COMPLEXITY ANALYSIS OF SOME PROBLEMS IN PLANAR GRAPHS, BOUNDED TREE-WIDTH GRAPHS, AND PLANAR POINT SETS

By

Prajakta Nimbhorkar

THE INSTITUTE OF MATHEMATICAL SCIENCES, CHENNAI.

A thesis submitted to the
Board of Studies in Mathematical Sciences

In partial fulfillment of the requirements

For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

October 2010

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Board, we recommend that the dissertation prepared by **Prajakta Nimbhorkar** entitled "Complexity Analysis of Some Problems in Planar Graphs, Bounded Tree-width Graphs, and Planar Point Sets" may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

---------- **Date :**
Chairman : V. Arvind (IMSc)

---------- **Date :**
Convener : Meena Mahajan (IMSc)

---------- **Date :**
Member : T. Kavitha (TIFR)

---------- **Date :**
Member : Venkatesh Raman (IMSc)

---------- **Date :**
Member : C. R. Subramanian (IMSc)

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.

---------- **Date :**
Guide : Meena Mahajan

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and the work has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution or University.

Prajakta Nimbhorkar

To My Parents and Family Members

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisor, Prof. Meena Mahajan, for her invaluable guidance, support, and encouragement which have led me towards the completion of this thesis. Working with her was a great learning opportunity. Besides her technical knowledge, I have been greatly influenced by her teaching, systematic way of working, clarity of thoughts, enthusiasm, and open-mindedness. I will always be grateful to her for all these things.

I would like to thank Prof. Samir Datta for the nice collaboration. He introduced me to the area of planar graphs and small-space complexity classes through the numerous discussions I had with him. He has always been enthusiastic to discuss problems, and share ideas, from which I have been greatly benefitted.

I wish to thank Prof. V. Arvind for introducing me to the interplay between algebra and computation and for the interesting discussions I had with him. I will forever be grateful to him for his invaluable support and encouragement.

I would like to thank Prof. Jaikumar Radhakrishnan for introducing me to the area of complexity theory, for several interesting discussions, as well as for his help, guidance, and encouragement.

I would like to thank all my co-authors - Prof. Samir Datta, Prof. Meena Mahajan, Prof. Kasturi Varadarajan, Prof. Thomas Thierauf, Bireswar, Nutan, and Fabian.

I also thank all the Computer Science faculty at the Tata Institute of Fundamental Research, and at The Institute of Mathematical Sciences for the knowledge of various aspects of theoretical computer science I gained from them during my course work and through seminars and discussions.

I am grateful to my M. Tech. advisor, Prof. Abhiram Ranade, for encouraging me to take up a research career in theoretical computer science. His guidance and clear thinking have always been an inspiration for me. The courses offered by him, and also by Prof. Sundar Vishwanathan, Prof. Ajit Diwan, and Prof. Ketan Mulmuley played a major role in developing my interest in theoretical computer science, and inspiring me to take up a research career in this area. I am thankful to all of them. I would like to thank Prof. Milind Sohoni for interesting discussions and guidance during my M. Tech. project.

I visited several academic institutes during my graduate studies, which provided me great opportunities to interact with researchers in my area. I would like to thank Prof. Benjamin Doerr for a visit to the Max Planck Institut Informatik, Prof. Jacobo Torán for inviting me to the University of Ulm, Prof. Sumit Ganguly for a visit to IIT Kanpur, Prof. Eldar Fischer for a visit to Technion and also for introducing me to the area of property testing.

I thank all the TCS students of IMSc and TIFR, with whom I had many interesting discus-

sions. Special thanks to Srikanth, Partha, Nutan, Somnath, Sreejith, Praveen, Karteek, Yadu, Kunal, Raghav, Sourav, Meghana for introducing me to several new problems in theoretical computer science through many wonderful discussions and presentations. I learned many things from them. Thanks to all my officemates for their cooperation as well as for interesting conversations. A warm thanks to all my friends at TIFR and IMSc for making my stay enjoyable and memorable. The walks and treks, the cooking, and the table tennis matches in the evenings are some of the memories I will always cherish.

I spent two years of my graduate studies at TIFR. I would like to thank TIFR and IMSc for the financial support and the wonderful infrastructure and working environment. Thanks to all the administrative staff at TIFR and IMSc for their cooperation, which made my stay extremely comfortable.

I would like to thank my parents for their constant support and encouragement. They have done a lot for me and I will always be grateful to them. A special thanks to my sister, Prachi, who has been a great companion right from my childhood. She has provided an invaluable support during my stay at TIFR. A heartfelt thanks to Maa, and Baba, who have always treated me like their daughter.

Last but not least, I would like to thank Partha, who has filled my life with joy. He has been a wonderful officemate, colleague, friend, and life partner.

Abstract

The focus of this thesis is on the complexity analysis of some computational problems in restricted graph-classes. The problems considered include graph isomorphism, various path problems like reachability, shortest path, and longest path computations. We investigate the space complexity of the graph isomorphism problem for planar graphs. The space complexity of path problems is considered for planar graphs, and k-trees. Another problem studied in the thesis is the clustering problem.

One of the main results on graph isomorphism included in the thesis is a log-space algorithm for isomorphism of planar graphs. This settles the complexity of planar graph isomorphism, since hardness for log-space is already known. A log-space algorithm is first described for isomorphism of 3-connected planar graphs, which is then used in the algorithm for planar graph isomorphism.

The results on path problems include an improved upper bound for computing the length of a longest path between two designated nodes in a planar DAG. We also present new upper bounds for counting the number of paths between two designated nodes in a planar DAG and in a single-sink DAG, under the promise that these numbers are bounded by a polynomial in the size of the graph. Reachability problem is also studied for directed k-trees and a log-space algorithm is given. Complexity of the shortest and longest path problems for directed acyclic k-trees has been analysed and log-space algorithms are described for these problems. We also give matching log-space hardness results, thereby settling the complexity of these problems for directed k-trees and directed acyclic k-trees respectively. These algorithms are applicable for partial k-trees, which are also known as graphs of tree-width at most k, provided a tree-decomposition for partial k-trees is given as input. The tree-decomposition for k-trees is known to be computable in log-space, but is not known for partial k-trees.

Another problem studied in this thesis is the k-means problem. It is a variant of the clustering problem. We prove that the k-means problem is NP-hard when the input is a set of points in two dimensions, and k is part of input. Earlier the hardness was known only for those instances where the number of dimensions is a part of input.

Contents

List of Figures

1 Introduction

The main goal of complexity theory is to prove non-trivial upper and lower bounds on the time and space requirements of various computational problems. The most widely used computational model is the *Turing machine model*. The computational power of a Turing machine depends on the time and space it is allowed to use, and also on whether it is *deterministic* or *non-deterministic*. Various complexity classes have been defined by imposing various time and space constraints on deterministic and non-deterministic Turing machines. Thus one way of proving a better upper bound for a problem is to show that the problem lies in a smaller complexity class. To show a lower bound for a problem, one can show that the problem is hard for a complexity class $\mathcal{C}$. A problem is hard for a class $\mathcal{C}$ if and only if each problem in $\mathcal{C}$ can be *reduced* to it under some acceptable notion of *reduction*. The complexity of a problem is said to be *settled* if it has the same upper and lower bound.

The emphasis of this thesis is mainly on the upper and lower bounds on the computational requirements of a Turing machine for solving certain fundamental computational problems on some restricted sets of inputs.

One of the main restrictions considered in the thesis is graph planarity. A natural problem that has been studied under this restriction is the problem of determining whether two given planar graphs are isomorphic. Another problem that has been considered on directed planar graphs, along with the additional restriction of acyclicity, is to compute the longest path length between two given nodes in the graph. We show that planarity significantly reduces the complexity of these problems under current complexity theoretic assumptions. Apart from this, the planarity restriction has been considered for a clustering problem called *the k-means problem*. Unlike the previous case, the planar (*i.e.* two-dimensional) k-means problem is shown to be computationally hard.

Another parameter is graph tree-width. We consider the reachability problem on graphs of bounded tree-width. With an additional restriction of acyclicity, we study the shortest and longest path problems on these graphs. In both of these problems, bounded tree-width is shown

to bring down the complexity of these problems under known complexity theoretic assumptions.

The thesis has been divided into three main parts, based on the problems considered. The *graph isomorphism* part contains results on planar graph isomorphism, the part on *path problems* includes results on reachability, and shortest and longest path problems in the graph classes mentioned above. These two problems are described in Section 1.2.1 and Section 1.2.2 respectively. The third part includes complexity analysis of the clustering problem known as the k-means problem, described in Section 1.2.3.

1.1 Preliminaries

We give some basic graph theoretic and complexity theoretic background here. This includes definitions of the graph classes and complexity classes considered in the thesis.

Graph Theory We first describe the notions related to connectivity in graphs, that are used in this thesis. Two main graph classes that are dealt with are that of *planar graphs*, and k*-trees*. We define these two classes and list some of their important properties here.

A graph $G = (V, E)$ is said to be *connected* if there is a path between every pair of vertices. G is called *biconnected* if it remains connected even after the removal of any one vertex and its incident edges. If G is not biconnected, then there are some vertices such that removing any one of them disconnects G into two or more components. Such vertices are called *articulation points*. G is said to be 3-*connected* if it remains connected even after the removal of any pair of vertices and their incident edges. If G is not 3-connected, then there is a pair of vertices whose removal disconnects G into multiple components. Such a pair is called a *separating pair*.

A graph is said to be *planar* if it can be drawn in a plane (or equivalently, on the surface of a sphere) without any two edges crossing each other. A *combinatorial embedding* or *rotation scheme* for a graph is a cyclic ordering of edges around each vertex. Let v be a vertex in a graph G and let E_v be the set of edges incident on v. Then a rotation ρ_v for v is a permutation on E_v. Rotation scheme or combinatorial embedding for G is defined as $\rho = \{\rho_v | v \in V\}$. For a planar graph G, a combinatorial embedding ρ is called a *planar combinatorial embedding* if there is a plane drawing of G that has the combinatorial embedding ρ.

Another graph class considered in this thesis is the class of k*-trees*.

Definition 1.1 *The class of* k-trees *is inductively defined as follows (see e.g. [42]):*

- *A clique with* k *vertices (*k*-clique for short) is a* k*-tree.*
- *Given a* k*-tree* G' *with* n *vertices, a* k*-tree* G *with* $n + 1$ *vertices can be constructed by introducing a new vertex* v *and picking a* k*-clique* X *(called the* support*) in* G' *and joining* v *to each vertex* u *in* X*. Thus,* $V(G) = V(G') \cup \{v\}$*,* $E(G) = E(G') \cup \{\{u, v\} \mid u \in X\}$*.*

A *partial k-tree* is a subgraph of a k-tree. The class of partial k-trees coincides with the class of graphs which have tree-width at most k.

Complexity Theory Now we define the complexity classes that appear in this thesis, and Figure 1.1 shows the relations among them.

We assume that a Turing machine has a read-only input-tape and a read-write work-tape. The complexity class *deterministic log-space* (L) is the class of languages accepted by a deterministic Turing machine that has a log-space work-tape. A complete problem for L is reachability in undirected graphs [77]. *Non-deterministic log-space* (NL) is the class of languages accepted by a non-deterministic Turing machine with a log-space work-tape. Directed reachability is a problem that is complete for this class. *Unambiguous log-space* (UL) is the class of languages accepted by a non-deterministic log-space machine that has at most one accepting path on any input. L and NL are closed under complement, but UL is not known to be closed under complement. The complement class of UL is coUL. The functional analogue of L is the class FL. Thus FL consists of functions that are computable by a deterministic log-space machine that also has a write-only one-way output-tape.

Now we consider the complexity classes where the Turing machine is allowed to have a stack, in addition to a log-space work-tape. An AuxPDA is an auxiliary pushdown automaton augmented with a log-space work-tape and running in time polynomial in the length of input. Alternatively, it can also be viewed as a log-space machine augmented with a stack and running in polynomial-time. It is called DAuxPDA, or UAuxPDA depending on whether the machine is deterministic, or unambiguous respectively. The class of languages accepted by a DAuxPDA is exactly the class LogDCFL, which is the class of languages that are log-space many-one reducible to some (deterministic) context-free language [82]. Similarly, the class of languages accepted by a non-deterministic AuxPDA is same as the class LogCFL, which is the class of languages that are log-space many-one reducible to a context-free language. However, UAuxPDA is not known to be equivalent to the class LogUCFL - the class of languages that are many-one reducible to an unambiguous context-free language. Only one-way containment LogUCFL$\subseteq$ UAuxPDA is known [58].

Additionally, we also consider complexity classes based on some circuit classes. NC is the class of languages recognized by poly-logarithmic depth bounded fan-in boolean circuits. AC is the class of languages recognized by poly-logarithmic depth unbounded fan-in boolean circuits. It is easy to see that NC=AC. In particular, NC^1 and AC^1 are the classes of languages recognized by log-depth boolean circuits of bounded and unbounded fan-in respectively.

Besides the language classes described so far, we describe here two function classes GapL and #L. #L is the class of functions f with the property that there is an NL machine M such that $f(x)$ is the number of accepting paths of M on input x. Thus the canonical complete problem

for #L is computing the number of paths of an NL machine. GapL is the class of functions f such that $f(x)$ is the number of accepting paths minus the number of rejecting paths of an NL machine on input x. *Computing the determinant of an integer matrix* is a complete problem for GapL under log-space reductions, and hence GapL can also be defined as the class of functions that are log-space reducible to the determinant computation problem for integer matrices.

The relations among these complexity classes are shown in Figure 1.1.

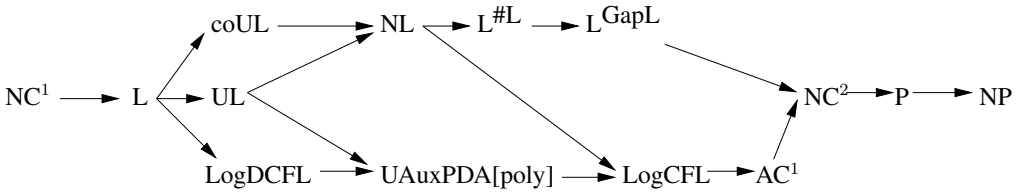

Figure 1.1: Relation among complexity classes

1.2 Problem Definitions and Related Results

The contributions of this thesis are in three main problems: graph isomorphism and canonization, path problems, and clustering problems. We define these problems here and give a brief overview of the previous work on these problems.

1.2.1 Graph Isomorphism and Canonization

Graph isomorphism (GI) is a fundamental computational problem, and its complexity is not yet settled. This problem has been extensively studied for a long time. Given two graphs G and H, GI involves determining if there is an edge-preserving bijection ϕ between the vertices of G and the vertices of H. Thus $\phi : V(G) \to V(H)$ is an isomorphism between G and H if it is a bijection such that $(\phi(u), \phi(v)) \in E(H) \Leftrightarrow (u, v) \in E(G)$. If such a bijection exists, then the two graphs are said to be *isomorphic*, denoted as $G \cong H$. The decision, counting, and search versions of GI are known to be polynomial-time equivalent (see *e.g.*[1, 68]).

GI is defined for directed as well as undirected graphs. However, GI for directed graphs is log-space many-one reducible to GI for undirected graphs [54]. If G and H are colored graphs, then an isomorphism ϕ between them must preserve the colors. Thus, if G and H have colors on their vertices, an isomorphism ϕ must have the additional property that u and $\phi(u)$ have the same color for all the vertices $u \in V(G)$. GI for colored graphs can be reduced to GI for uncolored graphs by AC^0-reductions (see *e.g.*[54]).

The problem can also be posed when G and H are in fact two copies of the same graph. In this case, the problem is called the *graph automorphism* problem and involves determining

whether there is an edge-preserving bijection from the vertex set of G to itself, which is different from the identity mapping. The graph automorphism problem is polynomial-time equivalent to the graph isomorphism problem. It can be seen that the set of automorphisms of a graph forms a group under composition.

Graph canonization is another problem of similar flavour. Given a graph class $\mathcal{G}$, a function f defined on $\mathcal{G}$ is said to compute *canonical forms* of $\mathcal{G}$ if it satisfies the following:

$$\begin{aligned} \forall G, H \in \mathcal{G} \quad &: \quad G \cong H \Leftrightarrow f(G) = f(H) \\ \forall G \in \mathcal{G} \quad &: \quad f(G) \cong G \end{aligned}$$

Given a function f that computes canonical forms, an isomorphism from G to $f(G)$ is called *a canonical labelling* of G. For a graph G, $f(G)$ is called the *canon* of G. The graph canonization problem involves computing such a function f. It is easy to see that the graph canonization problem is at least as hard as GI. Whether the two problems are equivalent is a long-standing open question. However, it is interesting to note that many polynomial-time algorithms, and parallel algorithms for GI on restricted graph classes mentioned above, in fact, provide a canonical labelling for the graphs.

The best known upper bound for the graph canonization problem is $\mathsf{FP}^{\mathsf{NP}}$ (see *e.g.* [14]), whereas it is easy to see that GI is in NP. This is because, given a bijection ϕ between the vertices of two graphs G and H, it is easy to verify whether ϕ is an isomorphism between G and H. GI is also known to be in the complexity class SPP [16]. However, no polynomial-time algorithm is known for GI. On the other hand, it is unlikely to be NP-complete. It is known that if GI is NP-complete, then the polynomial hierarchy collapses to its second level [24, 80]. The seminal paper by Babai and Luks takes a group-theoretic approach to graph canonization [22]. A subexponential-time algorithm for GI is known [18]. From complexity theory perspective, the best known lower bound for GI is GapL [86].

There are polynomial-time algorithms for GI on various graph classes like graphs of bounded degree [65], bounded genus [71], bounded tree-width [39], bounded color classes [66], planar graphs [45, 46], some fixed minor-free families of graphs [75], etc. For GI on 3-connected planar graphs, an $O(n^2)$ algorithm is known due to [92], which was extended to general planar graphs by [44]. An $O(n^2)$ algorithm for GI on planar graphs is also given by [56], whereas a linear-time algorithm is known due to [46]. For random graphs, the Weisfeiler-Lehman method [21, 20] produces a canonical form with high probability.

The parallel complexity of GI in various graph classes has been studied. An NC algorithm for trees was given by [29]. For planar graphs, an AC^1 algorithm is known due to [72], [76] and [89]. For graphs with colored vertices and bounded color classes, an NC algorithm was given by [66]. Additionally, GI for graphs with bounded color classes is known to be in the #L hierarchy

[17]. For GI in bounded tree-width graphs, a TC^1 upper bound is known due to [39], which has been improved recently to LogCFL [31].

Recent research has been focussed on finding out the space-complexity of GI on these graph classes, and matching hardness results. The goal has been to show GI for various graph classes to be complete for some natural complexity class. For instance, a log-space algorithm for GI on trees is known due to [62]. GI on trees is known to be L-hard when the input trees are given in pointer representation [50, 69], whereas it is NC^1-complete when they are represented as strings. L-completeness is also known for GI on partial 2-trees (also known as *series-parallel graphs*) [14], and k-trees [53]. For tournaments, the GapL lower bound of GI in general graphs is known to hold [90].

1.2.2 Path Problems

Path problems form another class of fundamental computational problems. We consider the following problems:

$$\begin{aligned}
\textsf{Reach} &= \{\ (G,s,t)\ \mid\ G \text{ contains a path from } s \text{ to } t\ \}\\
\textsf{Distance} &= \{\ (G,s,t,k)\ \mid\ G \text{ contains a path of length} \leq k \text{ from } s \text{ to } t\ \}\\
\textsf{Long-Path} &= \{\ (G,s,t,k)\ \mid\ G \text{ has a simple path of length} \geq k \text{ from } s \text{ to } t\ \}\\
\#\textsf{Path} &= \{\ (G,s,t,1^k)\ \mid\ G \text{ has exactly } k \text{ simple paths from } s \text{ to } t\}
\end{aligned}$$

These problems have varying complexities. Clearly, Distance and Long-Path are at least as hard as Reach, since $(G,s,t) \in$Reach if and only if $(G,s,t,n) \in$Distance if and only if $(G,s,t,0) \in$Long-Path. Reach is L-complete for undirected graphs [77], and NL-complete for directed graphs. Distance is NL-complete for directed as well as undirected graphs, and even for directed acyclic graphs [83]. Long-Path is NP-complete as the Hamiltonian cycle problem reduces to Long-Path. However, it is NL-complete for acyclic graphs. #Path is #P-hard, whereas for the class of directed acyclic graphs, #Path is #L-complete. A better upper bound of NL is known for #Path in directed acyclic graphs, when there is a promise that the number of s to t paths in the graph is bounded by a polynomial in the size of the graph [6].

The complexities of these problems have been analyzed for various graph classes. For directed planar graphs, Reach is known to be in $\mathsf{UL} \cap \mathsf{coUL}$ due to [25], whereas Distance is known to be in $\mathsf{UL} \cap \mathsf{coUL}$ due to [84]. For *series-parallel graphs*, which are also known as *partial* 2*-trees*, all the above problems are L-complete [49].

1.2.3 Clustering

Clustering is another interesting and important problem. Given a set of objects, the clustering problem involves partitioning the objects into different chunks based on their pairwise *similarities* and *differences*. The notion of *similarity* can be defined in various ways. In the geometric

setting, the objects are points in a Euclidean space, and the notions of *similarity* and *difference* are in fact some function of pairwise distances of the objects. Several variants of the clustering problem have been defined in the geometric setting *e.g. k-median, k-center, k-means*. Given a set S of n points in $\mathbb{R}^m$, the goal in the k-means problem is to find k points, called *centers*, in $\mathbb{R}^m$ so as to minimize the sum of the squares of the Euclidean distance of each point in S to its nearest center. In the k-median problem, the objective is to minimize the sum of the Euclidean distance of each point in S to its nearest center. In the k-center problem, the objective is to cover the points using k disks of minimum radius. Note that, in the geometric setting, instead of the Euclidean distance, the problems can be defined using L_p distance for any p. The k-median and k-center problems defined above are known to be NP-hard (see *e.g.* [70]).

We focus on the k-means problem here. The problem has been extensively studied. Lloyd [63] proposed a very simple and elegant local search algorithm that computes a certain local (and not necessarily global) optimum for this problem. Har-Peled and Sadri [41] and Arthur and Vassilvitskii [10, 9] examine the question of how quickly this algorithm and its variants converge to a local optimum. Lloyd's algorithm also does not provide any significant guarantee about how well the solution that it computes approximates the optimal solution. Ostrovsky et al. [74] and Arthur and Vassilvitskii [11] show that randomized variants of Lloyd's algorithm can provide reasonable approximation guarantees.

The k-means problem has also been studied directly from the point of view of approximation algorithms. There are polynomial-time algorithms that compute a constant-factor approximation to the optimal solution; see, for instance, the local search algorithm analyzed by Kanungo et al. [51]. If k, the number of centers, is a fixed constant, then the problem admits polynomial-time approximation schemes [34, 57]. If both k and the dimension m of the input are fixed, the problem can be solved exactly in polynomial time [47].

Drineas et al. [35], Aloise et al. [7], and Dasgupta [32] show that the k-means problem is NP-hard when the dimension m is part of the input even for $k = 2$. However, there has been no known NP-hardness result when the dimension m is fixed and k, the number of clusters, is part of the input.

While clustering problems generally tend to be NP-hard even in the plane, there are surprising exceptions – the problem of covering a set of points by k balls so as to minimize the sum of the radii of the balls admits a polynomial time algorithm if we use L_1 balls, and a $(1 + \varepsilon)$-approximation algorithm that runs in time polynomial in the input size and $\log \frac{1}{\varepsilon}$ for the usual Euclidean balls [37].

1.3 Results and Organization of the Thesis

The contributions of this thesis and the organization of subsequent chapters is given here. As stated earlier, the contributions can be broadly divided into three parts: graph isomorphism, path problems, and clustering. The contributions in graph isomorphism include a log-space algorithm for GI on 3-connected planar graphs (Chapter 2), and a log-space algorithm for GI on planar graphs (Chapter 3). Both of these results also give a canonical labelling of the graphs. In path problems, we show that the Long-Path problem in a planar directed acyclic graph is in UL $\cap$ coUL. The result and some of its extensions are described in Chapter 4. Another result is a log-space algorithm for Reach in directed k-trees, and for Distance and Long-Path computation in directed acyclic k-trees. These results appear in Chapter 5. Our log-space algorithms on GI and on path problems crucially use the seminal result by [77], which proves that undirected reachability is computable in L. In the clustering problem, we prove that the k-means problem is NP-hard even in two dimensions. The result is described in Chapter 6. Following is a brief overview of the contributions mentioned above:

Log-space Algorithm for Isomorphism and Canonization of 3-connected Planar Graphs We give a log-space algorithm for isomorphism and canonization problem in 3-connected planar graphs. The previously known upper bound was UL $\cap$ coUL due to [84]. A lower bound of L has also been given by [84]. Thus our result settles the complexity of this problem.

A crucial fact used in our algorithm is a result due to [93], which states that 3-connected planar graphs have a unique combinatorial embedding on the sphere, up to reflection. Moreover, a combinatorial embedding of a planar graph is known to be computable in L [5, 77].

The main steps in our canonical labelling algorithm are to get a combinatorial embedding of the given 3-connected planar graph, traverse the graph using a *universal exploration sequence* and output the list of edges traversed, then relabel the vertices of the graph in the order of their first occurrence in this list. This forms a labelling of the vertices, which is a function of the combinatorial embedding, and the choice of the edge and the vertex from which the traversal of the graph is started. This gives only $O(n)$ possible ways of labelling the vertices. A log-space transducer cycles through all these choices and chooses that labelling which leads to the lexicographically smallest list of edges. This list of edges forms the canon of the graph.

The notion of *universal exploration sequence (UXS)* used here was introduced in [55]. For a d-regular graph on n vertices, a UXS is a sequence of offsets which guides the traversal of a graph. A UXS has the property that, if a graph is connected, then its traversal according to the UXS visits all the vertices of the graph. Such a sequence is known to be computable in L [77]. We give the details in Chapter 2.

Log-space Algorithm for Planar Graph Isomorphism and Canonization We give a log-space algorithm for planar graph isomorphism and canonization. This improves the previously known upper bound AC^1 due to [72]. As GI is L-hard for trees [69], it also implies the same lower bound for planar graphs. Thus our result settles the complexity of the planar graph isomorphism problem. Recently, this result has been extended to graphs that exclude either a $K_{3,3}$ or a K_5 as minor [33].

The algorithm has the following main steps: Decomposition of a planar graph into biconnected components and construction of the *biconnected component tree*, decomposition of the biconnected components into triconnected components and construction of the *triconnected component tree*, canonization of the triconnected components using the canonization algorithm for 3-connected planar graphs, and canonization of the entire graph through canonization of the biconnected components. Each of the steps can be implemented in FL. Biconnected component tree is a tree-structure on biconnected components and articulation points. Triconnected component tree is a tree-structure on triconnected components and separating pairs in the graph. While dealing with these tree-structures, we crucially use the log-space algorithm given in [62] for GI on trees. The details of the algorithm and its complexity analysis are given in Chapter 3.

Longest Paths in Planar DAGs in $\mathsf{UL} \cap \mathsf{coUL}$ We investigate the complexity of the Long-Path problem for planar directed acyclic graphs and show that the NL upper bound can be improved to $\mathsf{UL} \cap \mathsf{coUL}$. This algorithm is based on the technique developed in [49]. From the technique in [49], it follows that Distance and Long-Path are equivalent for series-parallel graphs, given oracle access to Reach. We show that this result also holds for planar DAGs. Moreover, we give a double inductive counting algorithm similar to that of [6] for Long-Path problem on *max-unique graphs*, which are graphs with a unique longest path between each pair of connected vertices. These algorithms are described in Chapter 4.

Another problem addressed in Chapter 4 is the #Path problem. We consider the problem when there is a promise that the number of s to t paths is bounded by a polynomial, and give new upper bounds for various graph classes. For planar DAGs with this promise, #Path can be computed by a UAuxPDA running in polynomial time. The same bound holds for computing the number of shortest or longest s to t paths in a planar DAG, when this number is bounded by a polynomial. For DAGs with a unique sink, #Path can be computed in LogDCFL, when the target node is the unique sink. These algorithms are based on a depth-first search technique.

Log-space Algorithms for Path Problems in k-trees In [49], Reach, Distance, and Long-Path problems have been proved to be L-complete for directed series-parallel graphs (also known as partial 2-trees). We extend this result to directed k-trees, where k is a constant, and thus show that Reach in directed k-trees, and Distance and Long-Path in directed acyclic k-trees are

L-complete.

k-trees are known to have a *tree-decomposition*, which has been used in our algorithms. The tree-decomposition is known to be computable in L due to [53]. The ideas used in our algorithms are different from those in [49]. The technique central to these results is a careful implementation of divide-and-conquer in L. For Distance and Long-Path, we also use a technique from [61] (also [26]), where it has been used in the context of parsing languages recognized by *visibly pushdown automata*. The results also hold for partial k-trees (also known as graphs of tree-width k), however, a tree-decomposition for partial k-trees is not known to be computable in L. The best known upper bound for computing this tree-decomposition is LogCFL due to [91]. Our results are applicable if a decomposition is given as the part of input. The details are given in Chapter 5.

