Some problems on planar and minor free graphs

By

Archit Chauhan

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

to

Chennai Mathematical Institute

September 2024

Plot-H1, SIPCOT IT Park,

Siruseri, Kelambakkam,

Tamilnadu - 603103

India

CHENNAI
MATHEMATICAL
INSTITUTE

Archit Chauhan

Plot-H1, SIPCOT IT Park
Padur Post, Siruseri
Tamil Nadu, India-603103
E-mail:archit@cmi.ac.in

DECLARATION

I declare that the thesis entitled **"Some problems on planar and minor free graphs"** submitted by me for the degree of **Doctor of Philosophy in Computer Science** is the record of academic work carried out by me under the guidance of Professor Samir Datta and this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this University or any other University or Institution of Higher Learning.

Archit Chauhan
Chennai Mathematical Institute
September 2024.

CHENNAI
MATHEMATICAL
INSTITUTE

Samir Datta

Plot-H1, SIPCOT IT Park
Padur Post, Siruseri
Tamil Nadu, India-603103
E-mail:samir@cmi.ac.in

CERTIFICATE

I certify that the thesis entitled **"Some problems on planar and minor free graphs"** submitted for the degree of **Doctor of Philosophy in Computer Science** by "Archit Chauhan" is the record of research work carried out by him under my guidance and supervision, and that this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this University or any other University or Institution of Higher Learning. I further certify that the new results presented in this thesis represent his independent work in a very substantial measure.

Chennai Mathematical
Institute
Date: September 2024.

Samir Datta
Supervisor.

ACKNOWLEDGEMENT

This thesis would not have been possible without the support of many people involved, not all of whom I will be able to list here. First and foremost, I would like to thank my advisor Prof. Samir Datta for all the support over the years, and for keeping the energy high always. Though discussions would sometimes be quite intense, they helped me learn the value of grit and perseverence, and have helped me become a better researcher.

I had the chance of some wonderful visits during the thesis. I would like to thank Prof. Rohit Gurjar for hosting me at IIT Bombay. I had some really insightful discussions that motivated future directions of research. I would like to thank Prof. Yuichi Yoshida for a wonderful visit to NII Tokyo, and for teaching me a lot of things in a very short time.

I was also fortunate to have collaborated with some amazing people during my PhD. I would like to extend my gratitude to Prof. Eric Allender, from whom I have learnt a lot. He has always been extremely encouraging, kind and supportive. I would like to thank Anish, Vimal, Chetan, Prof. M. Praveen for all the work we did together.

I am deeply indebted to many wonderful teachers that I have had the chance to learn from during the course of my life so far. I would like to thank the Professors at CMI, especially Prof. Partha Mukhopadhyay, Prof. Prajakta Nimbhorkar, Prof. M. Praveen, Prof. Nithin Varma, Prof. Aiswarya Cyriac, Prof. G. Philip, Prof. Sriram Nambiar for all the wonderful lectures and discussions, both in and outside of academics. I would like to thank Prof. Rajat Tandon for his beautiful courses and discussions at IIIT Hyderabad, University of Hyderabad, that helped me make the decision to go for higher studies in academics. Finally I would like to thank my teachers from pre-college, particulary Arshad Majeed sir and S. P. Neelam sir, for introducing me to the beauty of mathematics and physics, and sowing the seeds of rigorous thinking.

The years at CMI went by fairly quickly, in part because of all the amazing friends I had there. I would like to thank Somnath, Alakh, Avinandan, Sarbojit, Sayantan for the unforgettable times as MSc batchmates and more. The trips organized by Alakh and Somnath to hunt for the finest filter coffee is something I'm sure we all miss even today. I will certainly miss Keerthan, Sonakshi, Sahil, Asif, Vishwa, Soumodev, Shakil for being wonderful flatmates and even better friends. Thanks in particular to Sonakshi and Keerthan for the cooking lessons during Covid lockdown. The 'chai pe charcha' (a euphemism for gossip about advisors etc.) with Vishwa and Asif made the PhD years an easier walk than it would have been otherwise. Shout-out to my gamer friends, in particular Shankar, Nanoty, Sricharan, Ron, Rajat, for the fun Dota sessions. I am also thankful to Abhranil, Adwitee, Alka, Amruta, Anirban, Arunava, Bhargava, Charles, Devendra, Govind, Joe, Nikhil, Pratibha, Preeti, Priya, Rajit, Roohani, Sajida, Sanjukta, Sarjick, and to my friends outside academia, especially Rahul and Vaibhav.

I consider myself fortunate to have chosen CMI for my graduate studies. It is an amazing institute and I hope it's legacy continues to grow and inspire. I am immensely thankful to all the staff and administration at CMI, who have always been very prompt, friendly, and make the atmosphere of the institute so warm and welcoming.

I cannot stress on how helpful the TCS Research Scholar Program of Tata Consultancy Services has been for my work. A lot of remote collaborations (especially during Covid) went smoothly thanks to the equipment I was able to purchase from their funds. Same holds for the visits I did for collaborations/paper presentations. I am deeply grateful for the support provided by the program.

Finally, this work has been made possible because of my family. I consider myself lucky to have parents who have always show unending patience, and given unconditional support. My siblings, who have always been there for me, no matter the circumstances. My nieces and nephews who have added much joy to my life. And my grandparents to whom I dedicate this thesis. My Late Dadima, who has always showered me with her love, and my Dadaji, who is the single most inspiring figure I know, and look up to.

Contents

Chapter 1

Introduction

A fundamental notion in theoretical computer science is that of *efficient computation*. Efficiently computable problems are those for which there exists an algorithm that can solve the problem in a short amount of time. More formally, the class P consists of problems that can be solved by a *Turing Machine* that takes time that is at most polynomial in the size of the input. There are many fundamental problems for which we know polynomial time algorithms to *verify the validity of a proposed solution*, but we don't know any polynomial time algorithm to solve the problem. This is captured by the famous 'P vs NP' question which, more than five decades after being posed [Coo71, Lev73], has still not been resolved. Attemps to attack the P vs NP problem however, have spawned a web of related questions, many of which are almost as fundamental, and despite being considered 'possibly easier' than P vs NP, have still not been resolved.

Among these are questions like 'P vs NC', which asks if problems that have a polynomial time algorithm on a *Turing Machine*, which is a sequential model of computation, can be solved significantly faster in a parallel model of computation. A classical model used for parallel computation is the Parallel Random Access Machine (or PRAM), which can be thought of as an abstract machine consisting of multiple processors, all of which have access to a shared memory. The class NC consists of problems that can be solved in polylogarithmic time by a PRAM with polynomially many processors.

Another fundamental question that has withstood attempts towards resolution is that of 'P vs BPP', which asks if a problem that can be solved by a randomized polynomial time algorithm can also be solved by a deterministic polynomial time algorithm. In other words, it asks if a problem that has a polynomial time algorithm using random bits, can be 'derandomized' without losing much efficiency in running time. The analogous question in the parallel setting is that of the equality of classes NC and RNC, where RNC is the class of problems that have a randomized NC algorithm. This question too is still open.

A step towards making the picture regarding these questions a bit clearer would be to look at various fundamental problems in theory of computation and try to pin down their exact complexity. A lot of these fundamental problems are graph theoretic in nature (12 of Karp's 21 NP-complete problems [Kar72] for example). Graphs indeed seem to be ubiquitous in computer science, both theoretically as well as in practice.

The first theme of this thesis is to look at the parallel complexity of the problem of Depth First Search in graphs or DFS. Depth First Search in graphs is an archetypal graph algorithm whose origin is in the 19-th century work of Trémaux [Die16][Chapter 1]. The DFS algorithm has played a pivotal role in the study of several problems such as biconnectivity [HT73b], triconnectivity [HT73a], topological sorting [Tar76],[Chapter 20 [CLRS22]], strong connectivity [Tar72, Sha81, Dij76], planarity testing and embedding [HT74] etc. Along with Breadth First Search (BFS), it is perhaps one of the best known algorithms to traverse graphs. However, while a parallel algorithm (i.e. an NC algorithm) has been known for BFS for some time [GM88][1], no such (deterministic) algorithm has been found for DFS. It was believed that DFS might be inherently sequential and Reif [Rei85] gave evidence supporting that by showing that lexicographically-first or lex-first DFS (i.e. a DFS traversal where the order of exploring the neighbours of the current node is predetermined in input) is P-complete in both directed as well as undirected graphs. Since it is widely believed that P $\neq$ NC, this result suggested that an NC algorithm for lex-first DFS is unlikely to exist. However the question of finding *a* DFS tree of a graph in NC, not necessarily the lex-first one, remained open. Surprisingly, it was shown by Aggarwal and Anderson [AA88] in 1987 that there does exist a *randomized* NC algorithm for DFS in undirected graphs. Subsequently, Aggarwal, Anderson and Kao [AAK90] gave an RNC algorithm for DFS in directed graphs also. Both the RNC algorithms mentioned above give a deterministic NC reduction to the problem of finding a minimum weight perfect matching in a bipartite graph. A recent breakthrough result by Fenner, Gurjar and Thierauf [FGT19] showed that such a perfect matching can be constructed in quasi-NC, thereby putting DFS in quasi-NC, for both directed as well as undirected graphs (the class quasi-NC consists of problems that can be solved in polylogarithmic time by a PRAM with *quasi*-polynomially many processors). However, a deterministic NC algorithm for DFS, either via a deterministic algorithm for perfect matching, or directly, has eluded us so far.

In restricted graph classes like planar graphs on the other hand, deterministic NC algorithms have long been known both in undirected as well as directed setting [Smi86, HY88, Hag90, Kao88]. Moreover, compared to the randomized NC algorithms for DFS in general graphs, the algorithms in planar graphs are much simpler and do not make use of a routine for perfect matching. Khuller [Khu90] also extended an NC algorithm for DFS in planar graphs to an NC algorithm for DFS in $K_{3,3}$-minor-free graphs. We make further progress in this direction by improving the parallel complexity of DFS in planar digraphs, bounded treewidth digraphs. We also extend the class of graphs for which NC algorithms for DFS are known to bounded genus digraphs and single-crossing-minor-free graphs[2].

The other theme of this thesis is to explore some problems related to parity of paths in directed graphs, in the sequential setting. Suppose we are given a graph G, a source vertex s, and a destination vertex t in it. The problem of finding a simple path from s to t is a fundamental problem known as the Reachability problem. A natural and well studied variant of it the problem of finding a path of *even length* from s to t. Lapaugh and Papadimitrou [LP84] showed that the undirected version of the problem can be solved in linear time, whereas as the directed version, which we refer to as *EvenPath*, is NP-complete.[3].

A related problem, that of finding a simple directed cycle of even length, called *EvenCycle* (which poly-time reduces to EvenPath), has also received significant attention. Polynomial-time algorithms have been known since long for the undirected version ([LP84, YZ97]), while the question of tractablility of the directed version was open for over two decades before

[1]Strictly speaking, we are not aware of NC algorithms for BFS. What we mean are algorithms that output a tree preserving the most important and widely used property of BFS, that every path from root to a vertex in the tree is a shortest path between them.

[2]As we will explain later, the term single-crossing-minor-free is used to denote a family of graph classes that include in particular, the class of $K_{3,3}$-minor-free graphs, and of K_5-minor-free graphs.

[3]The problem of finding a path of *odd* length from s to t easily reduces to EvenPath by adding a dummy out neighbour t' of t and querying for an even path from s to t'.

polynomial-time algorithms were given by McCuiag, and by Robertson, Seymour and Thomas [MRST97]. More recently, Björklund, Husfeldt and Kaski [BHK22] gave a randomized polynomial time algorithm for finding a *shortest* even directed cycle in directed graphs.

Since EvenPath is NP-hard, it is natural to investigate graph classes for which EvenPath would be tractable. Gallucio and Loebl [GL94] gave a polynomial-time algorithm for finding an even cycle in planar directed graphs, in which they develop a routine for a restricted variant of EvenPath (when s, t lie on a common face, and there are no even directed cycles left on removal of that face). Following that, Nedev [Ned99] in 1999 showed that EvenPath is in P in planar digraphs. We are not aware of any work following that of Nedev's, for EvenPath, until the recent work in [Sha21] which give a polynomial time algorithm for EvenPath in digraphs with at most one crossing (i.e. single-crossing graphs). We extend the class of graphs for which EvenPath is tractable, by giving a polynomial-time algorithm for EvenPath in single-crossing-minor-free digraphs. We also give solutions to two other related problems that occur along the way, that might be of independent interest. We explain our results in more detail in the rest of this chapter.

1.1 Our results on parallel algorithms for DFS

We first discuss our results related to DFS. For planar graphs, deterministic parallel algorithms to construct a DFS tree have been known, in both undirected as well as directed setting. For undirected planar graphs, Smith [Smi86] gave a deterministic parallel algorithm running in $O(\log^3 n)$ time. This was improved further in subsequent works [HY88] and Hagerup [Hag90] gave a parallel $O(\log n)$ time algorithm. In fact, it is observed in [ACDM19] that using Reingold's result on connectivity [Rei08], the algorithm of Hagerup is a Logspace reduction to computing distances in undirected planar graphs, thus putting it in the class UL ∩ co-UL ([TW10]). For directed planar graphs, Kao [Kao88] first gave a deterministic NC algorithm for DFS running in time $O(\log^7 n)$ on EREW PRAM. Subsequent papers optimized on runtime as well as number of processors [KK93, Kao95], leading to an algorithm for DFS in planar digraphs running in $O(\log^{10} n)$ time on a CRCW PRAM with $O(n)$ processors [KK93], and an algorithm for DFS in strongly connected planar digraphs running in $O(\log^5 n)$ time on a CRCW PRAM with $O(n/\log n)$ processors [Kao95].

1.1.1 Depth First search in directed planar graphs

We first aimed at generalizing the machinery developed by Hagerup [Hag90] for undirected planar graphs to directed planar graphs. The hope was to reduce the problem of constructing a DFS tree in directed planar graphs to that of computing distances in directed planar graphs, using a Logspace reduction. This would lead to a UL ∩ co-UL algorithm using [TW10]. Though we were not able to get a UL ∩ co-UL algorithm, we were able to come up with a graph decomposition that is a generalization the decomposition of Hagerup and get an AC^1(UL ∩ co-UL) algorithm, improving the previous bound of AC^{10} by Kao and Klein [KK93].

The decomposition of [Hag90] proceeds by embedding the graph in the plane identifying the set of 'outermost' cycles, that is those not contained inside any other cycles of the graph. These along with things in their exterior form the first layer of the graph. Then they peel this layer of and proceed similarly. This gives a layering of the graph, where the layers are attached to each other in a 'tree like manner'. They show that this layering can be computed efficiently in parallel. Moreover each layer is a very simple graph: a 'tree of cycles', and a particular kind of DFS traversal of each layer can be computed efficiently. Since the layers are attached in a tree like manner, they show that the DFS trees of each layer that were computed, can be patched up consistently in parallel.

For directed graphs we extend this idea by doing a two step nested sublayering. The first layer consists of the the set or 'outermost' *clockwise* cycles of the graph and things external to it. Then we peel this layer off and proceed similarly, again giving a 'tree like layering' of the graph. But since these layers can have complex nesting of counter clockwise cycles, for *each layer* obtained in the first step, we further divide it into sublayers as done in the first step, but by using *counter-clockwise* cycles. The final sublayers thus obtained again are attached to each other in a 'tree like manner'. Instead of a 'tree of cycles' in the layering for planar graphs, each sublayer in our layering for planar digraphs is a 'DAG (directed acyclic graph) of meshes'.

Unfortunately, it turns out that the meshes in the sublayers may not always have DFS traversals that can be patched up together consistently. Thus with a slight compromise from our initial goal, we proceed via using this decomposition to construct a balanced path separator of a graph.

The lemma we prove can be stated as:

Lemma 1.1 ([ACD21, ACD22]). *Given a planar digraph G and a vertex r in it, we can construct an $(\frac{11}{12}, r)$-path separator in* $\mathsf{UL} \cap \mathsf{co\text{-}UL}$.

However, we observed later that by plugging in some results on reachability in planar digraphs by Bourke, Tewari and Vinodchandran [BTV09] into the algorithm of Kao and Klein [KK93], also yields a $(\frac{2}{3}, r)$-path separator for planar digraphs in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$. In that sense, our decomposition is not crucial for the $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ bound for a balanced path separator, but gives an alternate algorithm. We nevertheless believe that our machinery might possibly have other applications in problems on planar digraphs. Using Lemma 1.1, we use a divide and conquer algorithm to construct a DFS tree as stated in the following theorem

Theorem 1.2 ([ACD21, ACD22]). *The problem of DFS in planar digraphs is in* AC^1*(*$\mathsf{UL} \cap \mathsf{co\text{-}UL}$*).*

Our method for constructing DFS trees using balanced path separators differs from the method of Kao and Klein [KK93]. They proceed via construction of partial DFS trees (developed in [AAK90]), whereas we sew the DFS trees of the smaller stongly connected components in a manner consistent with the DFS traversal of the DAG they are a part of.

1.1.2 Depth First Search in bounded treewidth graphs

We further explore this thread by investigating deterministic parallel algorithms for other classes of graphs. Surprisingly, we did not find any explicit result on parallel DFS for bounded treewidth graphs in the literature. The machinery developed by Kao [Kao88] to merge a small number of paths that form a balanced separator of a graph into a single path separator however, immediately gives an $\mathsf{AC}^1(\mathsf{L})$ bound for DFS in bounded treewidth graphs, both directed as well as undirected. We improve this bound and show the following result:

Theorem 1.3. *DFS in bounded treewidth digraphs is in* L.

