $k = 2$ in Figure 4.1. The corresponding tree structure is in Figure 3.2. We mark the fragment of the tree node containing $\{v_7, v_9, v_{10}\}$ by gray. The node is the second one in the sequence, and its implicit ID is 2. The parent ID is 4, which is the last tree node.*

I/O Efficient Query Processing. Based on $UCEF\text{-}Index$, we can use a similar idea as Algorithm 6 to answer (k, η)-core queries. We call the query processing algorithm for the external index UCEF-Query. Given an integer k and

a probability threshold η, we sequentially scan the $UCEF\text{-}Index$ for k. Since tree nodes are arranged in a non-increasing order of η-thresholds, each scanned node naturally satisfies line 4 of Algorithm 6. After loading all nodes with η-threshold not less than η, we derive the tree structure based on the parent ID of each node. The tree size is bounded by $O(n)$, and the result can be computed using lines 8 – 21 of Algorithm 6. The I/O cost of UCEF-Query is still optimal.

Theorem 9. *Given an integer k and a probability threshold η, the I/O complexity of* UCEF-Query *is $O(|C|/B)$, where $C = \{u \in V | \eta\text{-}threshold_k(u) \geq \eta\}$, and B is the block size.*

4.3 Local η-Threshold Computation

We can naively perform an in-memory algorithm, e.g., UCF-Construct*, to construct $UCEF\text{-}Index$. Specifically, for each integer k, we always first process the vertex with the smallest k-probability. Under the $O(n)$ memory limitation, we load neighbors of each vertex from external memory for computing k-probability and release the memory for loading neighbors of the next vertex. However, this strategy incurs significant I/O cost due to frequent random access of the external memory.

To improve the efficiency, we propose a new framework, called UCEF-Construct, tailored for the external memory setting. The framework releases the order limitation and computes the η-threshold of each vertex only using the neighbors' η-thresholds. Given an integer k and a probability value p, let $N_k^p(u)$ be the neighbors of u whose η-threshold is at least p in the (k, η)-core, i.e., $N_k^p(u) = \{v \in N(u) | \eta\text{-}threshold_k(v) \geq p\}$. The key theorem to support UCEF-Construct is given as follows.

Theorem 10. *Given an integer k, a probability value p, and a vertex u with $core(u) \geq k$, we have $p = \eta\text{-}threshold_k(u)$ iff*

1. *the k-probability of u in $N_k^p(u)$ is not smaller than p, i.e., $k\text{-}prob(u, \mathcal{G}[\{u\} \cup N_k^p(u)]) \geq p$; and*
2. *there does not exists a probability value p' s.t. $p' > p$ and p' satisfies condition (1).*

Example 8. *We give an example to explain Theorem 10. We consider the vertex v_4 in Figure 1.1. Given $k = 2$, the η-thresholds for neighbors of v_4 can be found in Figure 3.1. Assume that $p = 0.48$. We have $N_2^{0.48}(v_4) = \{v_2, v_3, v_5\}$. The probability that v_4 connects at least 2 neighbors in $N_2^{0.48}(v_4)$ is 0.3. Since $0.3 < 0.48$, p is not the η-threshold of v_4, which does not satisfy the condition 1 in Theorem 10. In this case, the correct η-threshold for v_4 should be 0.3.*

Based on Theorem 10, the procedure local_thres in Algorithm 9 shows the pseudocode for locally computing the η-threshold of u. Given a vertex u and a set of vertices N, we also use N to represent the induced subgraph of u and N, i.e., $\mathcal{G}[\{u\} \cup N]$, for ease of presentation when context is clear. For example, in line 7 of local_thres, $k\text{-}prob(u, N_i)$ is short for $k\text{-}prob(u, \mathcal{G}[\{u\} \cup N_i])$.

Lemma 12. *Given an integer k, a query vertex u and the η-thresholds for its neighbors N,* local_thres *correctly computes the η-threshold for u.*

Proof. According to the condition 1 of Theorem 10, all η-thresholds for the selected neighbors in computing k-probability are not smaller than p, and we have $\hat{t}(u) \leq p_1$. The computed k-probability is not smaller than p, and we have $\hat{t}(u) \leq p_2$. By setting $\hat{t}(u) = \min(p_1, p_2)$, we guarantee $\hat{t}(u)$ in each iteration always satisfies the condition 1 of Theorem 10. Note that we start from $i = k$ since p_2 would be 0 if $i < k$.

The condition 2 in Theorem 10 actually guarantees that p is the largest possible value satisfying the condition 1. local_thres computes such p in a bottom-up strategy. Specifically, we can find that the variable p_1 is monotonic decreasing, and p_2 is monotonic increasing. As a result, $\min(p_1, p_2)$ first monotonically increases to a peak and then monotonically decreases. Line 8 of local_thres checks whether the current $\hat{t}(u)$ reaches the peak. Once $\min(p_1, p_2)$ stops increasing, the procedure breaks the iteration and derives the correct $\hat{t}(u)$. □

In line 7 of local_thres, we do not need to compute k-$prob(u, N_i)$ from scratch in each iteration. Based on Equation 3.3, we maintain $Pr[deg(u, N_i) = j]$ for $0 \leq j \leq k-1$ in each iteration. The first iteration takes $O(k^2)$ time. In the iteration i with $i > 1$, assume $N_i = N_{i-1} \cup \{v\}$. We have $Pr[deg(u, N_i) = j] = (1 - p_{(u,v)}) \cdot Pr[deg(u, N_{i-1}) = j] + p_{(u,v)} * Pr[deg(u, N_{i-1}) = j - 1]$. Therefore, the total time complexity of line 7 is $O(k \cdot Deg(u))$, which is the same as that of computing k-probability of u in $\mathcal{G}$.

4.4 Semi-external Algorithms

Based on Theorem 10, we give the pseudocode of UCEF-Construct in Algorithm 9. To derive core numbers of all vertices under the semi-external setting, we adopt the algorithm SemiCore* in [69]. SemiCore* uses $O(n)$ memory space and computes core numbers in several iterations of sequentially reading the external graph. For each integer k, we compute η-thresholds in lines 3–13.

For each vertex u, we maintain a core number $core(u)$, a probability value $\hat{t}(u)$ as an estimation of η-$threshold_k(u)$, and an indicator to represent whether the vertex is active. Lines 3–4 initialize $\hat{t}(u)$ and mark all vertices in the k-core as active. Line 10 computes a new $\hat{t}(u)$ according to the current $\hat{t}$ values of neighbors. Lines 12–13 mark possibly influenced neighbors as active. The

Algorithm 9: UCEF-Index Construction

Input: An uncertain graph $\mathcal{G}(V, E, p)$
Output: *UCEF-Index* of $\mathcal{G}$

compute $core(u)$ for all $u \in V$;[69]
for $k \leftarrow k_{max}$ **to** 1 **do**
  foreach $u \in V : core(u) \geq k$ **do**
    $\hat{t}(u) = 1$, and mark u as active;
  while *active vertex exists* **do**
    foreach $u \in V : u$ *is active* **do**
      mark u as inactive;
      load $N(u)$ from the disk;
      $\hat{t}_{old} \leftarrow \hat{t}(u)$;
      $\hat{t}(u) \leftarrow \mathsf{local\_thres}(u, N(u), k)$;
      if $\hat{t}(u) = \hat{t}_{old}$ **then continue**;
      foreach $v \in N(u) : \hat{t}(u) < \hat{t}(v) \leq \hat{t}_{old}$ **do**
        mark v as active;
  `// construct` η`-tree for` k
  sort vertices of k-core in non-increasing order of $\hat{t}$ values, and write their neighbors accordingly in external memory;
  invoke Algorithm 8 to construct η-tree;

Procedure $\mathsf{local\_thres}(u, N, k)$:
  sort vertices in N in non-increasing order of $\hat{t}$ values;
  $\hat{t}(u) \leftarrow 0$;
  for $k \leq i \leq |N|$ **do**
    $p_1 \leftarrow$ the i-th $\hat{t}$ value in N;
    $N_i \leftarrow$ the first i vertices in N;
    $p_2 \leftarrow$ compute k-$prob(u, N_i)$;
    if $\hat{t}(u) \geq \min(p_1, p_2)$ **then break**;
    $\hat{t}(u) \leftarrow \min(p_1, p_2)$;
  return $\hat{t}(u)$;

η-threshold computation terminates if there is no active vertex. For each vertex u, $\hat{t}(u)$ is always an upper bound of η-$threshold(u)$, which never increases and converges to η-$threshold(u)$ finally. The proof of the algorithmic correctness is similar to that of [49, 69], and we omit the details here.

Note that the condition in line 12 significantly reduces unnecessary active vertices. We explain the rationale as follows. For a vertex u, the computation of $\hat{t}(u)$ of each vertex is based on the neighbors in $N_k^{\hat{t}(u)}(u)$ according to Theorem 10. $\hat{t}(u)$ requires to be updated if $N_k^{\hat{t}(u)}(u)$ changes. Since $\hat{t}(u)$ for any vertex u never increases, we mark a vertex v as active if its neighbor u leaves $N_k^{\hat{t}(v)}(v)$. In line 12, $\hat{t}(v) \leq \hat{t}_{old}$ means that u is in $N_k^{\hat{t}(v)}(v)$ when computing $\hat{t}(v)$, and $\hat{t}(u) < \hat{t}(v)$ means that u leaves $N_k^{\hat{t}(v)}(v)$.

Theorem 11. *The I/O complexity of computing η-thresholds of all vertices for every possible k is $O(\frac{l \cdot m}{B})$, where l is the total number of iterations, and B is the block size.*

η-Tree Construction. We construct η-trees in lines 14–15. In the in-memory algorithm to construct η-tree (Algorithm 8), vertices are processed in a non-increasing order of their η-thresholds. We adopt the same idea here and create a temporary file to arrange neighbors of vertices in k-core in such order. As a result, we can construct the tree by sequentially reading the required vertex neighbors from external memory in only one iteration. The temporary file can be constructed using a traditional external-sorting algorithm with the I/O complexity $O(\frac{m}{B} \log_{\frac{M}{B}} \frac{m}{B})$, where M is the memory size [43]. The semi-external setting allows us to use $O(n)$ memory. Since $n^2 \geq m$, $\log_{\frac{n}{B}} \frac{m}{B}$ can be regarded as a constant, and the I/O complexity of external sorting is reduced to $O(m/B)$. For an integer k, the I/O cost of η-tree construction is bounded by $O(m/B)$, and the overall I/O cost of Algorithm 9 is still $O(\frac{l \cdot m}{B})$.

4.5 Further Optimizations

It is obvious to see that the dominating cost in Algorithm 9 is incurred by computing η-thresholds. We propose several optimizations to reduce the I/O cost of this step and further improve the efficiency of Algorithm 9.

4.5.1 Reducing η-Threshold Estimations

Recall that for each vertex u in Algorithm 9, we initialize $\hat{t}(u)$ by 1, which is a very loose upper bound of the η-threshold for u. We reduce unnecessary η-threshold computations by setting a relatively tighter upper bound. Given an integer $1 < k \leq k_{max}$ and an arbitrary vertex u, we can naturally use $\eta\text{-}threshold_{k-1}(u)$ as an upper bound of $\eta\text{-}threshold_k(u)$ based on Lemma 5. To implement this idea, we adopt a bottom-up strategy which computes η-thresholds from $k = 1$ to $k = k_{max}$. In this way, we set $\hat{t}(u) = \eta\text{-}threshold_{k-1}(u)$ in line 4 of Algorithm 9.

4.5.2 Partial Neighbor Loading

According to Theorem 10, the η-threshold computation only requires the neighbors whose core numbers are not smaller than k. We reduce the I/O cost of the η-threshold computation by only loading necessary neighbors of each vertex. We implement this idea by sorting neighbors of each vertex after computing core numbers in line 1 of Algorithm 9. Specifically, we load neighbors of each vertex from the external memory. We sort the neighbors in non-increasing order of their core numbers and write back to the external memory. The I/O complexity of this step is $O(m/B)$. Given the sorted neighbors of each vertex, in line 8 of Algorithm 9, we sequentially read neighbors one by one from external memory until a neighbor is found with the core number smaller than k. This step reduces the I/O cost of line 8 from $O(|N(u)|/B)$ to $O(|N_k(u)|/B)$.

4.5.3 Vertex Ordering

Compared with the in-memory index construction, Algorithm 9 may perform line 10 several times for each vertex u until $\hat{t}(u)$ converges to η-$threshold_k(u)$ even using a tighter upper bound in Subsection 4.5.1. Intuitively, $\hat{t}(u)$ will be close to η-$threshold_k(u)$ if $\hat{t}$ values for all neighbors are close to their η-thresholds. A special case is shown as follows.

Lemma 13. *Given an integer k, assume that all vertices are sorted in a non-decreasing order of their η-thresholds for k, i.e., $\forall u, v \in V, \eta$-$threshold_k(u) \leq \eta$-$threshold_k(v)$ if $u < v$. Line 10 of Algorithm 9 performs only once in computing η-$threshold_k(u)$ for every vertex u.*

In the case of the lemma, vertices in line 6 are processed in non-decreasing order of η-thresholds. $\hat{t}(u)$ derived in line 10 is exactly η-$threshold_k(u)$ according to Theorem 10. Based on Lemma 13, we aim to improve the efficiency of Algorithm 9 by postponing the η-threshold computations of some vertices if their η-thresholds are relatively large with a high probability. To implement this idea, we sort vertices in external memory after line 1 of Algorithm 9 using several heuristic rules. We first arrange vertices in a non-decreasing order of their core numbers. We break a tie by considering the probability that the vertex u has at least one neighbor. Specifically, given two vertices u and v with $core(u) = core(v)$, we assign u to the front if $Pr[deg(u, \mathcal{G}) \geq 1] < Pr[deg(v, \mathcal{G}) \geq 1]$. The computation of $Pr[deg(u, \mathcal{G}) \geq 1]$ for all vertices u takes $O(m/B)$ I/Os since neighbors of each vertex are required.

We perform an external sorting algorithm to rearrange the graph structure according to the new vertex order, which takes $O(m/B)$ I/Os, similar to the discussion in Section 4.4. Note that the neighbor ordering discussed in Subsection 4.5.2 can be done as a byproduct in the vertex ordering.

4.6 Experiments

We conducted extensive experiments to evaluate the performance of our proposed solutions. The datasets and experiment settings are the same as that in Section 3.6.