NP-hardness of the Planar k-means Problem Dasgupta [32] raised the question of whether k-means is hard in the plane. We answer this question in affirmative. We give a polynomial-time reduction from *planar* 3-*SAT* to the planar k-means problem. The planar 3-SAT problem is a variant of 3-SAT, where a graph drawn on the clauses and variables in a 3-SAT formula is restricted to be planar. This problem is proved to be NP-hard in [60]. Given a planar 3-SAT formula, our reduction involves embedding its clause-variable graph in a two-dimensional integer grid (see *e.g.* [4]), and then constructing the planar k-means instance from this grid embedding. The details of the reduction are described in Chapter 6.

Part I

Graph Isomorphism and Canonization

2

Isomorphism and Canonization of 3-connected Planar Graphs

The graph isomorphism problem GI involves determining whether there is an edge-preserving bijection between the vertex-sets of two given graphs. That is, given two graphs $G = (V, E)$ and $H = (W, F)$, determine whether there exists a bijection $\phi : V \to W$ such that $(u, v) \in E \Leftrightarrow (\phi(u), \phi(v)) \in F$. It is an important and well-studied problem with as yet unsettled complexity. In this chapter, we consider this problem for 3-connected planar graphs.

We give a log-space algorithm for GI on 3-connected planar graphs. This improves the previously known upper bound of UL $\cap$ coUL by [84]. We also provide a way to give canonical labels to the vertices of a graph in log-space. Thus the main result of this chapter is the following:

Theorem 2.1 *Given two 3-connected planar graphs G and H, deciding whether G is isomorphic to H is complete for* L*. Given a 3-connected planar graph G, constructing a canon for G is complete for* FL*.*

In fact, the isomorphism algorithm proceeds by constructing canons of the two given graphs, and then compares the canons to check whether the graphs are isomorphic. We recall the following definition of a *canon* and *canonical labelling*:

Definition 2.2 *Given a graph class $\mathcal{G}$, a function f defined on $\mathcal{G}$ is said to compute* canonical forms *of $\mathcal{G}$ if it satisfies the following:*

$$\begin{aligned} \forall G, H \in \mathcal{G} \quad &: \quad G \cong H \Leftrightarrow f(G) = f(H) \\ \forall G \in \mathcal{G} \quad &: \quad f(G) \cong G \end{aligned}$$

Given a function f that computes canonical forms, an isomorphism from G to $f(G)$ is called a canonical labelling *of G, and $f(G)$ is called a* canon *of G.*

We generalize this definition to *canonical code* of a graph, which is a function f not only of the given graph G but also of one or more parameters like an edge or a vertex of G or a combinatorial embedding of G. The canonical code has the following property: Given two graphs G and H along with the parameters on which the canonical code is defined, the canonical codes are equal if and only if there is an isomorphism ϕ between G and H such that ϕ also maps the given parameters of G to the corresponding given parameters of H.

2.1 Comparison with the Previous Approach

The $\mathsf{UL} \cap \mathsf{coUL}$ algorithm of [84] for canonization of 3-connected planar graphs first obtains a combinatorial embedding of the given graph and then proceeds by constructing a spanning tree of the graph, which depends on the combinatorial embedding and a fixed starting edge of the graph. For a fixed choice of combinatorial embedding and a starting edge, the spanning tree is unique and hence it is said to be *canonical* when the graph, its combinatorial embedding, and a starting edge are given. The $\mathsf{UL} \cap \mathsf{coUL}$ algorithm of [84] then traverses this tree and outputs a canonical list of edges, which depends only on the spanning tree. The vertices of the graph are relabelled according to their first occurrence in this list to get a canonical labelling. The canonical spanning tree construction needs shortest path computations, which can be done in $\mathsf{UL} \cap \mathsf{coUL}$ [84].

The list of edges is a canonical code of the graph, which is a function of the graph, the combinatorial embedding and the starting edge chosen for the spanning tree construction. A log-space transducer cycles through all the choices of the combinatorial embedding and the starting edge, and selects the choice which leads to the lexicographically smallest canonical code. This serves as the canon of the graph.

Our approach bypasses the spanning tree construction and thereby eliminates the need of distance computation. Instead, to construct a list of edges, our algorithm first makes the graph 3-regular and then traverses it using a *universal exploration sequence*. The rest of the steps remain the same as in [84]. The details are given in the next section.

Both the approaches crucially use the result by [93] that a 3-connected planar graph has precisely two combinatorial embeddings, and hence there are only $O(n)$ ways to obtain a canonical code. Moreover, a combinatorial embedding of a planar graph is known to be computable in FL [5, 77].

2.2 Our Algorithm

As described in the previous section, our algorithm uses a *universal exploration sequence (UXS)*. The definition and properties of a UXS are given below.

2.2.1 Universal Exploration Sequence

The notion of a universal exploration sequence was first introduced in [55]. Let $G = (V, E)$ be a d-regular graph, with given combinatorial embedding ρ. The edges around any vertex u can be numbered $\{0, 1, \ldots, d-1\}$ according to ρ arbitrarily in clockwise order. A sequence $\tau_1\tau_2 \ldots \tau_k \in \{0, 1, \ldots, d-1\}^k$ and a starting edge $e_0 = (v_{-1}, v_0) \in E$ define a walk $v_{-1}, v_0, \ldots, v_k$ on G as follows: For $0 \leq i \leq k$, if (v_{i-1}, v_i) is the s^{th} edge of v_i, let $e_i = (v_i, v_{i+1})$ be the ℓ^{th} edge of v_i, where $\ell = (s + \tau_i) \mod d$.

Definition 2.3 (Universal Exploration sequences (UXS) [55]:) *A sequence $\langle \tau_1, \tau_2, \ldots, \tau_\ell \rangle \in \{0, 1, \ldots d-1\}^\ell$ is a universal exploration sequence for d-regular graphs of size at most n if for every connected d-regular graph on at most n vertices, any numbering of its edges, and any starting edge, the walk obtained visits all the vertices of the graph. Such a sequence is called an (n, d)-universal exploration sequence.*

The following lemma suggests that UXS can be constructed in L [77]:

Lemma 2.4 *There exists a log-space algorithm that takes as input $(1^n, 1^d)$ and produces an (n, d)-universal exploration sequence.*

2.2.2 Outline of Our Approach

The main steps involved in the algorithm can be outlined as follows:

1. Given a 3-connected planar graph $G = (V, E)$, find a planar embedding ρ of G.
2. Make the graph 3-regular canonically for this embedding ρ to obtain an edge-coloured graph G' as described in Algorithm 1.
3. Find the canon of G' using Algorithm 2. To do this, an $(n, 3)$-UXS is constructed and the graph is traversed according to it. During the traversal, the edges traversed are output. This gives a list of edges. The vertices of the graph are then relabelled in the order of their occurrence in this list. The list of edges in lexicographic order with these new labels on vertices is a canonical code that depends on the graph and the choice of its combinatorial embedding, and an edge and one of its end-points where the traversal is started.

Step 1 is known to be in FL [5, 77]. We prove that steps 2 and 3 can also be done in log-space. Step 3 uses a UXS to traverse the graph. Step 2 essentially does the preprocessing in order to make step 3 applicable.

To get a canon of G, the minimum canonical code is obtained by cycling through all the choices of starting edge and starting vertex, and both the combinatorial embeddings. As there are only linearly many choices, a log-space transducer can cycle through all of them.

2.2.3 Making the Graph 3-regular

In this section, we describe the preprocessing step to make the graph 3-regular. In Section 2.2.4, we use Reingold's construction for UXS [77] to come up with a canonical code. This preprocessing step is essential, as Reingold's construction [77] for UXS requires the graph to have constant degree. In Lemma 2.5, we prove that the preprocessing step preserves isomorphism of graphs. That is, two graphs are isomorphic if and only if they are isomorphic after the preprocessing step. As the embedding of the new graph is inherited from the given graph, even the new graph has only two possible embeddings.

Algorithm 1 Procedure to get a 3-regular planar graph G' from 3-connected planar graph G.

Input: A 3-connected planar graph G with planar combinatorial embedding ρ.
Output: A 3-regular planar graph G' on $2m$ vertices, with edges coloured 1 and 2 and planar combinatorial embedding ρ'.

1: **for all** $v_i \in V$ **do**
2: Replace v_i by a cycle $\{v_{i1}, \ldots, v_{id_i}\}$ on d_i vertices, where d_i is the degree of v_i.
3: The d_i edges $e_{i1}, \ldots, e_{id_i}$ incident to v_i in G are now incident to $v_{i1}, \ldots, v_{id_i}$ respectively.
4: Colour the cycle edges with colour 1.
5: Colour $e_{i1}, \ldots, e_{id_i}$ with colour 2.
6: **end for**

We describe the preprocessing steps in Algorithm 1. Note that the new graph thus obtained has $2|E|$ vertices.

Lemma 2.5 *Given two 3-connected planar graphs G_1 and G_2 with combinatorial embeddings ρ_1 and ρ_2, there is an isomorphism between G_1 and G_2 that preserves the combinatorial embeddings, if and only if there is an isomorphism between G'_1 and G'_2 that respects the colors on the edges.*

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two 3-connected planar graphs with planar combinatorial embeddings ρ_1 and ρ_2 respectively. Let $\phi : V_1 \to V_2$ be an isomorphism between the oriented graphs (G_1, ρ_1) and (G_2, ρ_2). By isomorphism of oriented graphs we mean that the graphs are isomorphic for the fixed embeddings, in our case ρ_1 and ρ_2. That is, if $\rho_{1_u} = u_1 \ldots u_d$, $\rho_{2_{\phi(u)}} = v_1 \ldots v_d$ and $\phi(u_1) = v_1$, then $\phi(u_i) = v_i$ for $1 \leq i \leq d$.

Construct G'_1 and G'_2 as described in Algorithm 1, replacing each vertex v of degree d by a cycle of length d, and colouring the new cycle edges with colour 1 and original edges with colour 2. The algorithm preserves the orientation of original edges from G_1 and G_2 and outputs the coloured oriented graphs (G'_1, ρ'_1) and (G'_2, ρ'_2).

Given an isomorphism ϕ between (G_1, ρ_1) and (G_2, ρ_2), we show how to derive an isomorphism ϕ' between (G'_1, ρ'_1) and (G'_2, ρ'_2). Let $u \in V_1$ with degree d and $\phi(u) = v$. Then we want

to derive an isomorphism between G'_1 and G'_2 which maps the d vertices in G'_1 corresponding to u to the d vertices in G'_2 corresponding to v. To fix the map between the cycles corresponding to u and v, we look at the edges incident on u and v.

Consider an edge $\{u, w\}$ in E_1. Let $\phi(w) = x$. Then $\{x, v\} \in E_2$. Let the corresponding edge in G'_1 be $\{u_i, w_j\}$ and that in G'_2 be $\{v_k, x_\ell\}$. Then we define a map $\phi' : V'_1 \to V'_2$ which is inherited from ϕ such that $\phi'(u_i) = v_k$ and $\phi'(w_j) = x_\ell$. It is easy to see that ϕ' is an isomorphism for edge-coloured oriented graphs (G'_1, ρ'_1) and (G'_2, ρ'_2).

Now we show how to obtain an isomorphism ϕ between (G_1, ρ_1) and (G_2, ρ_2), given an isomorphism ϕ' between (G'_1, ρ'_1) and (G'_2, ρ'_2). Let $e = \{v_{i_p}, v_{i_q}\} \in E'_1$ and the corresponding edge $e' = \{\phi'(v_{i_p}), \phi'(v_{i_q})\} \in E'_2$. Let v_{i_p} and v_{i_q} correspond to the same vertex v_i in G_1. Then colour of e and e' is 1. Thus ϕ' maps copies of the same vertex of G_1 to copies of a single vertex of G_2. Hence a map ϕ can be derived from ϕ' in a natural way. It is easy to see that ϕ is an isomorphism between oriented graphs (G_1, ρ_1) and (G_2, ρ_2). ■

2.2.4 Obtaining the Canonical Code

Lemma 2.5 from the previous section suggests that for given embeddings ρ_1, ρ_2 of G_1 and G_2, it suffices to check the 3-regular oriented graphs (G'_1, ρ'_1) and (G'_2, ρ'_2) for isomorphism. Thus, a canonical code of a graph G can be constructed using the graph G'. The Procedure $canon(G, \rho, v, e = (u, v))$ described in Algorithm 2 does this using a universal exploration sequence.

Algorithm 2 Procedure $canon(G, \rho, v, e = (u, v))$

Input: 3-connected planar graph $G = (V, E)$ and its combinatorial embedding ρ, starting vertex v, starting edge $e = (u, v)$.

Output: Canon of (G, ρ, v, e).

Construct a degree 3, edge-colored graph $G' = (V', E')$ using Algorithm 1.
Construct an $(n, 3)$-universal exploration sequence U.
With starting vertex $v'_i \in V'$ and edge $e = (v'_i, u'_j)$ incident on it, traverse G' according to U and ρ', outputting the labels of the vertices.
Relabel the vertices of G according to the first occurrence of any of their copies in this output sequence.
Output the list of edges of G in lexicographically increasing order.

To prove the correctness of Algorithm 2, we show that two graphs have the same canon if and only if they are isomorphic, with the given choice of combinatorial embedding, starting edge, and starting vertex.

Lemma 2.6 *Let G_1, G_2 be 3-connected planar graphs on n vertices. Let ρ_1 be a combinatorial embedding of G_1. Let $v \in V_1$ and $e_1 = (u_1, v_1) \in E_1$. Then*

1. *If $G_1 \cong G_2$, then there is a choice ρ_2, v_2, e_2 such that*
   $canon(G_1, \rho_1, v_1, e_1) = canon(G_2, \rho_2, v_2, e_2)$.
2. *Let ρ_2 be a combinatorial embedding of G_2 and let $v_2 \in V_2$, $e_2 = (u_2, v_2) \in E_2$. If $canon(G_1, \rho_1, v_1, e_1) = canon(G_2, \rho_2, v_2, e_2)$ then $G_1 \cong G_2$.*

Proof. If $G_1 \cong G_2$ then there is a bijection $\phi : V_1 \to V_2$. We first show that the corresponding embedding of G_2 can be obtained from ρ_1 and ϕ. Let $\rho_{1_v} = (v_1, \ldots, v_d)$ be the permutation of edges incident on v in ρ_1. Then the corresponding permutation of edges incident on $\phi(v)$ is $(\phi(v_1), \ldots, \phi(v_d))$. This gives a permutation of edges for each vertex in V_2, which form the corresponding combinatorial embedding ρ_2 for G_2.

Also, there is an isomorphism ϕ' between the 3-regular graphs G_1' and G_2'. Let $e_1 = (u, v) \in E_1$. Then $e_2 = (\phi(u), \phi(v)) \in E_2$. Let e_1 and e_2 be chosen as starting edges and v and $\phi(v)$ as starting vertices. Let their corresponding edges in G_1' and G_2' be $(u_j, v_i) \in E_1'$ and $(\phi'(u_j), \phi'(v_i)) \in E_2'$ respectively. Let T_1 and T_2 be the output sequences in Step 3 of Algorithm 2. If a copy of a vertex $w \in V_1$ occurs at a position l in T_1 then a copy of $\phi(w) \in V_2$ occurs at position l in T_2 as the oriented graphs (G_1', ρ_1') and (G_2', ρ_2') are isomorphic, and the same UXS is used for their traversal from corresponding starting edges and starting vertices. Thus the labellings obtained from the sequences have the property that $u \in V_1$ and $\phi(u) \in V_2$ get identical labels for all the vertices $u \in V_1$. This gives the same lexicographically increasing sequence of edges for G_1 and G_2, and hence $\sigma_1 = \sigma_2$.

Now we prove the converse. Let $\sigma_1 = \sigma_2 = \sigma$. The labels of the vertices in σ are a relabelling of vertices of V_1 and V_2, with the property that $\forall (i, j) \in \sigma : (i, j) \in E_1 \Leftrightarrow (i, j) \in E_2$. These relabellings are permutations, say π_1 and π_2 of V_1 and V_2. Then $\pi_1 \cdot \pi_2^{-1} : V_1 \to V_2$ is an edge-preserving bijection *i.e.* an isomorphism between G_1 and G_2. ■

Clearly, each of the above steps can be performed in FL. This proves Theorem 2.1.

2.3 Discussion

We have shown that 3-connected planar graphs can be canonized in log-space which, along with the known log-space hardness, implies that 3-connected planar graph isomorphism and canonization are L-complete.

In Chapter 3, we describe a log-space algorithm for isomorphism and canonization of planar graphs, which uses the algorithm described in this chapter.

3

Planar Graph Isomorphism and Canonization

Planar graphs form an important class of graphs. We study the complexity of graph isomorphism and canonization on this class of graphs. The problem definitions and a brief overview of known results is given in Section 1.2.1.

In Chapter 2, a log-space algorithm for isomorphism and canonization of 3-connected planar graphs is described. In this chapter, we give a log-space algorithm for planar graph isomorphism and canonization. This improves the previously known upper bound of AC^1 of [72]. Also, as isomorphism and canonization of trees is L-complete [62, 50, 69], our algorithm implies that planar graph isomorphism and canonization are L-complete thereby settling the complexity of these two problems on planar graphs. This result is interesting as the known lower bound for isomorphism of general graphs is GapL [86].

3.1 A Brief Overview

We give an overview of the canonization algorithm here. In fact, we describe an ordering algorithm that provides a total order on the input graphs. Thus, if two graphs are isomorphic, it returns equality; otherwise it provides an order between them. We call this order as *isomorphism order*. Once it is clear how to get a total order on the graphs, constructing the canon for a graph is quite straight forward.

It suffices to consider simple, undirected graphs *i.e.* undirected graphs without parallel edges or loops. For isomorphism of planar graphs that are not simple, there are log-space many-one reductions to isomorphism of simple planar graphs (cf. [54]). Also, if the given graphs are not connected, the connected components can be identified in L using the algorithm for undirected reachability by [77]. Then the connected components of the two graphs are pairwise compared for isomorphism. Keeping track of the pairwise comparisons using counters suffices to find the isomorphism order of the two graphs, and can be done in L. Henceforth, we assume that the input to the algorithm consists of two simple, undirected, connected planar graphs.

The canonization algorithm consists of the following steps:

1. Decompose the given planar graph into its biconnected components and construct a *biconnected component tree*. This step is known to be in log-space and is described in Section 3.2.

2. Decompose biconnected planar components into their triconnected components to obtain a *triconnected component tree* in log-space. This is essentially a parallel implementation of the sequential algorithm of [45] (Section 3.3).

3. Invoke the canonization algorithm for 3-connected planar graphs described in Chapter 2 to canonize the triconnected components of the graph.

4. Canonize biconnected planar graphs using their triconnected component trees. This step crucially uses Lindell's algorithm [62] for tree-canonization. Section 3.5 contains the details of this step. A brief overview of Lindell's algorithm is given in Section 3.4.

5. Canonize the planar graph using its biconnected component tree, by substituting the biconnected components with their triconnected component trees (Section 3.6).

We show that each of the above steps can be done in FL. It is necessary that the decompositions in Steps 1 and 2 be *canonical*, *i.e.* dependent only on the graph and not on its representation.

Remark 3.1 *Our procedures for Steps* 1 *and* 2 *do not use the properties of a planar graph, and hence are applicable to general graphs.*

3.2 Construction of Biconnected Component Tree

To construct a *biconnected component tree* of the given connected planar graph G, it is first decomposed into its biconnected components as follows:

Decomposition into Biconnected Components: The input is a connected planar graph. Recall that an *articulation point* in a connected graph is a vertex whose removal splits the graph into two or more connected components. The decomposition procedure along with its complexity analysis is given in the following lemma. As stated earlier, the procedure does not use planarity and is hence applicable to all graphs.

Lemma 3.2 *The decomposition of a connected graph into its biconnected components can be done in* FL.

Proof. We first describe the decomposition procedure and then the complexity analysis.

The Procedure: The decomposition of a connected graph $G = (V, E)$ into its biconnected components is done in the following steps:

1. Identify and output all the articulation points in the graph.
2. Identify the connected components formed by the removal of all the articulation points. Put a copy of each of the articulation points into the components formed by its removal. Put the corresponding edges as well.

Correctness and Complexity analysis: We show that each of the above steps can be implemented in FL, and it correctly lists the biconnected components of G.

1. To check whether a vertex $v \in V$ is an articulation point, remove v from the graph and for each pair of vertices $\{a, b\}$, check whether a is reachable from b. This can be done in L by making oracle queries to the undirected reachability algorithm of [77]. If the answer is negative for any pair of vertices, then v is an articulation point. In this case, output v. Cycle through all the vertices $v \in V$ and output their labels if they are articulation points.

   Although this procedure has a recursive nature, it can be implemented in parallel due to the fact that the biconnected components formed are the same irrespective of the order of removal of articulation points. Moreover, an articulation point remains an articulation point even after the removal of other articulation points.

2. Identification of the biconnected components is actually done in two steps. Now, the input is the graph G and a list of articulation points of G. Identification of the biconnected components is based on the fact that two biconnected components never share an edge, as they can share at most one vertex. Therefore, a log-space transducer cycles through every pair of edges $e = \{u, v\}$ and $e' = \{u', v'\}$ and checks whether every two vertices from $\{u, v, u', v'\}$ remain reachable even after the removal of all of the articulation points from the list. This can be checked in L by oracle access to undirected reachability.

   Note that some or all of the end-points of e and e' may be articulation points. Further, if say v or v' is an articulation point, then it is also removed while checking whether say u and u' remain reachable after the removal of all of the articulation points. If every two vertices from $\{u, v, u', v'\}$ remain reachable after the removal of all of the articulation points, then e and e' are called *inseparable edges*. Such inseparable pairs of edges are output.

   Another log-space transducer identifies the components from this list. Note that the biconnected components are indeed the equivalence classes of edges under the *inseparability* relation described above. The input to the transducer is the list of inseparable pairs of

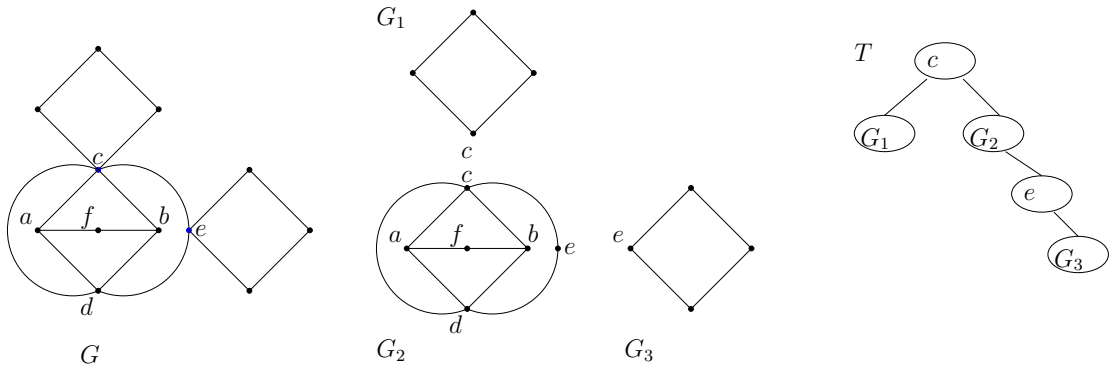

Figure 3.1: Decomposition into Biconnected Components

edges. To list the components, the log-space transducer cycles through all the edges in lexicographically increasing order (according to the given labels of vertices of G). For an edge e, it checks whether e forms an inseparable pair with an edge e' which is lexicographically smaller than e. If so, the component containing e must have been output earlier. Otherwise it lists all the edges that form inseparable pairs with e. These edges together form one biconnected component.

■

Construction of Biconnected Component Tree: Once the biconnected components of a connected planar graph are identified, it is straight forward to construct the biconnected component tree. The biconnected component tree is defined as follows:

Definition 3.3 *Given a connected graph G with its set of articulation points $A = \{a_1, \ldots, a_r\}$ and set of biconnected components $B = \{B_1, \ldots, B_s\}$, define a graph $G' = (V', E')$ where $V' = A \cup B$ and $E' = \{\{a_i, B_j\} | a_i$ has a copy in $B_j\}$. We call A as* articulation point nodes (or a-nodes) *and B as* biconnected component nodes (or b-nodes) *in G'. It can be seen that the graph G' defined above is a tree, with all the leaves of G' as b-nodes. This is called the* biconnected component tree *of G. Moreover, G' can be constructed in* FL.

Figure 3.1 shows an example of the decomposition.

3.3 Construction of Triconnected Component Tree

For each biconnected component in the biconnected component tree constructed by the procedure described in Section 3.2, we construct a triconnected component tree. For constructing

triconnected component tree, a biconnected component has to be decomposed into *triconnected components* by removing *separating pairs*. A pair of vertices is a *separating pair* if its removal breaks the biconnected graph into two or more components.

Decomposition into Triconnected Components A *triconnected component* of a biconnected graph is a 3-connected graph, or a simple cycle, or a 3-bond. A 3-*bond* is a graph on two vertices joined by three edges. A biconnected component can be decomposed by removing separating pairs. A sequential algorithm for recursive removal of separating pairs is given by [45]. Whenever a separating pair $\{a, b\}$ is removed from a graph, its copy appears in all the components formed. Further, each of the components formed contain the edge $\{a, b\}$, irrespective of whether it is an edge in the original graph. Such edges are called *virtual edges*. To distinguish virtual edges which are edges in the original graph from those which are not, 3-bonds are introduced. Thus there is a 3-bond corresponding to a separating pair $\{a, b\}$, if $\{a, b\}$ is an edge in the original graph.

After removing the separating pairs, the sequential algorithm of [45] combines simple cycles, if they are split at an intermediate step. This ensures uniqueness of the decomposition [67]. To get this decomposition in log-space, we do not remove separating pairs from a simple cycle. Thus simple cycles are also considered as triconnected components.

It is immediate that a procedure similar to that in Section 3.2 needs to be designed, where separating pairs are removed and triconnected components are identified. However, there is the following difficulty: Unlike in the case of articulation points, two or more separating pairs may *conflict* with each other. Two separating pairs $\{u, v\}$ and $\{x, y\}$ are said to be *conflicting* if, on removal of $\{u, v\}$, $\{x, y\}$ no longer remains a separating pair. Figure 3.2 shows an example of such conflicting separating pairs: *e.g.* $\{a, c\}$ and $\{b, d\}$ are conflicting separating pairs. If two

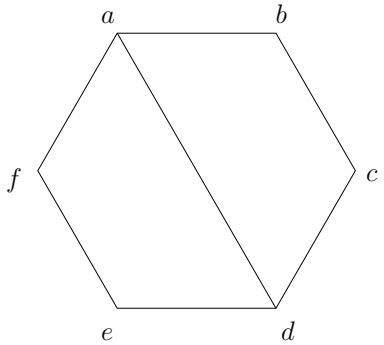

Figure 3.2: Conflicting Separating Pairs

separating pairs conflict, they can not be removed simultaneously. To overcome this difficulty,

we characterize a set of *non-conflicting* separating pairs and remove only this set. The following lemma gives this characterization, and shows that the decomposition obtained by removing such separating pairs is unique. The basic idea is that, if the vertices of a separating pair are connected by three or more vertex-disjoint paths, then such a separating pair cannot be separated by removal of any one of the other separating pairs. Such a separating pair is called a 3-*connected separating pair*.

Lemma 3.4 *In a simple undirected biconnected graph G, the removal of* 3*-connected separating pairs gives a unique decomposition, irrespective of the order in which they are removed.*

Proof. Let G be a biconnected graph with 3-connected separating pairs $\{s_1, \ldots, s_k\}$.

Consider a sequential removal of these separating pairs from G in an arbitrary order, say $\sigma = (s_1, \ldots, s_k)$. Let $H_1, \ldots, H_l$ be the biconnected components formed after removal of $s_1, \ldots, s_{i-1}$ where $i < k-1$, and let H_j be the component containing s_i. Let $s_i = \{u, v\}$ and $s_{i+1} = \{w, x\}$ (not necessarily all distinct). If s_{i+1} is not contained in H_j, then we get the same components by interchanging their positions in σ. So consider the case when they are contained in the same biconnected component, say H_j. Assume that s_i and s_{i+1} are 3-connected separating pairs in H_j as well, and hence there are at least three vertex-disjoint paths between u, v and between w, x. We prove that interchanging the positions of s_i and s_{i+1} in σ gives the same decomposition.