It is easy to see that given an undirected graph, if we make all the edges of it bidirected and find a DFS tree of the new directed graph, its underlying undiricted tree will be a valid DFS tree of the original undirected graph. This immediately gives the following corollary:

Corollary 1.4. *DFS in (undirected) bounded treewidth graphs is in* L.

Our result uses a logspace version of Courcelle's theorem [Cou90, EJT10], a result in model theory that allows adding linear order in a structure without significantly increasing its treewidth [CF12, BD15], and a simple charecterization of lex-first DFS trees in terms of lex-first paths from root to a vertex [dlTK95].

1.1.3 Depth First Search in bounded genus graphs

Since planar graphs have genus zero, a natural extension of planar graphs are graphs that are embeddable on a suface of higher genus. We give a parallel algorithm for DFS in graphs embeddable on surfaces whose genus is at most a fixed constant (such families of graphs are referred to as bounded genus graphs).

We first find a constant number of disjoint cycles so that if any large component remains on removing them, then it must be planar. For directed graphs, the cycles might not be directed cycles, but are such that they can be written as a union of at most two directed paths. We then find a balanced path separator of that remaining planar graph. This results in a balanced 'multipath' separator of the entire graph consisting of constantly many paths, and we use Kao's trick to combine them into a single balanced path separator and then proceed via the divide and conquer method of [ACD22]. Our result can be stated as the following theorem:

Theorem 1.5. *DFS is in* $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ *for directed graphs of bounded genus.*

We again get the following corollary from the explanation in the previous subsection.

Corollary 1.6. *DFS is in* $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ *for undirected graphs of bounded genus.*

1.1.4 Depth First Search in single-crossing-minor-free graphs

Robertson and Seymour showed [RS93] that single-crossing-minor-free graphs can be decomposed by 3-clique sums into smaller pieces such that each piece is either planar, or of bounded treewidth. It was shown in [EV21] that the decomposition can be found in NC. This decomposition theorem is a simpler version of the more complicated theorem by Roberstson and Seymour for general H-minor-free graphs, and a more general version of the decomposition result for $K_{3,3}$-minor free graphs, which states that $K_{3,3}$-minor free graphs can be decomposed by 2-clique sums into pieces (their triconnected components basically) that are either planar or exactly the graph K_5 [Hal43, Asa85, Vaz89]. Building on the results for DFS in planar graphs and bounded treewidth graphs, we show that:

Theorem 1.7. *DFS is in* NC *for undirected single-crossing-minor-free graphs.*

The basic idea is to find a path separator (cycle separator in fact) and use divide and conquer as described in [Smi86]. To find a path separator, the strategy at a high level is similar to that of [Khu90], to decompose the graph and find a separator in the 'central' piece. The algorithm first computes the clique-sum decomposition tree described by [RS93] in NC [EV21] and then finds a central piece in this decomposition, which is a piece such that subgraphs in the decomposition attached to it have small weights. There are two cases to analyze: either the central piece is of bounded treewidth, or it is planar. Hereon the technicalities differ from that of [Khu90], since for $K_{3,3}$-minor free graphs, the decomposition only involves separating pairs whereas for single-crossing-minor-free graphs, separating triplets (that is three vertices which if removed from the graph, result in more than one connected component) are involved as well, making things a little more complicated.

The case when it is of bounded treewidth is handled easily by further refining the tree decomposition. For the case when the central piece is planar, we show that in order to find a balanced separator of the input graph, it suffices to project the weights of the attached subgraphs on the vertices/edges/faces of this central piece and then find a balanced *cycle separator* of this piece. However there are some technical issues caused due to the presence of 'virtual edges' that are used to compute the decomposition. To resolve them, we construct certain 'gadgets' or 'mimicking networks', which are small graphs that have the same connectivity properties as a larger graph. We project the gadgets corresponding to the attached subgraphs on the vertices/edges/faces of the central piece and show that it preserves the planarity of the piece. Then we use the algorithm from [Mil86] to find a balanced cycle separator in the modified central piece, and show that it can be lifted to the original graph while maintaining the balanced separator property.

1.2 Our results on problems on parity of paths

We extend the class of graphs for which EvenPath is tractable, by giving a polynomial-time algorithm for EvenPath in single-crossing-minor-free digraphs. We show that in bounded-treewidth digraphs using Courcelle's theorem. Along with Nedev's algorithm for EvenPath in planar digraphs and Robertson- Seymour's decomposition for single-crossing-minor-free graphs into planar and bounded treewidth pieces, this gives a recipe for a polynomial time algorithm in single-crossing-minor-free digraphs. However there are technical issues (which we will explain in the thesis), that prevent a straightforward dynamic programming solution using these. To overcome these challenges, we develop two ingredients which might be of independent interest.

Parity Mimicking Networks The first is the construction of gadgets that we call *parity-mimicking networks*, which are graphs that preserve the parities of various paths between designated terminal vertices of the graph they mimic. Since the graph pieces in Robertson-Seymour's tree decomposition of single-crossing-minor-free graphs are joined to each other via a clique of at most three vertices, we construct parity mimicking networks for upto three terminal vertices. The idea of mimicking networks has been used in the past for other problems, like flow computation [HKNR98, HKNR98, CE13, KR13, KR14], and in perfect matching [EV21]. The ideas we use for constructing parity mimicking networks however, do not rely on any existing work. For technical reasons, we require our parity mimicking networks to be of bounded treewidth and planar, with all terminals lying on a common face. These requirements make it more challenging to construct them (or even to check their existence), than might seem at a first glance. To state our result more formally, we need some definitions.

We first define the parity configuration of a graph, which consists of subsets of $\{0, 1\}$ for each pair, triplet of terminals, depending on whether there exists 'direct' or 'via' paths of parity even, odd, or both (we use 0 for even parity and 1 for odd). We formalise this below.

Definition 1.8. Let L be a directed graph and $T(L) = \{v_1, v_2, v_3\}$ be the set of terminal vertices of L. Then, $\forall i, j, k \in \{1, 2, 3\}$, such that i, j, k are distinct, we define the sets $Dir_L(v_i, v_j)$, and $\mathit{Via}_L(v_i, v_k, v_j)$ as:

- $\mathit{Dir}_L(v_i, v_j) = \{p \mid$ there exists a path of parity p from v_i to v_j in $L - v_k$ $\}$
- $\mathit{Via}_L(v_i, v_k, v_j) = \{p \mid$ there exists a path of parity p from v_i to v_j via v_k *and* there does not exist a path of parity p from v_i to v_j in $L - v_k\}$

We say that the Dir_L, Via_L sets constitute the *parity configuration* of the graph L with respect to $T(L)$. We call the paths corresponding to elements in Dir_L, Via_L sets as *Direct paths* and *Via paths*, respectively.

We now define parity mimicking networks.

Definition 1.9. A graph L' is a parity mimicking network of a another graph L (and vice versa), if they share a common set of terminals, and have the same parity configuration, $\mathcal{P}$, with respect to the terminals. We also call them parity mimicking networks of parity configuration $\mathcal{P}$.

We now state our lemma regarding existence and efficient construction of parity mimicking networks satisfying the desired constraints:

Lemma 1.10. *[CDGS24] Suppose L is a graph with terminals $T(L) = \{v_1, v_2, v_3\}$, and suppose we know the parity configuration of L with respect to $\{v_1, v_2, v_3\}$. We can in polynomial-time, find a parity mimicking network L' of L, with respect to $\{v_1, v_2, v_3\}$ which consists of at most 18 vertices, and is also planar, with v_1, v_2, v_3 lying on a common face.*

It might be of interest to see if a simpler construction exists (perhaps a constructive argument to route paths, that has eluded us so far), that avoids the hefty case analysis we have to do, and also if they can be constructed for more than three terminals.

Disjoint paths with constraints on total parity The other ingredient we develop arises from a natural variant of another famous problem. Suppose we are given a graph G and vertices $s_1, t_1, s_2, t_2 \dots s_k, t_k$ (we call them terminals) in it. The problem of finding pairwise vertex disjoint paths, from each s_i to t_i is a well-studied problem called the disjoint paths problem. In undirected graphs, the problem is in P when k is fixed [RS95, RRSS91], but NP-complete otherwise [Lyn75]. For (directed) graphs, the problem is NP-complete even for $k = 2$ [FHW80]. In planar graphs, it is known to be in P for fixed k [Sch94, CMPP13, LMP$^+$20]. We consider this problem, with the additional constraint that the sum of lengths of the s_i-t_i paths must be of specified parity. We hereafter refer to the parity of the sum of lengths as total parity, and refer to the problem as DisjointPathsTotalParity. In the undirected setting, a stricter version of this problem has been studied, where each s_i-t_i path must have parity p_i that is specified in input. This problem was shown to be in P for fixed k, by Kawarabayashi et al. [KRW11]. However much less is known in the directed setting. We do not yet know if it is tractable in planar graphs, even for $k = 2$. We show that in some special cases, i.e., when there are four terminals, three of which lie on a common face of a planar graph, DisjointPathsTotalParity can be solved in polynomial time for $k = 3$. To state our result, we define the problem more formally, adapted to our setting:

Definition 1.11. Given a graph G and four distinct terminals v_1, v_2, v_3, and v_4 in $V(G)$, the DisjointPathsTotalParity problem is to find a set of three pairwise disjoint paths, from v_1 to v_2, v_2 to v_3, and from v_3 to v_4, such that the total parity is even, if such a set of paths exist, and output no otherwise.

The problem where total parity must be odd can be easily reduced to this by adding a dummy neighbour to v_4. The result we show is the following:

Lemma 1.12. *[CDGS24] Let G be a graph, and v_1, v_2, v_3, v_4 be four vertices of G. Both decision as well as search versions of DisjointPathsTotalParity for these vertices as defined above can be solved in polynomial time in the following cases:*

1. *If G has constant treewidth.*
2. *If G is planar and v_1, v_2, v_3 lie on a common face of G.*

We do this by showing that under the extra constraints, the machinery developed by [Ned99] can be further generalized and applied to find a solution in polynomial time. The question of tractability of DisjointPathsTotalParity in planar graphs, without any constraint of some terminals lying on a common face is open, and would be interesting to resolve. A polynomial-time algorithm for it (for fixed k), would yield a polynomial-time algorithm for EvenPath in graphs with upto k crossings, which is currently unknown.

Combining these two ingredients allows us to proceed with dynamic programming and show the following result:

Theorem 1.13. *[CDGS24] Given an H-minor-free graph G for any fixed single-crossing graph H, the EvenPath problem in G can be solved in polynomial time.*

Chapter 2

Preliminaries

2.1 Complexity classes

We give a brief introduction to parallel complexity classes. For more formal and detailed introduction, we refer the reader to the texts by Arora and Barak [AB09] and Vollmer [Vol99].

2.1.1 Complexity Classes based on Turing Machines

We assume familiarity with the notion of *Turing Machines* (deterministic and non-deterministic), and classes such as P and NP.

The randomized versions of deterministic classes like P are defined by giving the machine access to *random bits*.

The class L (for Logspace) consists of languages (or decision problems) which can be decided by Turing Machines that use only $O(\log n)$ cells of their work tape. The corresponding non-deterministic class is NL. A non-deterministic Turing machine is said to be *unambiguous* if on every input, there is at most one accepting computation path. The class of problems which can be decided by an unambiguous logspace-bounded non-deterministic Turing machines is referred to as UL. The class co-UL is its complement. Decision problems of languages can be seen as functions with a binary $0/1$ output. Though the above classes are defined for such binary valued functions, they can be extended to functions whose output can be any binary string. A function f being computable in P or L are naturally defined (We refer to a Turing machine that outputs a string as a *transducer*). To define the concept of a function being 'computable in NL', we note the following theorem from [JK89] where they show that the following three conditions are equivalent and each of them can serve as a definition:

1. f is computed by a L machine with an oracle from NL.
2. f is computed by a L-uniform NC^{1}circuit family with oracle gates for a language in NL.
3. $f(x)$ has length bounded by a polynomial in $|x|$, and the set $\{(x, i, b)$: the i^{th} bit of $f(x)$ is $b\}$ is in NL.

In particular thus, we can say that a function f is in NL if there is an NL machine that can compute the i^{th} bit of the output. The proof of the equivalence above relies on the fact that the class NL is closed under complement (i.e. NL=co-NL) [Sze88, Imm88]. Since the class UL is not yet known to be equal to its complement co-UL, it is less clear as to how to define the notion of "$f \in$ UL". However the the class UL $\cap$ co-UL is obviously closed under complement and the proof of [JK89] carries over to the class UL $\cap$ co-UL. That is, the following conditions are equivalent:

1. f is computed by a L machine with an oracle from UL $\cap$ co-UL.
2. f is computed by a L-uniform NC^1 circuit family with oracle gates for a language in UL $\cap$ co-UL.
3. $f(x)$ has length bounded by a polynomial in $|x|$, and the set $\{(x, i, b)$: the i^{th} bit of $f(x)$ is $b\}$ is in UL $\cap$ co-UL.

Thus if any of the above conditions hold, we can say that "f is computable in UL $\cap$ co-UL."

It is conjectured that for logspace Turing machines non-determinism can be made unambiguous, i.e., NL = UL [ARZ99, RA00]. The conjecture remains open though some progress has been made towards proving it. Allender and Reinhardt [RA00] showed that they are indeed equal in the non-uniform setting, i.e. NL/poly = UL/poly. The problem of reachability (which we call REACH), i.e. finding a path from a source vertex to a destination vertex in a directed graph is NL-complete [Jon75]. The problem of finding the length of a shortest path from a source vertex to a destination vertex in a directed graph (which we call DIST) is also NL-complete. For directed planar graphs, the following are two important results on functions in UL $\cap$ co-UL that we use in our work:

Theorem 2.1. *1.* REACH *in directed planar graphs is in* UL $\cap$ co-UL *[BTV09].*

2. DIST *in directed planar graphs is in* UL $\cap$ co-UL *[TW10].*

An important fact we will use is that the composition of two logspace-computable functions is also logspace-computable (see Chapter 4 [AB09]).

The proof also carries over to functions computable in L^C for any oracle C, thus in particular for functions computable in UL $\cap$ co-UL [TW10]. We make implicit use of this fact frequently when presenting our algorithms. For example, we may say that a colored labeling of a graph G is computable in UL $\cap$ co-UL, and that, given such a colored labeling, a decomposition of the graph into layers is also computable in logspace, and furthermore, that—given such a decomposition of G into layers—an additional coloring of the smaller graphs is computable in UL $\cap$ co-UL, etc. The reader need not worry that a logspace-bounded machine does not have adequate space to store these intermediate representations; the fact that the final result is also computable in UL $\cap$ co-UL follows from closure under composition. In effect, the bits of these intermediate representations are re-computed each time we need to refer to them.

2.1.2 The PRAM model

We briefly describe the model of PRAMs (see [Vol99, KR90] for a more detailed discussion). A PRAM has multiple synchronous processors, all having unit time access to a shared memory. Each processor also has a local memory. Processors can read from or write into a cell of their local memory or a cell of the shared memory (therefore they can communicate via the shared memory). There are various models of PRAMs characterized by rules for resolving conflicts regarding access of cells of the shared memory. The most common one we use is the Concurrent Read Concurrent Write, or the CRCW model, where multiple processors can simultaneously read from a shared memory cell, and simultaneously attempt to write to a shared memory cell. Within CRCW model, there is further classification of models based on how concurrent writes are resolved. For us, CRCW means the ARBITRARY model, where concurrent writes are resolved by enabling an arbitrary processor to succeed. There are also other models like COMMON, where the write succeeds only if all processors are attempting to write the same value, and PRIORITY, where the least indexed processor succeeds. These models have different powers, but for our purpose, that level of fine distinction is not required (see [KR90]). Apart from CRCW, there are other PRAM models like Concurrent Read Exclusive Write (or CREW), where multiple processors cannot simultaneously attempt to write at a shared memory cell, and Exclusive Read Exclusive Write (or EREW), where multiple processors can neither read simultaneously from a shared memory cell, nor attempt to write. As shown in [KR90], CREW can simulate any program run in CRCW with the same number of processors, with an overhead multiplicative factor of $O(\log n)$. We use CRCW(T(n)) to denote the class of problems which can be solved in $O(\text{T}(n))$ time by a CRCW PRAM with polynomial number of processors. If we assume the number of processors to be the same, the following hierarchy follows easily from definitions:

$$\text{EREW}(T(n)) \subseteq \text{CREW}(T(n)) \subseteq \text{CRCW}(T(n))$$

2.1.3 Boolean Circuits

Next we describe boolean circuits. We can think of a boolean circuit as a directed acyclic graph (DAG) where each edge represents a wire that carries a $0/1$ bit, each source node is labelled with a $0/1$ bit (an input bit), and all nodes other than the source nodes are labelled by boolean logic gates {AND, OR, NOT}. Some of the sink nodes, usually just one, are designated as output nodes. We sometimes refer to the nodes (except input nodes) as gates. The number of wires entering/exiting a gate is called the fan-in/fan-out of the gate respectively. If the input bits are specified, the circuit computes the output bits in the natural way.

The *size* of the circuit is defined as the number of nodes (gates) in the circuit. The *depth* of a circuit is defined as the length of the longest path from a source node (input) to an output node.

A *family* of boolean circuits is a sequence of boolean circuits denoted by $\{C_n\}_{n\in\mathbb{N}}$, where the circuit C_n has n input nodes. If $x \in \{0,1\}^n$ is some input C_n, we denote the output of C_n on x by $C_n(x)$. If $T : \mathbb{N} \to \mathbb{N}$ is a function, we say that the family $\{C_n\}_{n\in\mathbb{N}}$ has size $T(n)$ if for every n, the size of C_n is at most $T(n)$. We say the family $\{C_n\}_{n\in\mathbb{N}}$ has depth $T(n)$ if for every n, the depth of C_n is at most $T(n)$. We say that a language L is decided by a family of boolean circuits $\{C_n\}_{n\in\mathbb{N}}$ if for every $x \in \{0,1\}^n, x \in L \Leftrightarrow C_n(x) = 1$.