4.6.1 Performance of Semi-external Query Processing

Evaluation-I: Query Performance on Different Datasets. The running time of UC-Online (Algorithm 3), UCO-Query (Algorithm 4), and UCF-Query (Algorithm 6) with the default parameters $k = 15$ and $\eta = 0.5$ on all datasets are shown in Figure 4.2. Regarding the external memory setting, the running time of UCEF-Query is also shown in the last bar for each dataset in Figure 4.2, and the corresponding number of I/Os is shown in Figure 4.3. The only difference between UCEF-Query and UCF-Query is that UCEF-Query loads the index from external memory. We can see the running time of UCEF-Query is between UCF-Query and UCO-Query on all datasets. For instance, on the Orkut dataset with over 100 million edges, UCEF-Query takes about 60ms, while UCF-Query and UCO-Query take approximately 17ms and 857ms, respectively. And the corresponding number of I/Os for UCEF-Query is 13896.

4.6.2 Performance of Semi-external Index Construction

We use UCF-Construct*-EM to represent the naive extension of UCF-Construct*, which loads neighbors of each vertex from external memory. We use UCEF-Construct* to denote our final algorithm for index construction in external memory with all optimizations in Section 4.5.

Evaluation-II: Memory Usage. In Figure 4.4, we report the memory usage of UCEF-Construct* with UCF-Construct*-EM and the in-memory algorithm

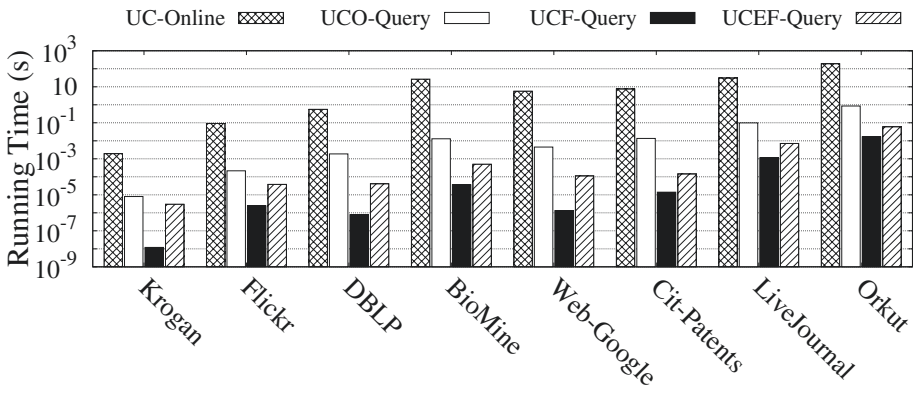


Figure 4.2: Query time on different datasets

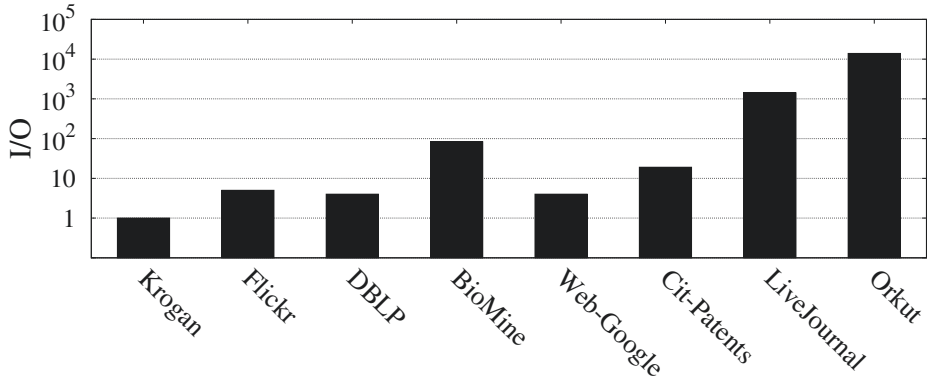


Figure 4.3: I/O cost of external query processing

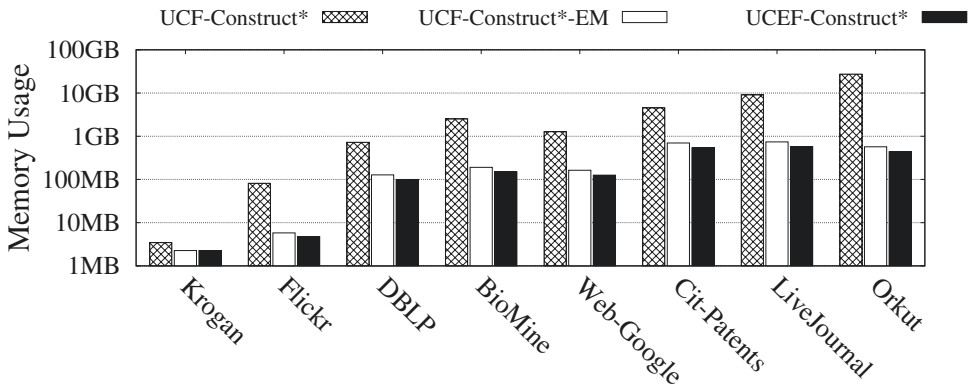


Figure 4.4: Memory usage for external index construction

UCF-Construct* as comparisons. We can see a considerable decrease in memory usage from UCF-Construct* to other external algorithms, since we limit the memory usage to $O(n)$. In the largest dataset Orkut, UCEF-Construct* takes about 430MB, while UCF-Construct* takes up to 26GB.

Evaluation-III: External Index Construction. The running time and I/O cost of our final algorithm UCEF-Construct* for external index construction are reported in Figure 4.5 and Figure 4.6, respectively. The performance of UCF-Construct*-EM is also reported as a comparison. We can find that the strategy for η-threshold computation in Theorem 10 is effective. In several datasets, UCEF-Construct* is one order of magnitude faster than UCF-Construct*-EM, and the I/O cost of UCEF-Construct* is almost two orders of magnitude smaller than that of UCF-Construct*-EM. For example, in DBLP, UCEF-Construct* and UCF-Construct*-EM take 11s and 189s, respectively.

Evaluation-IV: Optimizations for External Index Construction. We evaluate the effectiveness of optimizations proposed in Section 4.5. We use UCEF-Construct to denote Algorithm 9 without any optimizations in Section 4.5 and record its running time and I/O cost in each dataset. We use

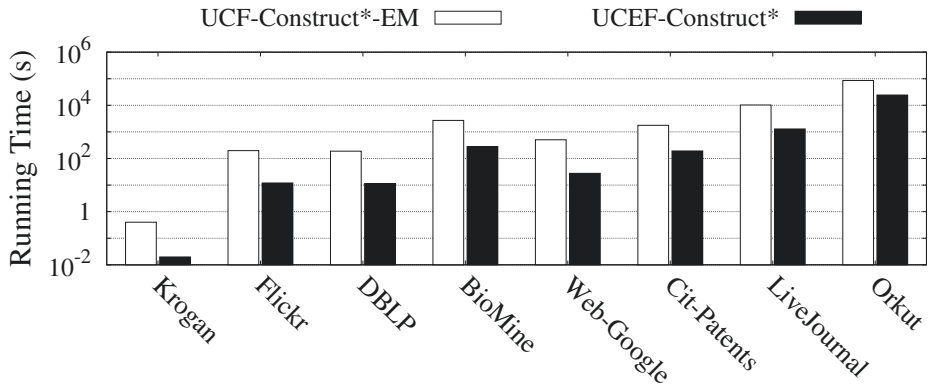


Figure 4.5: Time cost for external index construction

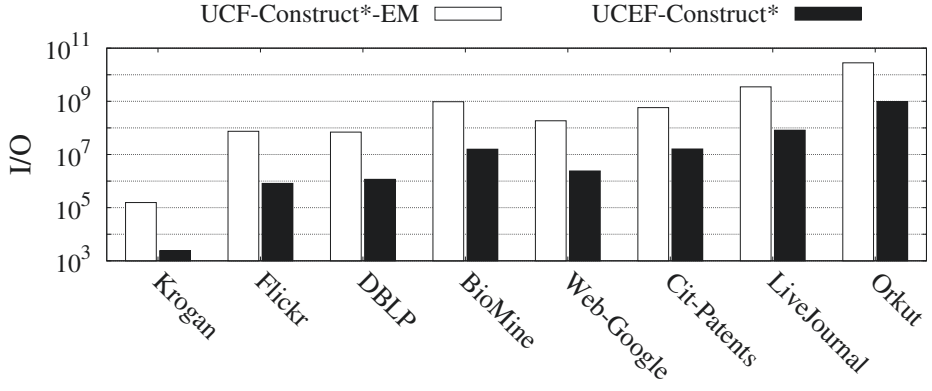


Figure 4.6: I/O cost for external index construction

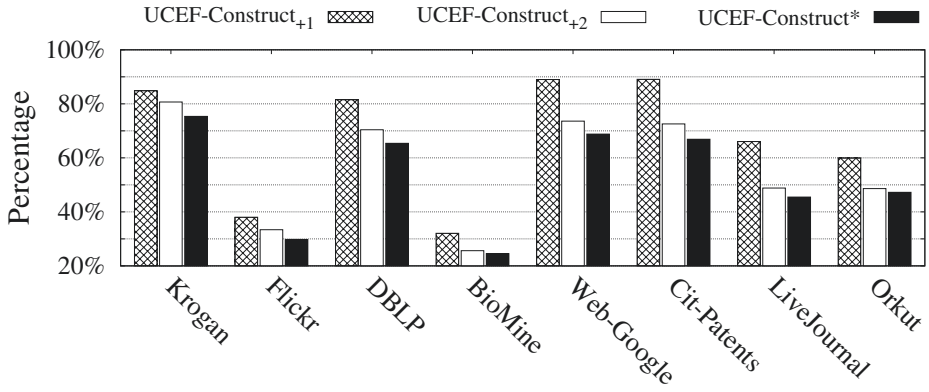


Figure 4.7: Speedup for external index construction

UCEF-Construct$_{+1}$ to denote the algorithm with the upper bound optimization in Subsection 4.5.1. We use UCEF-Construct$_{+2}$ to denote the algorithm with both upper bound optimization in Subsection 4.5.1 and neighbor ordering optimization in Subsection 4.5.2. Recall that UCEF-Construct* is the final algorithm with all three optimizations. We record the running time and I/O cost of these algorithms. For each dataset, we compute the percentages that the running time and the I/O cost of these algorithms account for those of UCEF-Construct. The results are shown in Figure 4.7 and Figure 4.8, respectively. The upper bound optimization is the most effective among them, and the speedup is obvious especially in large datasets. Note that in several small datasets of Figure 4.8, UCEF-Construct$_{+2}$ takes a little more I/Os than UCEF-Construct$_{+1}$ due to the external sorting of vertex neighbors. However, this optimization reduces a large number of unnecessary neighbors for the η-threshold computation and still achieves a speedup.

Evaluation-V: Scalability of External Index Construction. We evaluate the scalability of UCEF-Construct* and UCF-Construct*-EM. We vary the size and the density of Orkut by randomly sampling vertices and edges from 20% to

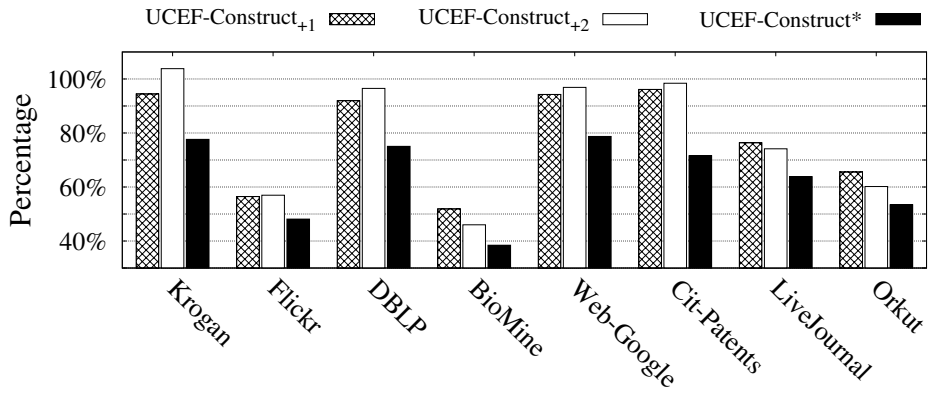


Figure 4.8: I/O reduction for external index construction

100%. When sampling vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set. The I/O cost is reported in Figure 4.9 (a) and Figure 4.9 (b). The running time is reported in Figure 4.9 (c) and Figure 4.9 (d).

4.7 Chapter Summary

This chapter presents an index-based solution for computing all the (k, η)-cores in uncertain graphs under semi-external memory setting. Our proposed index, called $UCEF\text{-}Index$, stores a tree structure for each integer k in the hard disk. The size of $UCEF\text{-}Index$ is also well-bounded by $O(m)$. Based on the index, the I/O cost for querying (k, η)-core is still optimal. We also propose an algorithm to construct the index in external memory. The results of extensive performance studies demonstrate the effectiveness of this index-based approach and the efficiency of the query algorithm under semi-external memory setting.