As $\{u, v\}$ are 3-connected, there are three vertex-disjoint paths between them, say ρ_1, ρ_2, ρ_3, such that vertices on at least one of them, say those on ρ_1, are separated from the vertices on the other two on removing s_i. We refer to this as ρ_1 being separated from ρ_2 and ρ_3 on removal of s_i. Let $X = \rho_1 \cup \rho_2 \cup \rho_3$, where X does not contain the vertices u and v. Consider the case when s_{i+1} is removed before s_i. We first prove that s_i remains a 3-connected separating pair even after removal of s_{i+1}:

1. **Case 1:** $|\{w, x\} \cap X| \leq 1$**.** Thus none or only one of w and x, say w, lies in X. Then the paths ρ_1, ρ_2, ρ_3 remain intact even after the removal of s_{i+1}, since a copy of w is retained on ρ_1 after removing s_{i+1}. Thus s_i continues to be a 3-connected separating pair.

2. **Case 2:** $w \in \rho_1, x \in \rho_2 \cup \rho_3$**.** In this case, removal of s_i can not separate ρ_1 and $\rho_2 \cup \rho_3$, since w and x are 3-connected. This contradicts the assumption that ρ_1 is separated from ρ_2 by removal of s_i.

3. **Case 3:** $w, x \in \rho_2 \cup \rho_3$**.** In this case, even after removal of s_{i+1}, ρ_1 can be separated from ρ_2 and ρ_3 by removal of s_i, and thus s_i continues to be a 3-connected separating pair.

4. **Case 4:** $w, x \in \rho_1$**.** In this case, the part of ρ_1 between w and x is replaced by a virtual edge on removal of s_{i+1}, but it still remains separable from ρ_2 and ρ_3. Thus s_i is still a 3-connected separating pair.

We also need to prove that, two vertices y, z lie in the same component on removing s_i followed by s_{i+1} if and only if they lie in the same component after removing s_{i+1} before s_i. This clearly holds if y, z form a 3-connected separating pair. So consider the case when they do not form a 3-connected separating pair. Further, assume that they are in the same component till $s_1, \ldots, s_{i-1}$ are removed, but are separated by the removal of s_i. In this case, y and z are in different components, say H_{j+1} and H_{j+2} respectively, when s_i is removed and s_{i+1} is not yet removed. However, at this stage, the vertices of s_{i+1} are in the same component, say H_{j+1}. Thus all the paths from z to w and x pass through u or v, and hence y and z are in different components even if s_{i+1} is removed before s_i.

This shows that in a sequence of removal of 3-connected separating pairs, two adjacent pairs can be interchanged. But a permutation on 3-connected separating pairs can be obtained from another one by interchanging adjacent separating pairs several times. This shows that 3-connected separating pairs uniquely partition the graph into triconnected components. Thus they can be removed in parallel. ■

It is clear that a simple cycle does not have a 3-connected separating pair. Therefore it is included in the set of triconnected components, without decomposing it.

Now it remains to show that the decomposition can be computed in FL.

Lemma 3.5 *The decomposition of a biconnected graph into its triconnected components can be computed in* FL.

Proof. We first describe the procedure and then the correctness and complexity analysis.

The Procedure: Given a biconnected graph $G = (V, E)$, the decomposition into triconnected components is similar to that in Section 3.2. The procedure has the following steps:

1. Identify and output all the 3-connected separating pairs in G.
2. Identify the connected components formed by the removal of all the 3-connected separating pairs. Put a copy of a separating pair into each of the components formed by its removal. Join the vertices in a copy of a separating pair by edges. We refer to these edges as *virtual edges*.
3. Output a 3-*bond* for each of the separating pairs listed in Step 1, if the vertices in that separating pair have an edge between them in G. All the copies of each separating pair are joined by virtual edges in Step 2. 3-bonds are included so as to keep track of which of the virtual edges are indeed present in the original graph.

Correctness and Complexity Analysis We show that each of the above steps can be implemented in FL, and the procedure correctly lists all the triconnected components of G.

1. To check whether a pair of vertices $\{u, v\}$ is a separating pair, remove it from G, cycle through every pair $\{a, b\}$ of the remaining vertices and check whether a and b are unreachable in $G \setminus \{u, v\}$. If this happens for any two vertices a and b, then $\{u, v\}$ is a separating pair. To check whether it is a 3-connected separating pair, cycle through all the pairs of vertices $a, b \in V \setminus \{u, v\}$, and check whether u and v remain reachable even after the removal of $\{a, b\}$. If there is no pair $\{a, b\}$ that disconnects u and v, then $\{u, v\}$ is a 3-connected separating pair. Output $\{u, v\}$.

2. Identification of triconnected components is done in two steps. Now the input is the biconnected graph G and a list of 3-connected separating pairs. We use the fact that two triconnected components never share three or more vertices. Thus a triple of vertices uniquely identifies a triconnected component. Therefore a log-space transducer cycles through each triple $\{a, b, c\}$ of vertices, and checks whether every two of them remain connected even after the removal of all the 3-connected separating pairs. If so, the triple is called an *inseparable triple*. Such triples are output.

   To identify triconnected components, a log-space transducer first outputs the lexicographically smallest inseparable triple, say $\{a, b, c\}$ and cycles through all the vertices v to check whether v forms inseparable triples with each pair of vertices in $\{a, b, c\}$. Thus v lies in the same component as that of $\{a, b, c\}$ if and only if $\{v, a, b\}$, $\{v, b, c\}$, and $\{v, a, c\}$ are all inseparable triples. This gives the first triconnected component. Then it considers lexicographically next smallest triple which has a vertex u that is not inseparable from $\{a, b, c\}$. This continues until all the triples are considered. The output is a list of triconnected components.

■

Construction of Triconnected Component Tree: Once the triconnected components of a biconnected graph are identified, it is straight forward to construct the triconnected component tree. The triconnected component tree is defined as follows:

Definition 3.6 *Given a biconnected graph G with its set of 3-connected separating pairs $X = \{x_1, \ldots, x_r\}$ and set of triconnected components $Y = \{Y_1, \ldots, Y_s\}$, define a graph $G' = (V', E')$ where $V' = X \cup Y$ and $E' = \{\{x_i, Y_j\} | x_i$ has a copy in $Y_j\}$. We call X as* separating pair nodes (or s-nodes) *and Y as* triconnected component nodes (or t-nodes) *in G'.*

It is easy to see that G' is a tree. This is called the triconnected component tree *of G.*

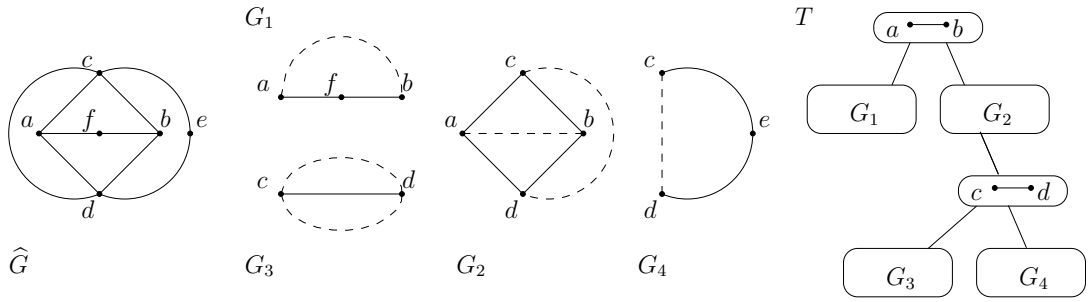

Figure 3.3: Decomposition into Triconnected Components

All the leaves of G' are t-nodes. Moreover, G' can be constructed in FL. Figure 3.3 shows an example.

3.4 Overview of Lindell's Algorithm

For isomorphism ordering of two graphs, we define an isomorphism ordering algorithm on their biconnected and triconnected component trees. This crucially uses Lindell's log-space algorithm for isomorphism ordering on trees. This algorithm is briefly described in this section.

The algorithm is based on an order relation $\leq$ on trees defined below. The order relation has the property that two trees S and T are isomorphic if and only if $S = T$. Because of this property it is called a *canonical order*. Clearly, an algorithm that decides the order can be used as an isomorphism test. Lindell also showed how to extend such an algorithm to compute a canon for a tree in log-space.

3.4.1 Isomorphism-ordering of two trees

Let S and T be two trees with roots s and t, respectively. The canonical order is defined as follows. $S < T$ if

1. $|S| < |T|$, or
2. $|S| = |T|$ but $\#s < \#t$, where $\#s$ and $\#t$ are the number of children of s and t, respectively, or
3. $|S| = |T|$ and $\#s = \#t = k$, but $(S_1, \ldots, S_k) < (T_1, \ldots, T_k)$ lexicographically, where it is inductively assumed that $S_1 \leq \ldots \leq S_k$ and $T_1 \leq \ldots \leq T_k$ are the ordered subtrees of S and T rooted at the k children of s and t, respectively.

The comparisons in steps 1 and 2 can be made in log-space. Lindell proved that even the third step can be performed in log-space using *two-pronged* depth-first search, and *cross-comparing* only a child of S with a child of T. This is briefly described below:

- Find the number of minimal sized children of s and t. If these numbers are different then the tree with a larger number of minimal children is declared to be smaller. If equality is found then remember the minimal size and check for the next size. This process is continued till an inequality in the sizes is detected or all the children of s and t are exhausted.

- If s and t have the same number of children of each size then assume that the children of s and t are partitioned into *size-classes* (referred to as *blocks* in [62]) in the increasing order of the sizes of the subtrees rooted at them. That is, the k children of s and t are partitioned into groups, such that the i-th group is of cardinality k_i and the subtrees in the i-th group all have size N_i, where $N_1 < N_2 < \cdots$. It follows that $\sum_i k_i = k$ and $\sum_i k_i N_i = n - 1$.

  Consider the partition of the children of s and t into different classes, depending on their isomorphism relation. We refer to these classes as *isomorphism-classes*. Clearly, isomorphism-classes are a refinement of size-classes. After Steps 1 and 2, the children are partitioned into size-classes, and Step 3 gives the refinement into isomorphism-classes. For this, the children in each size-class are recursively compared as follows: Let k be the number of children in the size-class currently being considered.

  Case 1, $k = 0$**.** Hence s and t have no children. They are isomorphic as all one-node trees are isomorphic. We conclude that $S = T$.

  Case 2, $k = 1$**.** Recursively consider the grand-children of s and t. No space is needed for making the recursive call. Once the recursive call is finished, the execution can be resumed by just looking at the node for which the recursive call had been made.

  Case 3, $k \geq 2$**.** For each of the subtrees S_j, compute its *order profile*. The order profile consists of three counters, $c_<$, $c_>$ and $c_=$. These counters indicate the number of subtrees of T in the size-class of S_j that are respectively smaller than, greater than, and equal to S_j. The counters are computed by comparing S_j with each of the T_i's, one at a time. Whenever an S_j with $c_< = 0$ is found, its $c_=$ counter gives the number of children of T that are isomorphic to S_j. They form the first isomorphism-class. This $c_=$ counter is remembered, and the same procedure is repeated for subtrees rooted at the children of t. If the $c_=$ counters are equal in these two cases, then S and T have the same number of isomorphic subtrees in this isomorphism-class, and we proceed to the next isomorphism-class. Otherwise the tree with a larger number of subtrees in this isomorphism-class is considered to be smaller.

To proceed to the next isomorphism-class, the above $c_=$ is remembered as *threshold* h, and the next isomorphism-class is formed by those children of s and t which have $c_<$ counter exactly equal to h. While going to the next isomorphism-class, h is updated by adding the current $c_=$ counter to it. The size-class is processed completely when $h = k$, and we go to the next size-class.

If all the size-classes are processed this way, without discovering an inequality, then we can conclude $S = T$.

3.4.2 Space-complexity Analysis

Let $|S| = |T| = n$, and the size-classes have subtrees of sizes $N_1 < N_2 < \ldots N_r$, where r be the number of size-classes. If the ith size-class has k children of s and t each, then $N_i \leq \frac{n}{k}$. The current order-profile and threshold need to be stored while making recursive comparison of two subtrees, which take $O(\log k)$ space. Since $\sum_i k_i N_i \leq n$, the following recursion equation for the space complexity holds. For each new size-class, the work-tape allocated for the former computations can be reused.

$$\mathcal{S}(n) = \max_i\{\mathcal{S}(N_i) + O(\log k_i)\} \leq \max_i\{\ \mathcal{S}\left(\frac{n}{k_i}\right) + O(\log k_i)\},$$

where $k_i \geq 2$ for all i. It is not hard to see that $\mathcal{S}(n) = O(\log n)$. Note that if $N_i > \frac{n}{2}$, then it is the only subtree in its size-class, and no space is used while making the recursive call.

3.4.3 Canonization of trees

Once it is clear how to get tree isomorphism order, it is straight forward to construct the canon of a tree. For this, Lindell's algorithm traverses the tree in depth-first order, where the subtrees rooted at the children of a node are traversed according to their isomorphism order. A '$[$' is printed while going down a subtree, $\circ$ while going over a subtree, and '$]$' while going up from a subtree.

3.5 Isomorphism and Canonization of Biconnected Planar Graphs

For isomorphism ordering of biconnected planar graphs, we define an isomorphism order on their triconnected component trees. Let T and T' be the triconnected component trees corresponding to two biconnected planar graphs G and H, respectively. We root T and T' at s-nodes $p = \{a, b\}$ and $p' = \{a', b'\}$, respectively, which are chosen arbitrarily. As there are only $O(n)$ s-nodes, an isomorphism test can cycle through all the possibilities of rooting these trees. The rooted trees are denoted as $T_{\{a,b\}}$ and $T_{\{a',b'\}'}$. They have s-nodes at odd levels and t-nodes at even levels.

Recall that s-nodes and t-nodes are separating pair nodes and triconnected component nodes respectively. Figure 3.4 shows two trees to be compared.

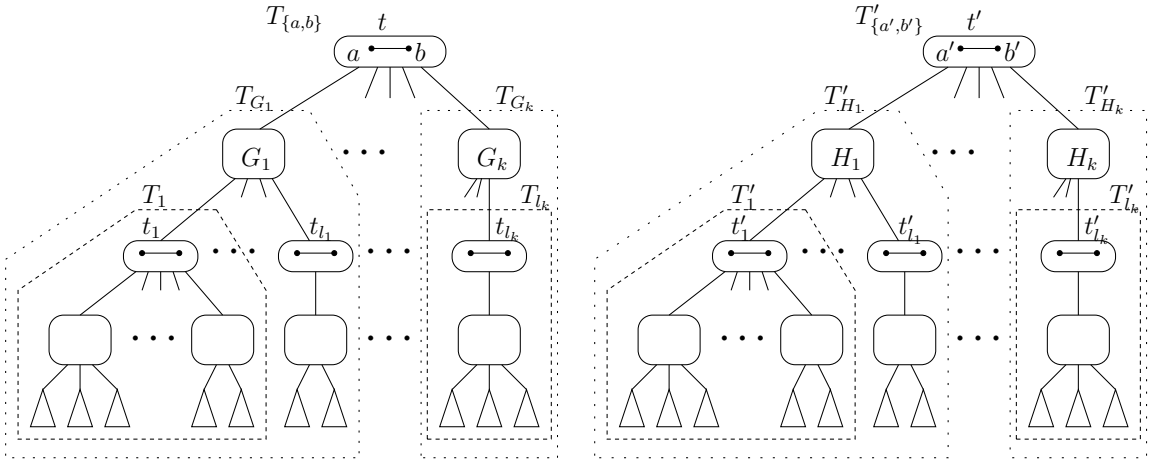

Figure 3.4: Triconnected component trees.

In Chapter 2, a log-space canonization algorithm for 3-connected planar graphs is described. Note that, an obvious way to canonize a triconnected component tree would be to invoke Algorithm 2 from Chapter 2 along with Lindell's algorithm.

However, this is not sufficient, since two non-isomorphic graphs may have the same triconnected component trees. See for example, Figure 3.5. The trees are the same, with corresponding

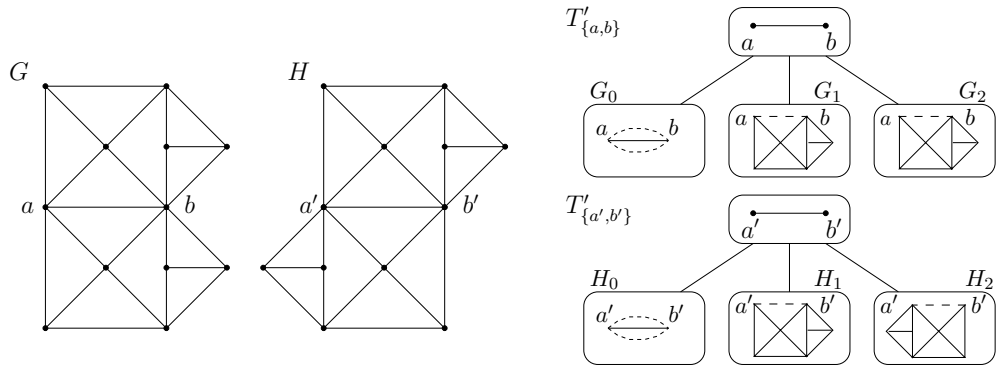

Figure 3.5: Non-isomorphic graphs with the same triconnected component trees

nodes being isomorphic. If a distinct color is given to the virtual edge $\{a, b\}$ in G_1, G_2 and to $\{a', b'\}$ in H_1, H_2, then any isomorphism between them maps $\{a, b\}$ to $\{a', b'\}$. Let ϕ_1 be such an isomorphism from G_1 to H_1. Then $\phi_1(a) = a'$ and $\phi_1(b) = b'$. Let ϕ_2 be an isomorphism from G_2 to H_2. Then $\phi_2(a) = b'$ and $\phi_2(b) = a'$. Due to this inconsistency between ϕ_1 and ϕ_2,

they can not be combined to get an isomorphism between G and H.

Remark 3.7 *The above example shows that, although non-isomorphic graphs can have same triconnected component trees, the vertex-labels have enough information to detect the non-isomorphism. This is a crucial fact in our algorithm.*

Remark 3.8 *As stated above, to ensure that the parent virtual edges of two components are mapped to each other by all the isomorphisms between the two components, the parent virtual edge can be given a distinct color. As only one distinct color is required, this coloring can be achieved by defining the parent virtual edge to be* larger *than any other edge in the component.*

As described next, while constructing a canonical code for a triconnected component, we get the parent virtual edge as the first edge in its canonical list of edges. This also ensures that the parent virtual edges of two components are always mapped to each other, and thus an explicit coloring is not required.

Therefore, we introduce an additional step in the isomorphism order defined in Lindell's algorithm. The details of all the steps in the isomorphism ordering are given below. Steps 1 and 2 are exactly same as in Lindell's algorithm. Step 3 needs some modifications, as the individual nodes of the tree are s-nodes or t-nodes, and the t-nodes have to be compared using the algorithm for isomorphism of 3-connected planar graphs.

3.5.1 Comparison of two triconnected component trees

Denote the isomorphism ordering of triconnected component trees by $<_{\mathrm{T}}$. In the isomorphism ordering, $T_{\{a,b\}} <_{\mathrm{T}} T'_{\{a',b'\}}$ if, in any of the steps below, $T_{\{a,b\}}$ is found to be *smaller than* $T'_{\{a',b'\}}$:

1. **Comparison of sizes:** In the first step of Lindell's algorithm, the sizes of the two trees are compared, and the tree with a smaller size is considered to be smaller. We define the size of a triconnected component tree as follows:

   Definition 3.9 *For a triconnected component tree T, the* size of a t-node C *of T is the number n_C of vertices in C. Note that the vertices in an s-node are counted in every t-node adjacent to it. The* size of an s-node *is* 2.

   The size of the tree T*, denoted by $|T|$, is the sum of the sizes of its t-nodes and s-nodes.*

   Note that the size of a triconnected component tree is polynomial in the size of the graph.

2. **Comparison of the number of children:** If the sizes of the two trees are equal, then the number of children of the roots of the two trees are compared. The tree whose root has

a smaller number of children is considered to be smaller. We denote the roots of the two trees as $t = \{a, b\}$ and $t' = \{a', b'\}$.

3. **Recursive comparison of subtrees:** If equality is found in the above two steps, then we make recursive comparisons of the subtrees of $T_{\{a,b\}}$ and $T'_{\{a',b'\}}$. For convenience, we denote $T_{\{a,b\}}$ and $T'_{\{a',b'\}}$ as T and T' respectively.

   After steps 1 and 2 above, we can assume that the children of p and p' are partitioned into *size-classes i.e.* partitioned according to their sizes. The size-classes are arranged in the increasing order of the size of the subtrees in them. As in Lindell's algorithm, the first step here is to compare the number of children of p and p' in each of the size-classes. If an inequality is found, then the tree with more children in a smaller size-class is considered to be smaller. Here onwards, we assume that p and p' have the same number of children in each of the size-classes.

   The recursive comparison is carried out by storing order profiles, and an inequality is returned exactly as in Lindell's algorithm. However, the comparison of two subtrees is more complex. We describe the details of one comparison here.

   Let T_i and T'_j be the subtrees rooted at a child C of p and a child C' of p', respectively. Note that C and C' are t-nodes, and have a virtual edge corresponding to their parents $p = \{a, b\}$ and $p' = \{a', b'\}$ respectively. C and C' are triconnected components *i.e.* they can be either 3-connected planar graphs or cycles or 3-bonds. We define 3-*bond*$<_T$*cycle*$<_T$3-*connected planar graph*, and report an inequality accordingly, if C and C' are of two different types. Otherwise we have the following possibilities:

   (a) C, C' **are** 3**-bonds:** In this case, they are leaves and are always equal.

   (b) C, C' **are cycles or** 3**-connected planar graphs:** If $|C| < |C'|$, we return $C <_T C'$. Otherwise, we start constructing and comparing the canonical codes for C and C' in all the possible ways. The details are given below.

Construction and comparison of canonical codes of C and C':

- *If C, C' are cycles* then canonical codes of C and C' are constructed by traversing them in both clockwise and counter-clockwise directions, starting from the virtual edges $\{a, b\}$ and $\{a', b'\}$ respectively. Thus there are two possible ways of constructing canonical codes of each of C and C'.
- *If C, C' are* 3*-connected planar graphs* then their canonical codes are obtained by calling the procedure 2 described in Chapter 2 for canonization of 3-connected pla-

nar graphs. The canonical codes for C are obtained by making the following calls: $canon(C, \rho, a, \{a, b\})$, $canon(C, \rho, b, \{a, b\})$, $canon(C, \rho', a, \{a, b\})$, $canon(C, \rho', b, \{a, b\})$. Here ρ and ρ' are the two possible combinatorial embeddings of C (and similarly for C'). Recall that $\{a, b\}$ is the parent of C in the tree T. Thus the canonization of C is done with only the virtual edge corresponding to its parent as the starting edge, for both of its end-points as starting vertices, and for both the combinatorial embeddings.

Remark 3.10 *The $canon$ procedure needs to be modified such that it takes an additional parameter i and returns the ith edge of the canonical code, along with the canonical as well as original labels of the end-points of that edge. It can be seen that the modified procedure also works in* FL.

Thus there are two possible canonical codes for a cycle and four for a 3-connected planar graph. A counter cnt is maintained, and the ith edge of all the possible codes of C and C' is obtained when $cnt = i$. The ith edges in all the codes are compared.

At any step i, if a code is found to be *larger* than another code, then the larger code is eliminated. In the end, if all the codes of C (respectively C') are eliminated then we return $C' <_T C$ ($C <_T C'$).

At a step i of the construction of the codes, a code c is considered to be *larger* than a code c' of C', if they are equal upto step $i - 1$, and one of the following holds at step i:

- the ith edge of c has a lexicographically larger canonical label than the ith edge of c' or
- the ith edges of c and c' have the same canonical labels but the ith edge of c is a virtual edge corresponding to a child of C, whereas the ith edge of c' is not a virtual edge of C', or
- ith edges of c and c' are virtual edges $e = (u, v)$ and $e' = (u', v')$ corresponding to a child of C and C' each, with the same canonical labels to u, u' and to v, v' with the condition that the canonical label of u (respectively u') is smaller than that of v (v'), and either of the two holds: (i) the recursive comparison of the subtrees T_e and $T'_{e'}$ rooted at e and e' results in $T_e >_T T'_{e'}$, or (ii) the recursive comparison of T_e and $T'_{e'}$ results in equality, but the *orientation* of e in T_e is $v \rightarrow u$, whereas that of e' is $u' \rightarrow v'$ in $T_{e'}$. The notion of *orientation* is defined as follows:

Definition 3.11 (Orientation of a separating pair:) *Consider a triconnected component tree T rooted at a separating pair $p = \{a, b\}$. The* orientation of p *is said to be $a \rightarrow b$ in*

T if, in all the possible canonical labelings of vertices which lead to the lexicographically smallest canonical code of T, the canonical label of a is smaller than that of b.

The root p is said to have no orientation *in T, if the smallest canonical code (i.e. canon) of T can be obtained with multiple ways of giving canonical labels to the vertices, and some of them have a smaller label for a than that of b, and some of them have a smaller label for b than that of a.*

Note that this is not a constructive definition. How to obtain the orientation of a separating pair in the subtree rooted at it is described in the next step.

4. **Comparison of Orientations:** If equality is returned in all the three steps above, then we can assume that the children of p and p' are partitioned into *isomorphism-classes*. Two children are in the same isomorphism-class if and only if they are isomorphic. We can also assume that an isomorphism-class of children of p and that of p' have the same number of children, otherwise an inequality is returned in Step 3 itself.

   As shown by the example in Figure 3.5, an additional step is needed at this stage. Intuitively, this step is required to check that pairwise isomorphisms of the children of p and p' can be consistently extended to an isomorphism of the entire graphs. To ensure this, we obtain and compare the *orientations* of p and p' in T and T' respectively. This is described below:

   Arrange the isomorphism-classes of children of p and p' in the increasing order, where the order is the isomorphism order of the subtrees in them obtained in the above three steps, and consider one isomorphism-class at a time starting from the smallest one. Let these classes be $(I_1, \ldots, I_p)$ and $(I'_1, \ldots, I'_p)$ respectively for T and T'. Let I_i and I'_i be the isomorphism classes currently being compared.

   A child C of $p = \{a, b\}$ is said to *give an orientation* $a \rightarrow b$ to p, if all the uneliminated canonical codes of C are those where a is taken as the starting vertex. This corresponds to $canon(C, *, a, \{a, b\})$ if C is a 3-connected planar graph, and $*$ denotes any of the two embeddings of C. If C is a cycle, then this corresponds to the traversal of the form $(a, b, \ldots, a)$.

   For an isomorphism-class I_i, count the number O_{i_1} of children that give $a \rightarrow b$ orientation to p and the number O_{i_2} of the children that give $b \rightarrow a$ orientation to p. Obtain the counters for I'_i as well. I_i is said to give $a \rightarrow b$ (respectively $b \rightarrow a$) orientation to p if $O_{i_1} > O_{i_2}$ ($O_{i_2} > O_{i_1}$). Note that either each of the children in I_i and I'_i gives an orientation to its parent, or none of them gives an orientation, as they are all isomorphic. If none of the children gives an orientation to its parent or if $O_{i_1} = O_{i_2}$, then I_i is said to be *symmetric about* p, and does not give any orientation to p.

Find the smallest isomorphism-class I_i that is *not* symmetric about p. The orientation given by I_i to p is considered to be the *reference orientaion*. For isomorphism-classes $I_j, j > i$, O_{j_1} is the number of children that give the reference orientation to p and O_{j_2} that give the opposite orientation to p. The *orientation counter* for I_j is $O_j = (O_{j_1}, O_{j_2})$. The orientation counter O'_j for I'_j is obtained in a similar way.

The comparison in this step is a lexicographic comparison of the orientation counters. That is, $T <_{\mathrm{T}} T'$ if equality holds in Steps 1 to 3 above, and $(O_1, \ldots, O_p) < (O'_1, \ldots, O'_p)$ lexicographically. If an isomorphism-class is symmetric about the root, then its orientation counter is considered to be $(0, 0)$.

The steps involved in the isomorphism ordering are summarized below:

Summary of the steps in the isomorphism order. The isomorphism order of two triconnected component trees T and T' rooted at separating pairs $p = \{a, b\}$ and $p' = \{a', b'\}$ is defined to be $T_{\{a,b\}} <_{\mathrm{T}} T_{\{a',b'\}'}$ if:

1. $|T_{\{a,b\}}| < |T_{\{a',b'\}}|$ or
2. $|T_{\{a,b\}}| = |T_{\{a',b'\}'}|$ but $\#p < \#p'$ or
3. $|T_{\{a,b\}}| = |T'_{\{a',b'\}}|$, $\#p = \#p' = k$, but $(T_{G_1}, \ldots, T_{G_k}) <_{\mathrm{T}} (T'_{H_1}, \ldots, T'_{H_k})$ lexicographically, where we assume that $T_{G_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T_{G_k}$ and $T'_{H_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T'_{H_k}$ are the ordered subtrees of $T_{\{a,b\}}$ and $T'_{\{a',b'\}}$, respectively.
4. $|T_{(a,b)}| = |T'_{(a',b')}|$, $\#p = \#p' = k$, $(T_{G_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T_{G_k}) =_{\mathrm{T}} (T'_{H_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T'_{H_k})$, but $(O_1, \ldots, O_p) < (O'_1, \ldots, O'_p)$ lexicographically, where O_j and O'_j are the orientation counters of the j^{th} isomorphism-classes I_j and I'_j.