Note that there is no restriction on *how the circuits of a family are generated*. It can be shown easily that there exists a family of polynomial size circuits that can decide a unary encoding of the Halting problem. To avoid such consequences, it is natural to impose some *uniformity* conditions on the ciruit families, so that there is a single algorithm to generate the circuits. A *logspace-uniform family* of circuits is a sequence of boolean circuits $\{C_n\}_{n\in\mathbb{N}}$ such that there is a logspace Turing machine that can output the description of C_n on input 1^n. Note that the condition of logspace-uniformity enforces the size of the circuits to be at most polynomial in the size of the input. There are other notions of uniformity as well, for example using the *deterministic log-time* Turing machine model (see [Ruz81, MIS90] for definition and discussions on these models and notions of uniformity).

We define the classes AC, NC:

Definition 2.2. For every $k \in \mathbb{Z}^+$, a language L is in the class *uniform* AC^k if L can be decided by logspace-uniform family of circuits $\{C_n\}$ where C_n has poly(n) size and $O(\log^k \mathrm{n})$ depth. The class AC is defined as $\bigcup_{i\geq 1}\mathrm{AC}^i$.

The class NC is defined similarly but with the restriction that the gates of the circuits in the family can have fan-in at most 2.

We will slightly abuse notation and drop the term uniform when referring to uniform AC.

The class quasi-NC is defined similar to the class NC with the modification that the size of circuits is quasi-polynomial in the size of input, that is, for input $x \in \{0,1\}^n$, the size of the circuit is *quasi-polynomial*, that is $2^{\log^{O(1)} n}$.

As observed in the previous subsection, languages can be thought of as functions with a single $0/1$ output, and complexity classes can be extended to boolean functions with a binary string as output. Though the classes AC,NC are defined for languages (single output gate), they can be easily extended to include such boolean functions by allowing circuits with multiple output gates.

We will also consider circuits which along with boolean gates {AND, NOT, OR} also have 'oracle gates' for some boolean funcitons. An oracle gate for boolean function f with n input bits is a gate that on input $x \in \{0,1\}^n$, gives the output $f(x)$. Suppose $\mathcal{C}$ is a complexity class (a set of boolean functions). We use AC$^k(\mathcal{C})$ to denote functions that can be computed by a family of AC^k circuits augmented with oracle gates that can compute functions in $\mathcal{C}$. An equivalent characterization of AC^k is in terms of the class CRCW$[\log^k n]$ of the PRAM model described abovewith number of processors $n^{O(1)}$ [KR90]. The class *randomized* NC or RNC can be defined by giving PRAMs access to random bits.

For all the classes mentioned above, we will slightly abuse notation and use the usual notations like AC, NC, NL etc when it is clear from the context that we are referring to the functional version of the complexity class. In our work, we will mostly try to optimize the complexity of various problems with respect to the following hierarchy:

$$\mathrm{L} \subseteq \mathrm{UL} \cap \text{co-UL} \subseteq \mathrm{UL} \subseteq \mathrm{AC}^1 \subseteq \mathrm{AC}^1(\mathrm{UL} \cap \text{co-UL}) \subseteq \mathrm{AC}^2 \subseteq \mathrm{NC}^3 \subseteq \mathrm{AC}^3 \ldots$$

2.2 Graph theory prelims

We assume some familiarity with the notion of graphs. We generally denote the graph formally as a pair $G = (V, E)$ where V is the set of *vertices* of G (sometimes denoted by $V(G)$), and $E \subseteq V \times V$ is the set of *edges* (sometimes denoted by $E(G)$). Usually we assume that G has n vertices. The graph G may be undirected, in which case E is a symmetric set, of G may be directed, in which case E need not be a symmetric set. If G is directed, and $e = (u, v)$ is an edge in E, we call u the *tail* of e, and v as the *head* of e. We will sometimes refer to the *underlying undirected graph* of G, which is the graph obtained by ignoring the directions of edges of G. We emphasize that when talking about concepts like treewidth, minors of directed graphs, we intend them to apply to the underlying undirected graphs. We will mostly deal with simple graphs, that is, we assume there are no self loops (edges of the type (u, u)) or multi-edges between two vertices. *size* of a graph we mean the number of vertices of the graph. If a graph is weighted, that is the vertices of the graph have weights, then we use the term size to mean the sum of the weights of all the vertices of the graph.

A graph G' is a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. If $S \subseteq V(G)$ is a set of vertices, then we denote by $G[S]$, the *induced subgraph* of S. If v is a vertex in G, we use

$G - v$ to denote the graph obtained by deleting the vertex v and all edges adjacent to v. If S is a set of vertices of G, $G - S$ is defined similarly. If $e = (u, v)$ is an edge in G, we use $G - e$ or $G - (u, v)$ to denote the graph obtained by deleting the edge (u, v) from G (just deleting the edge, not the end points u, v). The degree of a vertex v is the number of edges incident on v. If (u, v) is an edge in a directed graph, we say that v is an *out-neighbour* of u and u is an *in-neighbour* of v. The *out-degree* (respectively *in-degree*) of a vertex v in a digraph is the number of out-neighbours (respectively the number of in-neighbours) of v. A graph G is said to be *k-degenerate* if every subgraph of G has at least one vertex of degree at most k. The *degeneracy* of a graph is the minimum k such that it is k-degenerate.

A walk in a graph (directed or undirected) is a sequence of vertices $v_{i_1}, v_{i_2}, \ldots$ such that for any two vertices $v_{i_l}, v_{i_{l+1}}$ adjacent in the sequence, $(v_{i_l}, v_{i_{l+1}}) \in E$. A *path* of G (sometimes referred to as a simple path) is a walk where no vertex is repeated. A *cycle* (also referred to as a simple cycle) is a walk where no vertex is repeated, except for the first and the last vertex in the sequence, which are the same. For directed graphs, the terms *path* and *directed paths* will usually be synonymous (the latter is sometimes used to emphasize context) and the terms *cycles* and *directed cycles* will also be synonymous, unless otherwise specified. We use the term *disjoint paths* to refer to paths that do not share any vertex. Paths that do not share any vertex other than the end points are called *internally disjoint paths*, though we sometimes drop the term 'internally' when it is clear from context that we mean internally disjoint paths.

If $A, B \subseteq V$, we say that $P = v_0 \ldots v_k$ is an $A - B$ path if $V(P) \cap A = v_0$ and $V(P) \cap B = v_k$. A set of vertices $S \subset V$ is called a *separating set* of G if removing the vertices of S from G results in at least two connected components. A graph G is called *k-vertex-connected*, or just *k-connected* since we will mostly deal with vertex connectivity, if there does not exist any separating set of G of size at most $k - 1$. It is generally assumed that a k-connected graph has at least $k + 1$ vertices (the empty graph is considered as disconnected.) A classical theorem on connectivity in graph theory is the following, due to Menger [Men27] (refer to Chapter 3 [Die16] for an exposition).

Theorem 2.3. *Let $G = (V, E)$ be a graph and $A, B \subseteq V$. Then the minimum number of vertices separating A from B in G is equal to the maximum number of internally disjoint $A - B$ paths in G.*

An important and widely studied problem is that of finding disjoint paths in a graph between designated vertices. We will use the following result by Khuller, Mitchell and Vazirani [KMV92] on finding two disjoint paths between designated vertices in an undirected graph, in parallel, .

Theorem 2.4. *[KMV92] Suppose we are given an undirected graph G and vertices s_1, t_1, s_2, t_2 in it. The problem of deciding if there exists an s_1-t_1 path and an s_2-t_2 path, both of which are disjoint, is in* CRCW$[\log n]$. *Moreover, if such paths exist, we can find such a pair in* CRCW$[\log n]$.

We also refer to 2-connected graphs as *biconnected graphs*. If a single vertex by itself forms a separating set of a graph, then we call it a *cut vertex*, or an *articulation point*. A *biconnected component* or *block* of a graph is a maximal biconnected subgraph. Any connected graph decomposes into a tree of biconnected components called the *block-cut tree* of the graph. A block containing at most one cut vertex is called a *leaf block*, as it corresponds to a leaf vertex in the block-cut tree. The block-cut tree of a graph is unique and can be computed in L using [Rei08].

Another useful consequence of Menger's theorem is the following lemma:

Lemma 2.5. *Let G be a 3-connected graph with at least 4 vertices. Then for any edge $e \in E(G)$, $G - e$ is biconnected.*

A digraph G is called *strongly connected* if for any two vertices u, v, there is a path from u to v and a path from v to u. A *strongly connected component* of a digraph, which we sometimes refer to as *SCC* is a maximal subgraph that is strongly connected. A digraph that does not contain any (directed) cycles is called as a *directed acyclic graph*, also referred to as a *DAG*. Any digraph can be decomposed into a DAG consisting of SCCs of the graph as nodes.

2.2.1 Spanning trees, Depth-First-Search trees

A tree is a graph (usually undirected) that is connected but has no cycles. Given an undirected graph G, a spanning tree of G is a subgraph of G that is a tree and includes all vertices of G. Given a spanning tree T of G, the edges of G in T are called the *tree edges* and the remaining edges are called *non-tree* edges. An important fact we will often use is the following:

Lemma 2.6. *Given a graph G, a spanning tree of G can be constructed in* L.

The proof follows from [Rei08] and [Rei84]. If a vertex r of a tree T is designated as *root* of the tree, then we naturally get a partial order on vertices of T. It is easy to see that for any vertex v, there is a unique path from r to v in T, and if a vertex u lies on that path, then u is called an *ancestor* of v, and v is called a *descendant* of u. The immediate ancestor of a vertex in a tree is called its *parent* (which is unique), and the immediate descendants are called the *children*.

Given a spanning tree T of G, consider a non-tree edge (u, v) in G. Let w be the *least common ancestor* (also called *LCA*)of u, v, that is, if any other vertex x is a common ancestor of u, v, then it is an ancestor of w. Then the paths from w to u, v respectively, along with the edge (u, v) form a cycle that we call the *fundamental cycle* of the edge (u, v) with respect to T.

For directed graphs, we use the concept of *arborescence*. Given a digraph G and a vertex r designated as the root, an arborescence of G with respect to r is a subgraph T of G such that the underlying undirected graph of T is a tree and every edge in T points away from the root. An arborescence with respect to a vertex might not always exist in a digraph, but it always exists if the digraph is strongly connected.

Given any node r in a directed graph G, a depth-first traversal of G starting at r is a traversal of all of the nodes reachable in G from r obtained by starting with r as the only node on a stack, and repeating the following steps until the stack is empty: (1) Pop a node v off the stack and ignore it if it has already been visited. (2) Otherwise mark it as *visited*, and push all unvisited out-neighbours of v onto the stack. Different depth-first traversals of G result if the out-neighbours of v are placed onto the stack in different orders. Each depth-first traversal gives rise to a *depth-first search tree* (namely, the directed tree rooted at r where the children of each node v are the out-neighbours x of v having the property that, when x is first marked *visited*, x has no in-neighbour other than v that has been marked as *visited*). We will assume here onwards that for the problem of DFS in digraphs, if we are given an input graph G and a designated root vertex r in it, then all the vertices of G are reachable from r. The definition is similar for undirected graphs.

Note that for undirected graphs, it is sufficient to output the edges that are part of a depth-first search tree. Any Euler tour of the tree (which can be performed in L) for example, would be a valid DFS traversal of the input graph. For directed graphs on the other hand, given a depth-first search tree T, it is possible to traverse T in an order that is not a depth-first traversal of G. Thus it is more correct to say that we are constructing a depth-first *traversal* of G, by augmenting the information regarding edges of the DFS tree with an ordering on the out-neighbours of each vertex in the tree that is followed in the DFS traversal we are outputting. But we follow the established convention by abusing notation and referring to depth-first search trees (DFS trees) and depth-first traversals interchangeably.

2.2.2 Planar graphs, dual, genus and embeddings

A planar graph is a graph that can be draw on a plane (or on a sphere) such that no two of its edges cross each other (For a detailed exposition of planar graphs, embeddings, with formal topological definitions, we refer the reader to [Chapter 4, [Die16]]). More formally, an *embedding* of a graph $G = (V, E)$ on a plane is a mapping that maps V to distinct points in $\mathbb{R}^2$ and E to internally disjoint simple curves in $\mathbb{R}^2$ such that the endpoints of the image of edge (u, v) are mapped to the images of vertices u, v. The *faces* of an embedded planar graph are the maximally connected regions of $\mathbb{R}^2$ that disjoint from the images of V, E. A graph embedded on $\mathbb{R}^2$ (often called as the *plane*) is called as a *plane graph*. In such an embedding, there is an unbounded face which is called as the *outer face* of G. (Note that in an embedding of a graph on the sphere, we can designate *any* face as the outer face).

Since embeddings of the above type are difficult to deal with on computers, we use the more relaxed notion of combinatorial embeddings of planar graphs. A combinatorial embedding or rotation scheme ϕ is a cyclic ordering of the edges around each vertex. such that there is a plane drawing of the graph respecting the cyclic order. More formally, for a directed graph G and an edge (u, v), $\phi(u, v) = (u, w)$ where (u, w) is the next edge incident to u, in a cyclic ordering (say anti-clockwise) of the edges around u in a plane drawing of G. A face of the embedded graph can be specified by its *boundary*, that is the edges we would encounter when we traverse the boundary of the face (clockwise or counter-clockwise). Since we will talk about connected graphs the boundary of a face will always be a single walk. A plane graph is often written as $G = (V, E, F)$, where F denotes the set of faces of G.

We will also use the concept of the *dual graph* of a planar graph. Suppose $G = (V, E, F)$ is a graph embedded in the plane. The dual graph $G^* = (F^*, E^*, V^*)$ is another plane graph. The vertex set F^* of G^* has a bijection with face set F of G. The face set V^* of G^* has a bijection with face set V of G (sometimes in literature $G^* = (V^*, E^*, F^*)$ is used for dual. We use $G^* = (F^*, E^*, V^*)$ so that we can denote the vertex of G^* that corresponds to face f_i of G, by $f_i{}^*$). The dual graph need not be a simple graph. It is easy to see that the give an embedded planar graph, the dual can be computed in logspace. An important fact that we will repeatedly use is that the problem of planarity testing is in logspace. This was shown by Allender and Mahajan [AM04] (they showed that it is in SL which was later shown to be equal to L by Reingold), and also by Datta, Prakriya [DP11].

Theorem 2.7. *Given an input graph G, we can check if it is planar or not in* L. *If it is planar, we can also construct an embedding of it in* L.

The idea of drawing a graph on a plane/sphere without edges crossing can be extended to more general *surfaces*. We give a very brief introduction here and refer the reader to [MT01] for more details. A *surface* is a two-dimensional manifold, which can be thought of as a shape that locally resembles the Euclidean plane. The genus of a surface is a measure of its complexity, defined as the maximum number of non-intersecting simple closed curves that can be drawn on the surface without separating it. For example, a sphere has a genus of 0, while a torus (doughnut-shaped surface) has a genus of 1. A torus can be seen as being formed from a sphere by adding a *handle* to it. An *orientable surface* is one that can be obtained by repeatedly adding handles to a sphere. Each handle introduces a 'hole' to the surface, increasing its genus by one. We deal only with orientable surfaces in this thesis. The genus of a graph G is the minimum integer g such that G can be embedded on a surface of genus g (without any edges crossing). By graphs of *bounded genus*, we mean a family of graphs of genus at most a constant g. Another important fact we will use is that given a graph and a fixed constant g, testing constructing an embedding of the graph on a surface of genus g is in Logspace. This generalizes the result of Theorem 2.7 stated above. We state the following theorem by Elberfeld, Kawarabayashi, (slightly modified by using the term genus instead of *Euler* genus):

Theorem 2.8. *[EK14] For every $g \in N$, there is a logspace DTM that, on input of a (connected) graph G of genus at most g, outputs an embedding $\prod$ of G with genus at most g.*

The statement of the theorem in [EK14] is for *Euler genus*, which is a more general notion than the one we have defined above, but that is not required for our purpose.

Simple closed curves (cycles) drawn on orientable surfaces can be classified as separating or non-separating. A separating cycle divides the surface into two disjoint regions. A non-separating cycle, on the other hand, does not divide the surface into separate regions. Hence if a graph is embedded on a surface, the cycles of a graph are either surface separating or surface-non separating. The notion of combinatorial embeddings described above for planar graphs can be extended to surface embeddings.

2.2.3 Balanced separators

We state the definition of *balanced separators*:

Definition 2.9. Consider an input graph $G = (V, E)$ of size n, where V, E denote the set of vertices and edges respectively. Let r be a vertex in V. If G is undirected, we use the term α-separator for some subgraph of G which if removed from G (i.e. its vertices are removed), results in connected components of size at most αn, where $0 < \alpha < 1$. A single path P, or a collection of disjoint paths $\{P_1, \ldots P_k\}$ for example, could form a balanced separator of G. We call refer to them as *path separators* and *multipath separators* respectively.

If G is directed, we have a similar definition, but the constraint is instead on the size of each *strongly* connected component left after removing the vertices of separator. An (α, r)-path separator is an α-path separator that starts at vertex r.

Note that if we have an algorithm to find an α-separator in a directed graph, then we can use that to find an α-separator in an undirected graph by making each edge into a bi-directed edge.

Also note that the definition translates to the case when vertices of G are weighted.