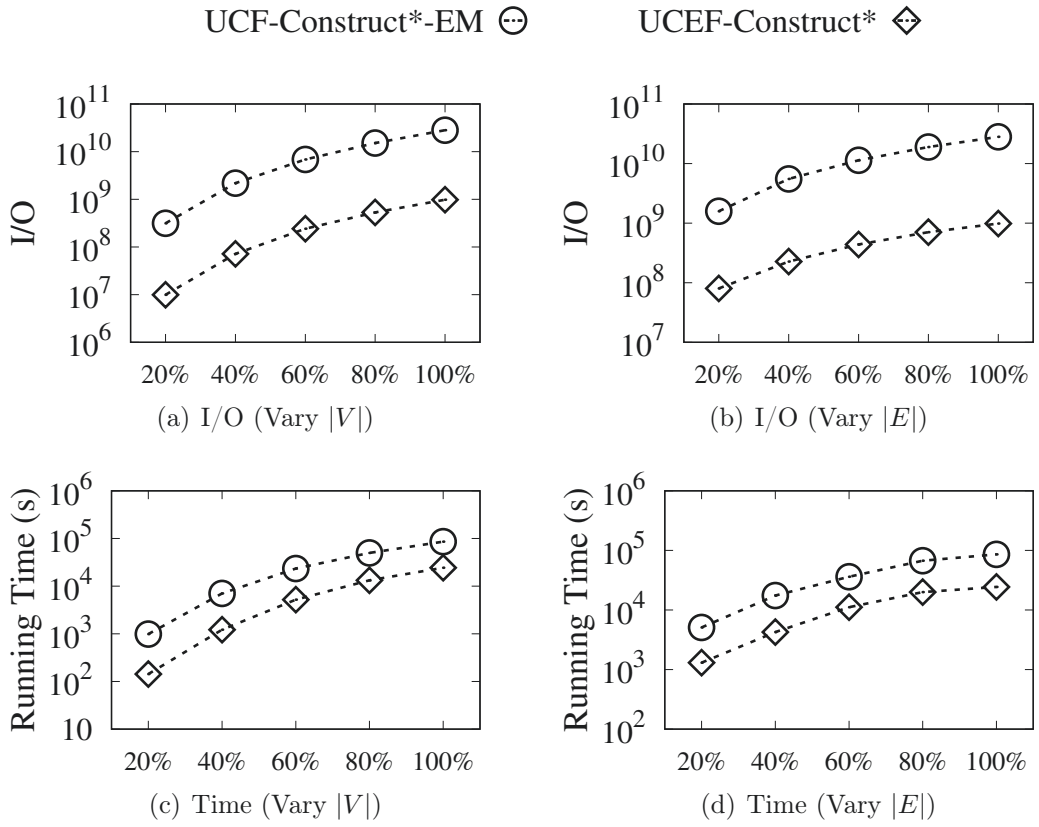


Figure 4.9: Scalability of external index construction on Orkut

Chapter 5

FULLY DYNAMIC DEPTH-FIRST SEARCH IN LARGE DIRECTED GRAPHS

5.1 Overview

In this chapter, we introduce a framework and corresponding algorithms for the DFS-Tree maintenance problem considering both edge insertion and deletion in general directed graphs. The work is published in [72] and the rest of this chapter is organized as follows. Section 5.2 introduces background knowledge about DFS and defines the research problem. Section 5.3 gives a framework for DFS-Tree maintenance. Section 5.4 gives the basic implementation for both edge insertion and deletion. Section 5.5 studies the optimizations. Section 5.6 reports the experiment result, and Section 5.7 summarizes the chapter.

5.2 Preliminary

Definition 9. (Depth-First Search) *Given a graph G, a depth-first search (DFS) traverses G in a particular order by picking an unvisited vertex v from the out-neighbors of the most recently visited vertex u to search, and backtracks to the vertex from where it came when a vertex u has explored all possible ways to search further. [21]*

For simplicity and without loss of generality, we add a virtual root vertex γ and connect γ to every vertex in G. We always perform the DFS traversal starting from γ and collect all vertices.

Definition 10. (DFS-Tree) *Given a graph G, a DFS-Tree of G, denoted by $\mathcal{T}_G$, is an ordered spanning tree formed by the process of DFS. [21]*

We omit the subscript G of $\mathcal{T}_G$ when the context is clear. Given a vertex u, we denote $\mathcal{C}(u)$ the children list of u in the DFS-Tree $\mathcal{T}$. Note that the DFS-Tree is not unique. There is a one-to-one correspondence between a vertex search order and a DFS-Tree. We give the pseudocode for computing the DFS-Tree in Algorithm 10, which is self-explanatory. Given an example graph G in Figure 1.2(a), one possible DFS-Tree $\mathcal{T}$ of graph G is shown in Figure 1.2(b).

Algorithm 10: DFS(u)

Input: a graph G, and a root vertex u in G
Output: a DFS-Tree $\mathcal{T}$

```
1 mark u as visited;
2 foreach v ∈ N_out(u): v is unvisited do
3 |   append v to the end of children list C(u) in T;
4 |   DFS(v);
```

The Validity of the DFS-Tree. Given a graph G and any search spanning tree $\mathcal{T}$ of G, the edges appearing in the tree are called *tree edges*. The remaining

edges (u, v) are called *non-tree edges* and are categorized into one the following four types:

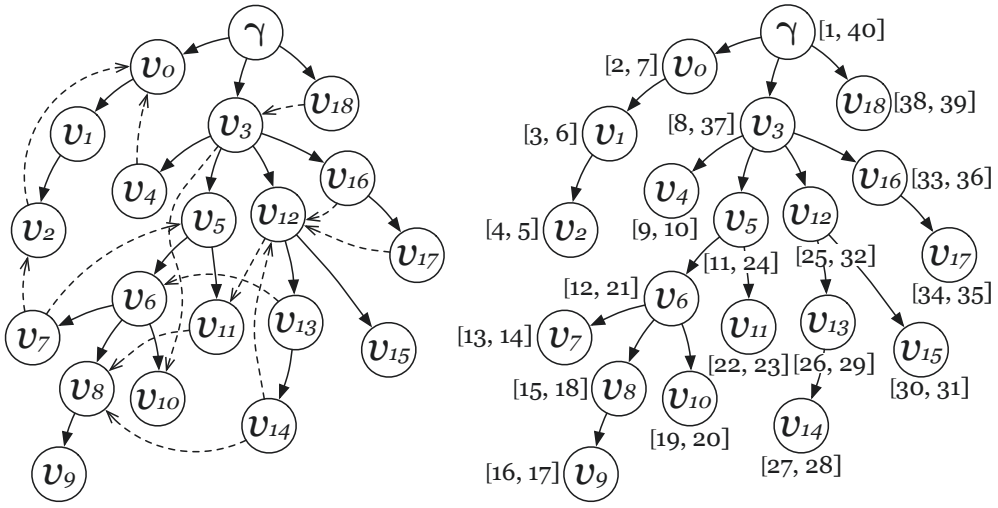
- (u, v) is a *forward edge* if u is an ancestor of v in $\mathcal{T}$.
- (u, v) is a *backward edge* if u is a descendant of v in $\mathcal{T}$.
- (u, v) is a *forward-cross edge* if u and v do not have an ancestor/descendant relationship, and u is visited before v in the preorder of $\mathcal{T}$.
- (u, v) is a *backward-cross edge* if u and v do not have an ancestor/descendant relationship, and u is visited after v in the preorder of $\mathcal{T}$.

Example 9. *We show the non-tree edges for the DFS-Tree $\mathcal{T}$ of Figure 1.2(b) in Figure 5.1(a). The edge (v_3, v_{10}) is a forward edge since v_3 is the ancestor of v_{10}. (v_2, v_0) is a backward edge since v_2 is a descendant of v_0. (v_7, v_2) is a backward-cross edge since v_7 is visited after v_2, and these two vertices do not have an ancestor/descendant relationship. There is no forward-cross edge in $\mathcal{T}$.*

Lemma 14. *Given a graph G, a search spanning tree of G is a DFS-Tree if and only if there is no forward-cross edge in G under this tree. [60]*

Problem Definition. In this work, we study the problem of maintaining the DFS-Tree in dynamic directed graphs. Formally, given a directed graph G and a DFS-Tree $\mathcal{T}$ of G, we aim to efficiently compute a search spanning tree of G without any forward-cross edge when an edge is inserted into or deleted from G.

Note that we only focus on the edge operation since the vertex update can be implemented by several edge updates.



(a) Types of non-tree edges (b) Time intervals

Figure 5.1: The non-tree edges and time intervals of the DFS-Tree $\mathcal{T}$

5.3 A Flexible Framework

In this section, we first introduce several important concepts in checking the validity of the DFS-Tree, then present a framework for the DFS maintenance problem.

5.3.1 Efficient Validity Check

As mentioned in the previous sections, a valid DFS-Tree does not contain any forward-cross edge. Note that for a deleted tree edge (s, t), we can first connect the tree by appending t to the end of the children list of the visual root γ. Several new forward-cross edges may appear as a result. Therefore, handling the edge deletion can also be transformed to the problem of eliminating froward-cross edges. Given an edge (s, t), we can efficiently check the edge type using the *time interval* of each vertex instead of scanning the out-neighbors (resp. in-neighbors) of s (resp. t) in the graph.

Definition 11. (TIME INTERVAL) *Given a DFS-Tree $\mathcal{T}$ and a vertex u, the time interval of u is denoted as $\mathcal{I}_{\mathcal{T}}(u) = [x, y]$, where $x = \mathcal{I}_{\mathcal{T}}(u).left$ is the discovery timestamp of u in the DFS traversal, and $y = \mathcal{I}_{\mathcal{T}}(u).right$ is the finish timestamp of u when its out-neighbors have been examined completely in the DFS traversal. [21]*

Corollary 1. *There exists a one-to-one correspondence between the DFS-Tree and the time interval of each vertex.*

We omit the subscript $\mathcal{T}$ of $\mathcal{I}_{\mathcal{T}}$ when the context is clear. We give the time interval of every vertex in the DFS-Tree $\mathcal{T}$ of Figure 1.2(b) in Figure 5.1(b). In the rest of this chapter, we always use the terms *discovery* u and *finish* u to represent the timestamps $\mathcal{I}(u).left$ and $\mathcal{I}(u).right$, respectively. The term *visit* u means either discovery u or finish u.

Given a timestamp t ($1 \leq t \leq 2n$) and the DFS-Tree $\mathcal{T}$, we denote $\mathcal{T}[t]$ the visited vertex at timestamp t, i.e., $\mathcal{T}[t] = v$ iff $\mathcal{I}(v).left = t \vee \mathcal{I}(v).right = t$. For example, $\mathcal{T}[12] = v_6$ in Figure 5.1(b). Given a time interval $\mathcal{I}$, we denote $\mathcal{T}(\mathcal{I})$ (or $\mathcal{T}[\mathcal{I}.left, \mathcal{I}.right]$) the visited vertex set no earlier than the timestamp $\mathcal{I}.left$ and no later than the timestamp $\mathcal{I}.right$, i.e., $\mathcal{T}(\mathcal{I}) = \{v \in V | \mathcal{I}.left \leq \mathcal{I}(v).left \leq \mathcal{I}.right \vee \mathcal{I}.left \leq \mathcal{I}(v).right \leq \mathcal{I}.right\}$. For example, $\mathcal{T}[8, 15] = \{v_3, v_4, v_5, v_6, v_7, v_8\}$ in Figure 5.1(b).

We always use the term *search spanning tree* to denote the tree structure that may contain forward-cross edges in the rest. Based on the concept of the time interval, a non-tree edge (s, t) in a search spanning tree is

- a forward edge if $\mathcal{I}(t) \subset \mathcal{I}(s)$,
- a backward edge if $\mathcal{I}(s) \subset \mathcal{I}(t)$,
- a forward-cross edge if $\mathcal{I}(s).right < \mathcal{I}(t).left$, or

- a backward-cross edge if $\mathcal{I}(t).right < \mathcal{I}(s).left$.

To efficiently check the edge types, we maintain the time interval of each vertex in addition to the tree structure.

5.3.2 The Framework

To eliminate the forward-cross edges in a search spanning tree, the general idea of our framework is to locate a candidate part of the tree, then reconstruct the tree structure and recompute the time intervals of this part. We propose a one-pass strategy. By one-pass, we mean to simultaneously update the tree structure and the time interval of each visited vertex. In addition, we also propose optimizations that dynamically refine the range in Section 5.5. The pseudocode of the framework is given in Algorithm 11.

Algorithm 11: DFS-Maintenance Framework

Input: a directed graph G, a search spanning tree $\mathcal{T}$ with forward-cross edges
Output: the updated DFS-Tree

```
1 set a candidate time interval CI;
2 r ← LCA(T[CI.left], T[CI.right]);
3 ts ← CI.left;
4 ConstrainedDFS(r);
5 return the updated DFS-Tree T;
```

We locate a part of the DFS-Tree by setting a candidate time interval $\mathcal{CI}$ (line 1). Let r be the lowest common ancestor (LCA) of the first visited vertex $\mathcal{T}[\mathcal{CI}.left]$ and the last visited vertex $\mathcal{T}[\mathcal{CI}.right]$ (line 2). We perform a constrained DFS starting from r (line 4). Here, by constrained, we mean only to visit the vertices falling in the candidate time interval $\mathcal{CI}$ during the DFS.

The pseudocode of ConstrainedDFS is given in Algorithm 12. Note that the initial state of every vertex in the graph is unvisited in all the algorithms proposed