We say that two triconnected component trees T_e and $T'_{e'}$ are *equal according to the isomorphism order*, denoted by $T_e =_{\mathrm{T}} T'_{e'}$, if neither $T_e <_{\mathrm{T}} T'_{e'}$ nor $T'_{e'} <_{\mathrm{T}} T_e$ holds. The following theorem states that two trees are equal according to the isomorphism ordering defined above, precisely when the underlying graphs are isomorphic.

Theorem 3.12 *The biconnected planar graphs G and H are isomorphic if and only if there is a choice of separating pairs e, e' in G and H such that $T_e =_{\mathrm{T}} T'_{e'}$ when rooted at e and e', respectively.*

Proof. Assume that $T_e =_{\mathrm{T}} T'_{e'}$. The argument is an induction on the depth of the trees that follows the definition of the isomorphism order.

Base case: $d = 2$. In this case, all the children of e and e' are leaves. As $T_e =_{\mathrm{T}} T'_{e'}$, e and e' have equal number of isomorphic children. Since the orientation counters match for each isomorphism-class, the isomorphism of the individual t-nodes can be extended to the isomorphism of the entire graphs.

Induction step: Let d be the depth of T_e and $T'_{e'}$. As the s-nodes and t-nodes appear at alternate levels, the subtrees rooted at the separating pairs at the next level have depth at most $d-2$. Assume the result holds for subtrees upto depth $d-2$. Let $G_1, \ldots, G_k$ be the children of e and $H_1, \ldots, H_k$ be the children of H such that $\forall i : T_{G_i} =_{\mathrm{T}} T_{H_i}$. Therefore we have $G_i \cong H_i$, with the corresponding virtual edges having isomorphic subtrees rooted at them with the same orientations. Thus the pairwise isomorphism of the children of G_i and H_i can be extended to the graphs corresponding to the subtrees rooted at G_i and H_i, for each i. Now, since e and e' have matching orientation counters, it is easy to see that an isomorphism between their corresponding children can be extended consistently to the entire graph.

The reverse direction holds obviously as well. If G and H are isomorphic and an isomorphism maps the separating pair $\{a, b\}$ of G to the separating pair $\{a', b'\}$ of H, then this also gives a pairwise isomorphism between a child of $\{a, b\}$ and a child of $\{a', b'\}$. Therefore an induction on the depth of the tree immediately leads to $T_{\{a,b\}} =_{\mathrm{T}} T_{\{a',b'\}}$. ■

3.5.2 Implementation of the Isomorphism Ordering in FL

We analyse the space-complexity of the isomorphism ordering. The first two steps of the isomorphism ordering can be computed in log-space as in Lindell's algorithm [62]. We show that steps 3 and 4 can also be performed in log-space.

- **Case 1: Comparison of two subtrees rooted at s-nodes** While comparing two subtrees rooted at s-nodes p and p', the order-profiles, and the orientation-counters need to be stored while making cross-comparisons of their children. This needs $O(\log k)$ space, where k is the number of children of p and p' in a particular size-class. Moreover, the reference orientation needs to be stored, once an isomorphism-class that is not symmetric about p and p' is found. This takes 1 bit. The 1 bit can be stored whenever two subtrees being compared are of size at most $\frac{n}{2}$, where n is the size of the subtrees rooted at p and p'. However, it can not be stored while comparing a child of p of size larger than $\frac{n}{2}$, with the corresponding child of p'. Note that such a child is unique, and is referred to as *the large child* of p and p'.

  To get around this problem, we compare the large child of p with that of p' before comparing any other size-classes, and store the result of this comparison. This takes $O(1)$ bits.

As all the other children of p and p' are of size at most $\frac{n}{2}$, these $O(1)$ bits can be stored until the comparison of the subtrees rooted at p and p' is completed.

- **Case 2: Comparison of two subtrees rooted at t-nodes** Consider the comparison of two subtrees rooted at t-nodes C and C' in T and T' respectively. If C and C' are of different types or they are 3-bonds, the result can be returned immediately. Consider the case when both C and C' are cycles or 3-connected planar graphs of the same size.

  In this case, we need to construct all the possible canonical codes for them, with the virtual edge corresponding to their parent as the starting edge. Recall that there are two possible canonical codes each when C and C' are both cycles, and four possible canonical codes each when they are 3-connected planar graphs. Therefore $O(\log |C|)$ space is needed for the construction of the canonical codes. A recursive call is made for comparing a child q of C with a child q' of C' when the corresponding virtual edges are encountered simultaneously in a canonical code of C and of C'. The entire work-tape contents can not be stored while making the call, and we still need to ensure that we store enough information so as to resume execution after completing the call. This is done as follows:

  We store the following on the work-tape while making a recursive call:

  1. A bit-vector indicating which canonical codes are already eliminated (as they were larger than the rest). This needs 8 bits.
  2. A bit-vector indicating the two canonical codes for which the current recursive call is being made.
  3. One bit each indicating the direction which e and e' get in the respective canonical codes. Recall that the direction indicates which end-point of e (and of e') gets a smaller canonical label in the canonical code.

  Each edge occurs exactly once in the canonical list of edges of C and C'. Therefore, after completing the call for e and e', we update the bit-vector of eliminated canons depending on the result of the recursive call, remember the virtual edges e and e' for which the call was made, and reconstruct the uneliminated canonical codes till e and e' are encountered in the respective canonical codes. This is precisely the point at which the recursive call was made. Thus, $O(1)$ space is needed while making the recursive call. $O(1)$ space can be used, unless e and e' are large children of C and C'.

  To get around this difficulty, we use the same method as in the previous case. Before starting the comparison of the subtrees rooted at C and C', check if they have a large child, and if so, compare the large children a priori and store the result of the comparison

in $O(1)$ space. This result is stored till the comparison of the subtrees rooted at C and C' is completed.

As seen above, while comparing two trees of size N, the algorithm uses no space for making a recursive call for a subtree of size larger than $N/2$. If a size-class has $k > 2$ children, then it uses $O(\log k)$ space while making a recursive call. In this case, each of the subtrees are of size at most $\frac{N}{k}$. If a size-class has only one child, and it is not a large child, then the algorithm uses $O(1)$ space. Hence we get the same recurrence for the space $\mathcal{S}(N)$ as that in Lindell's algorithm:

$$\mathcal{S}(N) \leq \max_j \mathcal{S}\left(\frac{N}{k_j}\right) + O(\log k_j),$$

where $k_j \geq 2$ for all j. Thus $\mathcal{S}(N) = O(\log N)$.

As the size N of each of the triconnected component trees T and T' is bounded by a polynomial in the number of vertices n in the graphs G and H, the space used is $O(\log n)$.

Theorem 3.13 *The isomorphism order between two triconnected component trees of biconnected planar graphs can be computed in* FL.

3.5.3 Canonization of Biconnected Planar Graphs

Once it is clear how to get the isomorphism order on triconnected component trees in log-space, it is straight forward to construct the canon similar to that in Lindell's algorithm described in 3.4.3. The details are given below.

Theorem 3.14 *A biconnected planar graph can be canonized in log-space.*

Proof. We assume that the canonization algorithm has oracle access to the isomorphism ordering algorithm (Section 3.5.1) and to the 3-connected planar graph canonization algorithm (Chapter 2). The root of the triconnected component tree T is chosen from all the s-nodes in T by cycling through all of them and choosing the one which gives the smallest tree according to the isomorphism order. Here onwards we assume that T is rooted at such a t-node, say $\{a, b\}$.

The canonization algorithm traverses T in the tree-isomorphism order as in Lindell's algorithm [62], outputting the canonical code of each of the nodes in pre-order, which is a list of edges including the virtual edges. In the second step, the final canon is computed from the canonical list, by relabelling the vertices according to the order of their first occurrence in this list. The details are given below.

Canonical list of a subtree rooted at an s-node: Consider a subtree $T_{(a,b)}$ rooted at the s-node $\{a, b\}$. The canonization algorithm computes the reference orientation of $\{a, b\}$ and outputs the edge in this direction. Then it recursively outputs the canonical lists of the subtrees of $\{a, b\}$ according to the increasing isomorphism order. Among isomorphic siblings, those which give the reference orientation to the parent are considered before those which give the reverse orientation. We denote this canonical list of edges $l(T, a, b)$. If the subtree rooted at $\{a, b\}$ does not give any orientation to $\{a, b\}$, and if it is not the root of T, then it appears in some t-node C. In this case, that orientation for $\{a, b\}$ is taken to be the order of the canonical labels of a and b in the canonical code of C.

Canonical list of a subtree rooted at a t-node: Consider the subtree T_C rooted at the t-node C. Let $\{a, b\}$ be the parent separating pair of C with reference orientation $a \rightarrow b$. If C is a 3-bond then output its canonical list $l(C, a, b)$ as (a, b). If C is a cycle then it has a unique canonical list with respect to the orientation (a, b), that is $l(C, a, b)$. Now we consider the case when C is a 3-connected component. Then C has two canonical codes with respect to the orientation (a, b), one for each of the two embeddings. Query the oracle for the embedding that leads to the lexicographically smaller canonical list and output it as $l(C, a, b)$. Recursively output the canonical lists of edges for each of the subtrees rooted at the children of C, in the order in which the corresponding virtual edges are present in $l(C, a, b)$.

The list of edges computed as described above still has the vertex-labels from the graph G. The vertices are relabelled according to the order of their first occurrence in this list, and a list of edges is output in the lexicographically increasing order. This is the canon of the given biconnected planar graph G. It is easy to see that the canon can be computed in FL. ■

3.6 Isomorphism and Canonization of Connected Planar Graphs

This section contains the description of the isomorphism order for connected planar graphs. To determine the isomorphism order of two connected planar graphs, we define an isomorphism order on their biconnected component trees. The construction of the biconnected component trees is given in Section 3.2. Recall that the biconnected component tree of a connected planar graph G has articulation point nodes called *a-nodes* and biconnected component nodes called *b-nodes*. These b-nodes are replaced with their respective triconnected component trees, and a new tree-like structure is constructed. An isomorphism order on this structure is then defined.

3.6.1 The Tree-Structure

In the isomorphism order of triconnected component trees, the canonization algorithm for 3-connected planar graphs was used as an oracle. However, in case of biconnected component trees, the canonization algorithm for biconnected planar graphs can not be used this way due to the following reasons:

1. In a triconnected component tree, once the parent of a t-node is fixed, it has only four possible canonical codes. In the case of a biconnected component tree, a b-node can have as many canonical codes as the number of separating pairs in it. Therefore they can not be constructed and compared simultaneously.

2. While comparing two t-nodes in a triconnected component tree, whenever a recursive call for their children resulted in an inequality, a canonical code is eliminated. This fact is used crucially to resume the execution after the recursive call. However, this is not the case in the canonization of b-nodes, since the subtrees of a triconnected component tree are compared several times.

To get around this problem, we give a tree-like structure that combines the biconnected component tree with the triconnected component trees of its b-nodes. We list below some simple facts about a biconnected component tree S:

1. S has a unique center. Recall that the center of a tree is the node in the tree, from which, the maximum distance to any node in the tree is minimized. It is easy to see that the center of a tree is the center of a longest path in it. As all the longest paths in S are of odd length, the center is unique.

2. An articulation point a has a copy in each biconnected component formed by its removal. However, it can have many copies in the triconnected component tree of a biconnected component B. This happens precisely when the copy of a in B is a part of a 3-connected separating pair.

3. Let a be the copy of an articulation point that appears in a biconnected component B. Let $T(B)$ be the triconnected component tree of B. Let C and D be two t-nodes in $T(B)$. If a copy of a appears in both C and D, then it appears in each node that is on the C to D path in $T(B)$. In other words, the nodes in $T(B)$ that contain a copy of a form a subtree of $T(B)$.

When an articulation point has several copies in the triconnected component tree of a biconnected component, we designate one of the copies as the *reference copy*. This copy is the one which occurs in the node closest to the root of the triconnected component tree, and thus depends on the choice of root of the tree.

3.6.2 Isomorphism Order of Biconnected Component Trees

Unlike in the case of triconnected component trees, there is no notion of orientation in case of biconnected component trees. Therefore the steps in the isomorphism order are exactly same as in Lindell's algorithm. However, the recursive comparison step is fairly complex in this case. The isomorphism order relation between two biconnected component trees is denoted by $<_{\mathtt{B}}$.

The size $|S|$ of a biconnected component tree S is defined as follows:

Definition 3.15 (Size of a biconnected component tree:) *Size of an a-node in S is 1. The size of a b-node is the size of its triconnected component tree. The size of a biconnected component tree is the sum of sizes of its nodes.*

Given two connected planar graphs G and H, let S and S' be their biconnected component trees. S and S' are rooted at a-nodes. As there are $O(n)$ a-nodes, a log-space transducer can cycle through all the choices of roots for S and S', to check whether $G \cong H$. Let the biconnected component trees S and S' be rooted at a-nodes a and a' respectively. The rooted trees are denoted by S_a and $S'_{a'}$. Define $S_a <_{\mathtt{B}} S'_{a'}$ if

1. $|S| < |S'|$, or
2. $|S| = |S'|$ but $\#a < \#a'$, or
3. $|S| = |S'|$, $\#a = \#a' = k$, but $(S_{B_1}, \ldots, S_{B_k}) <_{\mathtt{B}} (S'_{B'_1}, \ldots, S'_{B'_k})$ lexicographically, where we assume that $S_{B_1} \leq_{\mathtt{B}} \cdots \leq_{\mathtt{B}} S_{B_k}$ and $S'_{B'_1} \leq_{\mathtt{B}} \cdots \leq_{\mathtt{B}} S'_{B'_k}$ are the ordered subtrees of S_a and $S'_{a'}$, respectively.

$S_a =_B S'_{a'}$ if and only if neither $S_a <_B S'_{a'}$ nor $S'_{a'} <_B S_a$ holds.

Details of the isomorphism ordering Steps 1 and 2 are straight forward. The details of Step 3 are given below:

Consider the comparison of two subtrees rooted at a-nodes a, and a'. If equality is found in steps 1 and 2, the children of each of a and a' can be assumed to be partitioned into size-classes. Then the isomorphism order on the subtrees rooted at a and a' is obtained exactly as in Lindell's algorithm, *i.e.* by making cross-comparisons and storing order profiles.

Consider the comparison of two subtrees S_B and $S'_{B'}$ rooted at b-nodes B and B'. Let the parents of B and B' be a-nodes a and a' respectively, and the triconnected component trees of B and B' be T and T' respectively. We define the copies of a and a' in T and T' to be *larger* than any other vertex, which ensures that they are always mapped to each other. This is equivalent to giving them a distinct color.

The comparison of S_B and $S'_{B'}$ is started by invoking the isomorphism order procedure for T and T'. It calls the isomorphism order procedure for biconnected component trees whenever

the reference copies of articulation points are encountered. However, to make this recursive call, the current order profiles computed so far during the comparison of T and T' need to be stored. Moreover, as T and T' need to be rooted at an s-node and we need to cycle through the choices of the root, the current root also needs to be stored. Thus, to ensure that the algorithm works in log-space, a key task is to suitably limit the number of choices of roots of T and T'. We defer the description of this task to a later point, and describe the rest of the details assuming that the number of choices of roots is suitably limited to a number k:

1. Cycle through the k choices of roots for T and T'. This is done by comparing T with a fixed choice of root with all the choices for T', one at a time. Order profile is used to keep track of the results of the comparisons. The aim is to compare the minimum canonical codes of T and T' and return the result. The choice of root of T that leads to the minimum canonical code is the one that gives $T <_{\mathtt{T}} T'$ for the maximum number of choices of roots for T'.

2. Now consider the comparison of T and T' for some choices of roots. This comparison is carried out using the isomorphism order procedure for triconnected component trees. During the comparison of T and T', if a copy of an articulation point is encountered in a canonical code of a t-node C of T but not in that of the corresponding t-node C' of T', then that canonical code for C is considered to be larger and eliminated.

   If copies of articulation points u and u' are encountered simultaneously in nodes C and C', check whether these are their reference copies. If so, make a recursive call to isomorphism order procedure for biconnected component trees, to compare the subtrees of S_B and $S'_{B'}$ rooted at u and u'. If the copies encountered are not the reference copies, then assume equality and proceed. While making the recursive call, the current order profile of C or C' is stored along with the bit-vector for already eliminated canonical codes.

Limiting the number of choices of root for a triconnected component tree Let S be a biconnected component tree, and B be a b-node in it. Let the parent of B in S be a-node a, and the children of B be $a_1, \ldots, a_\ell$. The triconnected component tree of B be $T(B)$. Let the children of B be divided into p size-classes, and let k_j be the number of children in the jth size-class. We refer to the size-classes which contain only one child of B as *singleton size-classes*, and the children in the singleton size-classes as the *singleton a-nodes*. Parent of B is also considered as a singleton a-node. We have the following cases:

1. **The center of $T(B)$ is an s-node:** This s-node serves as the unique choice of root for $T(B)$. In the rest of the cases, we assume that the center of $T(B)$ is a t-node C. Also, assume $T(B)$ be rooted at C, and the reference copies of a and $a_1, \ldots, a_\ell$ are decided accordingly.

2. **There is a singleton a-node a_i that does not have a reference copy in C:** In this case, the reference copy of a_i in $T(B)$ appears in some t-node C' which is different from C. There is a unique path between C and C' in $T(B)$. The neighbor of C on this path is an s-node and serves as the unique choice of the root of $T(B)$.

   For the remaining cases, we assume that all the singleton a-nodes have reference copies in C.

3. **C is a cycle:** In this case, C has two possible canonical codes, with the parent a-node as the starting vertex, and its two incident edges as the starting edges. The children of C corresponding to the virtual edges that appear first in the two canonical codes form the choices for root of $T(B)$. There are two such choices.

   Here onwards, assume that C is a 3-connected planar graph.

4. **There are 3 or more singleton a-nodes:** The singleton a-nodes in C can be colored distinctly. Thus C has 3 vertices, each of which is colored with a different color. C is a 3-connected planar graph. From Corollary 3.20 below, when three vertices are fixed, C has at most one non-trivial automorphism, and hence it can have at most two minimum canonical codes. As in the previous case, the children of C corresponding to the virtual edges that appear first in these two canonical codes serve as the two choices for root of $T(B)$.

5. **There are at most two singleton a-nodes:** In this case, consider a non-singleton size-class of the children of B, that contains the minimum number of children of B, say k. If any one or more of these children do not have their reference copies in C, the number of choices for root of $T(B)$ can be limited to at most k, as in the Case 2 above.

   If all the k a-nodes have their reference copies in C, then from Lemma 3.16, C has at most $O(k)$ canonical codes. The number of choices for root of $T(B)$ can thus be limited to $O(k)$ as in Case 4 above.

The following lemma gives a relation between the number of automorphisms of the center C of the triconnected component tree $T(B)$, and the number of a-nodes that have their reference copy in C, when C is a 3-connected planar graph.

The lemma is stated in a more general form where some vertices of the graph are colored. Thus, we can assume that the reference copies of a-nodes in ith size-class are colored with color i. Moreover, the reference copy of the parent a-node is colored distinctly.

Lemma 3.16 *Let G be a 3-connected planar graph with colors on its vertices such that one vertex a is colored distinctly, and let $k \geq 2$ be the size of the smallest color class apart from the one which contains a. Then G has at most $4k$ automorphisms.*

To prove Lemma 3.16, we refer to the following results.

Lemma 3.17 *(P. Mani)(*See *e.g.* [19]*)Every triconnected planar graph G can be embedded on the 2-sphere as a convex polytope P such that the automorphism group of G coincides with the automorphism group of the convex polytope P formed by the embedding.*

Lemma 3.18 [15, 19, 12] *For any convex polytope other than tetrahedron, octahedron, cube, icosahedron, dodecahedron, the automorphism group is the product of its rotation group and $(1, \tau)$, where τ is a reflection. The rotation group is either C_k or D_k, where C_k is the cyclic group of order k and D_k is the dihedral group of order $2k$.*

Proof of Lemma 3.16. Let H be the subgroup of the rotation group, which permutes the vertices of the smallest color class among themselves. Then H is cyclic since the rotation group is cyclic. Let H be generated by a permutation π.

Notice that a non-trivial rotation of the sphere fixes exactly two points of the sphere viz. the end-points of the axis of rotation. Then, the following claim holds.

Claim 3.19 *In the cycle decomposition of π each non-trivial cycle has the same length.*

Proof of Claim 3.19. Suppose π_1, π_2 are two non-trivial cycles of lengths $p_1 < p_2$ respectively in the cycle decomposition of π. Then π^{p_1} fixes all elements of π_1 but not all elements of π_2. Thus $\pi^{p_1} \in H$ cannot be a rotation of the sphere which contradicts the definition of H. ■

As a consequence, the order of H is bounded by k, since the length of any cycle containing one of the k colored points is at most k. ■

This leads to the following corollary:

Corollary 3.20 *Let G be a 3-connected planar graph with at least 3 colored vertices, each having a distinct color. Then G has at most one non-trivial automorphism.*

Proof. An automorphism of G has to fix all the colored vertices. Consider the embedding of G on a 2-sphere. The only possible symmetry is a reflection about the plane containing the colored vertices, which leads to exactly one non-trivial automorphism. ■

Note that if the triconnected component C is one of the exceptions stated in Lemma 3.18, then it has $O(1)$ size. Then it has $O(1)$ minimum canonical codes, and a set of $O(1)$ separating pairs can be chosen as the possible choices for root of $T(B)$ as in Case 4 above.

Theorem 3.21 *Given two connected planar graphs G and H, and their biconnected component trees S and S', $G \cong H$ if and only if there is a choice of articulation points a in G and a' in H such that $S_a =_{\mathtt{B}} S'_{a'}$.*

Proof. Assume $S_a =_{\mathtt{B}} S'_{a'}$. The argument is an induction on the depth of the trees that follows the definition of the isomorphism order.

Base case: $d = 2$ In this case, all the children of a and a' are leaves. As $S_a =_{\mathtt{B}} S'_{a'}$, a and a' have equal number of isomorphic children, the isomorphism of the individual b-nodes can be extended to the isomorphism of the entire graphs.

Induction step: Let d be the depth of S_a and $S'_{a'}$. As the a-nodes and b-nodes appear at alternate levels, the subtrees rooted at the a-nodes at the next level have depth at most $d - 2$. Assume the result holds for subtrees up to depth $d - 2$. Let $G_1, \ldots, G_k$ be the children of a and $H_1, \ldots, H_k$ be the children of a' such that $\forall i : S_{G_i} =_{\mathtt{B}} S_{H_i}$. Therefore we have $G_i \cong H_i$, with the corresponding copies of articulation points having isomorphic subtrees rooted at them. Thus the pairwise isomorphism of the children of G_i and H_i can be extended to the graphs corresponding to the subtrees rooted at G_i and H_i, for each i. By coloring the copies of a in each of the G_is and that of a' in each of the H_is distinctly, it is easy to ensure that the pairwise isomorphism between the children of a and a' can be extended to an isomorphism between G and H.

The reverse direction holds obviously as well. If G and H are isomorphic and there is an isomorphism that maps an articulation point a of G to an articulation point a' of H. This also gives a pairwise isomorphism between children of a and the corresponding children of a'. Therefore an induction on the depth of the tree immediately leads to $S_a =_{\mathtt{B}} S_{a'}$. ■

3.6.3 Implementation of the Isomorphism Ordering in FL

We analyse the space-complexity of the isomorphism ordering. The first two steps can be implemented in FL as in Lindell's algorithm. We show that the third step can also be implemented in FL. The following two cases arise:

1. **Comparison of the subtrees rooted at two a-nodes:** This is a simple case and the storage requirement is exactly same as that of Step 3 in Lindell's algorithm. While comparing the children of two a-nodes from the same size-class, the order-profile of one child needs to be stored at a time. This takes $O(\log k)$ space where k is the number of children in that size-class. If the size-class has only one child of each a-node, nothing needs to be stored while comparing them recursively.

2. **Comparison of the subtrees rooted at two b-nodes:** Consider the comparison of the subtrees S_B and $S'_{B'}$ rooted at b-nodes B and B' respectively. Let a and a' be their parent a-nodes and $T(B)$ and $T(B')$ be their respective triconnected component trees. The comparison of the sizes of the subtrees and the number of children of B and B' can

clearly be done in log-space. So assume the children of B and B' be divided into size-classes. If each of B and B' have a large child, compare the large children and store the result. Recall that a child of B is considered to be a *large child* if the size of the subtree rooted at it is more than half of the size of the subtree rooted at B. If B and B' each have only one singleton child, compare them recursively and store the result. Thus no space is used while making this recursive call.

Now cycle through the k possible choices for roots of $T(B)$ and $T(B')$, where k is as described above. Thus k is either $O(1)$, or it is $O(k')$ where k' is the minimum cardinality of a non-singleton size-class of the children of B and B'. $O(\log k)$ space is used for storing the current roots of $T(B)$ and $T(B')$.

While comparing $T(B)$ and $T(B')$ for a fixed choice of roots, the recursive calls are made in two cases: to compare s-nodes and t-nodes in $T(B)$ and $T(B')$, and to compare a-nodes which are children of B and B' in S_B and $S'_{B'}$. Two a-nodes are compared when their reference copies are encountered while comparing the s-nodes or t-nodes in $T(B)$ and $T(B')$. We consider an a-node in S_B to be a child of that s-node or t-node of $T(B)$, which contains its reference copy. The sizes of the subtrees rooted at the s-nodes and t-nodes of $T(B)$ are computed accordingly. A large child of B is an exception as the result of the comparison of large children is computed and stored a priori. Therefore the storage used while making recursive comparisons is same as in the case of triconnected component trees (Section 3.5.2).

The recurrence for the space required is given by

$$\mathcal{S}(N) \leq \max_j \mathcal{S}\left(\frac{N}{k_j}\right) + O(\log k_j),$$

where $k_j \geq 2$ for all j. Thus $\mathcal{S}(N) = O(\log N)$.

3.6.4 Canonization of Planar Graphs

Once it is clear how to get the isomorphism order on biconnected component trees in log-space, obtaining the canon of a connected planar graph is straight forward. It is similar to the canonization of biconnected planar graphs described in Section 3.5.3. The details are given in the proof of the following theorem:

Theorem 3.22 *A connected planar graph can be canonized in log-space.*

Proof. Assume that the canonization algorithm has oracle access to the isomorphism ordering algorithm. Given a connected planar graph G with biconnected component tree S, a log-space

transducer cycles through all the a-nodes to choose a root for the tree that leads to the lexicographically smallest canonical code.

For a particular choice a of root, the canonization algorithm traverses the tree S according to the isomorphism order and outputs the canonical codes of the subtrees in pre-order. Canonical code of a b-node is the canonical code of the biconnected planar graph obtained by traversing its triconnected component tree in isomorphism order. As the canonical code of a triconnected component is a list of edges, this first pass gives a canonical list of edges of G.

In the second pass, another log-space transducer relabels the vertices of G according to their first occurrence in the list. These labels lead to a *canonical labelling* of the vertices of G. A list of edges of G in lexicographically increasing order according to the canonical labels of their end-points is the canon of G. It is easy to see that both of these steps can be implemented in FL. ■

3.7 Discussion

This chapter gives a log-space algorithm for canonization of planar graphs, which settles the complexity of planar graph canonization. An interesting question is to use this technique for other graph classes. There has been some progress in this direction due to [33], which describes log-space algorithms for canonization of graphs that exclude either a $K_{3,3}$ or a K_5 minor. These algorithms crucially use the fact that 3-connected components of a $K_{3,3}$-free graph and 4-connected components of a K_5-free graph are planar, and thus they can be canonized using a UXS. It is interesting to find other minor-closed families where similar properties hold and hence the same techniques apply.

Another question is to extend this result to bounded-genus graphs. The difficulty here is that even highly connected genus k graphs do not have a unique embedding on a genus k surface, and hence the idea of UXS can not be applied for their canonization.

Part II

Path Problems

4

Longest Paths in Planar DAGs

We now turn towards the complexity analysis of path problems in some restricted graph classes. This chapter consists of some results on path problems in directed graphs, and in planar directed acyclic graphs. The main problem considered here is computation of the longest path length between two designated nodes in a planar DAG. It is shown that this computation can be done in UL $\cap$ coUL. Moreover, it can be generalised to some other classes of DAGs. The problem of counting the number of paths in DAGs is considered next, and a promise version of this problem is shown to be in LogDCFL.