For brevity, we will occasionaly drop the fraction, or even the term *balanced*, and use the term path/cycle separator, when it is clear from context that we mean a balanced path/cycle separator.

Any simple cycle C in an embedded planar graph G will divide it into two regions, the *interior* and *exterior* of C, which we denote by $int(C)$ and $ext(C)$ respectively. Typically, if a face of G is marked as outer face then the region containing the outer face is referred to as the exterior and the other region ass the interior. Suppose the vertices, edges and faces of G are assigned weights. Then we use $w(int(C))$ to denote the sum of weights of vertices, edges, faces that are contained $int(C)$. The term $w(ext(C))$ is defined similarly. The vertices, edges that lie on C itself are usually not counted since it they are part of they separator and will be removed when recursing. Another variation of cycle separators for planar graphs we will use is one where both the weigth of the interior, as well as the weight of exterior of the cycle separator are small. We will define these in detail in Chapter 3.

For a planar graph that is embedded on the plane, we will use the notion of *laminar family* of cycles. A set of cycles of the graph form a laminar family if for any two cycles in the family:

- either they share at most one vertex in common and neither of them lies in the interior region of the other cycle,
- or they are vertex disjoint and one of them lies in the strict interior of the other.

More generally, we call a family of sets laminar if for any two subsets of the family, either one is a subset of the other, or they are both disjoint.

2.2.4 Minors, treewidth, clique sum decomposition

A graph H is called a *minor* of G if it can be formed from G by deleting edges, vertices and by contracting edges (in any sequence). If H is not a minor of G, we say that G is H-minor-free. If $\mathcal{F}$ is a family of graphs such that H is not a minor of any graph in it, then we say that $\mathcal{F}$ is H-minor-free, or that H is a forbidden minor of $\mathcal{F}$. It is a well known theorem of Wagner [Wag37] that the set of graphs for which both $K_{3,3}$ and K_5 are forbidden minors, is exactly the set of all planar graphs. A family $\mathcal{F}$ is called *minor closed* if for every graph in $\mathcal{F}$, all its minors are also in $\mathcal{F}$. Planar graphs are an example of a minor closed family. It is not a coincidence that they can be characterized by two graphs $K_{3,3}$ and K_5 as their forbidden minors. A deep result of Robertson and Seymour states that *any* minor closed family can be completely characterized by a finite set of forbidden minors [RS04].

We refer the reader to [Die16] for formal definitions of *tree decompositions* and treewidth. Informally, a tree decomposition T_G of a graph G is a tree where each node (sometimes called a bag) corresponds to a subset of vertices of G such that:

- Every vertex of G belongs to at least one bag of T_G.
- For every edge (u, v) in G, there exists a bag containing both u, v.
- For every vertex v of G, the bags that contain v form a connected subtree of T_G.

The width of a tree decomposition is the size of the largest bag minus one. The *treewidth* of a graph is the minimum width among all possible tree decompositions of G.

A particular type of tree decompositions we will use are *clique sum* decompositions, which are obtained when decomposing the graph along separating sets of some sizes. In a k-clique sum decomposition, we decompose the graph as long as there are separating sets of size k or smaller in the decomposed pieces. In this thesis, we will use a only encounter upto 3-clique sum decompositions, that is, we only decompose the graph along a separating vertex, a *separating pair* (pair of vertices), or a separating *triplet* (set of three vertices).

We reiterate here the definition of clique-sums:

Definition 2.10. A k-clique-sum of two graphs G_1, G_2 can be obtained from the disjoint union of G_1, G_2 by identifying a clique in G_1 of at most k vertices with a clique of the same number of vertices in G_2, and then possibly deleting some of the edges of the merged clique.

We can think of this operation in reverse. Suppose $c = \{v_1, v_2, v_3\}$ is a set of three vertices in G such that $G - c$ has two connected components G'_1, G'_2. Let G_1 denote the induced subgraph $G[V(G'_1) \cup c]$, and G_2 denote the induced subgraph $G[V(G'_2) \cup c]$ (that is, G_1, G_2 are graphs formed by separating G along c, keeping c intact in both the separated parts). We will also refer to the graph G_1 as the graph obtained by *augmenting* G'_1 with c. If the graph induced by c in G is not a clique, add virtual edges between the vertices of c to make c a clique in both G_1 and G_2. We say G is a *proper* clique sum of G_1, G_2 if both G_1, G_2 are isomorphic to proper minors of G (every minor of G is proper except G itself).

Then we can represent this decomposition as a tree consisting of three nodes, two of them corresponding to the graphs G_1, G_2, and one node corresponding to the 'clique' c to which nodes for G_1, G_2 are attached. We call the graphs G_1, G_2 as *pieces* in this decomposition. The nodes corresponding to G_1, G_2 in the decomposition tree are called *piece nodes* and the node corresponding to c is called the *clique node*. We can write G as $G_1 \oplus_c G_2$.

If G_1, G_2 above were to contain another separating vertex/pair/triplet, we could decompose them further. At every step when separating along a separating pair/triplet, we add virtual edges to make the separating pair/triplet a clique, if it is not already one. We can keep track of which edges in the graph are virtual edges and which are the real edges. Thus we can repeatedly apply this procedure to decompose any graph G into smaller pieces and the decomposition would result in a tree consisting of piece nodes and clique nodes, where the edges of the tree represent the incidence relation between the pieces and cliques (see [DHN+04, CE13, EV21] for more details and examples on clique sum decompositions). See Fig. 2.2 for an example of the decomposition (we use red color for clique nodes and blue color for piece nodes). We generally use T_G to denote this tree. It is easy to see that this is a valid tree decomposition since all the required properties are satisfied.

Although it is not necessary to have clique nodes to define a clique sum decomposition, we use them because in the case when multiple piece nodes in T_G are glued at a single common clique, a clique node for that clique gives a convenient way to represent the incidence relation in T_G. At times when it is clear that only two pieces are attached at a clique, we may skip drawing clique nodes in figures. We will also abuse notation sometimes and use the terms *piece* and *piece node* interchangeably (similar with *clique* and *clique node*).

The following is a theorem from [RS93] on clique sum decomposition of single-crossing-minor-free graphs that we will use.

Theorem 2.11 (Robertson-Seymour [RS93]). *For any single-crossing graph H, there is an integer τ_H such that every graph with no minor isomorphic to H is either*

1. *the proper 0-, 1-, 2- or 3-clique-sum of two graphs, or*
2. *planar, or*
3. *of treewidth $\leq \tau_H$.*

Thus, every H-minor-free graph, where H is a single-crossing graph, can be decomposed by 3-clique sums into graphs that are either planar or have treewidth at most τ_H. Polynomial time algorithms are known to compute this decomposition [DHN+04, KW11, GKR13], and also NC algorithms [EV21].

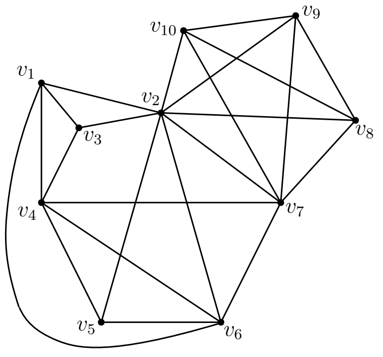


FIGURE 2.1: An example of a graph G.

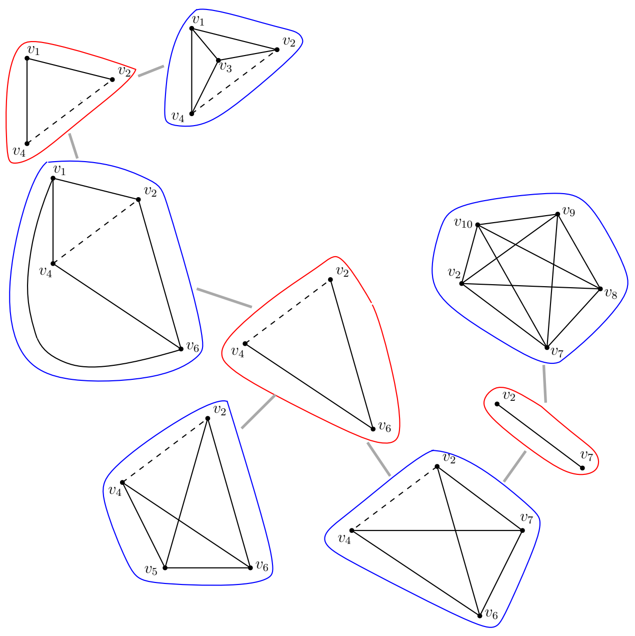


FIGURE 2.2: A clique sum decomposition of G. Red nodes are the clique nodes and blue node the piece nodes. Dashed edges denote virtual edges.

We emphasize that the bounded treewidth/planarity condition on the pieces we get in the decomposition, is along with the virtual edges taken into account. An important fact regarding this decomposition is the following. Suppose G_1 is a planar piece in a 3-clique sum decomposition of an single-crossing-minor-free graph G, and suppose $c = \{v_1, v_2, v_3\}$ is a clique of vertices in G_1 (possibly with virtual edges) along which G_1 was separated from one or more pieces (that is, in the decomposition tree T_G, the piece node corresponding to G_1 is adjacent to a clique node corresponding to c). Then the vertices v_1, v_2, v_3 lie on a common face in G_1. If this were not the case then vertices in the interior of c would be disconnected from the vertices in the exterior of c in G_1, which means that G_1 could have been decomposed further along c.

For any graph G in general, a 2-clique sum decomposition is obviously a special case of 3-clique sum decomposition since we only separate along cut vertices and separating pairs. A particular variant of 2-clique sum decomposition is widely used, including in our work : the decomposition into the *triconnected components* of G, also referred to as the *SPQR* tree of G. We emphasize however that in our work (and also in some other works in the literature), the term *triconnected* differs from the term 3-connected. The triconnected components of a graph will either be a 3-connected graph, a simple cycle, or a *dipole*. A dipole is a multi-graph with two vertices that are connected by multiple edges (except for one, the other edges in the dipole would be virtual edges from the decomposition). We refer the reader to [Lan37, HT73a, DBT90, DLN+22] for more details on the definitions regarding triconnected components and their decomposition. The reason to not decompose cycle components further is because it is useful in many scenarios and it also makes the triconnected decomposition unique. The algorithm of Hopcroft and Tarjan [HT73a] decompose the graph fully by 2-clique sums first (that is, even cycles are decomposed), and then recombine some bonds and triangles repeatedly so that finally some of the components are simple cycles. They call these the triconnected components of the graph and show that they are unique. The algorithm in [DLN+22] follows a slightly different approach by not decomposing cycle components in the first place. We use the following definition of 3-*connected separating pairs* from [DLN+22]:

Definition 2.12. Let $G = (V, E)$ be a biconnected graph. A separating pair $\{a, b\}$ is called 3-connected if there are three vertex-disjoint paths between a and b in G.

Note that a cycle does not contain any 3-connected separating pairs. They show that by decomposing the graph only along 3-connected separating pairs, we get exactly the same components as defined in the earlier works of [HT73a, DBT90]. The following is an important lemma from [DLN+22] that we will use.

Lemma 2.13. *[DLN+22] The triconnected components as well as the triconnected decomposition tree of a graph can be computed in* L.

2.3 Logic prelims

We refer the reader to standard texts like [End01] for details on syntax and semantics of first order and second order logic. Like first order logic, the symbols on second order logic consist of the usual logical symbols (like $=, \forall, \ldots$ etc) and non-logical symbols that include symbols for constants, relations, and functions of various arities. A logical *structure* $\mathcal{A}$ consists of all the symbols described in the syntax, a *domain of discourse* $|\mathcal{A}|$, and an *interpretation function* that maps non-logical symbols (symbols for constants, relations, functions) to elements, relations, and functions on $|\mathcal{A}|$. In first order logic, quantifiers can be applied to variables that range over elements of $|\mathcal{A}|$. In second order logic on the other hand, we have more types of variables that can range over relations, functions on elements of $|\mathcal{A}|$. Quantifiers can be applied on such variables.

Monadic second order logic is a restriction of second order logic where the quantification is allowed only on *unary* relations on $|\mathcal{A}|$, that is, on sets of elements of $|\mathcal{A}|$. Quantification is not allowed on functions or non-unary relations on $|\mathcal{A}|$.

We will use the notion of Gaifman graph of a logical structure:

Definition 2.14. Given a logical structure $\mathcal{A}$, with a finite universe of discourse $|\mathcal{A}|$, and relation symbols which are interpreted as relations of $\mathcal{A}$, the Gaifman graph of $\mathcal{A}$, $G_{\mathcal{A}}$ is a graph whose vertices are the elements of $|\mathcal{A}|$, and for $u, v \in |\mathcal{A}|$, we add an edge between them iff u, v both occur in a relation tuple of $\mathcal{A}$.

The treewidth of the structure is the treewidth of its Gaifman graph. We will use the following extension of Courcelle's theorem [Cou90] by [EJT10]:

Theorem 2.15. *[EJT10] For every $k \geq 1$ and every MSO-formula ϕ, there is a logspace DTM that on input of any logical structure $\mathcal{A}$ of tree width at most k, decides whether $\mathcal{A} \models \phi$ holds.*

Chapter 3

Review and revised analysis of existing results on parallel DFS

We restate some of the existing algorithms related to DFS and analyze their complexity with respect to circuits/space bounded complexity classes using more modern results.

Most algorithms for parallel DFS in undirected graphs proceed via finding a balanced path separator in the graph (we refer the reader to Section 2.2.3 for prelims on separators), starting from the root [Smi86, HY88, AA88, Khu90]. This path would become a branch of the DFS tree and we could recurse on the components attached to the path while backtracking on it. Since the path is a balanced path separator, the size of these components attached to the path is guaranteed to be at most a constant fraction of the graph. Therefore we would get a DFS tree in $O(\log n)$ phases. For directed graphs also, a divide and conquer strategy using path separators is used, but there are more technicalities involved as we will see.

It was shown by Kao in [Kao88] that all digraphs have balanced path separators (the argument also works for undirected graphs). The proof goes via considering a DFS tree and showing that one of the paths from the root to a vertex in the tree will be a $\frac{1}{2}$-path separator for the graph. The challenge thus is to find a path separator in NC. Kao also showed that if we have a small number of disjoint paths that together constitute an α-separator (we call these as multi-path separators) in digraphs (by small we mean upto polylogarithmically many), then we can trim and merge them into a single α-path separator in NC. If we could somehow merge and reduce a multi-path separator consisitng of a large number of paths , say $O(n)$ many, to a multi-path separator consisting of upto polylogarithmically many paths in NC, then we would immediately have an NC algorithm for DFS, as all vertices of the graph (or any half of them) constitute a trivial set of path separators. Such a routine to reduce a multi-path separator consisting of $O(n)$ many paths to one consisting of constantly many paths is indeed the heart of the RNC algorithm of Aggarwal and Anderson [AA88] (and also [AAK90]), and this is where the randomized routine for finding a minimum weight perfect matching is used. We also remark that in the literature of separators, an aspect of focus is often to minimize the size of the separator [LT79, Mil86]. For our purpose of constructing a DFS tree, the length of the path/cycle separator is not important.

We have five subsections in this chapter. First we present an algorithm to find a kind of balanced cycle separator in weighted undirected planar graphs by Miller [Mil86]. We will use it in Chapter 5. Then we present the machinery developed by Kao [Kao88, Kao95] to merge small number of path separators into a single path separator. Next we show an algorithm by [KK93] to construct a balanced path separator in planar digraphs and give a modern analysis of it. Then we discuss the methods to construct DFS trees using balanced path separators in undirected as well as directed graphs. In the last subsection, we will present the algorithm of Hagerup [Hag90] that constructs a DFS tree of an undirected planar graph very efficiently, bypassing the approach of recursively constructing balanced separators. The decomposition of [Hag90] would serve as context to the decomposition for planar digraphs in our work [ACD22] that we will present in the next chapter.

We give a list of parallel DFS algorithms in some graph classes with best running times, that we are aware of prior to our work. Here we use the notation CRCW[T(n),P(n)] to denote an algorithm running on CRCW PRAM in time T(n) using P(n) many processors.

TABLE 3.1

Graph class	**Existing parallel DFS algorithms**
Planar undirected	CRCW[$O(\log n)$, n] by Hagerup [Hag90]
Planar directed	EREW[$O(\log^7 n)$, n^4] by Kao [Kao88],
	CRCW[$O(\log^{10} n)$, n] by Kao and Klein [KK93],
	also CRCW[$O(\log^5 n)$, $n/\log n$] by Kao [Kao95] for *strongly connected* planar digraphs.
Bounded genus (directed or undirected)	Not known
Bounded treewidth (directed or undirected)	Not explicitly stated but a bound of AC^1(L) follows from Kao's machinery and [EJT10]
Single-crossing-minor free	Not known CRCW[$O(\log^3 n)$, n] by Khuller [Khu90] for $K_{3,3}$-minor-free graphs.

3.1 Cycle Separators in undirected planar graphs

Consider the following, slightly more general notion of cycle separators in planar graphs, that was introduced in [Mil86]:

Definition 3.1. Let G be an undirected planar graph, and $w : V(G) \cup E(G) \cup F(G) \to \mathbb{Q}^+$ be a normalised weight function assigning weights to vertices, edges and faces of G that sum upto 1. Let $\alpha \in [\frac{1}{2}, 1)$. We call a cycle C an α-*interior-exterior* cycle separator if $w(int(C)) \leq \alpha$, and $w(ext(C)) \leq \alpha$.

Having a routine to find a separator of this kind can easily be used to find a path separator in an unweighted biconnected graph by assigning a weight function that gives zero weight to faces and edges. Miller showed in that such a weighted cycle separator always exists if G is biconnected and if no face is too heavy, and also that such a separator can be constructed in CRCW$[\log n]$[1].