Algorithm 12: ConstrainedDFS(u)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left then
3      I(u).left ← ts, ts ← ts + 1;
4  foreach v ∈ N_out(u): I(v) ∩ CI ≠ ∅ ∧ I(v) ⊅ CI ∧ v is unvisited do
5      if I(v).left ≥ CI.left then
6          v' ← T[ts − 1];
7          if v' = u then
8              reassign v to the first element in C(u);
9          else
10             reassign v to the next element of v' in C(u);
11     ConstrainedDFS(v);
12 if I(u).right ≤ CI.right then
13     I(u).right ← ts, ts ← ts + 1;
```

in this study. The variables $\mathcal{CI}$ and ts are global variables. We update the discovery time of u if the original discovery time of u falls in the interval $\mathcal{CI}$ (lines 2–3). Then, we recursively discover the out-neighbor v of u if v falls in $\mathcal{CI}$ (lines 4–11). Note that in line 4, $\mathcal{I}(v) \cap \mathcal{CI} \neq \emptyset \wedge \mathcal{I}(v) \not\supset \mathcal{CI}$ is equivalent to $v \in \mathcal{T}(\mathcal{CI})$. In line 7, $v' = u$ means that v is the first discovered child after discovering u. Otherwise, v' is the last finished child of u before v. We recursively search the constrained neighbors of v by invoking ConstrainedDFS(v) in line 11. Finally, we update the finish time of u if its original finish time falls in the interval $\mathcal{CI}$ (lines 12–13).

5.3.3 Framework Analysis

Correctness Analysis. We prove the correctness of Algorithm 11 in this subsection. We use $\mathcal{T}$ to denote the input search spanning tree in Algorithm 11. We add the subscript *new* in the tree notation (shown as $\mathcal{T}_{new}$) when necessary for clarity to represent the updated search spanning tree returned by Algorithm 11.

Lemma 15. *Given any time interval $\mathcal{CI}$ in a search spanning tree $\mathcal{T}$, the lowest common ancestor of all vertices visited during $\mathcal{CI}$ is the same as that of the first and the last visited vertices during $\mathcal{CI}$, i.e.,* $\mathsf{LCA}(\mathcal{T}[\mathcal{CI}.left], \mathcal{T}[\mathcal{CI}.right]) = \mathsf{LCA}(\mathcal{T}(\mathcal{CI}))$.

Proof. Let $u = \mathcal{T}[\mathcal{CI}.left]$, $v = \mathcal{T}[\mathcal{CI}.right]$, w be the LCA of u and v, and w' be the LCA of $\mathcal{T}(\mathcal{CI})$. Since both w and w' are the ancestor of u and all the ancestors of u are on the simple path from u to the root, either w and w' have an ancestor-descendant relationship or they are the same vertex. To derive $w' = w$, we show w is not the ancestor of w' and vice versa.

First, w is not the ancestor of w'. Otherwise, the LCA of u and v would be w'. Second, we prove that w is not the descendant of w' by contradiction. Assume that w is the descendant of w'. There is a vertex u' in $\mathcal{T}(\mathcal{CI})$ such that u' is not the descendant of w. Then, either u' is the ancestor of w, or u' and w do not have an ancestor-descendant relationship. For the first case, we have $\mathcal{I}(u') \supset \mathcal{CI}$, and for the second case, we have $\mathcal{I}(u') \cap \mathcal{CI} = \emptyset$ since $\mathcal{I}(u') \cap \mathcal{I}(w) = \emptyset$ and $\mathcal{I}(w) \supseteq \mathcal{CI}$. Both two cases contradict that u' is in $\mathcal{T}(\mathcal{CI})$. □

Lemma 16. *For each vertex $v \in \mathcal{T}(\mathcal{CI})$ in Algorithm 11, either $v = r$ or there is a tree path from r to v such that each vertex in the path (excluding r) is in $\mathcal{T}(\mathcal{CI})$.*

Proof. We prove it by contradiction. Assume that there is a vertex $v \in \mathcal{T}(\mathcal{CI})$ and $v \neq r$, and the parent v' of v satisfies $v' \neq r$ and $v' \notin \mathcal{T}(\mathcal{CI})$. Then, either $\mathcal{I}(v') \cap \mathcal{CI} = \emptyset$ or $\mathcal{I}(v') \supset \mathcal{CI}$.

For the first case, we have $\mathcal{I}(v) \cap \mathcal{CI} = \emptyset$ because $\mathcal{I}(v) \subset \mathcal{I}(v')$. This contradicts that $v \in \mathcal{T}(\mathcal{CI})$. For the second case, r is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$ based on Lemma 15. However, if $\mathcal{I}(v') \supset \mathcal{CI}$, we hold that v' is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$, since v' is the common ancestor of $\mathcal{T}[\mathcal{CI}.left]$ and

$\mathcal{T}[\mathcal{CI}.right]$, and v' is a descendant of r. This contradicts that r is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$. □

Based on the above two lemmas, there is an invocation for ConstrainedDFS() for each vertex $v \in \mathcal{T}(\mathcal{CI})$, and we do not lose any vertex belonging to $\mathcal{T}(\mathcal{CI})$ in the constrained DFS. On the other hand, given the limitation in line 4 of Algorithm 12, we guarantee that if a vertex v is visited in the original tree during the interval $\mathcal{CI}$, v will also be visited in the updated tree during $\mathcal{CI}$. A formal lemma is given as follows.

Lemma 17. *For each vertex v in Algorithm 11, $\mathcal{I}_{\mathcal{T}_{new}}(v).left$ (resp. $\mathcal{I}_{\mathcal{T}_{new}}(v).right$) falls in $\mathcal{CI}$ if and only if $\mathcal{I}_{\mathcal{T}}(v).left$ (resp. $\mathcal{I}_{\mathcal{T}}(v).right$) falls in $\mathcal{CI}$, i.e., $\mathcal{T}(\mathcal{CI}) = \mathcal{T}_{new}(\mathcal{CI})$.*

Theorem 12. *Given an input search spanning tree $\mathcal{T}$ of G, Algorithm 11 computes a valid DFS-Tree if (i) there is no forward-cross edge (u, v) in $\mathcal{T}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI})$; and (ii) there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge \mathcal{I}(v).left > \mathcal{CI}.right$.*

Proof. The updated search spanning tree $\mathcal{T}_{new}$ computed by Algorithm 11 is a DFS-Tree if and only if there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$. Two vertices $u, v \in V$ can be considered in the following four cases: (i) $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$, (ii) $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$, (iii) $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$, and (iv) $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$.

For the case (i), Algorithm 12 guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$. Because if the edge (u, v) exists, v would be visited in the invocation of ConstrainedDFS(u).

For the case (ii), the condition (ii) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$. If $\mathcal{I}(v).left \leq \mathcal{CI}.right$, either $\mathcal{I}(v).right < \mathcal{CI}.left$ or $\mathcal{I}(v) \supset \mathcal{CI}$, since $v \notin$

$\mathcal{T}(\mathcal{CI})$. For both two cases, $\mathcal{I}(v).left \leq \mathcal{CI}.left$, considering that $\mathcal{I}(u).right \geq \mathcal{CI}.left$ since $u \in \mathcal{T}(\mathcal{CI})$, the edge (u, v) cannot be a forward-cross edge since $\mathcal{I}(u).right \geq \mathcal{I}(v).left$.

For the case (iii), the condition (i) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$. On the one hand, if $\mathcal{I}(u).left > \mathcal{CI}.right$ or $\mathcal{I}(u) \supset \mathcal{CI}$, we will have $\mathcal{I}(u).right \geq \mathcal{CI}.right \geq \mathcal{I}(v).left$. Therefore, the edge (u, v) cannot be a forward-cross edge. On the other hand, if $\mathcal{I}(u).right < \mathcal{CI}.left$, we assume that there is a forward-cross edge (u, v) in $\mathcal{T}_{new}$, then edge (u, v) in $\mathcal{T}$ may be a tree edge, forward edge, backward edge, forward-cross edge, or backward-cross edge. Firstly, edge (u, v) cannot be a tree edge or forward edge in $\mathcal{T}$. If so, we would have $\mathcal{I}(u) \supset \mathcal{I}(v)$, and this contradicts $\mathcal{I}(u).right < \mathcal{CI}.left$. Secondly, edge (u, v) cannot be a backward edge in $\mathcal{T}$. If so, it would be a backward edge in $\mathcal{T}_{new}$ too. Specifically, $u \in \mathcal{T}(r)$, and u is visited before $\mathcal{T}[\mathcal{CI}.left]$ during DFS. Following Algorithm 12, we visit vertices in the preorder of $\mathcal{T}$ before visiting $\mathcal{T}[\mathcal{CI}.left]$. Therefore, v is an ancestor of u in $\mathcal{T}_{new}$, and edge (u, v) is a backward edge. Thirdly, edge (u, v) cannot be a backward-cross edge in $\mathcal{T}$. If so, we would have $\mathcal{I}(v).right < \mathcal{I}(u).left < \mathcal{I}(u).right < \mathcal{CI}.left$, and this contradicts $v \in \mathcal{T}(\mathcal{CI})$. Considering aforementioned conditions, edge (u, v) must be a forward-cross edge in $\mathcal{T}$. We have that if edge (u, v) is not a forward-cross edge in $\mathcal{T}$, it will not be a forward-cross edge in $\mathcal{T}_{new}$ under the condition $\mathcal{I}(u).right < \mathcal{CI}.left$.

For the case (iv), the condition (i) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$. Since the time interval of a vertex not in $\mathcal{T}(\mathcal{CI})$ will not change during Algorithm 11, if $\mathcal{I}(u).right \geq \mathcal{I}(v).left$ in $\mathcal{T}$, we will have $\mathcal{I}(u).right \geq \mathcal{I}(v).left$ in $\mathcal{T}_{new}$ too. □

Theorem 13. *Given a directed graph G and a search spanning tree $\mathcal{T}$ of G, the*

running time of Algorithm 11 is bounded by $O(\sum_{u \in \mathcal{T}(\mathcal{CI}) \cup \{r\}} d_{out}(u))$*, where* r *is the LCA of all vertices in* $\mathcal{T}(\mathcal{CI})$*.*

5.4 Implementations

5.4.1 Edge Insertion

We give the basic implementation for the edge insertion in this subsection. The pseudocode is shown in Algorithm 13.

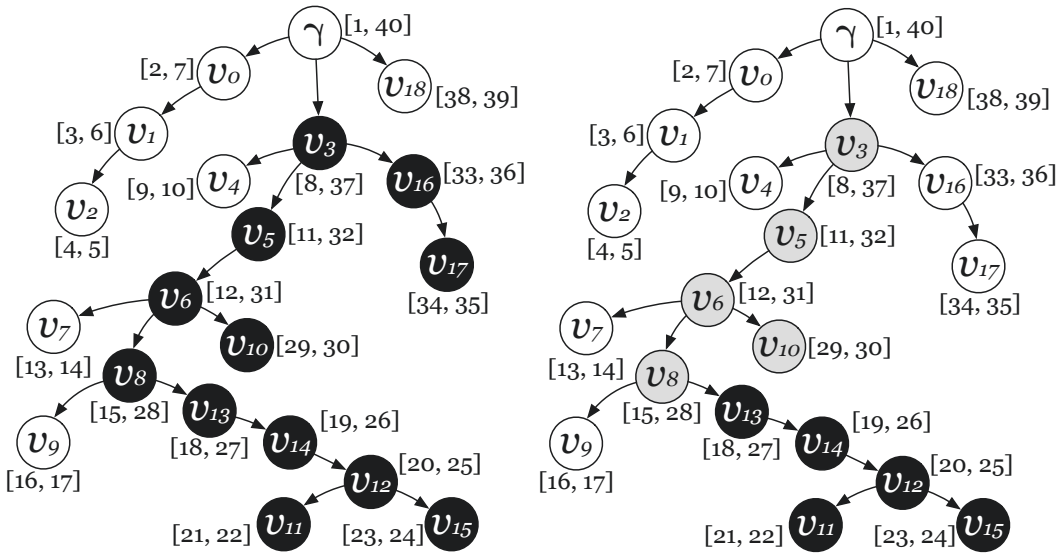
Algorithm 13: DFS-Insert

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t) in G
Output: the updated DFS-Tree

```
1 insert (s,t) into G;
2 if I(s).right > I(t).left then return T;
3 r ← LCA(s,t);
4 CI ← [I(s).right, I(r).right];
5 ts ← CI.left;
6 ConstrainedDFS(r);
7 return the updated DFS-Tree T;
```

We do nothing and return the original tree if (s, t) is not a forward-cross edge in line 2. We compute the LCA of s and t as the search root of ConstrainedDFS() in line 3 and set the candidate interval as $[\mathcal{I}(s).right, \mathcal{I}(r).right]$ in line 4. ConstrainedDFS() is invoked in line 6. A running example is given as follows.

Example 10. *Given the directed graph* G *in Figure 1.2(a) and its DFS-Tree* $\mathcal{T}$ *in Figure 1.2(b), the updated DFS-Tree* $\mathcal{T}_{new}$ *computed by Algorithm 13 for an inserted edge* (v_8, v_{13}) *is shown in Figure 5.2(a). The LCA of* v_8 *and* v_{13} *in the original DFS-Tree* $\mathcal{T}$ *is* v_3*.* v_3 *is also the LCA of all black vertices, which is supported by Lemma 15. According to Figure 5.1(b), the candidate time interval*



(a) Result of Algorithm 13 (b) Result of Algorithm 15

Figure 5.2: The updated DFS-Tree for the inserted edge (v_8, v_{13}) in the graph G

$\mathcal{CI}$ is assigned by $[18, 37]$. The vertices falling in the candidate time interval $\mathcal{CI}$ are marked in black. The time intervals of vertices which do not belong to $\mathcal{T}(\mathcal{CI})$ (white vertices) do not change in the algorithm.

Lemma 18. *In Algorithm 13, there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI})$ and $\mathcal{I}(v).left > \mathcal{CI}.right$.*

The correctness of Algorithm 13 is guaranteed by combining Theorem 12 and Lemma 18. We give the time complexity of Algorithm 13 as follows.

Theorem 14. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t), the running time of Algorithm 13 is bounded by $O(\sum_{u \in \mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(r).right]} d_{out}(u))$, where r is the LCA of s and t.*

5.4.2 Edge Deletion

We explain the basic implementation for the edge deletion in this subsection. The pseudocode is shown in Algorithm 14. The pseudocode is self-explanatory, so we omit the detailed description here.

Algorithm 14: DFS-Delete

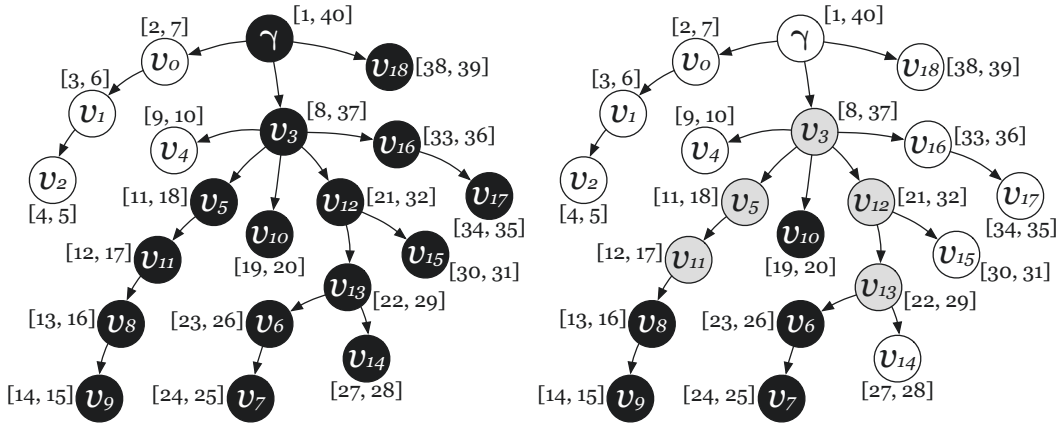
Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s, t) in G
Output: the updated DFS-Tree