4.1 Summary of the Main Results

We recall the following problem definitions from Section 1.2.2:

$$\begin{aligned}
\textsf{Reach} &= \{\ (G,s,t)\ \mid\ G \text{ contains a path from } s \text{ to } t\ \} \\
\textsf{Distance} &= \{\ (G,s,t,k)\ \mid\ G \text{ contains a path of length} \leq k \text{ from } s \text{ to } t\ \} \\
\textsf{Long-Path} &= \{\ (G,s,t,k)\ \mid\ G \text{ has a simple path of length} \geq k \text{ from } s \text{ to } t\ \} \\
\textsf{\#Path} &= \{\ (G,s,t,1^k)\ \mid\ G \text{ has exactly } k \text{ simple paths from } s \text{ to } t\}
\end{aligned}$$

We consider the combination of planarity and acyclicity for the Long-Path problem in directed graphs, and show that the NL upper bound can be improved to UL $\cap$ coUL in this case (Section 4.2). This is done by showing that the Long-Path problem in planar DAGs can be reduced to the Distance problem in planar DAGs, with oracle access to Reach. This improves the previously known bound of NL that is known for longest paths in DAGs. We also show that the same bound holds for Distance and Long-Path problems in toroidal DAGs. *Toroidal DAGs* are those which can be drawn on a torus without any two edges crossing each other. These are precisely the graphs of genus 1.

Section 4.3 describes a UL $\cap$ coUL algorithm for computing the length of a longest path in *max-unique* DAGs with a single sink. *Max-unique DAGs* are the DAGs with the property that the longest path between each pair of vertices is unique. In [78], min-unique graphs are

considered and a UL ∩ coUL algorithm for shortest path computation in min-unique graphs is described. Thus, it is shown that the min-uniqueness property helps in getting a UL ∩ coUL algorithm for shortest path computation. The algorithm is based on double inductive counting technique. We show that the max-uniqueness property can be used in a similar way to get a UL ∩ coUL algorithm for longest path computation in DAGs, with an extension of the double inductive counting technique.

Some versions of the #Path problem in DAGs are considered in Section 4.4. In particular, we consider the #Path problem on planar DAGs, and on single sink DAGs, when there is a promise that the number of s to t paths is bounded by a polynomial in the size of the graph. We show that under this promise, the number of s to t paths in planar DAGs can be computed by a UAuxPDA running in polynomial-time, whereas the number of s to t paths in a DAG that has t as the only sink, can be computed in LogDCFL. Further, we show that the number of longest or shortest s to t paths in a planar DAG can be computed by a UAuxPDA running in polynomial-time, given the promise that this number is bounded by a polynomial in the size of the graph.

For related graph-theoretic and complexity-theoretic background, and previously known results, we refer to Section 1.1 and Section 1.2.2, respectively.

4.2 Reducing Longest Paths to Shortest Paths in DAGs

For any subclass $\mathcal{C}$ of graphs, let Reach($\mathcal{C}$), Distance($\mathcal{C}$), and Long-Path($\mathcal{C}$) denote the restriction of these problems to instances from $\mathcal{C}$.

The main result of this section is as follows:

Theorem 4.1 Long-Path*(Planar DAG)* ∈ UL ∩ coUL.

We show that the Long-Path and the Distance problems in planar DAGs are equivalent via log-space reductions, given oracle access to Reach. This is an extension of the following result of [49]:

Lemma 4.2 ([49]) Distance*(Series-parallel) and* Long-Path*(Series-parallel) are equivalent.*

The Reach and Distance problems on planar DAGs are known to be in UL ∩ coUL due to the following results:

Lemma 4.3 ([25]) Reach*(Planar) is in* UL ∩ coUL.

Lemma 4.4 ([84]) Distance*(Planar) is in* UL ∩ coUL.

Thus Theorem 4.1 follows from Lemmas 4.3 and 4.4, which use the fact that $\mathsf{L}^{\mathsf{UL} \cap \mathsf{coUL}} = \mathsf{UL} \cap \mathsf{coUL}$, and from Lemma 4.5 below:

Lemma 4.5 Distance*(Planar DAG) and* Long-Path*(Planar DAG) are equivalent via log-space reductions that have access to the oracle* Reach*(Planar DAG).*

The proof of Lemma 4.5 is a generalization of the proof given in [49] for series-parallel graphs. The generalization in fact works for any class of acyclic graphs that is closed under subdivision and vertex deletion. In particular, it works for planar DAGs. We present below, in Theorem 4.6, the result of [49] simplified by specialising to unweighted graphs, and stated for such (more general) classes of graphs. Lemma 4.5 is an obvious corollary.

Theorem 4.6 *Let $\mathcal{C}$ be any subclass of directed acyclic graphs closed under subdivisions and vertex deletions. There is a function f, computable in log-space with oracle access to* Reach$(\mathcal{C})$*, that reduces* Distance$(\mathcal{C})$ *to* Long-Path$(\mathcal{C})$ *and* Long-Path$(\mathcal{C})$ *to* Distance$(\mathcal{C})$.

Proof. Let $G = (V, E)$ be the given directed acyclic graph, in which we want to find the longest, or the shortest, path between given vertices s and t. Construct a new graph $G' = (V', E')$ as follows:

For each $u \in V$, define $P_u = \{x \in V \mid \text{there is a path from } x \text{ to } u \text{ in } G\}$. Note that u is in P_u for all u. Next define $E_u = \{(x, y) \in E \mid x \in P_u, y \notin P_u\}$. Since G is acyclic, all outgoing edges of u are in E_u.

Let ρ be any s to t path. For every vertex u, ρ has at most one edge from E_u. This can be seen as follows: If there is a path from t to u ($t \in P_u$), then there is a path from every vertex on ρ to u via t, so no edge of ρ is in E_u. If there is no path from s to u, then there is no path from any vertex on ρ to u, so again no edge of ρ is in E_u. Now if $s \in P_u$ but $t \notin P_u$, then along the path ρ, we transit from being in P_u to being outside P_u exactly once. Let this transition occur on edge (x, y). Then (x, y) is in E_u, and no other edge of ρ can be in E_u. Thus

$$|\rho \cap E_u| = \begin{cases} 1, & \text{if } \ s \in P_u \text{ and } t \notin P_u, \\ 0, & \text{otherwise.} \end{cases}$$

To obtain G', we replace each edge $e = (u, v)$ by a path of length l_{uv} determined as follows:

$$\begin{aligned} l_{uv} &= 2\left(\sum_{x \in V : (u,v) \in E_x} \text{out-degree}(x)\right) - 1 \\ &= 2\left(\sum_{x \in V : u \in P_x, v \notin P_x} \text{out-degree}(x)\right) - 1 \end{aligned}$$

Since G is acyclic, the vertex u itself always qualifies in the above sum, and so l_{uv} is positive.

For any pair of vertices s, t we define the quantity K_{st} as follows:

$$K_{st} = \sum_{x \in V : s \in P_x, t \notin P_x} \text{out-degree}(x)$$

Now the crucial claim: for each pair of vertices s, t, each s to t path ρ in G of length $|\rho|$ (in terms of number of edges) is transformed by the above construction to a path in G' of length *exactly* $2K_{st} - |\rho|$. This is because the length of the transformed path is

$$\begin{aligned}
\sum_{\substack{(u,v)\in E \\ (u,v)\in\rho}} l_{uv} &= \sum_{\substack{(u,v)\in E \\ (u,v)\in\rho}} \left[2 \left(\sum_{x \in V : u \in P_x, v \notin P_x} \text{out-degree}(x) \right) - 1 \right] \\
&= 2 \left(\sum_{\substack{(u,v)\in E \\ (u,v)\in\rho}} \sum_{x \in V : u \in P_x, v \notin P_x} \text{out-degree}(x) \right) - |\rho| \\
&= 2 \sum_{x \in V} \left(\text{out-degree}(x) \sum_{e \in \rho \cap E_x} 1 \right) - |\rho| \\
&= 2 \left(\sum_{x \in V} \text{out-degree}(x) \cdot |\rho \cap E_x| \right) - |\rho| \\
&= 2 \sum_{x \in V : |\rho \cap E_x| = 1} \text{out-degree}(x) - |\rho| \;=\; 2K_{st} - |\rho|
\end{aligned}$$

It thus follows that the longest (shortest) path in G is mapped to the shortest (longest, respectively) path in G'. In fact, if the s to t paths are ordered monotonically with respect to length, then the above transformation precisely reverses this ordering. Hence the reduction function f maps (G, s, t, k) to $(G', s, t, 2|K_{st}| - k)$.

The reduction can be computed in log-space with oracle access to $\mathsf{Reach}(\mathcal{C})$, where all queries involve only the graph G. This is because obtaining G' as well as computing K_{st} merely involve finding the sets P_u and E_u and adding up out-degrees. ■

Longest Path in Toroidal DAGs In [4], Reach in toroidal graphs has been shown to be reducible to Reach in planar graphs by log-space truth-table reductions. From Theorem 4.1 and the construction of [4], we get the following corollary, which states that Distance and

Long-Path problems on toroidal DAGs can be reduced to the corresponding problems on planar DAGs by log-space truth-table reductions.

Corollary 4.7

$$\begin{aligned} \textsf{Distance}(\textit{Torus}) &\leq_m^{\log} \textsf{Distance}(\textit{Planar}) \\ \textsf{Long-Path}(\textit{Toroidal DAG}) &\leq_m^{\log} \textsf{Long-Path}(\textit{Planar DAG}) \end{aligned}$$

We give a brief overview of the reduction given in [4]:

Consider a directed graph G embedded on a torus. Let C be a cycle in the underlying undirected graph. A *side* of C is the set of vertices of G that are connected to some vertex on the cycle by paths, such that the paths do not cross the cycle. This notion of side suffices for toroidal graphs. Thus a cycle has two sides, called *the left side* and *the right side*. A cycle C is said to be *surface non-separating* if the two sides have a non-empty intersection. Note that a planar graph does not have such a cycle. If G is non-planar but is toroidal, it has at least one such cycle.

The reduction in [4] proceeds by finding such a cycle C, which can be found by computing a spanning tree of G. A spanning tree of a graph is known to be computable in L [73, 77]. Once a spanning tree is computed, one of the fundamental cycles with respect to the spanning tree is known to be surface non-separating. Whether a given cycle C is surface non-separating can be checked in L [4].

The next step of the reduction is to cut the torus along C to get a cylinder. Then make two copies of C - one on each side of this cylinder. To preserve connectivity properties, $2n+1$ copies of this cylinder are pasted together such that the original copy is in the middle and has n copies on each side, and all the copies together form a long cylinder. This new graph G' is cylindrical and hence planar. It can be shown that there is a directed s to t path in G if and only if there is a directed path in G' from s in the original copy of G to any of the copies of t. This completes the reduction.

4.3 Longest Paths in DAGs via Double Inductive Counting

The main result of this section is that in max-unique DAGs where t is the unique sink, the length of the unique longest s to t path can be computed in UL ∩ coUL. Recall that max-unique DAGs are those which have a unique longest path between every pair of vertices (u, v), whenever v is reachable from u. Formally, we give a proof of the following theorem:

Theorem 4.8 *There is a nondeterministic log-space machine M that, given as input a directed acyclic max-unique graph G with a unique sink t, and any other vertex s, finds the length of the longest s to t path unambiguously.*

In [78], double inductive counting is used to unambiguously test reachability in min-unique graphs. In [84], the same technique is used, in combination with the weighting technique from [25], to compute shortest paths in planar graphs. For computing longest paths, however, the technique cannot be used as it is, but needs a further small, but crucial, modification.

The modified technique is described in detail in Algorithms 3, 4 and 5. The algorithms use two counters c_k and Σ_k. The counter c_k stores the number of vertices having a path of length at least k to t. The counter Σ_k is the sum of the lengths of longest paths to t for those vertices whose longest path to t is of length less than k. Thus we define the following:

$$D(v) = \text{ Length of the longest path from } v \text{ to } t.$$

$$S_k = \{v \mid D(v) \geq k\}, \qquad c_k = |S_k|$$

$$\Sigma_k = \sum_{v \in V \setminus S_k} D(v), \qquad T = \sum_{v \in V} D(v)$$

While we describe and analyze the procedure in detail below, here is a quick overview for those familiar with the method from [78]. The crucial new parameter we need is T, the total length of all longest paths. At the outset, we nondeterministically guess a value M which is our estimate of T. At the end, when we have reached a value of k for which $c_k = 0$, we check whether M equals Σ_k. This allows us to make the procedure unambiguous. The additional condition that t is the unique sink allows us to initialise the counters: every vertex has a path, and hence a longest path, to t, and so $c_0 = n$.

The algorithms described here compute the longest path length when additionally s is the only source. After showing that these algorithms are correct, we discuss how to remove this restriction.

Algorithm 3 Main

Input: G, s, t
guess nondeterministically $M = \sum_{v \in V} D(v)$ with $n - 1 \leq M \leq n^2$
$c_0 \leftarrow n, \Sigma_0 \leftarrow 0, k \leftarrow 0$
while $c_k \neq 0$ **do**
    $k \leftarrow k + 1$
    Update (compute c_k and Σ_k)
end while
if $\Sigma_k \neq M$ **then**
    halt and **reject**
else
    output $D(s) = k - 1$, **accept** and **halt**
end if

Algorithm 4 Update: Compute c_k and Σ_k, given c_{k-1} and Σ_{k-1}

Input: $G, s, t, k, c_{k-1}, \Sigma_{k-1}$
$c_k \leftarrow c_{k-1}, \Sigma_k \leftarrow \Sigma_{k-1}$
for all $v \in V$ **do**
  if Test$(G, k-1, c_{k-1}, \Sigma_{k-1}, v)$=true **then**
    if for all out-neighbours x of v,
  Test$(G, k-1, c_{k-1}, \Sigma_{k-1}, x)$=false **then**
      $c_k \leftarrow c_k - 1, \Sigma_k \leftarrow \Sigma_k + k - 1$
    end if
  end if
end for

Algorithm 5 Test: An unambiguous procedure to test if $D(v) \geq k$

Input: $G, s, t, k, c_k, \Sigma_k, v$
count $= n$, sum $= 0$, path.to.v=true, sum$' = 0$
for all $x \in V$ **do**
  guess nondeterministically if $D(x) \geq k$
  if guess is no **then**
    guess a path of length $l < k$ from x to t.
    if this fails **then**
      reject and **halt**
    end if
    count $\leftarrow$ count $- 1$
    sum $\leftarrow$ sum $+ l$
    if $x = v$ **then**
      path.to.v =false
    end if
  else
    guess a path of length $l' \geq k$ from x to t
    if this fails **then**
      reject and **halt**
    end if
    sum$' \leftarrow$ sum$' + l'$
  end if
end for
if count $= c_k$ and sum $= \Sigma_k$ and sum$' +$ sum $= M$ **then**
  return path.to.v
else
  reject and **halt**
end if

It is clear that these procedures can be implemented in nondeterministic log-space. Claims 4.9, 4.10, 4.11 and 4.12, stated and proved below, show that these procedures unambiguously find the length of the longest path from s to t in a max-unique acyclic graph G where t is the only sink.

Claim 4.9 *If the guessed value of M is correct (i.e. $M = T$), then algorithm* Test*, given the correct values of c_k and Σ_k as input, reports a decision on exactly one run.*

Proof. The procedure Test, on each run R, guesses an x to t path R_x for each vertex x. Depending on its guess for $D(x) \geq k$, it adds the length of R_x to either sum or sum$'$. Finally these have to add up to M for Test to report a decision.

When $M = T$, M is indeed the sum of all $D(x)$. This can match sum + sum$'$ exactly when all the guessed paths R_x are longest. Since G is max-unique, this happens on exactly one run. ■

Claim 4.10 *For any guessed value of M, given the correct values of c_k and Σ_k as input, all the runs of algorithm* Test *that do not lead to rejection always return the correct decision.*

Proof. As described in the preceding proof, each run of Test guesses a path R_x for each x. It may guess a path of length shorter than $D(x)$, but not longer. Since count is decremented only when it guesses that $D(x) < k$, and for other guesses some witnessing path of length at least k is found, at the end the value of count is at most as large as c_k.

Suppose on some run Test returns a decision. Then on this run count $= c_k$. Suppose further that the decision is wrong.
Case 1: $D(v) < k$, but Test reports that it is larger. This cannot happen, since Test has to find a witnessing path of length at least k.
Case 2: $D(v) \geq k$, but Test reports that it is smaller. Then this run of Test does not account for v in count. So at the end of the run, count $< c_k$, a contradiction. ■

Claim 4.11 *If the queries $(D(v) \geq k)$ are answered correctly by* Test*, then given c_{k-1} and Σ_{k-1}, the values of c_k and Σ_k are updated correctly by algorithm* Update.

Proof. Update starts by assuming that $S_k = S_{k-1}$ and so $c_k = c_{k-1}$. Note that $S_k \subseteq S_{k-1}$, so Update only has to detect when to remove vertices from its current S_k.

For each v, Update checks whether $D(v) \geq k-1$ and $D(u) < k-1$ for all out-neighbours u of v. If this holds, then the longest path from v to t is of length exactly $k-1$ and $v \notin S_k$. Thus the procedure decrements c_k by 1 and increments Σ_k by $k-1$.

So if all the queries are answered correctly by Test, then what Update does is correct. ■

Claim 4.12 *The algorithm* Main *is correct and unambiguous.*

Proof. Main starts with the correct values of c_0 and Σ_0. From Claims 4.10 and 4.11, the correctness of Main is immediate. In particular, the final value of Σ_k is always correct.

If $M = T$, then by Claim 4.9, procedure Test always returns a decision, unambiguously. Thus exactly one run of Main (amongst those where $M = T$ was guessed) leads to a decision, and this decision is correct.

If $M > T$, then no run of Test, at any stage k, can trace paths adding up to M. So Test, and hence Update, and Main have no accepting run.

If $M < T$, consider the runs on which Test and Update proceed to finally compute Σ_k. Since Main is correct, we know that $\Sigma_k = T$. Now the check $M = \Sigma_k$ fails and Main rejects and halts. ■

A straightforward modification will handle the case when s is not the unique source. Just keep one additional special counter for the vertex s. Initialise this counter to 0. At each stage k, after c_k and Σ_k are computed, run Test to check if $D(s) \geq k$ and if so, set this counter to k. At the end of Main, report the value of this counter.

Alternate Proof of Theorem 4.1

The above algorithm can be used to give an alternate proof of Theorem 4.1. The idea is to (1) trim the graph using oracle queries to Reach(Planar) so that t is the only sink, (2) embed it in a grid with suitable edge weights using the embedding algorithm of [4], and (3) use the weighting scheme of [25] so that the resulting planar DAG, say H, is min-unique. As further noted in [84], the shortest path in H corresponds to a shortest path in G. It is straightforward to see that H is also max-unique, with the longest path in H corresponding to some longest path in G. Thus Theorem 4.8 is applicable. Computing the longest path length in G from that in H is also straightforward.

4.4 Extensions of the Longest Paths Algorithm: Search, Counting

In this section we consider variations of our planar Long-Path algorithm: finding a longest path, finding multiple longest paths, and counting the number of paths under some promise.

4.4.1 Finding a Longest Path

We show that for planar DAGs, the search version of Long-Path is also in UL ∩ coUL.

Theorem 4.13 *A longest path between two designated nodes s and t in a planar DAG can be found in* $\mathsf{L}^{\mathsf{UL} \cap \mathsf{coUL}}$ *and hence in* UL ∩ coUL.

Proof. The log-space machine computes the length of the longest path from s to t by asking queries to the Long-Path oracle, with the values of k starting from n, till a 'yes' answer is obtained. Recall that Long-Path $= \{(G, s, t, k) \mid G$ has a simple path of length $\geq k$ from s to $t\}$. Let this length of the longest s to t path be l. Then find the neighbor v of s that has length $l - 1$ path to t. Output v and continue, till finally t is output. ■

4.4.2 Finding Multiple Longest Paths

We consider another variation of Long-Path. Given a planar DAG G, the algorithm in Section 4.4.1 produces a longest path ρ. Can we find the (length of the) longest path other than ρ? Note that this may well have the same length as ρ, because G itself need not be max-unique.

We proceed as follows: Let l be the length of ρ, a longest path from s to t. For each edge e in ρ, let l_e denote the length of the longest path in $G \setminus \{e\}$. Then the length of the second longest path is the maximum of l_e over e in ρ, and such a path can be found by using the Algorithm of Section 4.4.1 on $G \setminus \{e\}$ for the appropriate edge e.

This can be generalised to finding the (length of the) k^{th} longest path, as long as k is a constant. Thus we have:

Theorem 4.14 *For each constant k, given a planar DAG G and vertices s, t in it, a list of k paths from s to t in G can be found in* UL $\cap$ coUL *such that every s to t path not listed is no longer than any listed s to t path.*

Remark 4.15 *The k longest paths may not be unique due to ties. The algorithm guarantees that any s to t path not listed by it is no longer than the shortest s to t path listed by it. But the k paths output by the algorithm are all distinct.*

4.4.3 Computing the Number of Paths: A Promise Version

The problem of counting the number of paths between two designated nodes in a DAG is known to be complete for #L. Consider a restriction of this problem when the number of such paths is bounded by a polynomial. With this restriction it is natural to believe that the counting problem is easier, because even Reach, which is otherwise NL-complete, is in LogDCFL for such graphs [27].

In [6], an NL upper bound for the counting problem under this promise was obtained, further substantiating this belief. We consider an additional (easily checkable) restriction where the DAG not only has polynomially bounded number of s to t paths but also t is the unique sink in the graph. In such graphs, we show that the counting problem can be solved in LogDCFL, proving Theorem 4.16 below:

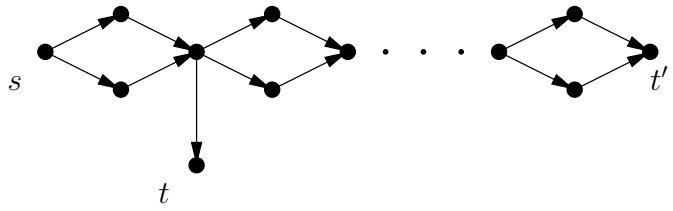

Figure 4.1: Example: DFS exploration when t is not the unique sink

Theorem 4.16 *Let $c > 0$ be a fixed constant. There is a* DAuxPDA *that, given a DAG G with n nodes and a unique sink t, and given a node s in G such that the number of paths from s to t in G is bounded by n^c, computes this number in polynomial time.*

Proof. Our procedure is a depth-first-search exploration of the graph. The algorithm explores the DAG starting from s. The number of s to t paths explored is stored in a variable count. It assumes an ordering on the labels of the vertices (lexicographical ordering will suffice) and an additional label 0 which is assumed to be the smallest in the ordering. It traverses the graph in a depth-first manner, putting visited vertices on the stack. The label of the vertex being visited in the current step is stored in the variable current. The traversal is started from s, taking the out-neighbour with the smallest label at each step. Whenever t is reached, count is incremented.

On reaching t, the algorithm backtracks by popping the stack, retrieving the vertex v visited just before t. The label of t is stored in a variable previous and the label of v is stored in current. If v has an out-neighbour u with a label larger than previous, the traversal is continued along the (v, u) edge, and the label of this out-neighbour is stored in the variable next. Otherwise, the backtracking process is continued by popping the stack again, storing the label of v in previous, and setting current to the newly popped label.

Thus, at any point of time, the stack contains nodes on the path from s to current (excluding current), and the vertex current is being explored in the forward or backward direction.

Note that for the algorithm to work in polynomial-time, it is essential that t is the unique sink in the graph. If there is another sink t', the algorithm will explore all s to t' paths as well, and this number may not be polynomially bounded. See *e.g.* Figure 4.1. The number of s to t path is 2, but the number of paths from s to t' is $2^{O(n)}$.

Clearly, the algorithm correctly computes the number of s to t paths. Furthermore, for each s to t path ρ, each edge on ρ is traversed exactly once in the forward direction. During the backtracking phase, after reaching t along ρ, an edge on ρ is traversed at most once in the backward direction.

Consequently, the algorithm can be implemented on a DAuxPDA running in polynomial time, if the number of s to t paths is bounded by a polynomial.

This completes the proof of Theorem 4.16. ■

4.4.4 Counting Paths in Planar DAGs: A Promise Version

We now consider applying Theorem 4.16 to planar DAGs and prove the following theorem:

Theorem 4.17 *Let $c > 0$ be a fixed constant.*

1. *There is a nondeterministic* AuxPDA *that, given a planar directed acyclic graph G with n nodes, and given nodes s and t in G such that the number of paths from s to t in G is bounded by n^c, proceeds unambiguously (rejects on all except one path) and computes this number in polynomial time.*

2. *There is a nondeterministic* AuxPDA *that, given a planar directed acyclic graph G with n nodes, and given nodes s and t in G such that the number of longest paths from s to t in G is bounded by n^c, proceeds unambiguously and computes this number in polynomial time. The same is true for shortest paths.*

The depth-first-search algorithm described in the proof of Theorem 4.16 may not be directly applicable, since the given DAG may have multiple sinks. These sinks can be removed in UL ∩ coUL by queries to Reach, to get a planar DAG with t as the only sink. Then the depth-first-search from the proof of Theorem 4.16 can be used. Overall, the combined algorithm uses log-space in a nondeterministic but unambiguous manner, and a stack in a deterministic manner, giving an algorithm that can be implemented on a UAuxPDA in polynomial time. Thus we get the first part of Theorem 4.17.

Our algorithm is similar to that given in [27] for testing reachability in the computation tree of an NL machine when each configuration of the machine is reachable by polynomially many paths.

Now consider the problem of computing the number of longest s to t paths when this number is bounded by a polynomial. Note that the total number of s to t paths need not be polynomially bounded, hence just removing the sinks other than t as above does not suffice.

Algorithm 6 describes a procedure that removes all the edges which do not appear on any longest path from s to t. It finds out the length of the longest s to t path in the planar DAG G in UL ∩ coUL using Theorem 4.1. Then G is layered and edges that are not a part of any s to t longest paths are removed. These are exactly the edges that go across more than one layer. Thus, in the resulting DAG, all s to t paths are the longest s to t paths. Moreover, t is the only sink in this new DAG.

Now we can use the depth-first-search from the proof of Theorem 4.16 for counting s to t paths in this graph.

Algorithm 6 Procedure to delete all s to t paths shorter than the longest path.

Input: Planar DAG $G = (V, E)$, two designated vertices s, t
Output: Graph G' that has no paths other than s to t longest paths
for all $v \in V$ **do**
  $layer(v) \leftarrow$ length of s to v longest path
end for
for all edges (u, v) **do**
  if $layer(v) - layer(u) = 1$ **then**
    output (u, v)
  end if
end for

If instead of longest paths, we wish to count shortest paths, and this number is bounded by a polynomial, a similar approach can be used. The distance of a node v from s should be used as its layer number in Algorithm 6 instead of the length of the longest path from s to v. From Lemma 4.4 we know that distance can be computed in UL ∩ coUL.

Combining the UL ∩ coUL layering procedure of Algorithm 6 with the DAuxPDA procedure of depth-first-search described in the proof of Theorem 4.16, we get the second part of Theorem 4.17. This number can also be determined in NL [6], and Theorem 4.17 gives a bound which is incomparable to this.

4.5 Discussion

We show that in planar and toroidal DAGs, detecting the presence or absence of long paths can be done in unambiguous log-space. In particular, detecting long paths and detecting short paths are equivalent, modulo reachability in these graphs. This result has recently been generalised in [85], where the same upper bound for the longest path problem is obtained for directed acyclic graphs which exclude either $K_{3,3}$ or K_5 as minors. For all these graph classes, the best known lower bound for all three problems — reachability, shortest path, and longest path — is log-space hardness. There is thus a gap between the lower and upper bounds. It is open whether the upper bound of UL ∩ coUL can actually be improved to log-space.

If the number of paths from s to t in a DAG is bounded by a polynomial in the graph size, then a result from [6] shows that this number can be computed in NL. We have given a different bound for two restrictions:

1. If t is the unique sink in the DAG, then this number can be computed by a polynomial time DAuxPDA and hence in LogDCFL.

2. If the DAG is planar, then this number can be computed by a polynomial time UAuxPDA.

This is the first (natural, we believe) instance we know of a problem that is not known to be in log-space, but is accepted by both UAuxPDA[poly] and NL machines. It suggests that UL should be an upper bound for this problem; establishing this is an interesting question.