[1]The theorem in [Mil86] also gives a bound on the size of the cycle separator, but that is not important for our purpose of constructing a DFS tree

We restate the theorem here:

Theorem 3.2. *[Mil86] Let G be a biconnected planar graph, and $w : V(G) \cup E(G) \cup F(G) \rightarrow \mathbb{Q}^+$ be an assignment of non-negative weights to vertices, edges, faces that sum upto 1, such that no face has weight more than $2/3$. Then we can construct a $2/3$-interior-exterior-cycle separator of G in* CRCW$[\log n]$.

3.2 Kao's machinery on path separators

Next, we reiterate an algorithm of Kao (see [Kao88, AAK90, Kao95]) that combines a small number of path separators into a single path separator in parallel, and observe that it is in L given an oracle for REACH. The trimming argument of the proof appeared in [Kao88] where Kao demonstrates that a path separator in a digraph can be converted to a directed cycle separator.

Theorem 3.3. *([Kao93, Kao95]) Let G be a digraph and $\{P_1, P_2\}$ be two paths that together form an α-separator of G for some $\alpha \in [\frac{1}{2}, 1)$. Given an oracle for* REACH*, we can in* L*, construct a single path P_3 that is also an α-separator of G.*

Again, the theorem is stated for digraphs, but the corresponding statement for undirected graphs also holds with appropriate changes. We reproduce the proof for completeness.

Proof. Let the two paths that together form an α separator be $P_1 = \{u_1, u_2 \dots u_p\}$ and $P_2 = \{v_1, v_2 \dots v_q\}$.

- Trim the path P_1 to $P_1' = \{u_1, u_2 \dots u_s\}$ $(s \leq p)$ such that separator property is just lost on not including u_s in the separator. That is, $P_1' \cup P_2$ is an α-separator of G, but $P_1' \backslash \{u_s\} \cup P_2$ is not. This means that in $G - (P_1' \backslash \{u_s\} \cup P_2)$, the vertex u_s is a part of a strongly connected component of size at least αn.

- Having trimmed P_1 to P_1', repeat the same step and trim P_2 to $P_2' = \{v_t, v_{t+1} \dots v_q\}$ $(t \geq 1)$ such that $P_1' \cup P_2'$ forms an α-separator, and the vertex v_t is part of some strongly connected component of size at least αn in the graph $G - (P_1' \cup P_2' \backslash \{v_t\})$.

- Thus in $G - (P_1' \backslash \{u_s\} \cup P_2' \backslash \{v_t\})$, there are two strongly connected components of size at least αn $(\alpha \geq 1/2)$, containing u_s, v_t resepectively. This means that they must be part of a single strongly connected component and hence there must be a path P from u_s to v_t in $G - (P_1' \backslash \{u_s\} \cup P_2' \backslash \{v_t\})$. Therefore we can concatenate the paths P_1', P, P_2' into a single path $P_1'.P.P_2'$, which is an α-separator of G.

While trimming paths in steps 1 and 2, we only need to remember the current end point of the trimmed path, which can be done in L. Other steps, like checking the size of leftover connected components and finding a path from u_s to v_t in the subgraph can be done in L given an oracle to REACH. □

Note that as a corollary, the above algorithm is in NL for general directed graphs, and in UL ∩ co-UL for planar directed graphs (using [BTV09]). For undirected graphs, the algorithm (with appropriate changes) is in L using Reingold's algorithm [Rei08]. As noted in [AAK90, KK93], if we have a multipath separator consisting of k many paths, say $\{P_1, P_2 \dots P_k\}$, we can combine them into a single path separator using k iterations of the above proceedure

(First merging P_1, P_2 into one path, then merging the result with P_3 and so on). If k is a constant, we can do this in L using k transducers. Thus we have the following corollary:

Corollary 3.4. *Let G be a digraph and k be a constant. Let $\{P_1, P_2, \dots P_k\}$ be k disjoint paths that form a multipath α-separator of G for some $\alpha \in [\frac{1}{2}, 1)$. Given an oracle for* REACH*, we can in* L*, construct a single path P_{k+1} that is also an α-separator of G.*

Another useful tool Theorem 3.3 gives is that of shifting the starting vertex of a path separator.

Corollary 3.5. *Suppose all vertices in G are reachable by a vertex, say r, and we have a balanced path separator P of G that does not include r. Then using an oracle for* REACH*, we can output a balanced path separator of G that starts at vertex r, in* L.

Proof. As shown in [Kao88], we can first convert the path separator P into a cycle separator of G. Now any path from r to this cycle separator will give us a path separator that starts at r. □

For bounded tree width graphs, we can find a tree decomposition in L using [EJT10], and we know that there is always a bag in the tree decomposition of a graph whose vertices together form a $1/2$-separator of the graph. Since REACH is known to be in L [EJT10] for digraphs of bounded treewidth, the following result follows using the corollary above:

Corollary 3.6. *If G is a digraph of bounded treewidth, we can find a $1/2$-path separator of G in* L.

We have stated the theorem and the corollaries for digraphs, but the corresponding statements for undirected graphs also hold true, with similar proofs.

3.3 Finding a path separator in planar digraphs

Since our results in the next chapter will use balanced path separators in planar digraphs, we present the algorithm in [KK93] to compute these. The parallel complexity in [KK93] was $O(\log^2 n)$ time using $O(n)$ many processors. We analyse it by plugging in more modern results on reachability and distance computation. The algorithm uses the following result of separator by Lipton and Tarjan [LT79].

Lemma 3.7. *Let G be an embedded planar undirected graph and T be a spanning tree of G rooted at r. Then there exist two vertices x, y such that the paths from r to x and from r to y in T together form a $2/3$-path separator of G.*

We refer to Lemma 2 of [LT79] for the proof (Their lemma also talks about a bound on the size of separator but that is not important to us). To give some more detail, they first add more edges to the graph (these can also be thought of as non-tree edges) until every face is a triangle, but keeping the graph planar. Then they show that their exists one non-tree edge (either original or an added one) (x, y) such that the fundamental cycle formed by it constitutes a $2/3$-*interior-exterior* cycle separator of the graph. Thus we can add a path from r to the least common ancestor of x, y to get the required two path separator.

Now we explain how [KK93] use it to construct a separator in planar digraphs. Given a planar digraph as input, the algorithm to find a path separator proceeds as follows:

1. Compute an embedding of the graph.

2. Compute the strongly connected components of G. If none of the them have size more than $2/3$rd of $|V(G)|$, then the empty set is a trivial path separator. Else look a the strongly connected component, say G_1, that is of size more than that.

3. Compute an arborescence T from any vertex r of G_1. To do this, first assign weights to edges as described in [BTV09] so that for every vertex v, there is a unique path from r to v in G_1 of minimum weight. Now we compute the minimum weight path from r to every vertex in G_1 using [TW10], and it is easy to see that this will be an arborescence of G_1.

4. Now we go over every pair of vertices (x, y), and check if the two paths in T: one from r to x, and the one diverging from the LCA of x, y to y, together form a $2/3$-path separator. By Lemma 3.7, one of them will give the required two path separator. Using Theorem 3.3 and Corollary 3.5, we can merge the two paths into a single $2/3$-path separator and shift the root as required.

Step 1 is in L using [AM04, DP11]. Step 2 is in UL ∩ co-UL using the result of [BTV09] that reachability in planar digraphs is in UL ∩ co-UL. The algorithm assigning the weights described in Step 3 is in UL ∩ co-UL [BTV09]. It is described in [TW10] that we can compute shortest paths between vertices of a graph in UL ∩ co-UL if the shortest path between every pair of vertices is unique. Step 4 is also in UL ∩ co-UL since we can enumerate over every pair of vertices in logspace and check for the separator property in UL ∩ co-UL.

Therefore we have the following result:

Theorem 3.8. *Given any planar digraph G, we can compute a $2/3$-path separator of G in* UL ∩ co-UL.

In the next chapter, we will show a different proof of this theorem, but with the fraction of separator being $11/12$ instead of $2/3$. We discovered this analysis of the above algorithm after the work in [ACD21, ACD22].

3.4 Application of Path Separators in Depth First Search

It is shown in [Smi86] that a proceedure to find a cycle separator (or a path separator in fact) in undirected graph, can be used to construct a DFS tree of the graph in parallel. We restate the result and argument in a slightly different form and with a more modern analysis.

Proposition 3.9. *[Smi86]*
Let $F_{sep}(H)$ be a function that takes as input an undirected graph H and returns a balanced path separator of H. Then there exists a uniform family of AC^1 *circuits, with oracle gates for F_{sep} and for functions computable in* L*, that take as input a graph G and a vertex r in it, and return a DFS tree of G rooted at r*

Proof. We can compute a DFS tree of G using the following proceedure:

1. Use the function F_{sep} to compute an α-path separator P of G. Use Corollary 3.5 to ensure that r is an end point of P. Let $G_1, G_2, \ldots G_l$ be the connected components of $G - P$. We can output these on a transducer in L using [Rei08].

2. For each connected component $G_i, i \in [1..l]$, consider an edge (u_i, v_i) such that $u_i \in G_i, v_i \in P$, and there is no other edge (u_i, v_i') with $u_i \in G_i, v_i' \in P$ such that v_i' occurs after v_i in P. In other words, (u_i, v_i) is an edge that connects G_i to P at a vertex furthest from r. It is clear that for each G_i, if we compute DFS tree of G_i rooted at u_i and attach them at v_i, then we get a valid DFS tree of G which corresponds to first traveling along P from r, and then exploring G_i's while backtracking on P. Thus we recursively compute the DFS trees of the G_i's in parallel and attach them to P as described.

Step 1 can be performed in logspace by [Rei08]. In step 2, the size of the G_i's is at most a constant fraction of the size of G since P is a balanced separator of G. Therefore the recursion depth is $O(\log n)$.

In terms of circuits, consider the following AC^0 circuit with recursive gates to compute the DFS tree. The first level of the circuit has an oracle gate for the logspace transducer described in step 1 which outputs all G_i's and the edges by which they are attached to P. The next level of the circuit contains recursive gates, that compute the DFS tree of G_i's with respect to u_i as described above. The third level of the circuit has gates to simply combine the DFS trees of G_i's with P as described and output a DFS tree of G. Since the size of G_i's reduces by a constant fraction in each iteration, the recursive gates at the second level can be expanded $O(\log n)$ times, leading to a circuit of $O(\log n)$ depth with oracle gates for the logspace transducers described. Therefore we get an AC^1 circuit with oracle gates for functions computable in logspace and for the function F_{sep}.

We can construct a logspace Turing machine that can output this circuit family by describing how to construct each level of the circuit, given the size of the input. □

For directed graphs however, although Aggarwal, Anderson and Kao in [AAK90] use directed path separators, they do it in a way different from the approach described above for undirected graphs. Suppose G is a digraph and P is an α-path separator of G. Then the weakly connected components $G_1, G_2, \ldots G_k$ could still be of large size, as removing P only bounds the size of the strongly connected components of the remaining graph. To limit the depth of recursion, the algorithm in [AAK90] uses path separators (cycle separators in fact) to construct a *partial* DFS tree T of G, that is, a tree which is a subtree of some DFS tree of G, with the same root. They construct a partial DFS tree such that in the remaining graph, the subgraphs *dangling* from the vertices of the partial tree T are of small size, and they show that computing the DFS trees of these dangling subgraphs in parallel and attaching them to T results in a DFS tree of G. The theorem they show regarding computing a DFS tree of a digraph given an algorithm to compute a directed cycle separator is the following:

Theorem 3.10 ([AAK90]). *Suppose a directed cycle separator of any n-vertex strongly connected directed graph can be computed in $T_c(n)$ time using $P_c(n)$ processors. Then a depth-first search spanning forest of any n-vertex digraph can be computed in $O(log^2 n(T_c(n) + log^2 n))$ time using $P_c(n) + MM(n)$ processors.*

For *strongly connected planar* digraphs, Kao [Kao95] gave an improved algorithm to convert a directed cycle separator into DFS trees of such graphs by additionally using proprties of planarity and strong connectivity on dangling subgraphs described in the framework of [AAK90]. The routine of [Kao95] computes a path separator in a strongly connected graph in CRCW[$\log^3 n$] using $n/\log n$ processors. They use this to build partial DFS trees recursively and construct a DFS tree in CRCW[$\log^5 n$] with $n/\log n$ processors.

In the next chapter where we will describe our work in [ACD22], we will show that to construct a DFS tree in a digraph given a routine for finding paths separators, we can in fact adapt the divide and conquer strategy of Proposition 3.9. To handle the large weakly connected components that might remain after removing a path separator, we write them as a DAG of SCCs. There is a simple algorithm to compute the lex-first DFS tree of a DAG with respect to any given ordering on the vertices [dlTK95]. We use this to compute DFS trees of the strongly connected componentsof the DAG in parallel, and sew them together.

3.5 Hagerup's decomposition for Planar Depth-First Search in undirected graphs

Let PLDIST denote the problem of computing distance between to vertices in an undirected planar graph. In this section we observe that Hagerup's CRCW[log n] time algorithm for DFS in planar graphs with some refinements and results on reachability, distance [Rei08, TW10] plugged into it, gives the following result:

Theorem 3.11. *DFS in planar graphs is logspace-Turing-reducible to PLDIST.*

Combined with the result of [TW10] stating that PLDIST is in UL ∩ co-UL, we get the following corollary.

Corollary 3.12. *DFS in planar graphs is in* UL ∩ co-UL.

Though we will not use this theorem in the results we present in our thesis, we will still present the algorithm as the decomposition given in [ACD22] is a generalization of Hagerup's decomposition given in [Hag90]. We preset some terms and notation from [Hag90] and introduce some more for ease of explanation.

Notations and Definitions

Definition 3.13. Let PLDFS be the class of all functions that map an embedded undirected (connected) planar graph G and a vertex $r \in V(G)$ to a set of arcs (or directed edges) A that constitutes a depth-first search tree rooted at r.

Let $G = (V, E)$ be an embedded undirected planar graph. Let $\mathcal{F}$ be the set of G's faces. Consider the face-incidence graph G_D. (This is similar to, but not identical to, the *dual graph*. Namely, the vertices of G_D are the faces of G, and two faces are adjacent in G_D if they share a vertex of G.)

Consider some initial face F_0. The *type* of a face F, denoted $Type(F)$, is its distance from F_0 in G_D. For $\alpha \in V \cup E$ define $Type(\alpha) = \{k \geq 0 | \alpha$ has an incident face F with $Type(F) = k\}$.

Note that G_D need not be planar. We will show in the algorithm how to compute distances of vertices in G_D using the face-vertex incidence graph which is planar.

The following is observed in [Hag90]:

Lemma 3.14. *For every* $\alpha \in V \cup E, Type(\alpha)$ *is an element of the sequence* $\{0\}, \{0, 1\}, \{1\}, \{1, 2\}, \{2\}, \cdots$.

This allows us to define a total order $<$ on set of vertex and edge types by

$$\{0\} < \{0,1\} < \{1\} < \{1,2\} < \{2\} < \{2,3\} < \dots \tag{3.1}$$

Call α *white* if $|Type(\alpha)| = 1$ and *black* if $|Type(\alpha)| = 2$.

Let $G_{\mathcal{B}}$ be the subgraph of G spanned by black edges. We start from a few more lemmas extracted from [Hag90]:

Lemma 3.15. *$G_{\mathcal{B}}$ contains all black vertices of G. Also, all vertices and edges inside a connected component of $G_{\mathcal{B}}$ have the same type.*

Lemma 3.16. *Let C be a simple cycle. If F_0 lies in the interior (exterior) of C and the exterior (interior) of C contains a face of type k, then C must contain a vertex of type $< k$.*

We refer to a biconnected component of a graph as a *block* of the graph.

Lemma 3.17. *Every block of $G_{\mathcal{B}}$ is a simple cycle.*

From now on, we will assume that F_0 is the external face of a connected graph G. It follows from Lemma 3.17 that the black edges of G constitute a set of cycles. Given any two black cycles in G, they are either disjoint, or they share a single vertex.

Next we diverge from [Hag90] and introduce some new terminology:

Definition 3.18. Let V_k denote the (white) vertices of type $\{k\}$ and let $V_{k,k+1}$ denote the (black) vertices of type $\{k, k+1\}$. We call the set $V_k \cup V_{k,k+1}$ the $k+1^{th}$ *layer*, denoted by $\mathcal{L}^{k+1}$. Observe that every vertex of G lies in some $\mathcal{L}^{k+1}$. We call the connected components of a layer *layered connected components* or *LCCs*. $\mathcal{LCC}^{k+1}$ denotes the set of connected components of the $k+1^{th}$ layer; we will use the notation L^{k+1} to refer to an element of $\mathcal{LCC}^{k+1}$.

See Fig. 3.2 for an example. Though we have defined $\mathcal{L}^{k+1}$ as a set of vertices, we will slightly abuse notation and also include edge sets $E_k \cup E_{k,k+1}$ in it. We will also at times refer to the subgraph of G that are formed by the edges and vertices in the layer. It will be clear from context if we are referring to the just the set of vertices of $\mathcal{L}^{k+1}$ or also the edges, when using the term $\mathcal{L}^{k+1}$.

Observe that $\mathcal{L}^1$ consists of all edges that are adjacent to the external face F_0, along with all vertices that are adjacent to F_0. The black edges of $\mathcal{L}^1$ form a set of cycles, and every other black edge of G lies in the interior of some black cycle of $\mathcal{L}^1$. There are no cycles consisting of white edges in $\mathcal{L}^1$ (because any such cycle would necessarily enclose a face, meaning that the edge would not be white).