```
1 delete (s, t) from G;
2 if (s, t) is not a tree edge then return T;
3 γ ← the virtual root of the DFS-Tree T;
4 CI ← [I(t).left, I(γ).right];
5 ts ← CI.left;
6 ConstrainedDFS(γ);
7 return the updated DFS-Tree T;
```

Example 11. *Given the directed graph G in Figure 1.2(a) and its DFS-Tree $\mathcal{T}$ in Figure 1.2(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 14 for a deleted edge (v_5, v_6) is presented in Figure 5.3(a). According to Figure 5.1(b), the candidate time interval $\mathcal{CI}$ is assigned by $[12, 40]$. Similar to Figure 5.2(a), the vertices falling in the candidate time interval $\mathcal{CI}$ are marked in black, and the time intervals of vertices which do not belong to $\mathcal{T}(\mathcal{CI})$ (white vertices) do not change in the algorithm.*

Lemma 19. *In Algorithm 14, there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI})$ and $\mathcal{I}(v).left > \mathcal{CI}.right$.*

The correctness of Algorithm 14 is guaranteed by combining Theorem 12 and Lemma 19. We give the time complexity of Algorithm 14 as follows.



(a) Result of Algorithm 14 (b) Result of Algorithm 17

Figure 5.3: The updated DFS-Tree for the deleted edge (v_5, v_6) in the graph G

Theorem 15. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s, t), the running time of Algorithm 14 is bounded by $O(\sum_{u \in \mathcal{T}[\mathcal{I}(t).left, \mathcal{I}(\gamma).right]} d_{out}(u))$, where γ is the virtual root of $\mathcal{T}$.*

5.5 The Improved Approaches

Drawbacks of Basic Solutions. Even though Algorithm 13 and Algorithm 14 correctly update the DFS-Tree, there is still much room for improvement. First, the key step in both Algorithm 13 and Algorithm 14 is to set a candidate time interval $\mathcal{CI}$. This interval can be very large, and many vertices will consequently be visited in ConstrainedDFS(). Second, all the out-neighbors of each discovered vertex will be scanned in ConstrainedDFS(). The total number of their out-neighbors can be very large especially in big graphs.

We propose several optimizations to improve the algorithmic efficiency. In response to the drawbacks of the basic solutions, we first adopt a strategy which dynamically refines the candidate time interval $\mathcal{CI}$. Specifically, we continuously adjust the candidate interval in the process of the algorithm, and the candidate

interval is theoretically guaranteed to be never larger than that of the basic solutions. In addition, we design a hybrid approach to perform the constrained DFS. By hybrid, we mean searching vertices by combining the graph search and the tree search. This avoids scanning all the out-neighbors of the visited vertices in the basic solutions. We introduce the details for edge insertion and deletion in Subsection 5.5.1 and Subsection 5.5.2, respectively.

5.5.1 Edge Insertion

Tightening Candidate Interval

We first give some key observations for tightening the candidate interval in the edge insertion algorithm.

Lemma 20. *Given an inserted edge (s,t), let C be the set of new descendants of s in Algorithm 13, i.e. $C = (\mathcal{T}_{new}(s) \setminus \{s\}) \cap \mathcal{T}(\mathcal{CI})$, we have $C = \mathcal{T}_{new}(t)$.*

Lemma 21. *Given an inserted edge (s,t), let w be the vertex in $\mathcal{T}_{new}(t)$ with the largest old right interval value, i.e., $w = \arg\max_{v \in \mathcal{T}_{new}(t)} \mathcal{I}(v).right$. There is no edge (u,v) such that $u \in \mathcal{T}_{new}(t)$ and $\mathcal{I}(v).left > \mathcal{I}(w).right$.*

Example 12. *Consider the graph G in Figure 1.2(a), the original DFS-Tree $\mathcal{T}$ in Figure 1.2(b), and the updated DFS-Tree $\mathcal{T}_{new}$ in Figure 5.2(a) for an inserted edge (v_8, v_{13}). The new descendants of v_8 is $C = \mathcal{T}_{new}(v_{13}) = \{v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$. v_{12} has the largest old right interval value 32 as shown in Figure 5.1(b). The set of vertices with a left time interval larger than 32 is $\{v_{16}, v_{17}, v_{18}\}$, and there is no edge from the vertex in $\mathcal{T}_{new}(v_{13})$ to the vertex in this set.*

Based on Lemma 21, we can use $[\mathcal{I}(s).right, \mathcal{I}(w).right]$ as the new candidate interval and guarantee the algorithmic correctness. To derive $\mathcal{I}(w).right$,

we dynamically update the candidate interval in the process of the edge insertion algorithm. Specifically, given an inserted edge (s,t), we can initialize the candidate interval as $\mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(t).right]$ since t must be in $\mathcal{T}_{new}(t)$. Every time a new vertex v is assigned as a descendant of t, we update $\mathcal{CI}.right$ to $\mathcal{I}(v).right$ if $\mathcal{I}(v).right > \mathcal{CI}.right$. The candidate interval stops updating once all the descendants of t are collected in the updated DFS-Tree $\mathcal{T}_{new}$. The following lemma shows that the search space of this new candidate interval is at most the same as that of Algorithm 13 in the worst case.

Lemma 22. *Given an inserted edge (s,t), let w be the vertex defined in Lemma 21. $\mathcal{I}(w).right \leq \mathcal{I}(r).right$, where r is the LCA of s and t.*

From Graph Search to Tree Search

We further improve the algorithmic efficiency by replacing a part of the graph search with tree search.

Lemma 23. *Given an inserted edge (s,t) in Algorithm 13, when finishing the visit of vertex t, i.e., $u = t$ in line 12 of Algorithm 12, there is no forward-cross edge in the graph.*

Based on Lemma 23, at the finish moment of vertex t, the tree is a valid DFS-Tree, and what we need is to update the time intervals of the remaining vertices. Therefore, we can simply search the tree instead of the graph to finish the update. We give an example to explain Lemma 23.

Example 13. *We continue to use Figure 5.2(a) for the inserted edge (v_8, v_{13}). When all the descendants of v_{13} are collected (the timestamp is 27), there is no forward-cross edge. At this moment, all remaining vertices still in the candidate interval are $\{v_3, v_5, v_6, v_8, v_{10}\}$. Compared with Figure 5.1(b), the tree structures*

of these remaining vertices do not change, and we only update their time intervals.

The Overall Algorithm

By combining these two optimization techniques mentioned above, we detail our final algorithm for edge insertion in this subsection. The pseudocode is presented in Algorithm 15.

Algorithm 15: DFS-Insert*

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t) in G
Output: the updated DFS-Tree

```
1 insert (s, t) into G;
2 if I(s).right > I(t).left then return T;
3 r ← LCA(s, t);
4 CI ← [I(s).right, I(t).right];
5 reassign t to the last element in C(s);
6 ts ← CI.left;
7 InsHybrid-DFS(r, t);
8 return the updated DFS-Tree T;
```

In line 4 of Algorithm 15, we initialize the candidate interval as $[\mathcal{I}(s).right, \mathcal{I}(t).right]$. We invoke the subroutine named InsHybrid-DFS instead of ConstrainedDFS, and the search root is still the LCA of s and t.

The pseudocode of InsHybrid-DFS is presented in Algorithm 16. Similar to Algorithm 12, we update the left time interval of u in lines 2–3 and the right time interval of u in lines 13–14 if necessary. We check whether u is in $\mathcal{T}(t)$ in line 4. If yes, we perform the graph search in lines 5–9. The constraint (line 6) in the graph search is set to be the same as that in line 4 of Algorithm 12. In line 8, we update the right bound of the candidate interval if necessary. If u is not in $\mathcal{T}(t)$, we perform the tree search in lines 11–12. Line 11 guarantees that

Algorithm 16: InsHybrid-DFS(u, t)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left then
3      I(v).left ← ts, ts ← ts + 1;
4  if u ∈ T(t) then
       // Graph Search
5      C(u) = ∅;
6      foreach v ∈ N_out(u): I(v) ∩ CI ≠ ∅ ∧ I(v) ⊅ CI ∧ v is unvisited do
7          reassign v to the last element in C(u);
8          CI.right ← max(CI.right, I(v).right);
9          InsHybrid-DFS(v, t);
10 else
       // Tree Search
11     foreach v ∈ C(u): I(v) ∩ CI ≠ ∅ do
12         InsHybrid-DFS(v, t);
13 if I(u).right ≤ CI.right then
14     I(u).right ← ts, ts ← ts + 1;
```

only the vertices falling in the candidate interval are visited. We give a running example of Algorithm 15 as follows.

Example 14. *Given directed graph G in Figure 1.2(a) and its DFS-Tree $\mathcal{T}$ in Figure 1.2(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 15 for the inserted edge (v_8, v_{13}) is shown in Figure 5.2(b). For each visited vertex, if a graph search is performed, the vertex is marked in black, or if a tree search is performed, the vertex is marked in gray. Consider the tree in Figure 5.2(a) derived by Algorithm 13. The candidate interval is $[18, 37]$, and there are 12 (black) vertices visited in the algorithm, whereas in Figure 5.2(b), the final stable candidate interval is $[18, 32]$. Only 10 (black and gray) vertices are visited in the algorithm, and we only visit the tree children instead of the graph out-neighbors of the 5 (gray) vertices inside.*

Theorem 16. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted*

edge (s,t)*, the running time of Algorithm 15 is* $O(|\mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(w).right]| + \sum_{u \in \mathcal{T}_{new}(t)} d_{out}(u))$*, where* w *is the vertex defined in Lemma 21.*

The left part of the complexity is the number of visited vertices in the candidate interval $[\mathcal{I}(s).right, \mathcal{I}(w).right]$, and the right part is the number of out-neighbors of all vertices in $\mathcal{T}_{new}(t)$. In brief, the left and right parts represent the tree search and graph search in Algorithm 15, respectively. Based on Lemma 22, the overall time complexity of Algorithm 15 is not larger than that of Algorithm 13.

5.5.2 Edge Deletion

In this subsection, we propose the optimizations for updating the DFS-Tree when a tree edge is deleted.

A Similar Optimization to Edge Insertion

Recall that given a deleted tree edge (s,t) in Algorithm 14, we straightforwardly append the subtree $\mathcal{T}(t)$ to the end of the children list of the virtual root γ. This essentially transforms the edge deletion problem to a forward-cross edge repairing problem. This method is inefficient due to the unpredictable generated forward-cross edges. A wide candidate interval (from $\mathcal{I}(t).left$ to $\mathcal{I}(\gamma).right$) is set to cover the possibly influenced vertices.

In order to tighten the candidate interval, one method is to adopt a similar optimization in Subsection 5.5.1, which dynamically updates the candidate interval. Specifically, we start the constrained DFS from the timestamp $\mathcal{I}(t).left$, i.e., $\mathcal{CI}.left = \mathcal{I}(t).left$, which is the same as that in Algorithm 14, and do not limit the right bound of the candidate interval initially. The DFS terminates immediately once all the vertices in $\mathcal{T}(t)$ are visited.

However, the improvement of this optimization is limited since the new ancestors of vertices in $\mathcal{T}(t)$ are unpredictable, and we need to scan all the out-neighbors of every vertex visited in the DFS.

Avoiding Unpredictable Graph Search

To improve the efficiency of the edge deletion, we adopt a new method that iteratively appends the vertices in $\mathcal{T}(t)$ to the tree as early as possible. For the vertices not in $\mathcal{T}(t)$, the tree search is performed and the time interval is updated in the tree. After visiting a special kind of vertex u which can be the parent of vertices in $\mathcal{T}(t)$, we start the graph search and collect a set of vertices in $\mathcal{T}(t)$ as the descendants of u. The algorithm terminates once all vertices in $\mathcal{T}(t)$ are appended to the tree. This new method further tightens the candidate interval, and the graph search is only performed on the vertices in $\mathcal{T}(t)$. We introduce the details below and start by giving several definitions for ease of understanding.

Definition 12. (Local Earliest Visit Time) *Given a vertex $u \in \mathcal{T}(t)$ and a vertex $p \in N_{in}(u) \cap (V \setminus \mathcal{T}(t))$, the local earliest visit time of u regarding p, denoted by $\mathcal{EV}_p(u)$, is the smallest integer i such that $i > \mathcal{I}(p).left$ and $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$.*

Definition 13. (Earliest Visit Time and Potential Parent) *Given a vertex $u \in \mathcal{T}(t)$, the earliest visit time of u, denoted by $\mathcal{EV}(u)$, is the smallest local earliest visit time of u, i.e., $\mathcal{EV}(u) = \min_{p \in N_{in}(u) \cap (V \setminus \mathcal{T}(t))} \mathcal{EV}_p(u)$. The corresponding vertex p of the minimum $\mathcal{EV}_p(u)$ is the potential parent of u, denoted by $\mathcal{PP}(u)$.*

Note that due to the existence of virtual root γ, every vertex in $\mathcal{T}(t)$ has a potential parent and earliest visit time. The condition $i > \mathcal{I}(p).left$ guarantees that we visit u as a child of p. Given for any other child $v \in \mathcal{C}(p)$, $\mathcal{I}(u) \cap \mathcal{I}(v) = \emptyset$.

The condition $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$ guarantees that i is the earliest position to append u to $\mathcal{C}(p)$ after the timestamp $\mathcal{I}(t).left$. We give an example as follows.

Example 15. *Reconsider the case of deleting the edge (v_5, v_6) in the DFS-Tree shown in Figure 5.1. The vertex set in the subtree $\mathcal{T}(v_6)$ is $\{v_6, v_7, v_8, v_9, v_{10}\}$. The vertex v_8 has three in-neighbors $\{v_{11}, v_{14}, \gamma\}$ that are not in $\mathcal{T}(v_6)$. The local earliest visit time $\mathcal{EV}_{v_{11}}(v_8)$ is 23, $\mathcal{EV}_{v_{14}}(v_8)$ is 28, and $\mathcal{EV}_{\gamma}(v_8)$ is 38. Therefore, the potential parent of v_8 is v_{11} and its earliest visit time is 23. In other words, the earliest position to append v_8 after the timestamp $\mathcal{I}(v_6).left = 12$ in the tree is the child of v_{11}.*

The vertex v_{10} has two in-neighbors $\{v_3, \gamma\}$ that are not in $\mathcal{T}(v_6)$. The local earliest visit time $\mathcal{EV}_{v_3}(v_{10})$ is 25, and $\mathcal{EV}_{\gamma}(v_{10})$ is 38. Thus, the potential parent of vertex v_{10} is v_3 and its earliest visit time is 25. In this case, v_{10} can be appended to $\mathcal{C}(v_3)$ after v_5, since v_5 has been visited before $\mathcal{I}(v_6).left = 12$, and $\mathcal{EV}(v_{10})$ should be larger than $\mathcal{I}(v_5).right = 24$.

Definition 14. (TRIGGER) *Given a vertex set $C \subseteq \mathcal{T}(t)$, the trigger of C is a vertex u with the smallest earliest visit time in C, i.e., $u = \arg\min_{v \in C} \mathcal{EV}(v)$.*

Note that it is possible that there are multiple triggers. We break the tie by randomly selecting one in the algorithm. An example to explain the above definitions is given below.