5

Path Problems in k-trees

This chapter continues the complexity analysis of path problems in restricted graph classes. Path problems are defined in Chapter 4. In this chapter, we analyze the complexity of these path problems in a graph class known as k-trees. The main results include a log-space algorithm and a matching log-space hardness result for reachability in directed k-trees described in Section 5.2, and a log-space algorithm for shortest and longest path computations in directed acyclic k-trees described in Section 5.3. Here it is assumed that k is a fixed constant. We also describe a simple log-space algorithm for shortest path computation in undirected k-trees in Section 5.4.

The class of k-paths forms a subclass of k-trees. The algorithms for k-trees use the corresponding algorithms for k-paths as a subroutine. Hence an algorithm for reachability in directed k-paths is given in Section 5.2.1, and an algorithm to find the length of the shortest and longest path in a directed acyclic k-path is given in Section 5.3.1. The corresponding algorithms for k-trees appear in Section 5.2.2 and Section 5.3.2 respectively.

5.1 Tree-Decomposition of k-trees

The definition and a tree-decomposition of k-trees are described here. The definition and decomposition are applicable to both directed and undirected k-trees. For directed graphs, the directions on the edges are ignored while defining k-trees and while computing their decomposition. Thus a directed graph is said to be a k-tree if its underlying undirected graph is a k-tree.

The graph class k-trees is defined in [42] and the definition is given below:

Definition 5.1 *The class of* k-trees *is inductively defined as follows.*

- *A clique with* k *vertices (*k*-clique for short) is a* k*-tree.*
- *Given a* k*-tree* G' *with* n *vertices, a* k*-tree* G *with* $n+1$ *vertices can be constructed by introducing a new vertex* v *and picking a* k*-clique* X *(called the* support *of* v*) in* G' *and*

then joining v to each vertex u in X. Thus, $V(G) = V(G') \cup \{v\}$, $E(G) = E(G') \cup \{\{u, v\} \mid u \in X\}$.

A *partial k-tree* is a subgraph of a k-tree. The class of partial k-trees coincides with the class of graphs which have tree-width at most k. The notion of tree-width is defined in [79]. Whether the given graph is a k-tree can be determined in log-space [13] but partial k-trees are not known to be recognizable in log-space.

In literature, several representations of k-trees have been considered [38, 13, 53]. We use the following representation given by Köbler and Kuhnert [53]:

Definition 5.2 *Let $G = (G, E)$ be a k-tree. The tree-representation $T(G)$ of G is defined by*

$$V(T(G)) = \{M \subseteq V \mid M \text{ is a } k\text{-clique or a } (k+1)\text{-clique}\}$$

$$E(T(G)) = \{\{M_1, M_2\} \subseteq V \mid M_1 \subsetneq M_2\}$$

In [53], it is proved that $T(G)$ is a tree and can be computed in log-space. In the rest of the chapter, we use G in place of $T(G)$. Thus, by *a k-tree G*, we always mean that G is in fact represented as $T(G)$. The term *vertices in G* refers to the vertices in the original graph, whereas *a node in G* and *a clique in G* refer to the nodes of $T(G)$. Partial k-trees also have a tree-decomposition similar to that of k-trees, but it is not known to be log-space computable. The best known upper bound for a tree-decomposition of partial k-trees is LogCFL due to [91].

The class *k-paths* is a sub-class of k-trees (e.g. see [40]). The recursive definition of k-paths is similar to that of k-trees. However, the restriction is that a new vertex can be added only to a particular clique called the *current clique*. After addition of a vertex, the current clique may remain the same, or may change by dropping a vertex and adding the new vertex in the current clique.

We consider the following representation of k-paths, referred to as *path-representation* here, which is based on the recursive definition of k-paths, and is known to be computable in log-space [13]:

Proposition 5.3 *Given a k-path $G = (V, E)$, for $i = 1, \cdots, m$, let X_i be the current clique at the ith stage of the recursive construction of the k-path. Let $V_1 = \cup_i X_i$ and $V_2 = V \setminus V_1$. We call the vertices in V_2 as spikes. The following facts are easy to see:*

1. *No two spikes have an edge between them.*
2. *Each spike is connected to all the vertices of exactly one of the X_i's.*
3. *X_i and X_{i+1} share exactly $k-1$ vertices*

The path-representation of G consists of a graph $G' = (V', E')$ where $V' = \{X_1, \ldots, X_m\} \cup V_2$ and $E' = \{(X_i, X_{i+1}) \mid 1 \leq i < m\} \cup \{(X, v) | X \text{ is a clique in } \in V', v \in V_2 \text{ has a neighbour in } X\}$.

5.2 Reachability

We give log-space algorithms to compute reachability in k-paths and in k-trees. Although the graphs considered in this section are directed, when we refer to any of the definitions or decompositions in the preliminary section we consider the underlying undirected graph.

5.2.1 Reachability in k-paths

Without loss of generality, we can assume that s and t are vertices in some k-cliques X_i and X_j, and not spikes. If s (respectively t) is a spike, then it has at most k out-neighbors (in-neighbors) and we can take one of the out-neighbors (in-neighbors) as the new source s' and new sink t' and check reachability. As there are only k^2 such pairs, we can cycle through all of them in log-space.

So now onwards we assume that s and t are not spikes. The algorithm is based on the observation that a simple s to t path ρ can pass through any clique at most k times. We use a divide- and-conquer approach similar to that used in Savitch's algorithm (which shows that directed reachability can be computed in $DSPACE(\log^2 n)$). The main steps involved in the algorithm are as follows:

1. *Preprocessing step:* Make the cliques disjoint by labelling different copies of each vertex with distinct labels and introducing appropriate edges. Compute reachabilities within each clique, even through its spikes, and then *remove the spikes*. (See *e.g.* Figure 5.1.) Number the cliques $X_1, \ldots, X_m$ from left to right.

2. Now assume that s and t are in cliques X_i and X_j respectively. Without loss of generality, we can assume $i \neq j$. This is because, if $i = j$, we can make another copy X_i' of X_i between X_i and X_{i+1}, join the copies of each vertex in X_i and X_i' by bidirectional edges to preserve reachabilities, remove the set of edges between X_i to X_{i+1} and introduce the same set of edges between X_i' and X_{i+1}. Choose the copy of s from X_i and that of t from X_i'. We also assume that $i < j$, as the case $i > j$ is analogous.

3. Divide the k-path into three parts P_1, P_2 and P_3 where P_1 consists of cliques $X_1, \ldots, X_i$, P_2 consists of cliques $X_i, \ldots, X_j$, and P_3 consists of cliques $X_j, \ldots, X_m$. (See *e.g.* Figure 5.2. The edges are not shown, but are assumed to be present between pairs of consecutive cliques.) Note that X_i (X_j) appears in both P_1 and P_2 (P_2 and P_3 respectively). Now we compute reachabilities of all pairs of vertices in X_i (X_j) when the graph is restricted to P_1 (respectively P_3). Then the reachability of t from s within P_2 is computed, using the previously computed reachabilities within P_1 and P_3.

The details are given below.

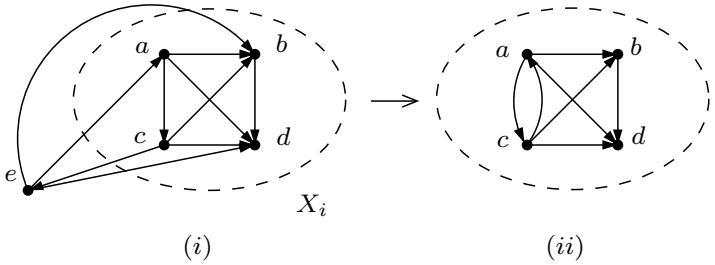

Figure 5.1: Removing spikes from a k-path preserving reachabilities

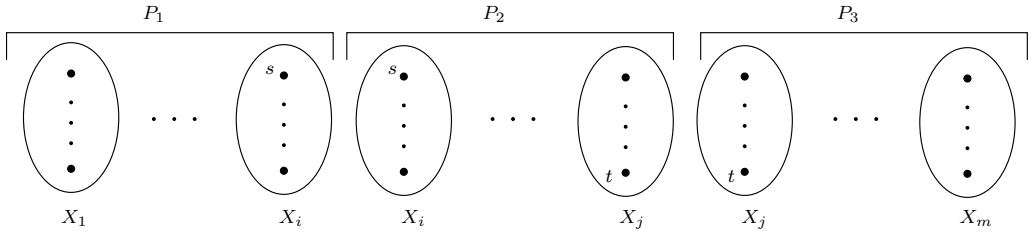

Figure 5.2: Division of the k-path into three parts

Preprocessing: Although adjacent k-cliques in a k-path decomposition share $k-1$ vertices, we perform a preprocessing step, where we give distinct labels to each copy of a vertex. Note that, once the k-path representation for G is computed, each vertex has its copy in several cliques which are consecutive in the path representation. Further, u and v have an edge in G if and only if they both belong to one or more common cliques. Thus, in the path representation, it suffices to keep the edge (u, v) between their copies within each of the common cliques. On the other hand, we join copies of the same vertex in two adjacent cliques by bidirectional edges. Thus the only inter-clique edges are the edges between two copies of the same vertex.

We perform another preprocessing step where we remove the spikes maintaining reachabilities between all pairs of vertices in each clique, and also compute reachabilities within each k-clique. (See *e.g.* Figure 5.1. The spike e is removed and (c, a) edge is added to the clique X_i as there is a path from c to a through e.) The following lemma proves that this can be done in log-space:

Lemma 5.4 *The spikes attached to each clique can be removed in log-space, maintaining reachability between every pair of vertices in the rest of the k-path.*

Proof. A simple path within a clique can be of length at most k. Further, as no two spikes have an edge between them, they can only introduce new reachabilities between pairs of vertices from the clique. Thus, even in the presence of spikes, length of a path between two vertices in a clique

can not exceed $2k$. There are at most $O(n^{2k})$ possible paths, which can be checked in log-space for constant k. ■

Remark 5.5 *There is another way to see this. A spike has edges to all the k vertices in the clique. There are 2^k ways to direct these edges. This partitions the spikes into 2^k equivalence classes. Two spikes which are in the same equivalence class can introduce exactly the same connectivity between any two vertices in the cliques. Thus it suffices to keep only one spike from each equivalence class. Hence the clique and the spikes together form a graph on at most $k + 2^k$ vertices, and as $k = O(1)$, $O(1)$ space suffices to remove the remaining spikes, maintaining the reachabilities among all the pairs of vertices in the clique.*

The Algorithm We describe an algorithm to compute pairwise reachabilities in X_i and X_j in P_1 and P_3 respectively, and also s-t reachability in P_2 using these previously computed pairwise reachabilities.

Algorithm 7 describes this reachability routine. The routine gets as input two vertices u and v, and two indices i and j. It determines whether v is reachable from u in the sub-path $P = (X_i, \ldots, X_j)$. This input is given in such a way that u and v always lie in $X_i \cup X_j$. Consider the case when both u and v are in X_i (or both in X_j). Let l be the center of P. Then a path from u to v either lies entirely in the sub-path $P' = (X_i, \ldots, X_\ell)$ or it crosses X_ℓ at most k times. Thus if $X_\ell = \{v_1, \ldots, v_k\}$ then for $\{v_{i_1}, \cdots, v_{i_r}\} \subseteq X_\ell$ we need to check reachabilities between u and v_{i_1} in P', then between v_{i_1} and v_{i_2} in $P'' = (X_\ell, \ldots, X_j)$ and so on, and finally between v_{i_r} and v. It suffices to check all the r-tuples in X_ℓ, where $0 \leq r \leq k$. The case when $u \in X_i$ and $v \in X_j$ (and vice versa) is analogous. In Algorithm 7, we present only one case where $u, v \in X_i$. Other three cases are analogous.

Thus at each recursive call, the length of the sub-path under consideration is halved, and $O(\log m)$ iterations suffice. We later prove that each iteration stores $O(1)$ data on stack and thus the algorithm works in log-space.

The following lemma gives the complexity analysis of the algorithm:

Lemma 5.6 *Algorithm 7 can be implemented in log-space.*

Proof. We assume that the algorithm uses a stack to store information while making recursive calls. We show that the total work-space and stack-space used is $O(\log n)$, which proves the lemma.

First, we show that $O(k \log k)$ stack-space is used for each recursive call. As u and v are either both in X_i or both in X_j or u in X_i and v in X_j or vice versa, this information can be stored in 2 bits while making the recursive call. Each vertex can be given a label within its clique, which needs $O(\log k)$ bits. Thus labels of u and v within their respective cliques can be stored

Algorithm 7 Procedure IsReach(u, v, i, j)

Input: Pre-processed k-path decomposition of graph G, clique indices i, j, vertex labels $u, v \in X_i \cup X_j$.
Output: Whether v is reachable from u in the sub-path $P = (X_i, \ldots, X_j)$.
if $j - i = 1$ **then**
    Compute the reachability directly, as the sub-path has only $2k$ vertices.
    Return the result.
end if
$l = \frac{j+i}{2}$
Case 1**:** $u, v \in X_i$
if IsReach(u, v, i, l) **then**
    Return 1;
else
    for $q = 1$ to k **do**
        $v_0 \leftarrow u$, $v_{q+1} \leftarrow v$
        for all q-tuples $(v_1, \ldots, v_q)$ of vertices in X_ℓ **do**
            if $\bigwedge_{\substack{x=0 \\ x \text{ even}}}^{q}$ IsReach(v_x,v_{x+1},i,l) $\wedge$ $\bigwedge_{\substack{x=1 \\ x \text{ odd}}}^{q}$ IsReach(v_x,v_{x+1},l,j) **then**
                Return 1;
            end if
        end for
    end for
end if
Case 2**:** $u, v \in X_j$ or $u \in X_i, v \in X_j$ or $u \in X_j, v \in X_i$
Analogous to Case 1

on stack. Next two parameters for a recursive call are either i, l or l, j. One bit is needed to distinguish between these cases. Knowing i and l (respectively l and j), j (i) can be recomputed on returning from the recursive call, and hence need not be stored. There are $O(k^k)$ q-tuples where $1 \leq q \leq k$. The algorithm considers them in lexicographic order. Thus $O(k \log k)$ bits are needed to store the id of the tuple currently under consideration. The position in the q-tuple of the pair of vertices for which a recursive call is made can be stored in $O(\log k)$ bits. This takes $O(k \log k)$ stack-space.

It can be seen that the calling routine can resume its execution by popping this information from the stack. The stack contains only $O(\log n)$ entries at any point of time, one corresponding to each possible value of $j - i$. As the value is halved after each recursive call, there are $\log n$ possible values. If $j - i = 1$, the algorithm uses only $O(k \log k)$ space to compute reachabilities.

■

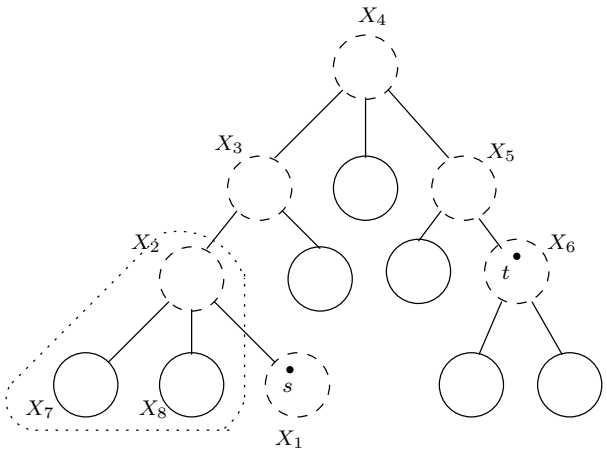

Figure 5.3: Example: A path in a k-tree and the subtrees attached to its nodes

5.2.2 Reachability in k-trees

Given a directed k-tree G in its tree decomposition and two vertices s and t in G, we describe a log-space algorithm that checks whether t is reachable from s. This algorithm uses Algorithm 7 as a subroutine. The algorithm involves the following steps:

1. **Preprocessing:** As in the case of k-paths, assign distinct labels to the copies of each vertex u in different cliques. Introduce a bidirectional edge between the copies of u in all the adjacent pairs of cliques. As reachabilities are maintained during this process, any copy of s and t can be taken as the new s and t. Let X_i and X_j be the nodes in G which contain s and t respectively.

2. **The Procedure:** After this preprocessing, we have a tree T with its nodes as disjoint k-cliques of vertices of G, and s and t are contained in cliques X_i and X_j. Compute the unique undirected path ρ between X_i and X_j in T in log-space. Each node on ρ has two of its neighbors on ρ, except X_i and X_j, which have one neighbor each. An s to t path has to cross each clique X_ℓ in ρ, and additionally, it can pass through the subtree attached to X_ℓ which is disjoint from ρ. (See *e.g.* Figure 5.3. $X_1, \ldots, X_6$ form ρ, shown with dotted circles. The subtree of X_2 is shown and consists of the nodes X_2, X_7, X_8.) We refer to such a subtree as *the subtree T_ℓ of X_ℓ*. We always consider T_ℓ to be rooted at X_ℓ. Thus T_ℓ is the subtree consisting of X_ℓ and those nodes in T which can be reached from X_ℓ without going through any other node on ρ.

   For each node X_ℓ on ρ, we pre-compute the pairwise reachabilities among the k vertices of X_ℓ in the tree T_ℓ. Thus a vertex j in X_ℓ is reachable from a vertex i in X_ℓ if there is a

directed path from i to j in G that passes through only those vertices which have a copy in the nodes of T_ℓ.

Note that once this is done for each node X_ℓ on ρ, we are left with ρ. As ρ is a k-path, we can use Algorithm 7 in Section 5.2.1 to compute reachabilities within ρ.

3. **Computing reachabilities within T_ℓ:** We do this inductively. If T_ℓ contains only one node X_ℓ, we have only k vertices, and their pairwise reachabilities within X_ℓ can be computed in $O(k \log k)$ space. We recursively find the reachabilities within the subtrees rooted at each of the children of X_ℓ in T_ℓ. Let the size of T_ℓ be N. At most one of the children of X_ℓ can have a subtree of size larger than $\frac{N}{2}$. Let X_a be such a child. Recursively compute the pairwise reachabilities for each pair of vertices in X_a within the subtree rooted at X_a. The reachabilities are represented as a $k \times k$ boolean matrix referred to as the *reachability matrix* M for the vertices in X_a, within the subtree rooted at X_a. We use M to compute the pairwise reachabilities of vertices in X_ℓ, within the subtree containing X_ℓ and the subtree rooted at X_a. This gives a new matrix M' of size k^2, and is stored on stack while computing the reachability matrix M'' for another child X_b of X_ℓ. The matrix M' is updated using M'', so that it represents reachabilities between each pair of vertices in X_ℓ within the subtree containing X_ℓ and the subtrees rooted at X_a and X_b. This process is continued till all the children of X_ℓ are processed. The matrix M' at this stage reflects the pairwise reachabilities between vertices of X_ℓ within the subtree T_ℓ.

Lemma 5.7 *The procedure described above can be implemented in log-space.*

Proof. Step 1 can be implemented in log-space in a straight forward way. While implementing Step 2, a log-space transducer computes the reachability matrices for each node X_ℓ on ρ when the graph is restricted to the subtree rooted at X_ℓ. Once these matrices are computed, the k-path algorithm is invoked, which works in log-space.

We now show that the reachability matrix described above for a node X_ℓ on ρ can be computed in log-space. Note that, while making a recursive call for reachabilities in the child X_a of X_ℓ, we do not store anything on stack. On the other hand, while making a recursive call for reachabilities in subsequent children of X_ℓ, we store the matrix M' constructed so far. As k is constant and the size of all the subtrees rooted at the children of X_ℓ excluding X_a is at most $\frac{N}{2}$, the stack depth is always $O(\log n)$ and thus the algorithm works in log-space. ■

5.2.3 Hardness for L

The L-hardness of reachability in k-trees follows from L-hardness of the problem of path ordering defined below [36]:

Definition 5.8 (Path Ordering $\mathrm{ORD}(P, u, v)$**)** *Given a directed path P, specified by giving for each vertex w (except the last) its successor $s(w)$ along the path, and given vertices u, v, decide if u precedes v in P.*

Lemma 5.9 *For any $k \geq 1$* ORD *is* AC^0 *many-one reducible to k-path reachability.*

Proof. As a directed path is a k-path for $k = 1$, the hardness of 1-path follows. Given an instance of ORD, and for a fixed value of $k \geq 2$, construct a k-path instance by introducing $k - 1$ new vertices and joining them to all the vertices in the given instance of ORD with outgoing edges.

Clearly, this is an instance of k-path reachability, and the reduction preserves reachability. The $k - 1$ new vertices form a clique with each vertex w and its successor $s(w)$ in the ORD instance.

■

5.3 Shortest and Longest Paths

We show that the shortest and longest paths in weighted directed acyclic k-trees can be computed in log-space, when the weights are positive and are given in unary. Throughout this section, the terms k-path and k-tree always refer to directed acyclic k-paths and k-trees respectively, with integer weights on edges and we here onwards omit the specification *weighted directed acyclic*. We use the following (weighted) form of Theorem 4.6 from Chapter 4: The proof is exactly similar to that of Theorem 4.6 and we omit it here.

Theorem 5.10 (See Theorem 4.6) *Let $\mathcal{C}$ be any subclass of weighted directed acyclic graphs closed under vertex deletions. There is a function f, computable in log-space with oracle access to* Reach($\mathcal{C}$)*, that reduces* Distance($\mathcal{C}$) *to* Long-Path($\mathcal{C}$) *and* Long-Path($\mathcal{C}$) *to* Distance($\mathcal{C}$)*, where* Reach($\mathcal{C}$)*,* Distance($\mathcal{C}$)*, and* Long-Path($\mathcal{C}$) *are the problems of deciding reachability, computing distance and longest path respectively for graphs in $\mathcal{C}$.*

We use this theorem to reduce the shortest path problem in k-trees to the longest path problem, and then compute the longest (that is, maximum weight) s to t path. The reduction involves changing the weights of the edges appropriately so that the shortest path becomes the longest path and vice versa. This gives a directed acyclic k-tree with positive integer weights on edges given in unary. Given a graph G in $\mathcal{C}$, and two designated nodes s and t, the first step of the shortest to longest path reduction involves removing *useless* vertices. A vertex v is considered to be *useless* if v is unreachable from s, or t is unreachable from v. These are precisely those

vertices which can not participate in any s to t path. This operation makes G a single-source single-sink DAG. Note that the class of k-trees is not closed under vertex deletions, however, we first find the tree-decomposition of the k-tree as in Section 5.1, and then find and delete useless vertices from this tree-decomposition. This makes the size of some of the cliques smaller than k, but the decomposition still has the property that adjacent cliques differ in at most one vertex, and this is sufficient.

The next step of the reduction involves negating the weights of all the edges, so that the shortest path becomes the longest path and vice versa. However, we need positive integer weights on edges. This is achieved by adding a sufficiently large positive number to the weight of each edge, such that the weight of each path increases by a fixed number. This increase in weights is polynomial in the original weights and in n. At the end of the reduction, we have a directed acyclic k-tree with positive integer weights on edges given in unary. We show that the maximum weight of an s to t path can be computed in log-space with a technique which uses ideas from [49]. The algorithm to compute maximum weight s to t path in k-trees uses the algorithm for computing maximum weight path in k-paths as subroutine. Therefore we first describe the algorithm for k-paths in Section 5.3.1

5.3.1 Shortest and Longest Paths in Directed Acyclic k-paths

Let G be a directed acyclic k-path and s and t be two designated vertices in G. The computation of maximum weight of an s to t path is done in five stages, described below in detail. The main idea is to obtain a log-depth circuit by a suitable modification of Algorithm 7, and to transform this circuit to an arithmetic formula over integers, whose value is used to compute the maximum weight of an s to t path in G. The log-space bound uses the following proposition:

Proposition 5.11 *A Boolean formula and an arithmetic formula over integers can be evaluated in log-space [28, 23, 30, 43]. See e.g. [2].*

Computing the maximum weight s to t path in G involves the following steps:

1. **Construct a log-depth Boolean formula from Algorithm 7:** Modify Algorithm 7 so that it outputs a circuit $\mathcal{C}$ that has nodes corresponding to the recursive calls made in Line 15 and the tuples considered in the **for** loop in Line 14.

   More specifically, a node q in $\mathcal{C}$ that corresponds to a recursive call IsReach(u, v, i, j) has children $q_1, \cdots, q_N$, which correspond to the tuples considered in that recursive call (**for**-loop on Line 12 of Algorithm 7). We refer to q as a *call-node* and $q_1, \ldots, q_N$ as *tuple-nodes*. A tuple-node q' corresponding to a tuple $(v_1, \ldots, v_N)$ has call-nodes $q'_1, \ldots, q'_N$ as its children, which correspond to the recursive calls made while considering the tuple

$(v_1, \ldots, v_N)$ (Line 15 of Algorithm 7). The leaves of $\mathcal{C}$ are those recursive calls which satisfy the **if** condition on Line 3 of Algorithm 7, thus they are always call-nodes. These are the recursive calls made for two adjacent cliques. As the depth of the recursion in Algorithm 7 is $O(\log n)$, the circuit $\mathcal{C}$ also has $O(\log n)$ depth. Hence it can be converted to a formula $\mathcal{F}$ by only a polynomial factor blow-up in its size. The maximum number of children of a node is $O(k^k)$ and hence the size of $\mathcal{F}$ is bounded by $O(k^{k \log n})$, which is polynomial in n for constant k.

2. **Prune the Boolean formula:** The next step is to remove those nodes and their subtress from the formula which return a "no" answer in the recursion. These are the subtrees which do not participate in any s to t path. Such subtrees are removed by a log-space transducer as follows:

   The internal call-nodes of $\mathcal{F}$ are replaced by $\vee$ gates and tuple-nodes are replaced by $\wedge$ gates. The leaves of $\mathcal{F}$ are replaced by 0 or 1 depending on whether the corresponding recursive call returned 0 or 1 in the **if** block on Line 3 of Algorithm 7. It can be seen that a sub-formula of $\mathcal{F}$ rooted at a call-node evaluates to 1 if and only if the corresponding recursive call returns 1 in Algorithm 7. Similarly, the sub-formula rooted at a tuple-node evaluates to 1 if and only if the conjunction corresponding to it (on Line 15 of Algorithm 7) evaluates to 1.

   Now, we evaluate in log-space the sub-formula rooted at each node of $\mathcal{F}$. Note that a node that evaluates to 0 does not contribute to any path from s to t, and hence its subtree can be safely removed.

3. **Transformation into a $\{+, max\}$-tree:** The new, pruned formula obtained in Step 2 is then relabelled: Each $\wedge$ label is replaced with a $+$ label and each $\vee$ label with a max label. Each leaf corresponds to calls of the form $IsReach(u, v, i, i+1)$. It is labelled with the length of the maximum weight u to v path confined within cliques i and $i+1$, which can be computed in $O(1)$ space. This weight is strictly positive, since the 0-weight leaves are removed in Step 2. Further, all the weights are in unary. Thus we now have a $\{+, max\}$-tree T with positive, unary weights on its leaves and $+$ or max labels on internal nodes. It is easy to see that the value of the $\{+, max\}$-tree T is the maximum weight of any s to t path in G.

4. **Transformation into a $\{+, \times\}$-tree:** The evaluation problem on the $\{+, max\}$-tree T obtained in Step 3 is then reduced to the evaluation problem on a $\{+, \times\}$-tree T' whose leaves are labelled with positive integer weights coded in binary. This reduction works in log-space and is similar to that of [49]. The reduction involves replacing a $+$-node of T with a $\times$-node, and a max-node with a $+$ node. The weight w of a leaf is replaced with

2^{mw}, where m is the sum of the weights of all the leaves of T plus one. The correctness of the reduction follows from Lemma 5.12, which gives the relation between the value of the $\{+, \max\}$-tree and that of the $\{+, \times\}$-tree. The proof closely follows to a similar lemma in [49].

5. **Evaluation of the $\{+, \times\}$ tree:** The last step is to evaluate the $\{+, \times\}$-tree, which can be done in log-space by Proposition 5.11.

Lemma 5.12 *Let the value of the $\{+, \times\}$-tree T' obtained by the above reduction be v', and that of the $\{+, \max\}$-tree T be v. Then $v = \lfloor \frac{\log_2 v'}{m} \rfloor$.*

Proof. Let u be the value of a node x in the T and u' be its value in the T'. Let s be the sum of the weights of the leaves in the subtree of T rooted at x. We claim that $2^{um} \leq u' \leq 2^{um+s}$ for all nodes x in T'. We proceed by simultaneous structural induction on T and T'. Once this is proved for the root *i.e.* $2^{vm} \leq v' \leq 2^{vm+s}$, where $s = m - 1$ for the root, we get $v \leq \frac{\log_2 v'}{m} < v + 1$ which proves the lemma.