The vertices in $\mathcal{L}^2$ are adjacent to the external face of the graph $G - \mathcal{L}^1$ (and in general the vertices in $\mathcal{L}^k$ are adjacent to the external face of $G - \bigcup_{i<k} \mathcal{L}^i$). Thus, in particular, each $L^k \in \mathcal{LCC}^k$ is outerplanar. Also, each $L^{k+1} \in \mathcal{LCC}^{k+1}$ lies in the interior of a black cycle in $\mathcal{L}^k$, and there are no cycles consisting entirely of white edges in $\mathcal{L}^k$ for any k.

Since each $L^k \in \mathcal{LCC}^k$ is outerplanar, given any starting vertex v in L^k, it is easy to construct a depth-first search tree $T_v(L^k)$:

- The root of $T_v(L^k)$ is v.
- For a white vertex x of $T_v(L^k)$, all edges adjacent to x are white. Since there are no white cycles, attach the acyclic collection of white edges that are adjacent to x.

- For a black vertex x of $T_v(L^k)$, if x is not a cut vertex (that is, x is an element of only one black cycle C) where x is the first element of C to be added to the tree, append the path through C starting at x (but do not add the edge that comes back into x, to maintain acyclicity) There are two directions that one could choose to walk along C; arbitrarily choose the counterclockwise traversal. Call this vertex x the *lead vertex* of C in $T_v(L^k)$.

- For a black vertex x of $T_v(L^k)$, if x is a cut vertex that is an element of different cycles $C_1, \ldots C_\ell$ not already in the tree, then (as in the previous case) append the path around each C_i in the counterclockwise direction, and make x the *lead vertex* of C_i in $T_v(L^k)$.

- Repeat until $T_v(L^k)$ spans L^k.

The important properties are (1) In a depth-first traversal of $T_v(L^k)$, when the lead vertex of a cycle C is encountered, all of the other vertices of C are visited before any other vertex is visited, and (2) the lead vertex for each cycle is determined entirely by v. We can thus denote this by $lead(C, v)$.

In order to see that $T_v(L^k)$ can be constructed in logspace, note first that (using reachability in undirected graphs [Rei08]) it is easy to find all of the cut vertices (see [AM04]) and hence to list all of the cycles, and to create a graph $T'_v(L^k)$ that is a spanning tree of the graph that results when each cycle in L^k is contracted to a single vertex. $T_v(L^k)$ is trivial to compute, given $T'_v(L^k)$.

Each $L^{k+1} \in \mathcal{L}^{k+1}$ is contained in a black cycle C in $\mathcal{L}^k$. There may be more than one connection between C and L^{k+1}. A contribution of [Hag90] is to show that there is an edge which we denote $e(v, L^{k+1})$ connecting C to some vertex u in L^{k+1} such that that a depth-first search tree for G incorporating $T_v(L^k)$ and $T_u(L^{k+1})$ can be constructed using the edge $e(v, L^{k+1})$. By induction, if we know the root of the depth-first tree for L^1, this determines the lead vertex of every black cycle in G and thus also uniquely determines the connection between every black cycle C and every LCC L connected to C. Let us denote this connection $e'(r, L)$, where $e'(r, L^{k+1})$ is $e(v, L^{k+1})$ for the choice $v = e'(r, L^k)$. Connecting the forest consisting of the trees $T_x(L)$ for various x and L along with the edges $e'(r, L)$ results in a depth-first tree for G.

We can now describe the algorithm at a high level:

1. Given an undirected graph G and a vertex r, determine if G is planar and connected, and if so, compute an embedding of G in the plane with r on the external face.

2. Construct the face-vertex incidence graph G'_D of G. This is the undirected bipartite graph on vertex set $V \cup \mathcal{F}$ that contains an edge $\{u, F\}$, for all $u \in V$ and $F \in \mathcal{F}$, exactly if F is incident on u in G. G'_D is clearly planar.

3. For each face of G (i.e., each vertex of G_D), compute its distance from the external face F_0. This is clearly half of its distance from the external face in G'_D, which is planar. This gives us distances in G_D.

4. Using this distance information, compute the type of each vertex and edge of G. This allows us to partition the vertex set of G into the graphs $\mathcal{L}^k$.

5. Partition each $\mathcal{L}^k$ into connected components, to obtain the set $\mathcal{LCC}^{k+1}$.

6. For each $L^k \in \mathcal{LCC}^{k+1}$ and for each v in L^k construct $T_v(L^k)$.

7. For each k, for each $L^{k+1} \in \mathcal{LCC}^{k+1}$, identify the black cycle C in $\mathcal{L}^k$ that contains L^{k+1}.

8. Construct a tree S whose nodes consist of vertices v_C for each black cycle of G and v_L for each LCC L. The root of S is the unique LCC $L^1 \in \mathcal{L}^1$. (It is unique because G is connected.) The children of any LCC $L^k \in \mathcal{L}^k$ are the black cycles in L^k. The children of any black cycle C in $\mathcal{L}^k$ are the LCCs L^{k+1} that are in the interior of C.

9. Label each edge (L, C) in the tree S with the name of the lead vertex of the black cycle C in the depth-first traversal of G starting at r, and label each edge (C, L) in the tree S with the name of the vertex of L that is traversed in the edge $e'(r, L)$.

10. Start constructing the tree T by incorporating the tree $T_r(L^1)$ (rooted at the root r).

11. Output the tree T that is composed of the trees $T_v(L)$ for each $L \in \bigcup_k \mathcal{L}^k$ and each $v \in L$, along with the edges $e'(r, L^{k+1})$ for each $L^{k+1} \in \mathcal{L}^{k+1}$, using the information computed in step 8. (Discard all of the trees $T_v(L)$ that are not connected to T, i.e., those trees where v is the "wrong" root.)

The first step is computable in logspace [AM04, Rei08]. The second step is computable in logspace. The third step is computable in logspace with an oracle for computing distance in planar graphs. Each of the remaining steps is easy to compute in logspace, using the fact that reachability in undirected graphs is logspace-computable [Rei08]. The step that requires the most explanation is step 9. In order to label an edge (L, C) in the tree, start a traversal of $T(r, L^1)$, which allows us in logspace to compute the lead vertex of each black cycle C' in $\mathcal{L}^1$. If C is not one of these cycles C', then the edge (L, C) is a descendent of one such C', and we can determine which one in logspace. Knowing the lead vertex of C', we can determine the edge of G that connects this cycle C' to the LCC L' such that the edge (C', L') is followed from the root of S to (L, C). Knowing this edge, we can determine the lead vertex of each cycle C'' such that (L', C'') is an edge of S. If C is also not any one of these cycles C'', then again we can in logspace determine the next node of S that needs to be traversed, and we can continue in this way until we finally reach the edge (L, C). It is important to note that we do not need to maintain the entire list of lead vertices that are encountered along the way. Note also that this procedure also gives us the information that is required to label vertices of the form (C, L). This procedure is repeated for each edge of S (and thus the lead vertices of each black cycle are recomputed many times).

The argument showing that the tree produced in this way is a depth-first tree is similar to the argument in [Hag90].

This completes the proof.

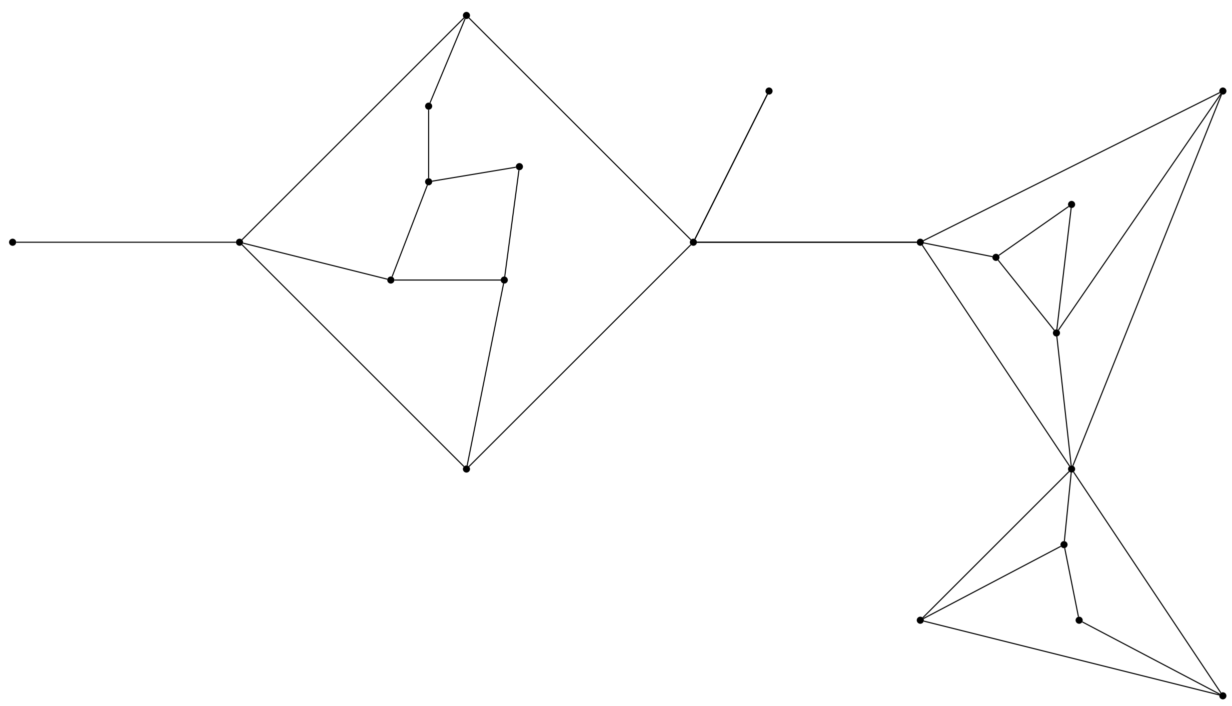


Figure 3.1: An example of a connected planar graph

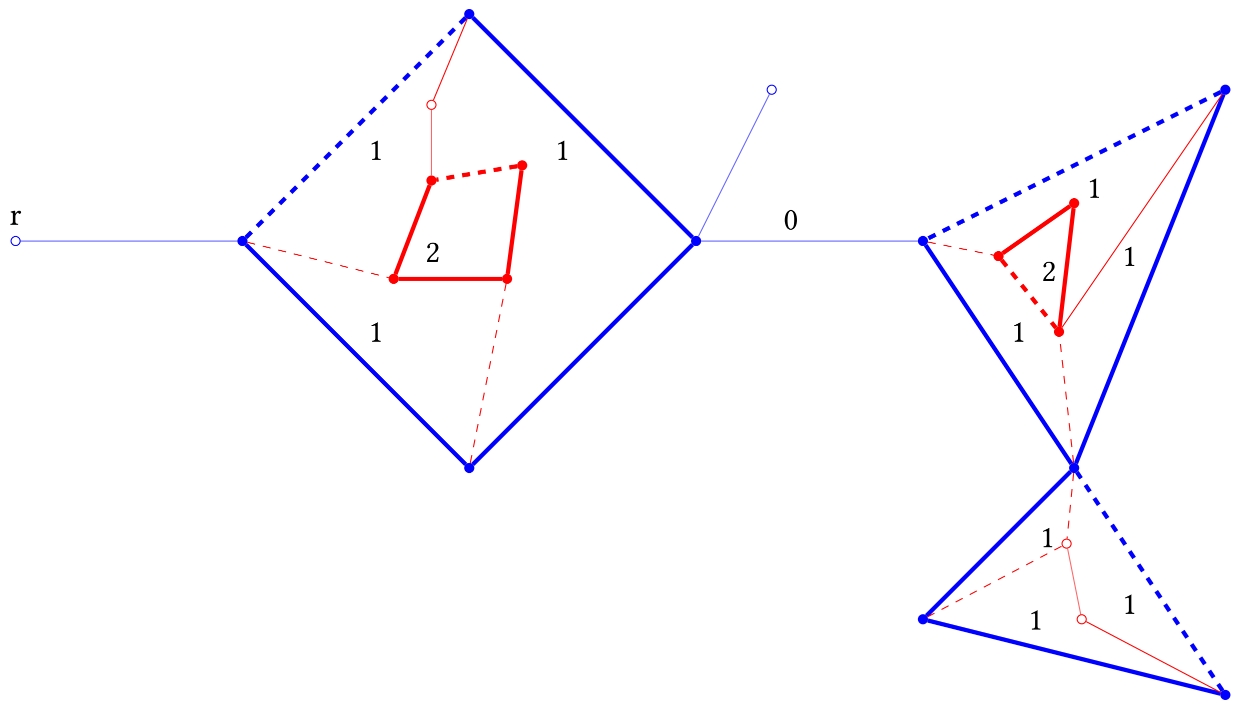


Figure 3.2: The example graph with layers. The numbers denote the distance of the faces from the outer face in the face-incidence dual. Blue edges and vertices belong to layer 1. Red edges and vertices belong to layer 2. The edges and vertices in bold denote the 'black' cycles (we have drawn colored here to differentiate the layers). Dotted lines are non-tree edges of a DFS tree rooted at vertex labelled r. The direction we choose for traversing cycles is counterclockwise.

Chapter 4

DFS in directed planar graphs

In this chapter, we will present our results on parallel complexity of DFS in directed planar graphs.

It is known that for directed acyclic graphs (DAGs), the lexicographically-least DFS tree from a given vertex can be computed in AC^1 [dlTK95]. (See also [dlTK01, AIK$^+$14, EHK15, NVG17, IO20, Hag20].) First, we observe that, given a DAG G and an ordering on vertices, computation of the lexicographically-least DFS tree in G logspace-reduces to the problem of reachability in G. Thus, for general DAGs, computation of a DFS tree lies in NL. For classes of graphs where the reachability problem lies in L, so does the computation of DFS trees in DAGs in those classes. One such class of graphs is planar DAGs with a single source (see [ABC$^+$09], where this class of graphs is called SMPDs, for **S**ingle-source, **M**ultiple-sink, **P**lanar **D**AGs).

For undirected planar graphs, we saw in the previous chapter that Hagerup's AC^1 algorithm [Hag90] for DFS can be put in $\mathsf{UL} \cap \text{co-}\mathsf{UL}$.

Our main contribution in the current paper is to show that a more sophisticated application of the ideas in [Hag90] leads to an $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ algorithm for construction of DFS trees in planar *directed* graphs. (That is, we show DFS trees can be constructed by unbounded fan-in log-depth circuits that have oracle gates for a funtion in $\mathsf{UL} \cap \text{co-}\mathsf{UL}$.) Since $\mathsf{UL} \subseteq \mathsf{NL} \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1$, the $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ algorithm can be implemented in AC^2.

There are two main lemmas that we show to get the result.

Theorem 4.1. *Given a planar digraph, and a vertex r designated as root, we can compute an $(\frac{11}{12}, r)$-path separator in* $\mathsf{UL} \cap \text{co-}\mathsf{UL}$.

To prove this theorem we use a generalization of the decomposition of [Hag90] that we saw in the previous chapter. Though we also saw in the previous chapter that this theorem (with a better fraction) follows from existing algorithms [KK93] with some tweaks, we believe that the graph decomposition we develop (which we show can be computed in $\mathsf{UL} \cap \text{co-}\mathsf{UL}$) may still be useful in further work on directed planar graphs.

The other theorem we show is related to using the path separator we find to construct a DFS tree.

Theorem 4.2. *Let $F_{sep}(H)$ be a function that takes as input a digraph H and returns a balanced path separator of H. Then there exists a uniform family of* AC^1 *circuits, with oracle gates for F_{sep}, functions computable in* $\mathsf{UL} \cap \text{co-}\mathsf{UL}$, *that take as input a digraph G and a vertex r in it, and return a DFS tree of G, rooted at r*

We differ here from the existing approaches to construct a DFS tree using path/cycle separators described in the previous chapter. Our algorithm involves a simple application of the result of DFS in DAGs to patch up DFS trees of the leftover strongly connected components that computed in parallel recursively.

These two theorems give the following corollary which is the main result of this work:

Corollary 4.3. *DFS in directed planar graphs is in* $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$.

4.1 Preliminaries

We refer the reader to Chapter 2 for basic preliminaries used in this chapter.

In this chapter when we call a digraph a *tree* (as opposed to a *DFS tree*) we mean only that the underlying *undirected* graph forms a connected acyclic graph. Similarly, if the underlying undirected graph is acyclic, the directed graph is said to be a *forest*, and when we refer to the k-connected components of G, we are referring to the subgraphs of G corresponding to the k-connected components of the underlying undirected graph.

Any (directed) cycle C in a plane graph divides the plane into two connected regions: the *interior* and the *exterior*. Any vertex not on C that is embedded in the interior region is said to be *enclosed* by C. In the previous chapter, we saw Hagerup's decomposition using the notion of *black* cycles. Here we will talk about colored cycles (red for clockwise and blue for counter-clockwise). We remark however that we do not use coloring of edges/vertices in the exclusive sense as is often used, that is, an edge/vertex for example could have both red and blue colors or none. When we say that v is *immediately enclosed by* the colored cycle C, it means that v is enclosed by C and there is no other colored cycle C' enclosing v whose interior is a subset of the interior of C. A subgraph H is *strictly enclosed* by C if no edge of H lies on C and every edge of H (except possibly its endpoints) is embedded in the interior of C.

4.2 DFS in DAGs Logspace-Reduces to Reachability

In this section, we observe that constructing the lexicographically-least DFS tree in a (not-necessarily planar) DAG G can be done in logspace given an oracle for reachability in G. But first, let us define what we mean by the lexicographically-least DFS tree in G:

Definition 4.4. Let G be a DAG, with some ordering on the neighbours of each vertex. (For example, with adjacency lists, we can consider the ordering in which the neighbours are presented in the list. But we will also need to consider different orderings.) For *any such ordering*, the lexicographic-least DFS traversal of G is the traversal done by the Algorithm 1.