Example 16. *Given the set $C = \mathcal{T}(v_6)$, the trigger is v_8. Its earliest visit time and potential parent are 23 and v_{11}, respectively. For the other vertices in $\mathcal{T}(v_6)$, we have $\mathcal{PP}(v_{10}) = v_3, \mathcal{EV}(v_{10}) = 25$ and $\mathcal{PP}(v_6) = v_{13}, \mathcal{EV}(v_6) = 27$. The virtual root γ is the only one in-neighbor not in $\mathcal{T}(v_6)$ for the vertices v_7 and v_9. We have $\mathcal{PP}(v_7) = \mathcal{PP}(v_9) = \gamma$ and $\mathcal{EV}(v_7) = \mathcal{EV}(v_9) = 38$.*

We explain the rationale behind these definitions. Firstly, we only allow the vertices in $\mathcal{T}(t)$ to append to the tree after the timestamp $\mathcal{I}(t).left$. This guarantees that there is no new forward-cross edge (u, v) such that $u \in \mathcal{T}(t)$ and v is visited before $\mathcal{I}(t).left$. Secondly, we find the earliest potential parent for each vertex in $\mathcal{T}(t)$, and this avoids the appearance of forward-cross edges due to the backward movement of vertices in $\mathcal{T}(t)$. We give examples as follows.

Example 17. *Following the previous example, consider vertex v_7. The only in-neighbor of v_7 not in $\mathcal{T}(v_6)$ is γ. According to Definition 12 and Definition 13, the earliest position to append v_7 is after v_3. If we omit the condition $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$ in Definition 12 and append v_7 as the first child of γ, this will generate two new forward-cross edges (v_7, v_2) and (v_7, v_5). Therefore, we only append vertices after the timestamp $\mathcal{I}(t).left$.*

On the other hand, consider vertex v_8. There are three in-neighbors not in $\mathcal{T}(v_6)$, v_{11}, v_{14} and γ. v_{11} is the potential parent. If we append v_8 to $\mathcal{C}(v_{14})$, it will generate a new forward-cross edge (v_{11}, v_8).

Based on Definition 12, Definition 13 and Definition 14, we compute the first trigger u and its potential parent p by scanning the in-neighbors of $\mathcal{T}(t)$. We compute the earliest visit time of u and append u to the children list of p accordingly. We perform the tree search and update the time intervals starting from the timestamp $\mathcal{I}(t).left$. Once visiting the trigger, we perform the graph search and collect its all descendants. Note that for each trigger, only the vertices in $\mathcal{T}(t)$ are appended to its new subtree. Once all descendants of the trigger are collected, we locate the next trigger and repeat this step. We terminate the procedure once all vertices in $\mathcal{T}(t)$ are visited. The following subsection gives a detailed algorithm and corresponding analysis.

The Overall Algorithm

We give our final algorithm for edge deletion in this subsection. The pseudocode is presented in Algorithm 17.

Algorithm 17: DFS-Delete*

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s,t) in G
Output: the updated DFS-Tree

```
1 delete (s,t) from G;
2 if (s,t) is a non-tree edge then return T;
3 delete t from C(s);
4 V_c ← T(t);
5 CI ← [I(t).left, +∞];
6 ts ← CI.left;
7 LocateNextTrigger();
8 while V_c is not empty do DelHybrid-DFS(r);
9 return the updated DFS-Tree T;
```

We mark all candidate vertices as V_c in line 4. The left bound of the candidate interval is initialized as $\mathcal{I}(t).left$ in line 5, and the search will start from this timestamp. We do not set the right bound since the algorithm will terminate once all vertices in V_c are visited. We compute the trigger in line 7 before performing the search and continuously invoke the DelHybrid-DFS() until V_c is empty. The pseudocodes of LocateNextTrigger() and DelHybrid-DFS() are given in Algorithm 18 and Algorithm 19, respectively.

In Algorithm 18, we first compute the trigger in V_c in line 1. Based on Definition 12 and Definition 13, we reassign the trigger to its earliest visit position in the children list of its potential parent from line 2 to line 7. Similar to the previous algorithms, r is the search entrance for the DelHybrid-DFS() and is initialized by the LCA of s and p (line 8). Note that unlike the previous methods, we dynamically update r, since we compute a new trigger in each invocation of Algorithm 18. The search entrance r must update to cover the new trigger and

Algorithm 18: LocateNextTrigger()

```
1 trigger ← get trigger in V_c based on Definition 14;
2 p ← PP(trigger);
3 v ← T[EV(trigger) − 1];
4 if v = p then
5 |   reassign trigger to the first element in C(p);
6 else
7 |   reassign trigger to the next element of v in C(p);
8 if r is undefined then r ← LCA(s, p);
9 else r ← LCA(r, p);
```

guarantee the correctness (line 9).

In Algorithm 19, we first update the left time interval of u if u falls in the candidate interval or $u \in V_c$ in lines 2–3. Note that even though $\mathcal{CI}.left$ is initialized by $\mathcal{I}(t).left$ and $\mathcal{I}(u).left \geq \mathcal{CI}.left$ for all $u \in V_c$ holds at the beginning, we update $\mathcal{CI}.left$ in the algorithm. Thus, condition $u \in V_c$ is necessary. The graph search is performed only for the vertices in V_c and is shown in lines 4–9. In line 7, we limit u's out-neighbors to those belonging to V_c. This is because we postpone visiting u and an out-neighbor $v \notin V_c$ must be visited before. Otherwise, the edge (u, v) would be a forward-cross edge. The tree search is shown in lines 11–17. In line 13, if v is a trigger, a set of vertices has been appended to the subtree of v. If V_c is empty, we actually finish updating the DFS-Tree, and the algorithm can terminate immediately. By setting $\mathcal{CI}.right$ as $ts - 1$, no vertex will be further visited. When V_c is not empty, we invoke LocateNextTrigger() to find the next earliest position and complete the DFS-Tree. In line 18, all visited vertices satisfy this condition until we set $\mathcal{CI}.right$ as $ts - 1$. We update the right time interval of u in line 19. Note that line 20 is an important step to avoid visiting the same vertices repeatedly and to guarantee the algorithmic correctness. Specifically, recall that r updates in Algorithm 18. Let r_{old} be the old one, and r_{new} be the updated value. Assume that r_{new} is an ancestor of