So we prove the claim for all the nodes x in T. When x is a leaf, clearly the claim holds as we set $u' = 2^{um}$. So consider the case when x is an internal node. Let x have ℓ children. Let the values of the subtrees of T rooted at the children of x be $u_1, \ldots, u_\ell$ and let their corresponding values in T' be $u'_1, \ldots, u'_\ell$. Let the sum of the weights of the leaves in the subtrees of T rooted at each of the children of x be $s_1, \ldots, s_\ell$ respectively. By induction hypothesis, the claim holds for each of the children, so $2^{u_i m} \leq u'_i \leq 2^{u_i m + s_i}$ for $1 \leq i \leq \ell$. Also $s = \sum_{i=1}^{\ell} s_i$.

Consider the case when x is a $+$-node in T and hence a $\times$-node in T'. Then $u = \sum_{i=1}^{\ell} u_i$ and $u' = \prod_{i=1}^{\ell} u'_i$. We have the following:

$$2^{um} \;=\; 2^{\sum_{j=1}^{\ell} u_j} \;\leq\; \prod_{j=1}^{\ell} u'_j \;=\; u' \;\leq\; \prod_{j=1}^{\ell} 2^{u_j m + s_j} \;=\; 2^{um+s}$$

Now consider the case when x is a $\max$-node in T. Therefore it is a $+$-node in T'. Also, $u = \max\{u_1, \ldots, u_\ell\} = u_1$, say, and $u' = u'_1 + \ldots + u'_\ell$. Let $s_i = \max\{s_1, \ldots, s_\ell\}$.We have the following:

$$\begin{aligned} 2^{um} \;&=\; 2^{\max\{u_1,\ldots,u_\ell\} m} \;\leq\; \sum_{j=1}^{\ell} 2^{u_j m} \;\leq\; \sum_{j=1}^{\ell} u'_j = u' \\ &\leq\; \sum_{j=1}^{\ell} 2^{u_j m + s_j} \;\leq\; \ell \cdot 2^{u_1 m + \max\{s_1,\ldots,s_\ell\}} \;=\; 2^{u_1 m + s_i + \log_2 \ell} \;\leq\; 2^{u_1 m + s} \end{aligned}$$

where the last inequality follows from the fact that each of the s_is are positive and that $\log_2 \ell \leq \ell - 1$. This completes the proof of the lemma. ∎

5.3.2 Shortest and Longest Paths in Directed Acyclic k-trees

Given a directed acyclic k-tree (in its tree-decomposition) G, two vertices s and t in G, and weights on the edges of G, encoded in unary, we show how to compute the maximum weight of an s to t path in G. Unlike the case of k-paths, the reachability algorithm for k-trees given in Section 5.2.2 can not be used to get a log-depth circuit. This is because the recursion depth of the algorithm is the same as the depth of the k-tree. Therefore we need to find another way of recursively dividing the k-tree into smaller and smaller subtrees, as we did for k-paths in Sections 5.2.1 and 5.3.1.

Our recursive splitting procedure for k-trees is based on the technique used in the following result of [61]:

Lemma 5.13 *(***Lemma** 6 **of [61], also see [26]***) Let M be a visibly pushdown automaton accepting well-matched strings over an alphabet Δ. Given an input string x, checking whether $x \in L(M)$ can be done in log-space.*

Before describing the procedure, we give some background on the notation used in the above lemma. A *visibly pushdown automaton* is a pushdown automaton with the additional restriction that the push/pop actions depend only on the current input symbol. An example of a language that can be recognized by such an automaton is the set of strings of *well-matched* parentheses. A string of opening and closing parentheses is said to be well-matched if and only if every opening parenthesis can be matched with a corresponding closing parenthesis on its right and vice versa.

The algorithm given in [61] works by recursively dividing the input string x into three disjoint, well-matched, smaller substrings such that, over two stages of this division, the length of each of the smaller strings is at most a $\frac{3}{4}$ fraction of the length of x. Thus the recursion terminates in $O(\log n)$ steps.

To use this algorithm, we order the children of each of the nodes of the k-tree in a particular way, then label the leaves with opening and closing parentheses in such a way that the concatenation of the labels of the leaves of the subtree rooted at each internal node forms a balanced parentheses expression x. We add dummy leaves if necessary. The algorithm of [61] is given x as input. The recursive splitting of x in the algorithm corresponds to splitting the k-tree into smaller subtrees.

Thus, using the algorithm, we can compute *a set of recursive separators for a tree*, defined below:

Definition 5.14 *Given a rooted tree T, separators of T are two nodes a and b of T such that*

1. *The subtrees rooted at a and b respectively are disjoint,*

2. *T is split into subtrees T_1, T_2, T_3 where T_1 consists of a, some of the children of a, and subtrees rooted at them, T_2 is defined similarly for b, and T_3 consists of the rest of the tree along with a copy of a and b each.*

3. *After two such stages of splitting, each of the subtrees consists of at most a $\frac{3}{4}$ fraction of the leaves of T.*

This process is done recursively until the number of leaves in the subtrees is two. Such a subtree is in fact a path.

A set of recursive separators of T consists of the separators of T and separators of all the subtrees obtained in the recursive process.

The following lemma gives the procedure to compute a set of recursive separators of a tree T:

Lemma 5.15 *Given a tree T, the set of recursive separators of T can be computed in log-space.*

Proof. The steps of the algorithm are as follows:

1. By adding dummy leaves, ensure that each internal node has an even number of children which are leaves, and there are at least two such children.

2. Arrange the children of each node from left to right such that the non-leaves are consecutive, and there is an equal number of leaves to the left and to the right of this group of non-leaves.

3. For each internal node, label the left half of its leaf-children with '(' and the right ones by ')'. This ensures that the leaves of the subtree rooted at each internal node form a balanced parentheses expression. Moreover, the converse also holds. That is, leaves which form a balanced parentheses expression are consecutive leaves in the subtree rooted at an internal node.

The leaves of T now form a balanced parentheses expression, and we run the algorithm of [61] on this string. The recursive splitting of the string into smaller and smaller substrings corresponds to the recursive splitting of T at some internal nodes, which satisfies Definition 5.14. This is ensured by the way the leaves are labelled. Each balanced parentheses expression corresponds to either a subtree rooted at an internal node or the subtrees rooted at some of the children of an internal node.

The subtrees obtained after every two stages of splitting a tree have at most $\frac{3}{4}$th of the number of leaves in the tree. Thus after every two stages of recursion, the number of leaves in the subtrees is reduced by a constant fraction. When there are only two leaves in a subtree, the recursion stops.

Moreover, the algorithm of [61] can output all the substrings formed at each stage of recursion in log-space. As a substring completely specifies a subtree of T, our procedure outputs the set of recursive separators for T in log-space. ■

Once an algorithm to compute the set of recursive separators for k-trees is known, a reachability routine similar to Algorithm 7 can be designed in a straight forward way. Obtain the set of recursive separators corresponding to two stages of the algorithm of [61]. To see this, let s and t be in cliques a and b of T, respectively. Let c and d be separators of T A simple s to t path can pass through the subtrees rooted at c and d at most $2k$ times. Thus, as in the case of k-paths, a reachability query for s to t is broken into $O(k^k)$ reachability queries on three smaller subtrees at each step, the subtrees being those rooted at c and d, and the subtree obtained by removing these two subtrees. The number of leaves in each of the subtrees obtained after two such stages of splitting is at most $3/4$ times that in the original tree. Thus the recursion terminates after $O(\log n)$ iterations and we get subtrees with two leaves. Such subtrees are in fact k-paths and we use the reachability algorithm for k-paths. This gives a log-space routine for reachability in k-trees, which has $O(\log n)$ depth of recursion. From the reachability routine, the computation of maximum weight path follows from the steps 1 to 5 described in Section 5.3.1. Therefore we get the following theorem:

Theorem 5.16 *Given a (weighted) directed acyclic k-tree G with unary weights on edges, and two designated vertices s and t in G, the maximum weight of a simple path from s to t can be computed in log-space.*

5.4 Distance Computation in Undirected k-trees

We give a simple log-space algorithm for computing the shortest path between two given vertices in an undirected k-tree. We use the decomposition of [52], where a k-tree is decomposed into layers, such that the following properties are satisfied:

1. Layer 0 is a k-clique. Each vertex in layer $i > 0$ has exactly k neighbors in layers $j < i$. Further, these neighbors of i which are in layers lower than that of i form a k-clique.

2. No two vertices in the same layer share an edge.

This decomposition is log-space computable [53]. Moreover, given two vertices s and t, it is always possible to find a decomposition in which t lies in layer 0. This can also be done in log-space. If both s and t are in layer 0, then there is an edge between s and t, which is the shortest path from s to t. Therefore assume that s lies in a layer $r > 0$. The following claims lead to a simple algorithm:

Claim 5.17 *The shortest s to t path never passes through two vertices u and v such that $layer(u) < layer(v)$.*

Proof. Assume the contrary *i.e.* let there be a shortest s to t path ρ which passes from a vertex in a lower layer to a vertex in a higher layer. Since $layer(s) > layer(t)$, there are three consecutive vertices u, v, w on ρ such that $layer(v) > layer(u)$ and also $layer(v) > layer(w)$. Then by the properties above, u and w are part of a k-clique and hence (u, w) is an edge in the graph. But this gives a shorter path than ρ by removing v, which contradicts our assumption. ■

Claim 5.18 *There is a shortest path from s to t passing through the neighbor of s in the lowest layer.*

Proof. Let $\{v_1, \ldots, v_k\}$ be the neighbors of s in layers lower than that of s. Thus they form a k-clique. Let $layer(v_1) < \ldots < layer(v_k)$. Let there be no shortest path from s to t that passes through v_1. Consider a shortest s to t path $\rho = (s, v_i, u_1, \ldots, u_r, t)$, $i \neq 1$. As $v_1, \ldots, v_k$ form a k-clique, $(v_1, v_i) \in E$. As $layer(v_1) < layer(v_i)$, $layer(u_1) < layer(v_i)$ and $(u_1, v_i) \in E$, it must be the case that $(v_1, u_1) \in E$. Thus v_i on ρ can be replaced by v_1 which contradicts the assumption that there is no shortest s to t path passing through v_1. ■

These claims suggest the following simple algorithm which can be implemented in log-space: Start from s and choose the next vertex from the lowest possible layer, at each step till we reach layer 0.

5.5 Discussion

In [49], the path problems for series-parallel graphs, also known as partial 2-trees have been completely characterized, and the corresponding problems for k-trees with $k > 2$ are mentioned as open questions. We resolve these open questions and show a matching L lower bound to complete the characterization of path problems in k-trees. All our log-space results hold directly only for k-trees and not for partial k-trees which are also equivalent to tree-width k graphs. The reason being that a tree decomposition for partial k-trees is apparently more difficult to construct (best known upper bound is LogCFL[91]) as opposed to k-trees (for which it can be done in L [53]).

However, we observe that, if the tree-decomposition of a partial k-tree is given as input, then the algorithms described in this chapter can be used for path problems in partial k-trees as well.

Part III

Clustering

6

The Planar k-means Problem is NP-hard

In this chapter, we consider a variant of the clustering problem, known as the k-means problem, and prove that the planar version of this problem is NP-hard. For a brief overview of the clustering problem and known results, see Section 1.2.3.

6.1 Background and Preliminaries

The clustering problem involves partitioning a finite set of objects into different chunks so as to minimize certain objective function. These chunks are called *clusters*. We consider the geometric version of this problem, where the objects are points in $\mathbb{R}^m$. We recall the definition of the k-means problem:

Definition 6.1 *Given a set of n points $P = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^m$, find a set of k points $B = \{b_1, b_2, \ldots, b_k\} \subset \mathbb{R}^m$ such that*

$$\sum_{i=1}^{n} [d(p_i, B)]^2$$

is minimized. This minimum value is denoted Opt(P, k).

Here $d(p_i, B)$ is the Euclidean distance from p_i to the nearest point in B; $d(p_i, B) = \min_{1 \le j \le k} d(p_i, b_j)$.

We consider the problem for $m = 2$, and refer to it as *planar k-means*. In particular, we consider the decision version: Is Opt$(P, k) \le S$? Here S is part of input.

Choosing the set B of k centers fixes a clustering $\mathcal{C}$ of the points in P, with each point going to its nearest center (breaking ties arbitrarily). On the other hand, if a subset of points $C \subseteq P$ is known to form a cluster, then the center of the cluster is uniquely determined as the centroid of the points in C. Thus we can talk of the cost of a cluster $C = \{p_1, \ldots, p_\ell\}$ and the cost of a

clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$:

$$\begin{aligned} \text{Opt}(C, 1) &= \text{Cost}(C) = \min_b \sum_{i=1}^{m} [d(p_i, b)]^2 \\ \text{Cost}(\mathcal{C}) &= \sum_{j=1}^{k} \text{Cost}(C_j) \end{aligned}$$

Thus $\text{Opt}(P, k)$ is the minimum, over all clusterings $\mathcal{C}$ of P into k clusters, of $\text{Cost}(\mathcal{C})$.

We show the NP-hardness of planar k-means by a reduction from planar 3-SAT [60].

Definition 6.2 *([60]) Let F be a 3-CNF formula with a set of variables $\{v_1, \ldots, v_n\}$ and clauses $\{c_1, \ldots, c_m\}$. We call $G(F)$=(V, E) the graph of F, where*

$$\begin{aligned} V &= \{v_i | 1 \le i \le n\} \cup \{c_j | 1 \le j \le m\} \\ E &= E_1 \cup E_2 \text{ where} \\ E_1 &= \{(v_i, c_j) | v_i \in c_j \text{ or } \bar{v_i} \in c_j\} \\ E_2 &= \{(v_j, v_{j+1}) | 1 \le j < n\} \cup \{(v_n, v_1)\} \end{aligned}$$

If $G(F)$ is a planar graph, F is called a planar 3-CNF formula. The planar 3-SAT problem is to determine whether a given planar 3-CNF formula F is satisfiable.

We note that our reduction, in fact, requires only the graph (V, E_1) to be planar. (Some of the literature in fact refers to this sub-graph as the graph of F, but we follow the convention from [60].)

Henceforth throughout this note, we use the term distance to mean square of Euclidean distance. That is, $\text{dist}(p, q) = [d(p, q)]^2$. We will be explicit when deviating from this convention.

We use the following well known or easily verifiable facts about the k-means problem [47, 35].

Proposition 6.3 *1. The cost of a cluster of points is half the average sum of distances from a point to the other points in the cluster:*

$$\textit{Cost}(C) = \frac{1}{2|C|} \sum_{p \in C} \sum_{q \in C; q \neq p} \textit{dist}(p, q)$$

In other words, if $C = \{p_1, p_2, \ldots, p_\ell\}$, then

$$\textit{Cost}(C) = \frac{1}{\ell} \sum_{i=1}^{\ell} \sum_{j=i+1}^{\ell} \textit{dist}(p_i, p_j)$$

In the following, we will use this form as the definition for the cost of a cluster. This makes the problem discrete: though there are uncountably many choices for the cluster centers, there are only finitely many partitions of P into k clusters.

2. *If, in an instance of the k-means problem, the given points form a multiset, then we say that a clustering is multiset-respecting if it puts all points at the same location into the same cluster.*

   Every instance of the k-means problem has a multiset-respecting optimal clustering.

3. *Let P be a multiset instance of the k-means problem, and let P' be the instance obtained by adding a point p to P. Then $\forall k$, Opt$(P, k) \leq$ Opt(P', k).*

4. *In particular, adding a point to a cluster cannot decrease the cost of that cluster; Cost$(C) \leq$ Cost$(C \cup \{p\})$.*

5. *If clustering $\mathcal{C}'$ refines clustering $\mathcal{C}$ (that is, every cluster in $\mathcal{C}$ is the union of some clusters in $\mathcal{C}'$), then Cost$(\mathcal{C}') \leq$ Cost$(\mathcal{C})$.*

6.2 Reduction from Planar 3-SAT to Planar k-means

Let F be the given planar 3SAT instance with n variables and m clauses. We construct an instance I of planar k-means corresponding to F. We list the required properties of this instance in Section 6.2.1. The correctness of the reduction is proved in Section 6.2.2. In Section 6.2.3, we describe a layout which indeed satisfies these properties. The reduction may introduce irrational coordinates in the resulting k-means instance. Rounding these irrational coordinates to sufficiently close rational points is described in Section 6.2.4.

6.2.1 Properties of the layout

The corresponding k-means instance I we construct will satisfy the following:

1. Corresponding to each variable x_i, there is a simple circuit s_i in the plane, with an even number of vertices marked on it. At each vertex on such a circuit, M copies of a point are placed. The circuits for different variables do not intersect.

   For each circuit, its vertices can be partitioned into pairs of adjacent vertices in two ways. We associate one of them (chosen arbitrarily) with the assignment $x_i = 1$ and the other with $x_i = 0$. We call the first pairing the 'true matching' and the other pairing the 'false matching'.

2. Let u, v be any two distinct vertices taken from any of the circuits (not necessarily the same circuit). If u and v are adjacent on some circuit, then the distance between them is β. Otherwise, the distance between them is at least 2β.

3. There is a point p_j corresponding to every clause C_j. If $x_i \in C_j$ ($\bar{x_i} \in C_j$) then there is a unique nearest edge (u, v) on the true (respectively false) matching of the circuit s_i such that p_j is equi-distant from u and v. It is at distance α from the midpoint of uv, and hence at a distance $\alpha + \frac{\beta}{4}$ from u and v. All vertices other than the endpoints of these nearest edges (two per literal in the clause, so at most six) are at a distance at least $\alpha + \frac{5\beta}{4}$ from p_j.

   Clause points p_j and p_l, for $l \neq j$, are at distance at least θ from each other.

4. The instance I consists of all the clause points, and M copies of a point at each vertex on each circuit s_i. The parameters satisfy

$$M \geq \frac{6\alpha m}{\beta} \qquad\qquad \theta \geq 2(M+1)\alpha m$$

5. The value of k is given by

$$k = \sum_{i=1}^{n} \frac{|s_i|}{2}$$

We ensure that the optimal k-means clustering puts the points in each circuit s_i into $\frac{|s_i|}{2}$ clusters by dividing them into either true pairs or false pairs. (Thus these clusters contain $2M$ points.) Every clause point p_j has at most three pairs of point locations at distance α from itself. It is clustered with one of these pairs if that pair forms a cluster in the circuit s_i. Otherwise, the optimal clustering puts p_j along with some pair of point locations that forms a cluster in the circuit it appears in. In particular, if x_i is assigned a value 1, then in the corresponding k-means clustering, points of s_i are clustered according to the true matching, otherwise they are clustered according to the false matching. Similarly, if the assignment to a variable x_i satisfies a clause c_j, then the clause point p_j is at distance $\alpha + \frac{\beta}{4}$ from the vertices of a cluster in s_i, otherwise it is at distance strictly greater than $\alpha + \frac{\beta}{4}$ from at least one vertex in every cluster in s_i.

We need to show that

1. A layout satisfying the above properties gives a correct reduction from planar 3-SAT to planar k-means. This is done in Section 6.2.2 below.

2. The layout is indeed possible for some choice of α, β, θ, M, and can be obtained in polynomial time. This is done in two stages: a layout with irrational coordinates is described in Section 6.2.3, and in Section 6.2.4 we describe how to eliminate the irrational coordinates.

6.2.2 Correctness of the Reduction

Consider clustering of only circuit points into k non-empty clusters.

Lemma 6.4 *1. Clustering the circuit points into consecutive pairs (i.e. into the true or the false matching for each variable) has cost* $\frac{kM\beta}{2}$.

2. Any other multiset-respecting clustering of circuit points has cost at least $\frac{kM\beta}{2} + \frac{M\beta}{3}$.

Proof. Let A be any matching-based k-means clustering of the circuit points. Then using Proposition 6.3(1) we can see that $\text{Cost}(A) = \frac{kM\beta}{2}$, since the cost of each cluster is $\frac{M\beta}{2}$.

Let B be some multiset-respecting clustering that does not correspond to a matching on the circuits. By the size of a cluster, we mean the number of distinct vertices (and hence all M points at that vertex) in it.

If the largest cluster in B has 2 vertices, then every cluster is a pair, and at least one pair is not consecutive on any circuit. Hence

$$\text{Cost}(B) \geq \frac{(k-1)M\beta}{2} + \frac{M^2(2\beta)}{2M} = \frac{kM\beta}{2} + \frac{M\beta}{2}$$

satisfying the claimed bound.

So now assume that B has some larger clusters too. Let B contain p clusters of sizes $l_1, \ldots, l_p$ more than 3 each, q clusters of size 3 each, r clusters of size 2 each, and s clusters of size 1 each. Then we have the following:

$$p + q + r + s = k \tag{6.1}$$

$$\sum_{i=1}^{p} l_i + 3q + 2r + s = 2k \tag{6.2}$$

Subtracting twice the first equation from the second, and using $p = \sum_{i=1}^{p} 1$, we get

$$s = \sum_{i=1}^{p} (l_i - 2) + q \tag{6.3}$$

For a cluster C of size $l \geq 4$, the best possible situation is that l pairs within the cluster are edges on some circuit. Thus the cost of such a cluster is at least

$$\text{Cost}(C) \geq \frac{1}{lM}\left[lM^2\beta + \left(\binom{l}{2} - l\right) M^2 2\beta\right] = (l-2)M\beta$$

Similarly, in a cluster of size 3, at most two pairs can be edges on a circuit (the circuits are of even length), so the cost is at least $4M\beta/3$.

The cost of the (p, q, r, s) clustering B thus satisfies:

$$\begin{aligned}
\text{Cost}(B) &\geq M\beta\left[\sum_{i=1}^{p}(l_i - 2) + \frac{4q}{3} + \frac{r}{2}\right] \\
&= \frac{M\beta}{2}\left[s + r + q + \frac{2q}{3} + \sum_{i=1}^{p}(l_i - 2)\right] \qquad \text{from } (Equation\ 6.3) \\
&\geq \frac{M\beta}{2}\left[s + r + q + p + \frac{2q}{3} + p\right] \qquad \because l_i - 2 \geq 2 \\
&= \frac{M\beta}{2}\left[k + p + \frac{2q}{3}\right] \qquad \text{from } (Equation\ 6.1) \\
&\geq \frac{kM\beta}{2} + \frac{(p+q)M\beta}{3} \\
&\geq \frac{kM\beta}{2} + \frac{M\beta}{3} \qquad \because p + q \geq 1
\end{aligned}$$

Thus any multiset-respecting clustering of circuit points that is not a matching based clustering has a cost larger than the matching based clusterings, and the difference is at least $\frac{M\beta}{3}$. ■

Lemma 6.5 *The formula is satisfiable if and only if there is a clustering of value at most* $\frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m$.

Proof. ($\Rightarrow$:) Consider one of the satisfying assignments of the formula. A clustering can be constructed from it as follows:

If $x_i = 1$ (respectively $x_i = 0$), cluster the points of s_i according to the true (false) matching. As every clause C_j is satisfied, fix one of the variables x_i that satisfies it. Put the clause point p_j with the nearest pair of s_i. If $x_i = 1$, then points of s_i are clustered into true matching pairs. Further, x_i appears in C_j in non-negated form, and so, by our construction, p_j is at a distance α from the midpoint of one of the true matching pairs. Thus p_j can be clustered with this pair. The cost of this cluster is

$$\begin{aligned}
\text{Cost(cluster)} &= \frac{1}{2M+1}\left(M^2\beta + 2M\left(\alpha + \frac{\beta}{4}\right)\right) \\
&= \frac{2M}{2M+1}\alpha + \frac{M\beta}{2}
\end{aligned}$$

The case $x_i = 0$ is analogous. Clustering all clause points in this way gives a clustering where m clusters contribute $\frac{M\beta}{2} + \frac{2M}{2M+1}\alpha$ each, and the remaining contribute $\frac{M\beta}{2}$ each, giving an overall value of $\frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m$.

($\Leftarrow$:) Suppose there is a clustering of value at most $\frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m$. By Proposition 6.3(2), we can assume that there is a multiset-respecting clustering $\mathcal{C}$ with this value. Let $\mathcal{C}'$ denote the

restriction of $\mathcal{C}$ to circuit points. By Proposition 6.3(4), adding the clause points cannot decrease the cost of the clustering; thus $\text{Cost}(\mathcal{C}') \leq \text{Cost}(\mathcal{C})$. Now we prove a series of claims:

1. The restriction of $\mathcal{C}$ to circuit points, $\mathcal{C}'$, has exactly k non-empty clusters.

   If $\mathcal{C}'$ has fewer clusters, then there is a cluster with more than 2 points. Refine the clustering by removing one point from such a cluster and putting it in a cluster by itself. Repeat until there are exactly k clusters, to get clustering $\mathcal{C}''$ of circuit points. By Proposition 6.3(5), $\text{Cost}(\mathcal{C}'') \leq \text{Cost}(\mathcal{C}')$. $\mathcal{C}''$ is not matching-based (since we created singleton clusters), so by Lemma 6.4, it contributes a value of at least $\frac{kM\beta}{2} + \frac{M\beta}{3}$. Since

   $$\frac{kM\beta}{2} + \frac{M\beta}{3} \leq \text{Cost}(\mathcal{C}'') \leq \text{Cost}(\mathcal{C}') \leq \text{Cost}(\mathcal{C}),$$

   we have

   $$\text{Cost}(\mathcal{C}) - \left(\frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m\right) \geq \frac{M\beta}{3} - \frac{2M}{2M+1}\alpha m \geq \frac{M\beta}{6} > 0,$$

   where the second inequality follows by our choice of M. We have reached a contradiction to our assumption about $\text{Cost}(\mathcal{C})$.

2. $\mathcal{C}'$ is a matching-based clustering. That is, in $\mathcal{C}$, all circuit points are clustered into a matching based clustering.

   If not, then by the argument used above for $\mathcal{C}''$, we obtain a contradiction to our assumption about $\text{Cost}(\mathcal{C})$.

3. No cluster in $\mathcal{C}$ has more than one clause point.

   If some cluster C has two or more clause points, then let u, v be the vertices of the matching in C, and let p, q be two distinct clause points in it. By Proposition 6.3(4), the cost of the cluster C is at least the cost of the cluster containing just u, v, p, q. Thus using Proposition 6.3(1), we have

   $$\text{Cost}(C) \geq \frac{1}{2M+2}\left[M^2\beta + 4M\left(\alpha + \frac{\beta}{4}\right) + \theta\right] = \frac{M\beta}{2} + \frac{4M\alpha + \theta}{2(M+1)}$$

All other clusters have a cost of at least $M\beta/2$ each, so the overall cost is at least

$$\begin{aligned}\text{Cost}(\mathcal{C}) &\geq (k-1)\frac{M\beta}{2} + \frac{M\beta}{2} + \frac{4M\alpha+\theta}{2(M+1)} \\ &\geq \frac{kM\beta}{2} + \frac{4M\alpha + 2(M+1)\alpha m}{2(M+1)} \quad \text{by our choice of } \theta \\ &> \frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m \qquad \text{a contradiction.}\end{aligned}$$

4. Each clause point is clustered with the nearest pair of circuit points, which should also be a matching pair in the matching based clustering.

   Every cluster containing a clause point has cost $\frac{M\beta}{2} + \frac{2M\alpha}{2M+1}$ if the circuit edge in the cluster is nearest the clause point, and has cost at least $\frac{M\beta}{2} + \frac{2M}{2M+1}\alpha + \frac{M\beta}{2M+1}$ otherwise.

Thus a satisfying assignment can be constructed from this clustering. ■

6.2.3 The Details of the Layout

We now describe the layout obtained from the planar 3SAT formula F that gives us the desired instance I of the k-means problem. Let $G = (V, E)$ be the associated planar clause-variable incidence matrix. (From Definition 6.2, $G = (V, E_1)$. Since $G(F)$ is planar, so is G.) Note that the vertex set V of G can be partitioned into two sets: X corresponding to variable vertices, and Y corresponding to clause vertices, and G is bipartite with $E \subset X \times Y$. All vertices in Y have degree at most 3, and all vertices in X have degree at most m.

1. Let $\mathcal{E}$ be a planar combinatorial embedding of G; such an embedding can be obtained in polynomial time, and even in log space. (See for instance [4].) $\mathcal{E}$ corresponds to some plane drawing of G and specifies, for each vertex v, the cyclic ordering of the edges incident on v in this drawing.

2. Construct a related bounded-degree planar graph H and an embedding $\mathcal{E}'$ as follows: replace each vertex $v \in X$ by a cycle C_v on m vertices, $v_1, v_2, \ldots, v_m$. Reroute the $d(v)$ edges incident on v in G to the first $d(v)$ of these vertices, in the same order as dictated by $\mathcal{E}$. It is straightforward to see that H is planar, and its embedding $\mathcal{E}'$ can be easily obtained from $\mathcal{E}$. The maximum degree of any vertex in H is 3. The vertex set of H is the disjoint union of X' and Y, where $X' = X \times [m]$.

3. Consider a plane drawing of H where vertices are embedded at points on an integer grid, and edges are embedded as rectilinear paths. Such a drawing can be obtained in polynomial time [59, 87], and even in logarithmic space [4].