That is, the lexicographically-least DFS tree is merely a DFS tree, but with the (very natural) condition that the children of every vertex are explored in the given order. Importantly, when we apply this procedure as part of our algorithm for DFS in planar graphs, the ordering on the neighbours of v will be determined *dynamically*. (Note that, in the algorithm that defines the lexicographically-least DFS traversal, no reference is made to the ordering of the neighbours of v until it is visited; thus it causes no problems if this ordering is not determined until that time.) Also, we will need to apply our algorithm to directed acyclic *multigraphs* (i.e., graphs with parallel edges between vertices) where there is a logspace-computable function $f(v, e)$ that computes the ordering of the neighbours of vertex v, assuming that v is entered using edge e – where e can also be "null" if v is the root of the traversal. (That is, if the DFS

```
**Input:** $(G, v)$
**Output:** Sequence of edges in DFS tree
visited[$v$] $\leftarrow 1$
visited[$w$] $\leftarrow 0$ for all $w \neq v$
for every out neighbour $w$ of $v$, in the given order do
    if visited[$w$] = 0 then
        print$(v, w)$
        $DFS(G, w)$
    end
end
```

Algorithm 1: Static DFS routine

tree visits vertex v from vertex x, and there are several parallel edges from x to v, then the ordering of the neighbours of v may be different, depending on which edge is followed from x to v.) [1]

As is observed in [dlTK95], the unique path from s to another vertex v in the lexicographically-least DFS tree in G rooted at s is the lexicographically-least path in G from s to v.[2]

Now consider the following simple algorithm for constructing the lexicographically-least path in a DAG G from s to v, shown in Algorithm 2:

```
**Input:** $(G, s, v, f)$
**Output:** Lexicographically-least path from $s$ to $v$ under $f$
$current \leftarrow s$; $e \leftarrow null$;
while $(current \neq v)$ do
    $child \leftarrow$ first child of $current$ (in the order given by $f(current, e)$)
    while $(REACH(child, v) \neq TRUE)$ do
        $child \leftarrow$ next child of $current$ (in the order given by $f(current, e)$)
    end
    $e \leftarrow$ a selected edge from $current$ to $child$; output $e$
    $current \leftarrow child$;
end
```

Algorithm 2: DAG DFS routine

The correctness of this algorithm is essentially shown by the proof of Theorem 11 of

[1] Let us give additional motivation for having a dynamically-computed ordering on the neighbours of v. We will be considering a DAG whose vertices consist of strongly-connected components (SCCs) of the original graph G. We will have already pre-computed *several* DFS trees of *each* SCC: one rooted at *each* node in the SCC. Our final DFS tree will consist of (a) one DFS tree for each SCC (where the root of the DFS tree for SCC C is some node $r_C \in C$) along with (b) a selected edge (v_D, r_C) connecting any two SCCs D and C that are adjacent in the DFS tree of the DAG. But of course, to fully specify the DFS tree, we also need to have an ordering on the neighbours of each vertex. In practice, we will be using the (precomputed) DFS tree of D (rooted at r_D) to determine the order of neighbours of vertex D in the DAG (whose vertices are SCCs). The "lexicographically least" property of our DFS tree of the DAG depends only on the ordering of the neighbours (and not on the selection of the specific edge between vertices in the directed acyclic multigraph).

[2] In case a more detailed definition is necessary, here is what is meant by "the lexicographically-least path from s to v". Let p and p' be two paths from s to v. They each start with s and agree up through some vertex x, and first differ at the next vertex. Let us say that p has the edge (x, w) and p' has the edge (x, w') The vertex x is entered via some edge e (where, if $x = s$, then e is the null edge). The neighbours of x are ordered according to $f(x, e)$. If w precedes w' in the ordering $f(x, e)$, then p precedes p' in the lexicographic ordering.

[dlTK95].

The algorithm for computing the lexicographically-least DFS tree rooted at s can thus be presented as the composition of two functions g and h, where $g(G, s) = (G, s, L)$, where L is a list, containing the lexicographically-least path from s to each vertex v. Note that the set of edges in the DFS tree in G rooted at s is exactly the set of edges that occur in the list L in $g(G, s) = (G, s, L)$. Then $h(G, s, L)$ is just the result of removing from G each edge that does not appear in L. The function h is computable in logspace, whereas g is computable in logspace with an oracle for reachability in G.

As discussed in Section 4.1, a DFS tree is not only a list of edges; one must also know the order in which to explore the children of a node. Given a node v with children x and y, in order to determine whether x should be visited prior to y, one can simply compute the lexicographically-least path from s to x and from s to y, and compare.

Since reachability in DAGs is a canonical complete problem for NL, we obtain the following corollary:

Corollary 4.5. *Construction of lexicographically-least DFS trees for DAGs lies in* NL.

Similarly, since reachability in planar directed (not-necessarily acyclic) graphs lies in UL$\cap$ co-UL [BTV09, TV12], we obtain:

Corollary 4.6. *Construction of lexicographically-least DFS trees for planar DAGs lies in* UL $\cap$ co-UL.

A planar DAG G is said to be an SMPD if it contains at most one vertex of indegree zero. Reachability in SMPDs is known to lie in L [ABC$^+$09].

Corollary 4.7. *Construction of lexicographically-least DFS trees for SMPDs lies in* L.

4.3 Overview of the Algorithm

The main algorithmic insight that led us to the algorithm in this paper was a generalization of the layering algorithm that Hagerup developed for *undirected* graphs [Hag90]. We show that this approach can be modified to yield a useful decomposition of *directed* graphs, where the layers of the graph have a restricted structure that can be exploited. More specifically, the strongly-connected components of each layer are what we call *meshes*; we exploit the properties of meshes to construct paths (which will end up being paths in the DFS trees we construct) whose removal partitions the graph into significantly smaller strongly-connected components.

The high-level structure of the algorithm is thus:

1. Construct a planar embedding of G.
2. Partition the graph G into layers (each of which is surrounded by a directed cycle).
3. Identify one such cycle C that has properties that will allow us to partition the graph into smaller weakly-connected components.
4. Depending on which properties C satisfies, create a path p from the exterior face either to a vertex on C or to one of the meshes that reside in the layer just inside C. Removal of p partitions G into weakly-connected components, where each strongly-connected component therein is smaller than G by a constant factor.

5. Let the vertices on this path p be $v_1, v_2, \ldots, v_k$. The DFS tree will start with the path p, and append DFS trees for subgraphs $G_1, G_2, \ldots, G_k$ to this path, where G_i consists of all of the vertices that are reachable from v_i that are not reachable from v_j for any $j > i$. (This is obviously a tree, and it will follow that it is a DFS tree.) Further, decompose each G_i into a DAG of strongly-connected components. Build a DFS tree of that DAG, and then work on building DFS trees of the remaining (smaller) strongly-connected components.

6. Each of the steps above can be accomplished in UL ∩ co-UL, which means that there is an AC^0 circuit with oracle gates from UL ∩ co-UL that takes G as input and produces the list of much smaller graphs $G_1, \ldots, G_k$, as well as the path p that forms the spine of the DFS tree. We now recursively apply this procedure (in parallel) to each of these smaller graphs. The construction is complete after $O(\log n)$ phases, yielding the desired $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ circuit family.

In the exposition below, we first layer the graph in terms of clockwise cycles (which we will henceforth call red cycles), and obtain a decomposition of the original graph into smaller pieces. We then apply a nested layering in terms of counterclockwise cycles (which we will henceforth call blue cycles); ultimately we decompose the graph into units that are structured as a DAG, which we can then process using the tools from Section 4.2. The more detailed presentation follows.

4.3.1 Degree Reduction and Expansion

Definition 4.8. (of $\mathsf{Exp}^{\circlearrowright}(G)$ and $\mathsf{Exp}^{\circlearrowleft}(G)$) Let G be a plane digraph. The "expanded" digraph $\mathsf{Exp}^{\circlearrowright}(G)$ (respectively, $\mathsf{Exp}^{\circlearrowleft}(G)$) is formed by replacing each vertex v of total degree $d(v) > 3$ by a clockwise (respectively, counterclockwise) cycle C_v on $d(v)$ vertices, where the $d(v)$ edges incident on v now connect to the $d(v)$ vertices on C_v (so that each of those vertices now has degree 3), respecting the cyclic ordering of edges around v.

We will also find it useful to refer to the process of converting $\mathsf{Exp}^{\circlearrowright}(G)$ (or $\mathsf{Exp}^{\circlearrowleft}(G)$) back to G, by *contracting* each expanded cycle C_v back to v.

$\mathsf{Exp}^{\circlearrowright}(G)$ and $\mathsf{Exp}^{\circlearrowleft}(G)$ each have maximum degree bounded by 3; i.e., they are *subcubic*. Next we define the clockwise (and counterclockwise) dual for such a graph and also a notion of distance.

Recall that for an undirected plane graph H, the dual (multigraph) H^* is formed by placing, for every edge $e \in E(H)$, a dual edge e^* between the face(s) on either side of e (see Section 4.6 from [Die16] for more details). Faces f of H and the vertices f^* of H^* correspond to each other as do vertices v of H and faces v^* of H^*. There is also a well-studied notion of duality for *directed* plane graphs. The graph that is called the *dual of a directed graph* in sources such as [BETT98, KN11, BK09, Kao93] corresponds to the edges of weight one in what we define below as the *clockwise dual of* G; for technical reasons we also include additional edges (in the reverse direction) of weight zero, and we also make use of a *counterclockwise dual*:

Definition 4.9. (of Duals $G^{\circlearrowright}$ and $G^{\circlearrowleft}$) Let G be a plane digraph. Then the clockwise dual $G^{\circlearrowright}$ (respectively, counterclockwise dual $G^{\circlearrowleft}$) is a weighted bidirected version of the undirected dual of the underlying undirected graph of G. If e is an edge in G with faces f and g to the left and right, respectively (in the direction of travel on e), then there is an edge with weight one in $G^{\circlearrowright}$ that is oriented from f^* to g^* (thus corresponding to rotating e 90 degrees in a clockwise direction). The edge in the other direction, from g^* to f^* receives weight zero. (The weights in $G^{\circlearrowleft}$ are the opposite, with the weight one edge resulting from a counterclockwise rotation, and the other direction having weight zero.). We inherit the definition of dual vertices and faces from the underlying undirected dual.

Definition 4.10. Let G be a plane subcubic graph, and let f and g be faces of G. Define $d^{\circlearrowright}(f,g)$ to be the weight of the minimal-weight path from f^* to g^* in $G^{\circlearrowright}$. We define $d^{\circlearrowleft}(f,g)$ similarly.

Definition 4.11. For a plane subcubic digraph G, let f_0 be the external face. Define the type $\mathsf{type}^{\circlearrowright}(f)$ (respectively, $\mathsf{type}^{\circlearrowleft}(f)$) of a face to be the singleton set $\{d^{\circlearrowright}(f_0,f)\}$ (respectively, $\{d^{\circlearrowleft}(f_0,f)\}$). Generalise this to edges e by defining $\mathsf{type}^{\circlearrowright}(e)$ (respectively $\mathsf{type}^{\circlearrowleft}(e)$) as the set consisting of the union of the $\mathsf{type}^{\circlearrowright}$ (respectively, $\mathsf{type}^{\circlearrowleft}$) of the two faces adjacent to e. Also, for a vertex v, define $\mathsf{type}^{\circlearrowright}(v)$ (respectively $\mathsf{type}^{\circlearrowleft}(v)$) to be the union, over all faces f incident on v, of $\mathsf{type}^{\circlearrowright}(f)$ (respectively, $\mathsf{type}^{\circlearrowleft}(f)$).

It is easy to see by definition of the duals that the types of adjacent faces can differ by at most one, and hence no vertex can be adjacent to faces with three distinct types. We formalise this in the lemma below:

Lemma 4.12. *In every subcubic graph G, the cardinality $|\mathsf{type}^{\circlearrowright}(x)|, |\mathsf{type}^{\circlearrowleft}(x)|$ where x is a face, edge or a vertex is at least one and at most 2 and in the latter case consists of consecutive non-negative integers.*

Further, if $v \in V(G)$ is such that $|\mathsf{type}^{\circlearrowright}(v)| = 2$, then there exist unique $u, w \in V(G)$, such that $(u,v),(v,w) \in E(G)$ and $|\mathsf{type}^{\circlearrowright}(u,v)| = |\mathsf{type}^{\circlearrowright}(v,w)| = 2$.

Proof. We first observe that if (f_1,f_2) is a dual edge with weight 1, then by the triangle inequality we have, $d^{\circlearrowright}(f_0,f_1) \le d^{\circlearrowright}(f_0,f_2) \le d^{\circlearrowright}(f_0,f_1)+1$. Now, since each vertex $v \in V(G)$ of a subcubic graph is incident on at most 3 faces the only case in which $|\mathsf{type}^{\circlearrowright}(v)| > 2$ corresponds to three distinct faces f_1,f_2,f_3 being incident on a vertex. But here the undirected dual edges form a triangle such that in the directed dual the edges with weight 1 are oriented either as a cycle or acyclically. In the former case by three applications of the above inequality, we get that $d^{\circlearrowright}(f_0,f_1) \le d^{\circlearrowright}(f_0,f_2) \le d^{\circlearrowright}(f_0,f_3) \le d^{\circlearrowright}(f_0,f_1)$, hence all 3 distances are the same. Therefore $|\mathsf{type}^{\circlearrowright}(v)| = 1$.

In the latter case, suppose the edges of weight 1 are $(f_1,f_2),(f_2,f_3),(f_1,f_3)$, then by the above inequality again we get: $d^{\circlearrowright}(f_0,f_1) \le d^{\circlearrowright}(f_0,f_2), d^{\circlearrowright}(f_0,f_3) \le d^{\circlearrowright}(f_0,f_1)+1$. Thus, both $d^{\circlearrowright}(f_0,f_2), d^{\circlearrowright}(f_0,f_3)$ are sandwiched between two consecutive values $d^{\circlearrowright}(f_0,f_1), d^{\circlearrowright}(f_0,f_1)+1$. Hence $d^{\circlearrowright}(f_0,f_1)$, $d^{\circlearrowright}(f_0,f_2)$, $d^{\circlearrowright}(f_0,f_3)$ must take at most two distinct values, and thus $|\mathsf{type}^{\circlearrowright}(v)| \le 2$. Moreover either $\mathsf{type}^{\circlearrowright}(f_1) \ne \mathsf{type}^{\circlearrowright}(f_2) = \mathsf{type}^{\circlearrowright}(f_3)$ or $\mathsf{type}^{\circlearrowright}(f_1) = \mathsf{type}^{\circlearrowright}(f_2) \ne \mathsf{type}^{\circlearrowright}(f_3)$. Let e_1,e_2,e_3 be such that, ${e_1}^{\circlearrowright} = (f_2,f_3), {e_2}^{\circlearrowright} = (f_1,f_3), {e_3}^{\circlearrowright} = (f_1,f_2)$. Then the two cases correspond to $|\mathsf{type}^{\circlearrowright}(e_1)| = |\mathsf{type}^{\circlearrowright}(e_2)| = 2, |\mathsf{type}^{\circlearrowright}(e_3)| = 1$ and to $|\mathsf{type}^{\circlearrowright}(e_1)| = 1, |\mathsf{type}^{\circlearrowright}(e_2)| = |\mathsf{type}^{\circlearrowright}(e_3)| = 2$, respectively. Noticing that e_1,e_3 are both incoming or both outgoing edges of v completes the proof for the clockwise case. The proof for the counterclockwise case is formally identical. $\square$

Definition 4.13. For a plane subcubic graph G as above, define $\mathsf{red}(G)$ to be a colored version of G, where vertices and edges with a type of cardinality two in $G^{\circlearrowright}$ are colored red, and all other vertices and edges are white. Similarly, define $\mathsf{blue}(G)$ to be the colored version of G, where vertices and edges with a type of cardinality two in $G^{\circlearrowleft}$ are colored blue, and all other vertices and edges are white.

We will see later how to apply both the duals in G to get red and blue layerings of a given input graph.

We enumerate some properties of $\mathsf{red}(G)$ and $\mathsf{blue}(G)$, where G is subcubic:

Lemma 4.14. *1. Red vertices and edges in* $\mathsf{red}(G)$ *form disjoint clockwise cycles.*

2. No clockwise cycle in $\mathsf{red}(G)$ *consists of only white edges (and hence white vertices).*

Similar properties hold for $\mathsf{blue}(G)$.

Proof. 1. Firstly, note that a red edge must have red endpoints, as they are adjacent to the same faces that the edge between them is adjacent to. It is immediate from Lemma 4.12 that if v is a red vertex, it has exactly one red incoming edge and one red outgoing edge, proving that they form disjoint cycles. Now consider a red cycle C. The type of each edge of C must be the same, since if there are two consecutive edges in C of different types, it would make the common vertex adjacent to at least three vertices of different types contradicting Lemma 4.12. This means that the distance in $G^{\circlearrowright}$ of each face bordering the "outside" of C from the external face is one less than the distance of each face bordering the "inside" of C. But in any *counterclockwise* cycle, the distance in $G^{\circlearrowright}$ from the external face to both sides of C are the same (by the way distances are defined in $G^{\circlearrowright}$). Thus C is clockwise.

2. Suppose C is a clockwise cycle. Consider the shortest path in $G^{\circlearrowright}$ from the external face to a face enclosed by C. From the Jordan curve theorem (Theorem 4.1.1 [Die16]), it must cross the cycle C. The edge dual to the crossing must be red.