Algorithm 19: DelHybrid-DFS(u)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left ∨ u ∈ V_c then
3      I(u).left ← ts, ts ← ts + 1;
4  if u ∈ V_c then
       // Graph Search
5      V_c    V_c \ {u};
6      C(u) = ∅;
7      foreach v ∈ N_out(u): v ∈ V_c ∧ v is unvisited do
8          reassign v to the last element in C(u);
9          DelHybrid-DFS(v);
10 else
       // Tree Search
11     foreach v ∈ C(u): I(v) ∩ CI ≠ ∅ ∨ v ∈ V_c do
12         DelHybrid-DFS(v);
13         if v = trigger then
14             if V_c is empty then
15                 CI.right ← ts − 1;
16             else
17                 LocateNextTrigger();
18 if I(u).right ≤ CI.right then
19     I(u).right ← ts, ts    ts + 1;
20     CI.left ← ts;
```

r_{old}. In the invocation of DelHybrid-DFS(r_{new}), we need to avoid searching the subtree $\mathcal{T}(r_{old})$ since the new intervals of the vertices inside have been allocated in the invocation of DelHybrid-DFS(r_{old}). In this case, $\mathcal{CI}.left$ has been updated to $\mathcal{I}(r_{old}).right + 1$. We give a running example for Algorithm 17 as follows.

Example 18. *Given the directed graph G in Figure 1.2(a) and its DFS-Tree $\mathcal{T}$ in Figure 1.2(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 17 for the deleted edge (v_5, v_6) is shown in Figure 5.3(b). Similar to the example of edge insertion, each visited vertex is marked in black if the graph search has been performed, and each visited vertex is marked in gray if a tree search has been performed. The visited vertex set is the union of the black and gray vertices.*

Figure 5.3(a) derived by Algorithm 14 visits 16 (black) vertices. Figure 5.3(b) only visits 10 vertices. The initial left bound of the candidate interval in Algorithm 17 is $\mathcal{I}(v_6).left = 12$, and the right bound when terminating the algorithm is 26. Of these 10 vertices, we visit the tree children instead of the graph out-neighbors for the 5 vertices.

Theorem 17. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s, t), the running time of Algorithm 17 is bounded by $O(|C| + \sum_{u \in \mathcal{T}(t)} d(u))$, where $|C|$ is the number of vertices visited in the candidate interval and $d(u) = d_{in}(u) + d_{out}(u)$.*

5.5.3 Batch Update

Our proposed algorithms can be easily extended to handle the batch update. Without loss of generality, a batch update can be considered as a group of edge insertions followed by a group of edge deletions.

Edge Insertion. We assume that all the inserted edges are forward-cross edges. For other kinds of edges, we always first add them into the

graph since they will never break the validity of the DFS-Tree. We denote the set of inserted forward-cross edges as $B_+ = \{(s_1, t_1), (s_2, t_2), ..., (s_b, t_b)\}$. We set $r = \mathsf{LCA}(s_1, t_1, s_2, t_2, ..., s_b, t_b)$ and the candidate interval $\mathcal{CI} = \bigcup_{1 \leq i \leq b} [\mathcal{I}(s_i).right, \mathcal{I}(\mathsf{LCA}(s_i, t_i)).right]$ in the batch update version of Algorithm 13. The optimization of tightening the candidate interval discussed in Subsection 5.5.1 can also be applied in the batch insertion. Considering Algorithm 15, besides the modification of r in Algorithm 13, we set $\mathcal{CI} = \bigcup_{1 \leq i \leq b} [\mathcal{I}(s_i).right, \mathcal{I}(t_i).right]$ and perform the graph search in Algorithm 16 when $\exists 1 \leq i \leq b : u \in \mathcal{T}(t_i)$.

Edge Deletion. Similarly, we assume that all the deleted edges are tree edges. We denote the set of deleted tree edges as $B_- = \{(s_1, t_1), (s_2, t_2), ..., (s_b, t_b)\}$. We set $\mathcal{CI} = [\min_{1 \leq i \leq b} \mathcal{I}(t_i).left, \mathcal{I}(\gamma).right]$ in the batch update version of Algorithm 14. The optimizations discussed in Subsection 5.5.2 can also be applied in the batch deletion. Considering Algorithm 17, besides the aforementioned modifications, we set $V_c = \bigcup_{1 \leq i \leq b} \mathcal{T}(t_i)$.

5.6 Experiments

We conducted extensive experiments to evaluate the performance of our proposed algorithms. The algorithms evaluated in the experiments are summarized as follows:

- ReDFS-Insert and ReDFS-Delete: The naive methods are discussed in Section 5.3. That is, Algorithm 10 is run if a forward-cross edge is inserted or a tree edge is deleted.
- DFS-Insert and DFS-Delete: Algorithm 13 and Algorithm 14.
- DFS-Insert$^+$ and DFS-Delete$^+$: DFS-Insert$^+$ represents Algorithm 13 with

the optimization of tightening the candidate interval discussed in Subsection 5.5.1. DFS-Delete$^+$ represents Algorithm 14 with a similar optimization discussed in Subsection 5.5.2.

- DFS-Insert* and DFS-Delete*: Algorithm 15 and Algorithm 17.

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with Intel Xeon 3.1GHz CPU and 128GB 1600MHz ECC DDR3-RAM.

Datasets. We conducted experiments on twelve publicly-available real-world graphs and divided them into two groups according to the data size.

Group one consists of six graphs with a relatively small size. Group two consists of six big graphs. The detailed statistics of these datasets are summarized in Table 5.1. All networks and corresponding detailed descriptions can be found in SNAP[1], KONECT[2], and WebGraph[3].

In group one, CiteSeer is a citation network extracted from the CiteSeer digital library. Web-Stanford is the web graph of *stanford.edu*. YahooAd is a lexical network containing adjacency word data from phrases in Yahoo advertisements. ProsperLoan is an interaction network containing information on loans between users of the *prosper.com* website. Wiki-Talk is a communication network containing users' discussions on Wikipedia. Web-Google is a web graph from a Google programming contest.

Group two consists of six big graphs. Flickr, LiveJournal and Twitter are online social networks. StackOverflow is an interaction network containing users' comments, questions, and answers on Stack Overflow. UK-2002 is a web graph obtained from a 2002 crawl of the *.uk* domain performed by UbiCrawler. Wiki-

[1] `http://snap.stanford.edu/data/index.html`
[2] `http://konect.cc`
[3] `http://law.di.unimi.it/datasets.php`

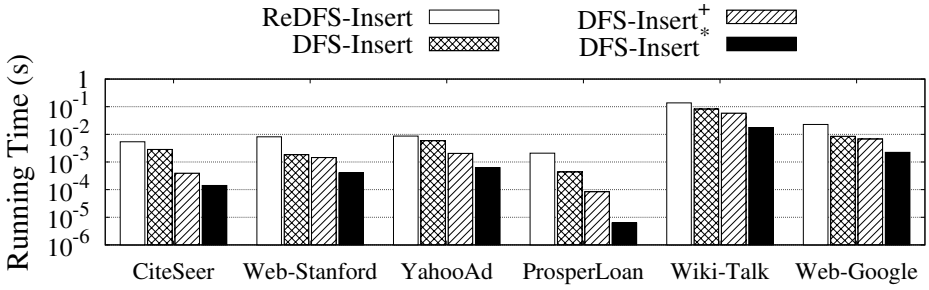
Table 5.1: Network statistics

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
CiteSeer	384,413	1,751,463	4.56
Web-Stanford	281,903	2,312,497	8.20
YahooAd	653,260	2,931,708	4.49
ProsperLoan	89,269	3,394,979	38.03
Wiki-Talk	2,394,385	5,021,410	2.10
Web-Google	875,713	5,105,039	5.83
Flickr	2,302,925	33,140,017	14.39
StackOverflow	2,601,977	63,497,050	24.40
LiveJournal	4,847,571	68,993,773	14.23
UK-2002	18,520,486	298,113,762	16.10
Wiki-Link	12,150,976	378,142,420	31.12
Twitter	41,652,230	1,468,365,182	35.25

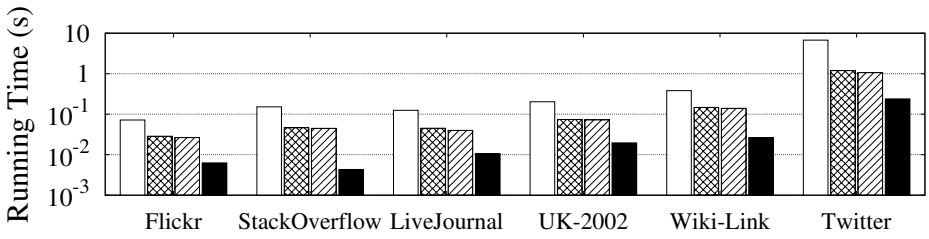
Link is a hyperlink network containing the wikilinks of the Wikipedia in the English language.

Input Preparation. In order to test the performance of our proposed algorithms, we randomly selected 10000 distinct existing edges in the graph for each test case. For edge deletion, we deleted these 10000 edges from the original graphs one by one. For edge insertion, we started from the graphs after removing these 10000 edges and corresponding DFS-Tree. We inserted them into the graphs one by one. For each algorithm, we calculated the average running time for each edge insertion (resp. deletion).

The rest of this section is summarized as follows. Subsection 5.6.1 provides the overall processing time of all the four algorithms. Then in Subsection 5.6.2, we evaluate the effectiveness of our proposed optimizations. Subsection 5.6.3 evaluates the scalability. Subsection 5.6.4 reports the memory usage.



(a) Small graphs



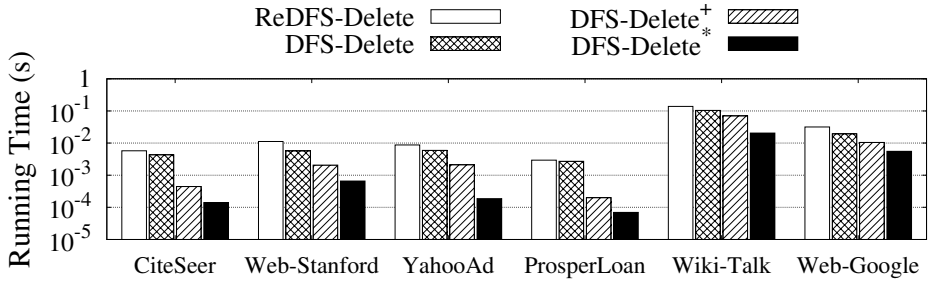
(b) Big graphs

Figure 5.4: Running time for edge insertion

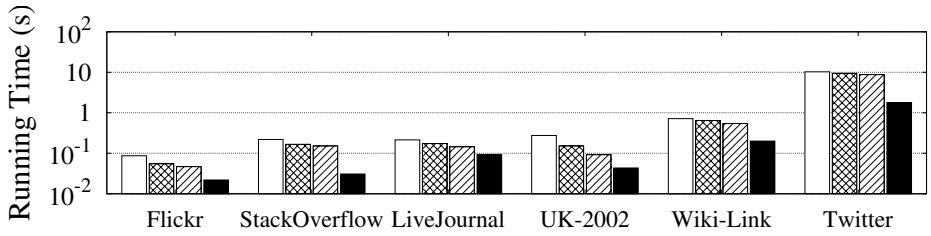
5.6.1 Overall Efficiency

Edge Insertions. Figure 5.4 shows the running time of all the four insertion algorithms on all datasets. Based on our proposed framework, DFS-Insert is more efficient than naively computing the DFS-Tree from scratch (ReDFS-Insert), and our proposed optimization techniques further improve the efficiency of DFS-Insert. For instance, the running time of DFS-Insert* on the smallest graph CiteSeer is $0.14ms$. Meanwhile, the running times of DFS-Insert$^+$, DFS-Insert and ReDFS-Insert are $0.39ms$, $2.84ms$ and $5.40ms$, respectively. On the largest graph Twitter with over 1 billion edges, DFS-Insert* takes around $0.24s$, while DFS-Insert$^+$, DFS-Insert and ReDFS-Insert take about $1.06s$, $1.19s$ and $6.75s$, respectively.

Edge Deletions. Figure 5.5 shows the running time of all the four deletion algorithms on all datasets. Similar to Figure 5.4, we witness a large improve-



(a) Small graphs

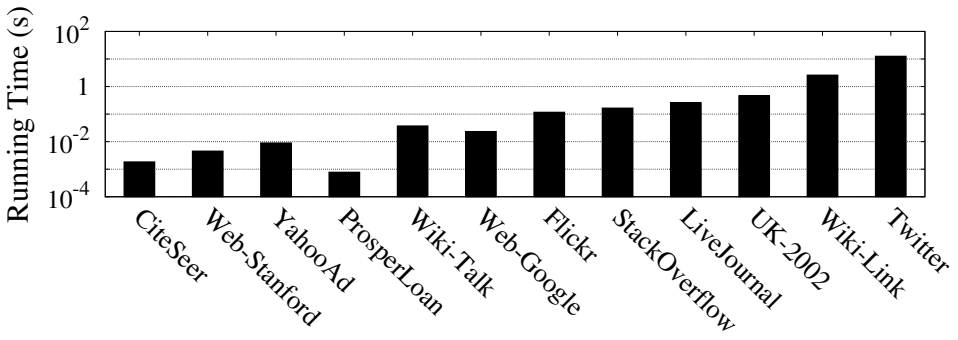


(b) Big graphs

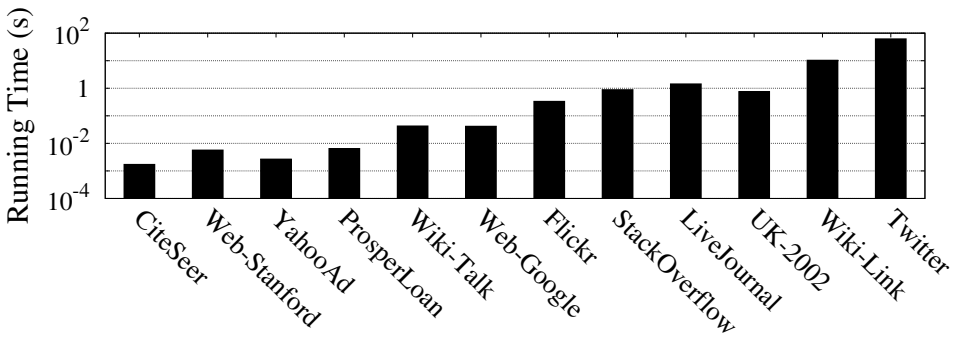
Figure 5.5: Running time for edge deletion

ment from the baseline algorithm to our final algorithm. For example, on the smallest graph CiteSeer, DFS-Delete* takes only $0.14ms$, while DFS-Delete$^+$, DFS-Delete and ReDFS-Delete take $0.44ms$, $4.33ms$ and $5.75ms$, respectively. On the largest graph Twitter, DFS-Delete* takes approximately $1.79s$, while the algorithms DFS-Delete$^+$, DFS-Delete and ReDFS-Delete take around $8.78s$, $9.45s$ and $10.27s$, respectively.

Update Distribution. Recall that it only takes constant time if the inserted edge is not a forward-cross edge or the deleted edge is not a tree edge. To clearly show the performance of our final algorithm, we report the percentages of the forward-cross edge insertion (DFS-Insert*) and the tree edge deletion (DFS-Delete*) in Table 5.2 and Table 5.3, respectively. We also report the average running times of DFS-Insert* and DFS-Delete* for these updates in Figure 5.6(a) and Figure 5.6(b), respectively.



(a) Edge insertions



(b) Edge deletions

Figure 5.6: Running time for tree updates

Table 5.2: Percentage of forward-cross edge insertions

CiteSeer	Web-Stanford	YahooAd	ProsperLoan
7.64%	8.85%	6.95%	0.78%
Wiki-Talk	Web-Google	Flickr	StackOverflow
47.41%	9.55%	5.36%	2.61%
LiveJournal	UK-2002	Wiki-Link	Twitter
4.07%	4.24%	1.01%	1.89%

Table 5.3: Percentage of tree edge deletions

CiteSeer	Web-Stanford	YahooAd	ProsperLoan
8.14%	11.5%	6.95%	1.07%
Wiki-Talk	Web-Google	Flickr	StackOverflow
47.73%	13.3%	6.52%	3.45%
LiveJournal	UK-2002	Wiki-Link	Twitter
6.46%	5.74%	1.92%	2.88%

5.6.2 Effectiveness of Optimizations

We further evaluate the effectiveness of our optimizations. We count the number of ConstrainedDFS() invocations in Algorithm 13 (resp. Algorithm 14) and denote it as c_n. This value represents the number of vertices performing graph search in Algorithm 13 (resp. Algorithm 14). We also count the numbers of vertices by graph search and tree search in our final algorithm, which are denoted by c_g and c_t, respectively. Specifically, for each invocation of InsHybrid-DFS() in DFS-Insert*(), we increase c_g by one if it is true in line 4 of InsHybrid-DFS(). Otherwise, we increase c_t by one. Similarly, for edge deletion, we increase c_g by one if it is true in line 4 of DelHybrid-DFS(). Otherwise, we increase c_t by one.

We report the value of c_t/c_n and c_g/c_n for both edge insertion and deletion on all datasets in Figure 5.7. The black part represents the ratio of visited vertices

by the tree search c_t/c_n, and the gray part represents the ratio of visited vertices by the graph search c_g/c_n.

The results show that in our final algorithms, the total number of visited vertices is reduced on all datasets, and there is a large proportion of tree search inside. Compared with the basic algorithms for both operations, we only need a very small number of graph searches. For example, the largest percentages of graph search (i.e., c_g/c_n) for edge insertion and edge deletion are about 17% and 23% on Twitter and LiveJournal, respectively. The lowest percentage is 0.000002% on YahooAd for both edge insertion and deletion.

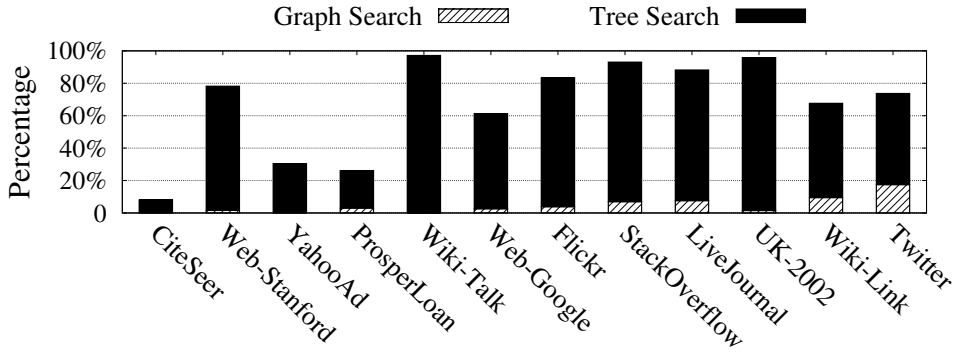
Overall, it is shown that our proposed optimization techniques not only reduce the overall visited vertices but also significantly replace the out-neighbor search by the tree children search.

5.6.3 Scalability Test

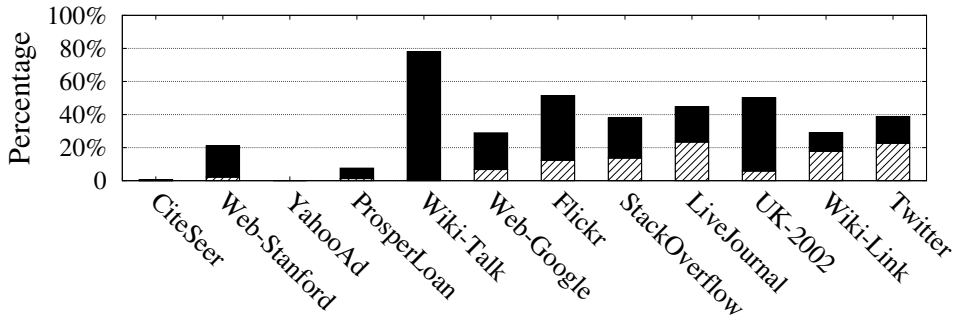
This experiment tests the scalability of our proposed algorithms. Due to the space limitation, we only report the results for two real-world graph datasets as representatives — UK-2002 and Twitter. The results on the other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges from 20% to 100%. When sampling vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set.