4. Inflate the grid by a factor of $b \geq 14$.

   This ensures, in particular, that every vertex or bend point u is at the centre of a big box B_u of size $b \times b$, and a small box S_u of size 6×6. The big boxes for different grid points have disjoint interiors, and thus contain no other vertex or bend point even on their boundaries.

   Consider an edge connecting vertex $[x, k] \in X'$ with vertex $y \in Y$. Replace it by a pair of parallel rectilinear paths separated by two grid squares. At the y end, join up these paths along the boundary of S_y. At the $[x, k]$ end, splice them along with the edges to $[x, k-1]$ and $[x, k+1]$ to form a continuous path. See Figure 6.1. Note that some additional rectilinear bends might be required at the $[x, k]$ end, (see, for example, $(x, 4)$ in Figure 6.1).

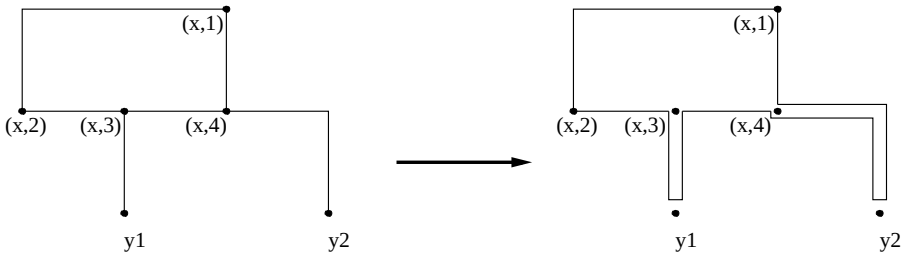

Figure 6.1: Creating circuits for variables

   For each vertex $x \in X$ (and hence for each variable in F), this process distorts the cycle C_x in H into a circuit t. Let t_i denote the circuit corresponding to variable x_i. Since t_i is a rectilinear circuit on a grid, it is of even length.

5. Each clause point y_j is now moved to the center of one of the grid squares touching it, the one that is to the North-West. Extend the three circuits "incident" to the clause point, if necessary, so that all incident circuits are at a Euclidean distance of precisely $\frac{5}{2}$ times the grid length from the moved clause point. See the layout and the modification in Figure 6.2.

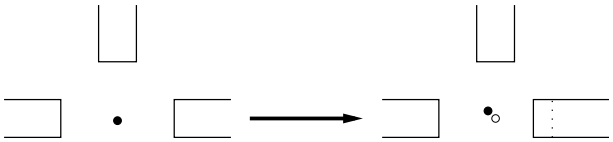

Figure 6.2: Repositioning clause points

6. For each circuit t_i, arbitrarily fix one of its perfect matchings as the true matching, and the other as the false matching.

   Let clause c_j contain variable x_i positively (negatively, respectively). If in the layout so far, y_j is nearest a true (false, resp.) edge of t_i, then nothing needs to be done. If, however,

the edge of t_i nearest y_j is a false (true, resp.) edge, further deform t_i in the area within B_{y_j} but outside S_{y_j}. Replace a sub-path of length two (on each parallel path) by a path of length three, with the vertices laid out on a regular semi-hexagon and hence at distance one from their neighbours on the circuit. Change the true/false matchings within B_{y_j} to be consistent with the labelling outside. This makes the edge nearest y_j a true edge if it was false earlier, and vice versa. The overall length of the circuit remains even. See Figure 6.3.

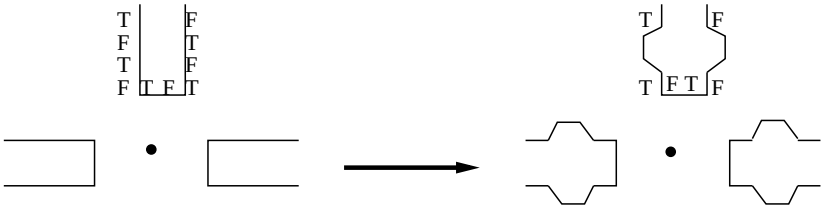

Figure 6.3: Adjusting the parity of circuits relative to clause points

Since the grid was inflated sufficiently, these distortions do not affect other vertices / bends.

After doing this distortion wherever needed, the resulting circuit for a variable is the required circuit s_i.

Figure 6.4 gives the complete layout for a small planar 3SAT instance.

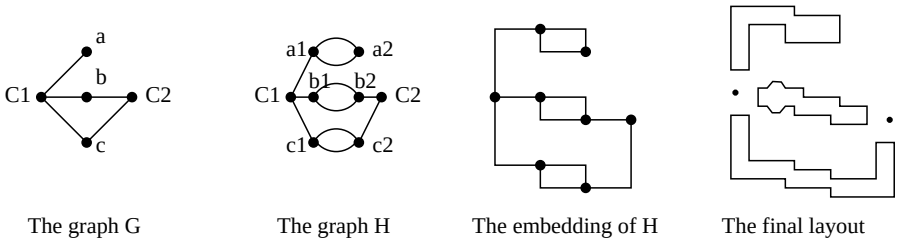

Figure 6.4: The layout for $F = (a \vee b \vee c) \wedge (\overline{b} \vee c)$

Let the squared unit length of the grid be β. Then $\alpha = (\frac{5}{2})^2\beta = 6.25\beta$. Any two clause points are separated either vertically or horizontally by b grid lengths, so the distance between them is at least $\theta = b^2\beta$.

Figure 6.5 shows the box B_u for a clause point u, and within this box the smallest distances are demonstrated. The nearest vertices are A_3 and A_4 (and the corresponding vertices on the other two circuits as well). The distances satisfy the following:

point pair	distance	point pair	distance
		u, A_2	$\alpha + 6\beta + \beta/4$
A_i, A_{i+1}	β	u, A_3	$\alpha + \beta/4$
A_1, A_3	3β	u, A	α
A_2, A_4	2β	u, A_4	$\alpha + \beta/4$
A_3, A_5	4β	u, A_5	$\alpha + 2\beta + \beta/4$

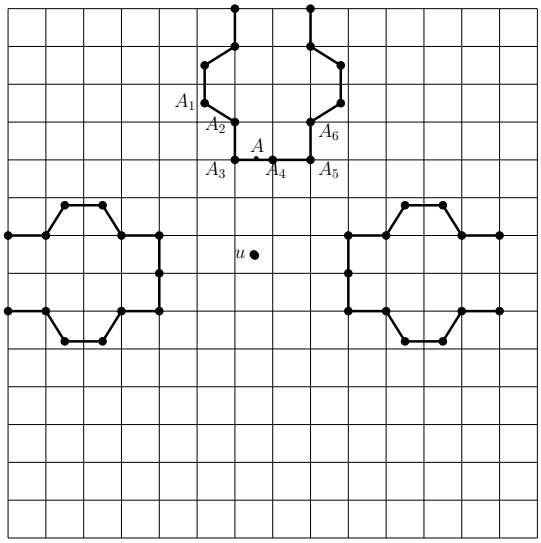

Figure 6.5: The distances α, β shown inside B_u for a clause point u.

It is straightforward to see that with $M = 38m$, $b = 28m$, all the required conditions on the parameters are satisfied.

6.2.4 Dealing with the irrational coordinates

In the last step of the layout, where we replace certain sub-paths of length 2 by sub-paths of length 3, the numerators of the point coordinates become irrational. Essentially, we introduce multiples of $\sqrt{3}$ in the numerator. However, there is a gap in Lemma 6.5 between the k-means clustering costs corresponding to satisfiable and unsatisfiable instances. So we may "round" these irrational points to sufficiently close rational points, while still preserving a non-zero gap.

Lemma 6.5 shows that the optimal k-means clustering cost is at most $\mu = \frac{kM\beta}{2} + \frac{2M}{2M+1}\alpha m$ in the case where the original planar 3-CNF formula is satisfiable; a quick glance at the proof shows that if the original formula is not satisfiable, the optimal k-means clustering cost is at least $\mu + \lambda$, where

$$\lambda = \min\left(\frac{M\beta}{6}, \frac{2M\alpha}{M+1}, \frac{M\beta}{2M+1}\right).$$

Let $\varepsilon = \frac{\lambda/2}{\mu+\lambda} < \frac{\lambda/2}{\mu}$.

Let d_{min} denote the smallest Euclidean distance between two different locations in the layout. We consider a simplistic rounding: for a location with coordinates (x, y) where, say, x is irrational (only one coordinate is irrational in the construction so far), move the location to (x', y) so that x' is rational, and $|x' - x| < \frac{\varepsilon d_{min}}{8}$. Observe that it is possible to find such an x' in polynomial time ([64], Theorem 1.4.7). Now if p and q denote two points in the original layout, and p' and q' denote the corresponding points in the rounded layout, then

$$d(p', q') < d(p, q) + \frac{\varepsilon d_{min}}{4} \leq d(p, q) \left[1 + \frac{\varepsilon}{4}\right]$$

(recall, $d(u, v)$ is the Euclidean distance between u and v) and so

$$\text{dist}(p', q') < \text{dist}(p, q) \left[1 + \frac{\varepsilon}{2} + \frac{\varepsilon^2}{8}\right] \leq \text{dist}(p, q) \left[1 + \varepsilon\right]$$

Similarly, we can see that $\text{dist}(p', q') > \text{dist}(p, q) \left[1 - \varepsilon\right]$. Thus,

$$(1 - \varepsilon)\text{dist}(p, q) < \text{dist}(p', q') < (1 + \varepsilon)\text{dist}(p, q).$$

Now, Proposition 6.3 (1) implies that for any clustering of the points, the ratio of the cost in the rounded layout to the cost in the original layout is strictly greater than $(1 - \varepsilon)$ and strictly less than $(1 + \varepsilon)$.

Thus, the optimal k-means clustering cost in the rounded layout is strictly less than $(1 + \varepsilon)\mu \leq \mu + \lambda/2$ when the input formula is satisfiable, and is strictly greater than $(1-\varepsilon)(\mu+\lambda) \geq \mu + \lambda/2$ when the input formula is not satisfiable.

6.3 Discussion

We have shown that the k-means clustering problem remains NP-complete even in two dimensions, when the number of centers k is part of the input. The NP-hardness of this problem has been independently observed by Andreas Vattani, [88].

There are still some unsettled issues regarding this hardness. An obvious question is whether there are natural parameters associated with planar k-means instances such that when these parameters are restricted in some way, the problem becomes tractable.

- One possible choice of the parameter is k, the number of centers itself. It is known that for planar k-means with constant values of k, there is a polynomial-time algorithm due to [47]. Our reduction places no bounds on the value of k; it is unrestricted (and in particular, can be as large as $\theta(n)$). A natural question to ask is where is the hardness threshold; at

what values of k does the planar k-means problem become NP-hard. For instance, for $k \in O(\log n)$, the algorithm by [47] runs in quasi-polynomial time, and thus is unlikely to be NP-hard. Our reduction shows hardness for a particular choice of ϵ. Is it hard for $k = n^{\epsilon}$ for every choice of $\epsilon \in (0, 1)$? It has been pointed out by Vattani ([88]) that this is indeed the case.

- Another possible parameter to examine is the ratio of the maximum distance between points to the minimum distance. In an instance generated in our reduction, this ratio is infinity, as there are points with distance 0. A small perturbation of the points will make this ratio finite, but still unbounded. (On the other hand, it will be polynomial in n.) If the ratio is known a priori to be, say, linear in n, does it make the problem easier?

  However, if we consider only different locations, then the ratio even in our reduction is bounded by a polynomial in n, as the grid itself is of size polynomial in n. It is not clear if linear ratio is possible preserving hardness.

Regarding approximability also, the picture concerning planar k-means is far from clear. The algorithm of [11] (a variant of Lloyd's algorithm), while providing approximation guarantees for general k-means, does not provide any better guarantees on planar instances. (Although the lower bound example constructed in [11] is for high dimensions, analogous planar instances can also be constructed.) However, its behaviour on planar instances is not fully understood. In particular, it is entirely possible that for any planar instance of k-means, the algorithm of [11] gives an $O(1)$-approximation with high probability. Even if this is not true, it may still hold for most planar instances. Settling this either way would be of some interest.

The most important open question is to determine whether there is a PTAS for the planar k-means problem. Note that for a very similar problem, the *planar k-median problem*, a PTAS is known to exist ([8]). This is despite the fact that unlike the 1-mean, the 1-median does not have a closed form solution.

7 Conclusion

We summarize the results stated in the previous chapters and conclude with a brief discussion on related open problems.

7.1 Summary of Results

The work presented in this thesis can be divided into three broad categories. In Chapters 2 and 3, we have considered the graph isomorphism and canonization problem on planar graphs. We have described a log-space algorithm for isomorphism and canonization of 3-connected planar graphs, which has been used to get a log-space algorithm for isomorphism and canonization of planar graphs.

In Chapters 4 and 5, we have considered path problems which include reachability, longest and shortest path computation, and counting paths. We have given a $\mathsf{UL} \cap \mathsf{coUL}$ algorithm for computing a longest path in planar DAGs, and have also presented new upper bounds on a promise version of counting paths. We have also described log-space algorithms for path problems in k-trees, which include log-space algorithms for reachability in directed k-trees and shortest and longest path computation in directed acyclic k-trees. and the k-means clustering problem in two dimensions.

Besides these results on space-complexity of graph problems summarized above, we have considered the complexity of the k-means problem. In Chapter 6, we prove that the k-means problem is NP-hard even in two dimensions, when the number of clusters is not fixed.

7.2 Discussion

We mention some related open problems. In the isomorphism and canonization problem, a big challenge is to improve the upper or lower bound for general graphs. It is also interesting to get better upper bounds for bounded genus graphs and minor-closed families of graphs. As genus

k graphs do not have a unique embedding on a genus k surface for $k > 1$, our technique is not directly applicable. It would also be interesting to consider interval graphs and intersection graphs of various objects *e.g.* axis-parallel rectangles in plane.

In the context of path problems, recently Reach and Distance problems in $K_{3,3}$ minor-free and K_5 minor-free graphs and Long-Path in $K_{3,3}$ minor-free and K_5 minor-free DAGs have been shown to be in UL $\cap$ coUL [85]. For planar directed graphs, improving the upper bound on reachability from UL $\cap$ coUL to L is an open question. Some partial progress has been recently made by [81], where a log-space algorithm for reachability in planar directed acyclic graphs with at most $O(\log n)$ sources is described. Even for the class of layered grid graphs, no upper bound better than UL $\cap$ coUL is known. (See *e.g.* [3].) It would be interesting to get a log-space algorithm, or a weaker but orthogonal upper bound of LogDCFL for this class. For graphs of bounded tree-width, better upper bound will be possible if the upper bound on constructing the tree-decomposition can be improved from LogCFL to a smaller complexity class.

In the planar k-means problem, it is a challenging question to obtain a PTAS or to prove APX-hardness.

List of Publications/Reports

[DDN10] Bireswar Das, Samir Datta, and Prajakta Nimbhorkar. Log-space algorithms for paths and matchings in k-trees. In *proceedings of* 27^{th} *International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2010.

[DLN08] Samir Datta, Nutan Limaye, and Prajakta Nimbhorkar. 3-connected planar graph isomorphism is in log-space. In *Proceedings of the 28th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 153–162, 2008.

[DLN+09] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Planar graph isomorphism is in log-space. In *CCC '09: Proceedings of the 24th Annual IEEE Conference on Computational Complexity*, pages 203–214, 2009.

[LMN10] Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar. Longest paths in planar dags in unambiguous logspace. *Chicago Journal of Theoretical Computer Science, CATS 2009 special issue*, 2010(8), 2010.

[MNV09] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is NP-hard. *Theoretical Computer Science, WALCOM 2009 special issue (To appear). A preliminary version appeared in WALCOM '09: Proceedings of the 3rd International Workshop on Algorithms and Computation,* pages 274–285. Springer-Verlag LNCS, 2009.

Bibliography

[1] Manindra Agrawal and V. Arvind. A note on decision versus search for graph automorphism. *Information and Computation*, 131(2):179–189, 1996.

[2] Eric Allender. Making computation count: Arithmetic circuits in the nineties. In L. Hemaspaandra, editor, *SIGACT News Complexity Theory Column*, volume 28. ACM SIGACT, 1997.

[3] Eric Allender, David Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45:675–723, 2009.

[4] Eric Allender, Samir Datta, and Sambuddha Roy. The directed planar reachability problem. In *Proc. 25th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 3821 of *LNCS*, pages 238–249., 2005.

[5] Eric Allender and Meena Mahajan. The complexity of planarity testing. *Information and Computation*, 189(1):117–134, 2004.

[6] Eric Allender, Klaus Reinhardt, and Shiyu Zhou. Isolation, matching and counting uniform and nonuniform upper bounds. *Journal of Computer and System Sciences*, 59(2):164–181, 1999.

[7] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of Euclidean Sum-of-Squares Clustering. In *Machine Learning*, volume 75, pages 245–248, 2009.

[8] S. Arora. Polynomial time approximation schemes for Euclidean tsp and other geometric problems. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 2, 1996.

[9] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the 22nd anual Symposium on Computational Geometry (SoCG)*, pages 144–153, 2006.

[10] David Arthur and Sergei Vassilvitskii. Worst-case and smoothed analysis of the icp algorithm, with an application to the k-means method. In *Proceedings of the 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 153–164, 2006.

[11] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.

[12] Michael Artin. Algebra. *Prentice Hall, India, New Delhi*, 1996.

[13] V. Arvind, Bireswar Das, and Johannes Köbler. The Space Complexity of k-Tree Isomorphism. In *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 822–833. Springer, 2007.

[14] V. Arvind, Bireswar Das, and Johannes Köbler. A logspace algorithm for partial 2-tree canonization. In *Computer Science Symposium in Russia (CSR)*, pages 40–51, 2008.

[15] V. Arvind and Nikhil Devanur. Symmetry breaking in trees and planar graphs by vertex coloring. In *The Nordic Combinatorial Conference (NORCOM)*, 2004.

[16] V. Arvind and Piyush P. Kurur. Graph isomorphism is in SPP. *Information and Computation*, 204(5):835–852, 2006.

[17] V. Arvind, Piyush P. Kurur, and T. C. Vijayaraghavan. Bounded color multiplicity graph isomorphism is in the #l hierarchy. In *Proceedings of the 20th Annual IEEE Conference on Computational Complexity (CCC)*, pages 13–27, 2005.

[18] László Babai. Moderately exponential bound for graph isomorphism. In *FCT '81: Proceedings of the 1981 International FCT-Conference on Fundamentals of Computation Theory*, pages 34–50. Springer-Verlag, 1981.

[19] László Babai. Automorphism groups, isomorphism, reconstruction. *Handbook of combinatorics*, 2:1447–1540, 1995.

[20] László Babai, Paul Erdös, and Stanley M. Selkow. Random graph isomorphism. *SIAM J. Comput.*, 9(3):628–635, 1980.

[21] Laszlo Babai and Ludik Kucera. Canonical labelling of graphs in linear average time. In *SFCS '79: Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 39–46, 1979.

[22] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM Symposium on Theory of Computing (STOC)*, pages 171–183, 1983.

[23] Michael Ben-or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, 1992.

[24] Ravi B. Boppana, Johan Hastad, and Stathis Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[25] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *ACM Transactions on Computation Theory*, 1(1):1–17, 2009.

[26] Burchard von Braunmühl and Rutger Verbeek. Input driven languages are recognized in log n space. In *Selected papers of the international conference on "foundations of computation theory" on Topics in the theory of computation*, pages 1–19, 1985.

[27] Gerhard Buntrock, Birgit Jenner, Klaus-Jörn Lange, and Peter Rossmanith. Unambiguity and fewness for logarithmic space. In *Proc. 8th International Symposium on Fundamentals of Computation Theory (FCT)*, volume 529 of *LNCS*, pages 168–179, 1991.

[28] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992.

[29] Samuel R. Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *Proceedings of the 5th Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 18–33, 1997.

[30] A. Chiu, G. Davida, and B. Litow. Division in logspace-uniform NC^1. *Theoretical Informatics and Applications*, 35:259–275, 2001.

[31] Bireswar Das, Jacobo Torán, and Fabian Wagner. Restricted space algorithms for isomorphism on bounded treewidth graphs. In *Proceedings of 27th International Symposium on Theoretical Aspects of Computer Science (To appear)*, 2010.

[32] Sanjoy Dasgupta. The hardness of k-means clustering. Technical Report CS2007-0890, University of California, San Diego, 2007.

[33] Samir Datta, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Graph isomorphism for $K_{3,3}$-free and K_5-free graphs is in log-space. In *Proc. of 29th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 145–156, 2009.

[34] W. Fernandez de la Vega, Marek Karpinski, Claire Kenyon, and Yuval Rabani. Approximation schemes for clustering problems. In *Proceedings of the 35th annual ACM Symposium on Theory of Computing (STOC)*, pages 50–58, 2003.

[35] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. In *Machine Learning*, volume 56, pages 9–33, 2004.

[36] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, 1997.

[37] Matt Gibson, Gaurav Kanade, Erik Krohn, Imran A. Pirwani, and Kasturi Varadarajan. On clustering to minimize the sum of radii. In *Proceedings of the 19th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–825, 2008.

[38] John G. Del Greco, N. Chandrasekharan, and R. Sridhar. Fast parallel reordering and isomorphism testing of k-trees. *Algorithmica*, 32(1):61–72, 2002.

[39] Martin Grohe and Oleg Verbitsky. Testing graph isomorphism in parallel by playing a game. In *33rd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4051 of *LNCS*, pages 3–14, 2006.

[40] Arvind Gupta, Naomi Nishimura, Andrzej Proskurowski, and Prabhakar Ragde. Embeddings of k -connected graphs of pathwidth k. *Discrete Applied Mathematics*, 145(2):242–265, 2005.

[41] Sariel Har-Peled and Bardia Sadri. How fast is the k-means method? In *Proceedings of the 16th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 877–885, 2005.

[42] Frank Harary and E. M. Palmer. On acyclic simplicial complexes. *Mathematica*, 15:115–122, 1968.

[43] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002.

[44] John E. Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

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

[46] John E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th annual ACM Symposium on Theory of Computing (STOC)*, pages 172–184, 1974.

[47] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted Voronoi diagrams and randomization to variance-based clustering. In *Proceedings of the 10th annual Symposium on Computational Geometry (SCG)*, pages 332–339, 1994.

[48] Andreas Jakoby and Till Tantau. Computing shortest paths in series-parallel graphs in logarithmic space. In *Complexity of Boolean Functions*, number 06111 in Dagstuhl Seminar Proceedings, 2006. http://drops.dagstuhl.de/opus/volltexte/2006/618.

[49] Andreas Jakoby and Till Tantau. Logspace algorithms for computing shortest and longest paths in series-parallel graphs. In *Proc. 27th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4855 of *LNCS*, pages 216–227, 2007. see also [48].

[50] Birgit Jenner, Johannes Köbler, Pierre McKenzie, and Jacobo Torán. Completeness results for graph isomorphism. *Journal of Computing and System Sciences*, 66(3):549–566, 2003.

[51] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the 18th annual Symposium on Computational Geometry (SCG)*, volume 28, pages 89–112, 2004.

[52] M. M. Klawe, D. G. Corneil, and A. Proskurowski. Isomorphism testing in hookup classes. *SIAM Journal on Algebraic and Discrete Methods*, 3(2):260–274, 1982.

[53] Johannes Köbler and Sebastian Kuhnert. The isomorphism problem for $\boldsymbol{k}$-trees is complete for logspace. In *Proceedings of Mathematical Foundations of Computer Science (MFCS)*, volume 5734 of *LNCS*, pages 537–548, 2009.

[54] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: its structural complexity*. Birkhäuser, 1993.

[55] Michal Koucký. Universal traversal sequences with backtracking. *Journal of Computing and System Sciences*, 65(4):717–726, 2002.

[56] Jacek P. Kukluk, Lawrence B. Holder, and Diane J. Cook. Algorithm and experiments in testing planar graphs for isomorphism. *Journal of Graph Algorithms and Applications*, 8(2):313–356, 2004.

[57] Amit Kumar, Yogish Sabharwal, and Sandeep Sen. A simple linear time $(\mathbf{1} + \boldsymbol{\epsilon})$ approximation algorithm for $\boldsymbol{k}$-means clustering in any dimensions. In *Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 454–462, 2004.

[58] Klaus-Jörn Lange. Complexity and structure in formal language theory. *Fundamenta Informaticae*, 25(3-4):327–352, 1996.

[59] Charles E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proceedings of the 21st annual IEEE Symposium on Foundations of Computer Science (SFCS)*, pages 270–281, 1980.

[60] David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11:329–343, 1982.

[61] Nutan Limaye, Meena Mahajan, and B. V. Raghavendra Rao. Arithmetizing classes around NC^1 and L. *Theory of Computing Systems, special issue for STACS 2007*, 46(3):499–522, 2010.

[62] Steven Lindell. A logspace algorithm for tree canonization (extended abstract). In *Proceedings of the 24th annual ACM Symposium on Theory of Computing (STOC)*, pages 400–404, 1992.

[63] Stuart P. Lloyd. Least squares quantization in pcm. In *IEEE Transactions on Information Theory*, volume 28, pages 129–136, 1982.

[64] László Lovász. An algorithmic theory of numbers, graphs, and convexity. *CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM*, 1986.

[65] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[66] Eugene M. Luks. Parallel algorithms for permutation groups and graph isomorphism. In *SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 292–302, 1986.

[67] Saunders Maclane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3:460–472, 1937.

[68] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8:131–132, 1979.

[69] Pierre McKenzie, Birgit Jenner, and Jacobo Torán. A note on the hardness of tree isomorphism. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity (CCC)*, page 101, 1998.

[70] Nimrod Megiddo and Kenneth J. Supowit. On the complexity of some common geometric location problems. *SIAM Journal on Computing*, 13:182–196, 1984.

[71] Gary Miller. Isomorphism testing for graphs of bounded genus. In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 225–235, 1980.

[72] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[73] Noam Nisan and Amnon Ta-Shma. Symmetric logspace is closed under complement. *Chicago Journal of Theoretical Computer Science*, 1995.

[74] Rafail Ostrovsky, Yuval Rabani, Leonard Schulman, and Chaitanya Swamy. The effectiveness of Lloyd-type methods for the k-means problem. In *Proceedings of the 47th annual IEEE Symposium on Foundations of Computer Science*, pages 165–176, 2006.

[75] Ilia N. Ponomarenko. The isomorphism problem for classes of graphs closed under contraction. *Journal of Mathematical Sciences (JSM, formerly Journal of Soviet Mathematics)*, 55, 1991.

[76] Vijaya Ramachandran and John H. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49:517–561, 1994.

[77] Omer Reingold. Undirected connectivity in log-space. *Journal of ACM*, 55(4), 2008.

[78] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal of Computing*, 29(4):1118–1131, 2000.

[79] Neil Robertson and P. D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[80] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal on Computing and System Sciences*, 37(3):312–323, 1988.

[81] Derrick Stolee, Chris Bourke, and N. V. Vinodchandran. A log-space algorithm for reachability in planar DAGs with few sources. In *Proceedings of the 25th Conference on Computational Complexity*, 2010 (To appear).

[82] I. Sudborough. On the tape complexity of deterministic context-free language. *Journal of Association of Computing Machinery*, 25(3):405–414, 1978.

[83] Till Tantau. Logspace optimization problems and their approximability properties. *Theory of Computing Systems*, 41(2):327–350, 2007.

[84] Thomas Thierauf and Fabian Wagner. The isomorphism problem for planar 3-connected graphs is in unambiguous logspace. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 08001 of *Dagstuhl Seminar Proceedings*, pages 633–644, 2008.

[85] Thomas Thierauf and Fabian Wagner. Reachability in $K_{3,3}$-free graphs and K_5-free graphs is in unambiguous log-space. In *Fundamentals of Computation Theory (FCT)*, volume 5699 of *LNCS*, pages 323–334, 2009.

[86] Jacobo Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing*, 33(5):1093–1108, 2004.

[87] Leslie G. Valiant. Universality considerations in vlsi circuits. In *IEEE Transactions on Computers*, volume 30, pages 135–140, 1981.

[88] Andrea Vattani. The hardness of k-means clustering in the plane. manuscript, 2009.

[89] Oleg Verbitsky. Planar graphs: Logical complexity and parallel isomorphism tests. In *24th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 682–693, 2007.

[90] Fabian Wagner. Hardness results for tournament isomorphism and automorphism. In *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 572–583, 2007.

[91] Egon Wanke. Bounded tree-width and LOGCFL. *Journal of Algorithms*, 16(3):470–491, 1994.

[92] Louis Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circuit Theory*, 13:142–148, 1966.

[93] Hassler Whitney. A set of topological invariants for graphs. *American Journal of Mathematics*, 55:235–321, 1933.