The running times of DFS-Insert* and DFS-Delete* for edge insertion and edge deletion of different percentages are reported in Figure 5.8 and Figure 5.9, respectively, with other algorithms used as comparisons.

It can be seen that DFS-Insert* and DFS-Delete* perform better than the other algorithms in all cases. For example, considering edge insertion on UK-2002, the running times of DFS-Insert*, DFS-Insert$^+$, DFS-Insert and ReDFS-Insert



(a) Edge insertions



(b) Edge deletions

Figure 5.7: Percentage of vertices performing graph search or tree search

are 1.09ms, 2.48ms, 10.29ms and 19.79ms, respectively, when sampling 20% vertices. When the sampling percentage of vertices reaches 100%, DFS-Insert*, DFS-Insert$^+$, DFS-Insert and ReDFS-Insert take about 18.30ms, 72.70ms, 74.26ms and 203.22ms, respectively.

Note that in Figure 5.8, the running time of several algorithms on Twitter sightly decreases when the sampling ratio increases in some cases. This is because in the 10000 update operations, we do not need to update the tree structure if an inserted edge is not a forward-cross edge or a deleted edge is not a tree edge. The proportion of the operations that lead to the tree update may be low in a large graph, so the average processing time for each operation may be small.

5.6.4 Memory Usage

Figure 5.10 shows the memory usage of all the four algorithms on all datasets. We do not use any auxiliary data structure in the final algorithm. Therefore, the memory usage of all the proposed algorithms is same.

5.7 Chapter Summary

This chapter introduces a framework and corresponding algorithms for the DFS-Tree maintenance problem considering both edge insertion and deletion in general directed graphs. Two groups of optimizations are also presented to speed up the algorithms. The results of extensive performance studies demonstrate the efficiency of our proposed algorithms.

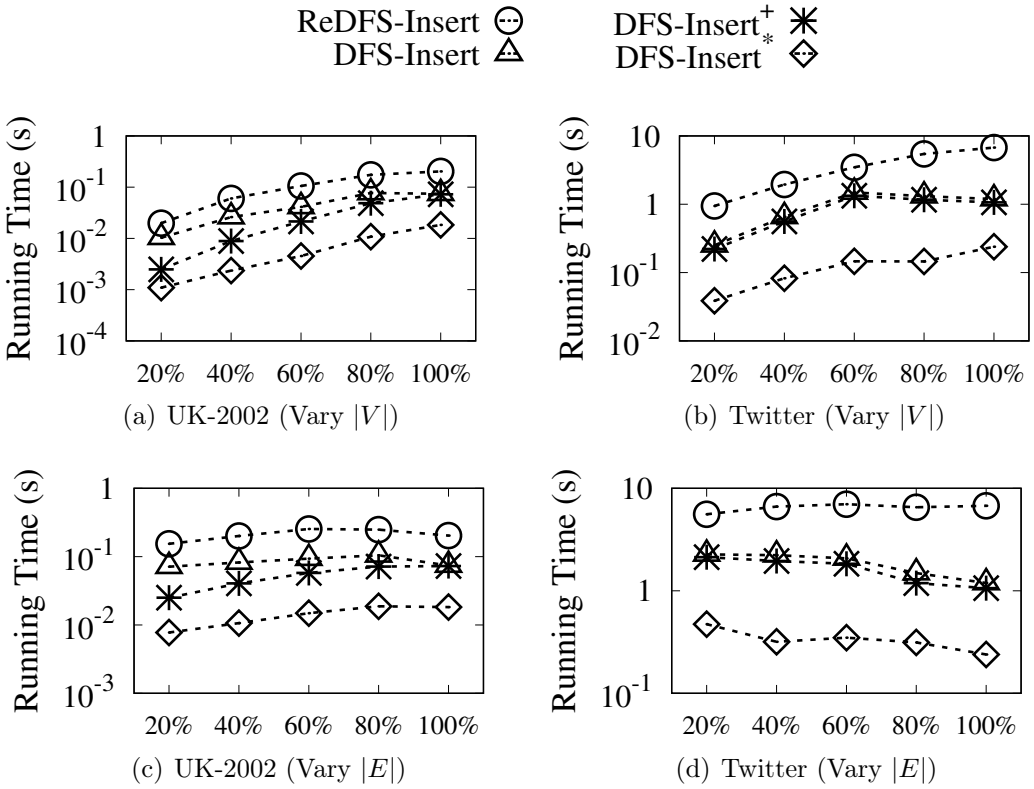


Figure 5.8: Scalability for edge insertion

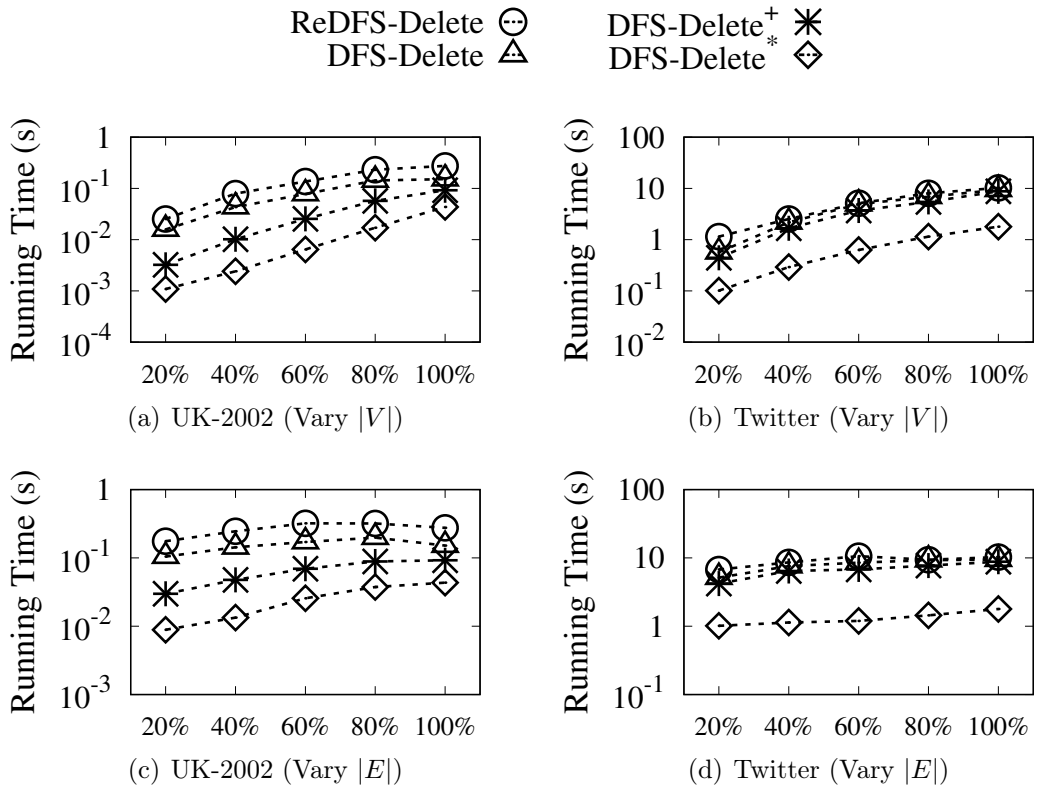


Figure 5.9: Scalability for edge deletion

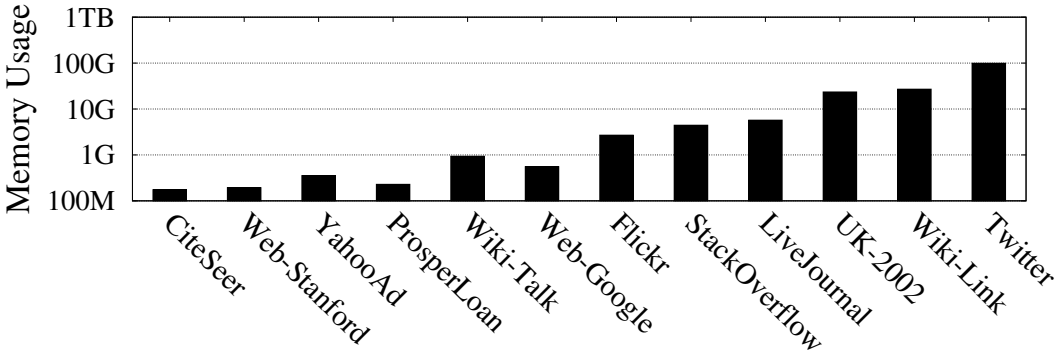


Figure 5.10: Memory usage

Chapter 6

EPILOGUE

In this thesis, we study the cohesive subgraph detection problem in large graphs. We propose index-based algorithms for computing k-cores in large uncertain graphs under internal memory and semi-external memory settings separately. The $UCF\text{-}Index$ maintains a tree structure for each integer k in the memory, and the $UCEF\text{-}Index$ stores these trees in the hard disk. The query algorithm based on $UCF\text{-}Index$ has optimal time complexity, and the query algorithm based on $UCEF\text{-}Index$ has optimal I/O complexity. The sizes of both two indexes are well-bounded by $O(m)$. We also design several optimizations to speed up the index construction algorithms. For the DFS-Tree maintenance problem, we introduce a framework and corresponding algorithms considering both edge insertion and deletion in general directed graphs. Two groups of optimizations are also presented to speed up the algorithms. We conduct extensive performance studies on several large real-world datasets to show our proposed algorithms' effectiveness and efficiency.

The research works in this thesis also open several future research directions. For the (k, η)-core computation, a potential task is to efficiently maintain the $UCF\text{-}Index$, given that many real-world graphs are highly dynamic. In addition,

approximate solutions can be designed to speed up the index construction. For the dynamic DFS problem, we have given a basic idea to handle the batch update in Subsection 5.5.3. A possible direction is to further improve the efficiency of the batch update. Another direction is to explore a vertex order and a corresponding DFS-Tree to reduce the update cost. Last but not least, developing a fully dynamic SCC detection algorithm based on the fully dynamic DFS algorithm.

BIBLIOGRAPHY

[1] E. Adar and C. Ré. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 30(2):15–22, 2007.

[2] C. C. Aggarwal. *Managing and mining uncertain data*. Springer, 2009.

[3] E. Akbas and P. Zhao. Truss-based community search: A truss-equivalence based indexing approach. *Proceedings of the VLDB Endowment*, 10(11):1298–1309, 2017.

[4] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in Neural Information Processing Systems*, pages 41–50, 2006.

[5] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 25–37. Springer, 2009.

[6] G. D. Bader and C. W. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(1):2, 2003.

[7] S. Baswana, S. R. Chaudhury, K. Choudhary, and S. Khan. Dynamic DFS in undirected graphs: Breaking the O(m) barrier. In *Proceedings of the*

27th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 730–739. SIAM, 2016.

[8] S. Baswana and K. Choudhary. On dynamic DFS tree in directed graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 102–114. Springer, 2015.

[9] S. Baswana, A. Goel, and S. Khan. Incremental DFS algorithms: A theoretical and experimental study. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 53–72. SIAM, 2018.

[10] S. Baswana, S. K. Gupta, and A. Tulsyan. Fault tolerant and fully dynamic DFS in undirected graphs: Simple yet efficient. In *44th International Symposium on Mathematical Foundations of Computer Science*, pages 65:1–65:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[11] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

[12] P. Boldi, F. Bonchi, A. Gionis, and T. Tassa. Injecting uncertainty in graphs for identity obfuscation. *Proceedings of the VLDB Endowment*, 5(11):1376–1387, 2012.

[13] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1316–1325. ACM, 2014.

[14] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[15] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 205–216. ACM, 2013.

[16] L. Chen, R. Duan, R. Wang, and H. Zhang. Improved algorithms for maintaining DFS tree in undirected graphs. *CoRR, abs/1607.04913*, 2016.

[17] L. Chen, R. Duan, R. Wang, H. Zhang, and T. Zhang. An improved algorithm for incremental DFS tree in undirected graphs. In *16th Scandinavian Symposium and Workshops on Algorithm Theory*, pages 16:1–16:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[18] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 51–62. IEEE, 2011.

[19] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by H*-graph. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 447–458. ACM, 2010.

[20] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[22] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 991–1002. ACM, 2014.

[23] H. De Fraysseix, P. O. De Mendez, and P. Rosenstiehl. Trémaux trees and planarity. *International Journal of Foundations of Computer Science*, 17(5):1017–1029, 2006.

[24] E. W. Dijkstra. *A discipline of programming*. Englewood Cliffs: Prentice-Hall, 1976.

[25] L. Eronen and H. Toivonen. Biomine: Predicting links between biological entities using network models of heterogeneous databases. *BMC Bioinformatics*, 13(1):119, 2012.

[26] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment*, 9(12):1233–1244, 2016.

[27] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[28] A. D. Fox, B. J. Hescott, A. C. Blumer, and D. K. Slonim. Connectedness of PPI network neighborhoods identifies regulatory hub proteins. *Bioinformatics*, 27(8):1135–1142, 2011.

[29] P. G. Franciosa, G. Gambosi, and U. Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information Processing Letters*, 61(2):113–120, 1997.

[30] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[31] Y. Gao, X. Miao, G. Chen, B. Zheng, D. Cai, and H. Cui. On efficiently find-

ing reverse k-nearest neighbors over uncertain graphs. *The VLDB Journal*, 26(4):467–492, 2017.

[32] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 44–50. AAAI Press, 2014.

[33] A. V. Goldberg. *Finding a maximum density subgraph*. University of California Berkeley, CA, 1984.

[34] J. Healy, J. Janssen, E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 137–148. Springer, 2006.

[35] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[36] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[37] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1311–1322. ACM, 2014.

[38] X. Huang, W. Lu, and L. V. Lakshmanan. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 77–90. ACM, 2016.

[39] R. Jin, L. Liu, and C. C. Aggarwal. Discovering highly reliable subgraphs in uncertain graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 992–1000. ACM, 2011.

[40] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *Proceedings of the VLDB Endowment*, 4(9):551–562, 2011.

[41] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 40–46. ACM, 1974.

[42] S. Khan. Near optimal parallel algorithms for dynamic DFS in undirected graphs. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 283–292. ACM, 2017.

[43] D. E. Knuth. *Art of computer programming, volume 3: Sorting and searching (2nd edition)*. Addison Wesley Longman Publishing Co., Inc., 1998.

[44] N. J. Krogan, G. Cagney, H. Yu, G. Zhong, X. Guo, A. Ignatchenko, J. Li, S. Pu, N. Datta, A. P. Tikuisis, et al. Global landscape of protein complexes in the yeast saccharomyces cerevisiae. *Nature*, 440(7084):637–643, 2006.

[45] R. E. Ladner. The circuit value problem is log space complete for P. *ACM SIGACT News*, 7(1):18–20, 1975.

[46] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5):509–520, 2015.

[47] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social

networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

[48] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[49] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[50] K. Nakamura and K. Sadakane. A space-efficient algorithm for the dynamic DFS problem in undirected graphs. In *International Workshop on Algorithms and Computation*, pages 295–307. Springer, 2017.

[51] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin. Efficient probabilistic k-core computation on uncertain graphs. In *2018 IEEE 34nd International Conference on Data Engineering (ICDE)*, pages 1192–1203. IEEE, 2018.

[52] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. K-nearest neighbors in uncertain graphs. *Proceedings of the VLDB Endowment*, 3(1-2):997–1008, 2010.

[53] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[54] J. H. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters*, 25(1):65–70, 1987.

[55] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

[56] A. E. Saríyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *Proceedings of the VLDB Endowment*, 10(3):97–108, 2016.

[57] A. E. Saríyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 927–937. ACM, 2015.

[58] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[59] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[60] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 282–292. ACM, 2002.

[61] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 939–948. ACM, 2010.

[62] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering*, 29(3):683–697, 2017.

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

[64] R. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2:160–161, 1974.

[65] R. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[66] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[67] A. Tucker. Chapter 2: Covering circuits and graph colorings. *Applied Combinatorics*, 49, 2006.

[68] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[69] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 133–144. IEEE, 2016.

[70] D. Wen, B. Yang, L. Qin, Y. Zhang, L. Chang, and R. Li. Computing k-cores in large uncertain graphs: An index-based optimal approach. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[71] B. Yang, D. Wen, L. Qin, Y. Zhang, L. Chang, and R.-H. Li. Index-based optimal algorithm for computing k-cores in large uncertain graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 64–75. IEEE, 2019.

[72] B. Yang, D. Wen, L. Qin, Y. Zhang, X. Wang, and X. Lin. Fully dynamic depth-first search in directed graphs. *Proceedings of the VLDB Endowment*, 13(2):142–154, 2019.

[73] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1):276–284, 2010.

[74] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing*, 53(2):352–369, 2010.

[75] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 337–348. IEEE, 2017.

[76] Z. Zhang, J. X. Yu, L. Qin, and Z. Shang. Divide and conquer: I/O efficient depth-first search. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 445–458. ACM, 2015.

[77] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 480–491. ACM, 2012.

[78] Z. Zou, H. Gao, and J. Li. Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 633–642. ACM, 2010.