□

The definitions above, which apply only to subcubic plane graphs, can now be extended to a general plane graph G, by considering the subcubic graphs $\mathsf{Exp}^{\circlearrowright}(G)$ (and $\mathsf{Exp}^{\circlearrowleft}(G)$). But first, we must make a simple observation about $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$ (respectively about $\mathsf{blue}(\mathsf{Exp}^{\circlearrowleft}(G))$).

Lemma 4.15. *Let $v \in V(G)$ be a vertex of degree more than 3. Let C_v be the corresponding expanded cycle in* $\mathsf{Exp}^{\circlearrowright}(G)$. *Suppose at least one edge of C_v is white in* $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$. *Then there is a unique red cycle C that shares edges with C_v.*

Proof. First we note that C_v does not contain anything inside it since it is an expanded cycle. By Lemma 4.14 we know that C_v has at least one red edge. Suppose it shares one or more edges with a red cycle R_1. Since both cycles are clockwise and C_v has nothing inside, the cycle R_1 must enclose C_v. Now suppose there is another red cycle R_2 that shares one or more edges with C_v. Then R_2 must also enclose C_v. But two cycles cannot enclose a cycle whilst sharing edges with it without touching each other, which contradicts the above lemma that all red cycles in a subcubic graph are vertex disjoint. □

The last two lemmas allow us to consistently contract the red cycles in $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$, in order to obtain a colored version of G which we call $\mathsf{Col}^{\circlearrowright}(G)$. We make this more precise in the following:

Definition 4.16. The colored graph $\mathsf{Col}^{\circlearrowright}(G)$ (respectively, $\mathsf{Col}^{\circlearrowleft}(G)$) is obtained by labeling a vertex $v \in V(G)$ having degree more than 3 as red iff the cycle C_v in $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$ has at least one red edge and at least one white edge. Otherwise the color of v is white[3]. All the edges of G, and all of the vertices of G having degree ≤ 3 inherit their colors from $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$. The coloring of $\mathsf{Col}^{\circlearrowright}(G)$ is similar.

We can now characterize the colorings in the graph $\mathsf{Col}^{\circlearrowright}(G)$:

Lemma 4.17. *The following hold:*

1. *A red cycle in $\mathsf{Col}^{\circlearrowright}(G)$ is not connected via a red edge to any vertex in its interior.*[4]
2. *Every 2-connected component of the red subgraph of $\mathsf{Col}^{\circlearrowright}(G)$ is a simple clockwise cycle.*

Proof. Both parts of the lemma follow, if we can establish that the red subgraph of $\mathsf{Col}^{\circlearrowright}(G)$ consists of a collection of connected components, each of which is a remnant of exactly one red cycle in $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$, where furthermore, each connected component consists of a collection of red cycles that intersect at cut vertices (as illustrated in Figure 4.5). Recall that $\mathsf{Col}^{\circlearrowright}(G)$ results from taking the subcubic graph $\mathsf{Exp}^{\circlearrowright}(G)$ and contracting each cycle C_v where v is a vertex in G of degree > 3. The red subgraph of $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$ consists of disjoint cycles, by Lemma 4.14. Contracting any cycle C_v in $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$ does not increase the number of red connected components. Thus each red connected component of $\mathsf{Col}^{\circlearrowright}(G)$ is a remnant of exactly one red cycle in $\mathsf{Exp}^{\circlearrowright}(G)$. If C_v contains all red vertices, then v is white in $\mathsf{Col}^{\circlearrowright}(G)$ and thus is not part of any red subgraph. If C_v contains both red and white vertices, then C_v consists of alternating red subpaths and white subpaths, and by Lemma 4.15 the red subpaths are all part of the same cycle; let us call it R. On contracting C_v, R is transformed into a collection of clockwise red cycles (let's call them $R_1, R_2, \ldots$) sharing a common cut-vertex v. Furthermore, for any other C_x that contains edges from R, after C_v is contracted, C_x now shares edges with exactly one of the cycles R_i. (This is because C_x is embedded inside R. If it is possible to start at a vertex in $C_v \cap R$, and travel to C_x and then back to C_v, it follows that C_x is embedded in the closed region between R and C_v. When C_v is contracted, that segment of R becomes one of the cycles R_i, and C_x is embedded inside it.) Thus, when C_x is contracted, R_i in turn is transformed into a collection of cycles with x as a cut vertex. Inductively, this establishes the claim that, in turn, completes the proof of the lemma.

□

Although the above lemmas have been proved for the clockwise dual, they also hold for counterclockwise dual with red replaced by blue.

4.3.2 Layering the Colored Graphs

Definition 4.18. Let $x \in V(\mathsf{Col}^{\circlearrowright}(G)) \cup E(\mathsf{Col}^{\circlearrowright}(G))$. Let $\ell^{\circlearrowright}(x)$ be one more than the minimum integer that occurs in $\mathsf{type}^{\circlearrowright}(x')$, for each $x' \in V(\mathsf{Exp}^{\circlearrowright}(G)) \cup E(\mathsf{Exp}^{\circlearrowright}(G))$ that is contracted to x. Further let $\mathcal{L}^k(\mathsf{Col}^{\circlearrowright}(G)) = \{x \in V(\mathsf{Col}^{\circlearrowright}(G)) \cup E(\mathsf{Col}^{\circlearrowright}(G)) : \ell^{\circlearrowright}(x) = k\}$. Similarly, define $\ell^{\circlearrowleft}(x)$ and $\mathcal{L}^k(\mathsf{Col}^{\circlearrowleft}(G))$. We call $\mathcal{L}^k(\mathsf{Col}^{\circlearrowright}(G))$ the k^{th} layer of the graph.

See Figure 4.6 for an example. It is easy to see the following from Lemma 4.17:

[3]This may seem counterintuitive. If C_v is not entirely red, then v participates in some red cycle containing edges not in C_v. Whereas if C_v is all red, then v is not connected to other red parts of G, and thus we color it white.

[4]The *interior* of a cycle is the subgraph of G induced on the vertices that are embedded inside C, but not on C.

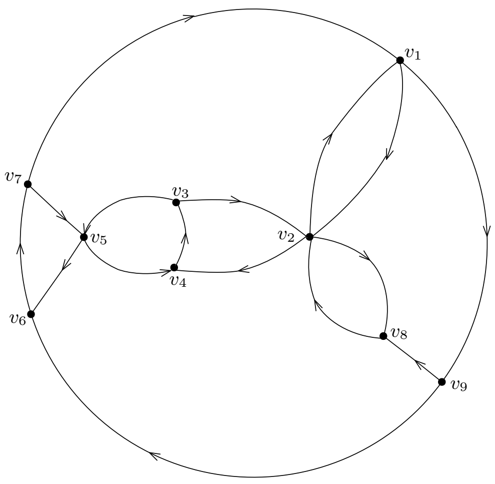


FIGURE 4.1: An example of a directed graph G.

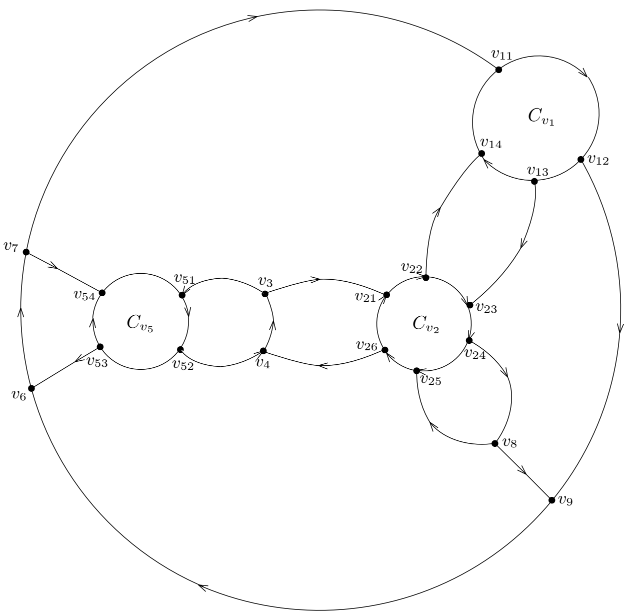


FIGURE 4.2: The graph $\mathsf{Exp}^{\circlearrowright}(G)$.

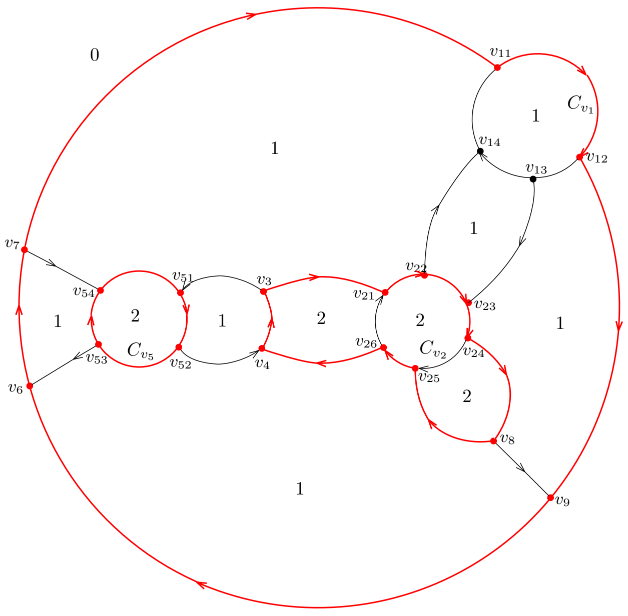


FIGURE 4.3: The graph $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$, along with types of the faces.

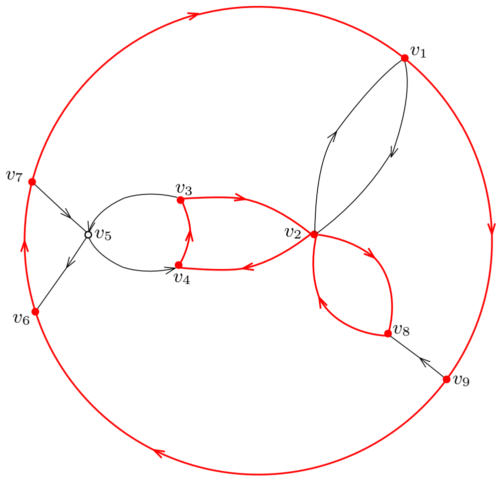


FIGURE 4.4: The graph $\mathsf{Col}^{\circlearrowright}(G)$. Notice that vertex v_5 was expanded into a red cycle, C_{v_5}, but is a white vertex in $\mathsf{Col}^{\circlearrowright}(G)$ because all of its edges were red in $\mathsf{red}(\mathsf{Exp}^{\circlearrowright}(G))$.

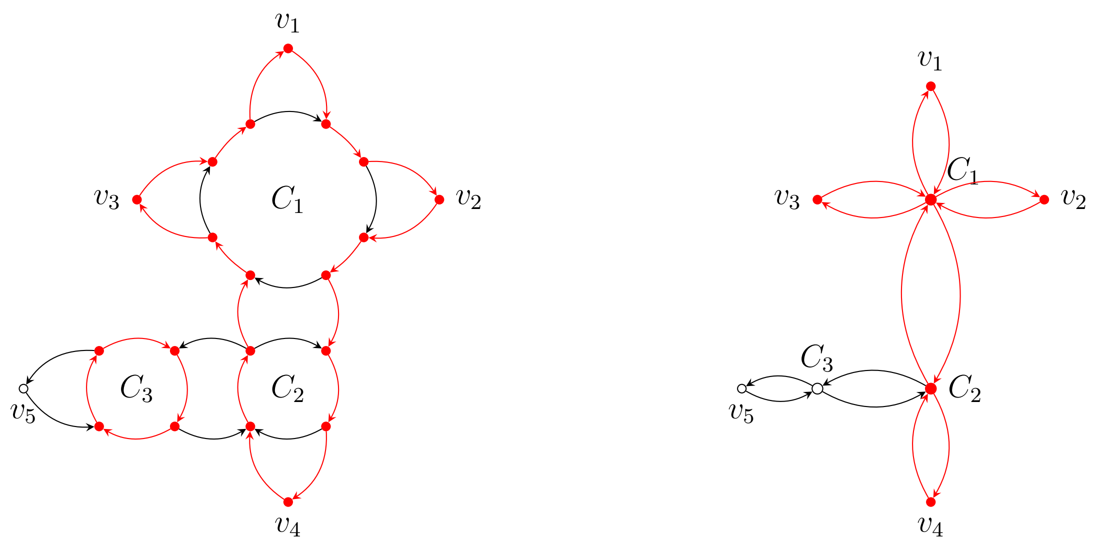


FIGURE 4.5: An example of contracting expanded cycles. The figure on right shows the graph G after contracting the expanded cycles C_1, C_2, C_3 in $\mathsf{Exp}^{\circlearrowright}(G)$.

Proposition 4.19. *For every $x \in V(\mathsf{Col}^{\circlearrowright}(G)) \cup E(\mathsf{Col}^{\circlearrowright}(G))$ the quantity $\ell^{\circlearrowright}(x)$ is one more than the number of red cycles that strictly enclose x in $\mathsf{Col}^{\circlearrowright}(G)$. All the vertices and edges of a red cycle of $\mathsf{Col}^{\circlearrowright}(G)$ lie in the same layer $\mathcal{L}^{k+1}(\mathsf{Col}^{\circlearrowright}(G))$ for the enclosure depth k of the cycle.*

We had already noted above that the red subgraph of G had simple clockwise cycles as its 2-connected components. We note a few more lemmas about the structure of a layer of G:

Lemma 4.20. *We have:*

1. *A red cycle in a layer $\mathcal{L}^{k+1}(\mathsf{Col}^{\circlearrowright}(G))$ does not contain any vertex/edge of the same layer inside it.*

2. *Any clockwise cycle in a layer consists of only red vertices and edges.*

Dually, a blue cycle in a layer does not contain any vertex or edge of the same layer inside it.

Remark 4.20.1. Notice that the conclusion in the second part of the lemma fails to hold if we allow cycles spanning more than one layer.

Proof. The first part is a direct consequence of Proposition 4.19. For the second part we mimic the proof of the second part of Lemma 4.14. Consider a clockwise cycle $C \subseteq \mathcal{L}^{k+1}(\mathsf{Col}^{\circlearrowright}(G))$ that contains a white edge e. Every face adjacent to C from the outside must have type$^{\circlearrowright} = \{k\}$ because C is contained in layer $k+1$. Then the type$^{\circlearrowright}$ of the faces on either side of e is the same and therefore must be $\{k\}$. Let f be a face enclosed by C that has type$^{\circlearrowright}(f) = \{k\}$. Thus it must be adjacent to a face of type$^{\circlearrowright} = \{k-1\}$. But this contradicts that every face inside and adjacent to C must have type$^{\circlearrowright} = \{k'\}$ for $k' \geq k$. $\square$

The lemmas above show that the strongly-connected components of the red subgraph of a layer consist of red cycles touching each other without nesting, in a tree like structure. This prompts the following definition:

Definition 4.21. For a red cycle $R \subseteq \mathcal{L}^{k}(\mathsf{Col}^{\circlearrowright}(G))$ we denote by G_R, the graph induced by vertices of $\mathcal{L}^{k+1}(\mathsf{Col}^{\circlearrowright}(G))$ enclosed by R.

Now we combine Definitions 4.16 and 4.18:

Definition 4.22. Each vertex or edge $x \in V(G) \cup E(G)$ gets a red layer number $k+1$ if it belongs to $\mathcal{L}^{k+1}(\mathsf{Col}^{\circlearrowright}(G))$ and a blue layer number $l+1$, if it belongs to $\mathcal{L}^{l+1}(\mathsf{Col}^{\circlearrowleft}(G_R))$ where $R \subseteq \mathcal{L}^{k}(\mathsf{Col}^{\circlearrowright}(G))$ is the red cycle immediately enclosing x. In this case, we say that x belongs to sublayer $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$.

Moreover this defines the colored graph $\mathsf{Col}(G)$ by giving x the color red if it is red in $\mathsf{Col}^{\circlearrowright}(G)$, and also giving x the color blue if it is blue in $\mathsf{Col}^{\circlearrowleft}(G_R)$. (Notice it could be *both* red and blue). The vertex x is white if it is white in both $\mathsf{Col}^{\circlearrowright}(G_R)$ and $\mathsf{Col}^{\circlearrowleft}(G_R)$.

By Proposition 4.19, we can also say that a sublayer $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$ thus consists of edges/vertices that are strictly enclosed inside k red cycles and inside l blue cycles that are contained *inside* the red cycle that immediately encloses them.

We now present some observations and lemmas regarding the structure of a sublayer.

Since every edge/vertex in $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$ has the same red *and* blue layer number, it is clear that there can be no nesting of colored cycles. Also we have:

Lemma 4.23. *Every clockwise cycle in a sublayer $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$ consists of all red edges and vertices and any every counterclockwise cycle in the sublayer consists of all blue vertices and edges. (Some edges/vertices of the cycle can be both red as well as blue)*

Proof. This is a direct consequence of Lemma 4.20 applied to the sublayer $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$, which is a (counterclockwise) layer in graph G_R for some red cycle R. □

Thus we can refer to clockwise cycles and counterclockwise cycles as red and blue cycles respectively.

Definition 4.24. For a red or blue colored cycle C of layer $\mathcal{L}^{k,l}(\mathsf{Col}(G))$, we denote by G_C the graph induced by vertices of $\mathcal{L}^{k',l'}(\mathsf{Col}(G))$ enclosed by C, where $\{k', l'\}$ is $\{k+1, 1\}$ or $\{k, l+1\}$ according to whether C is a red or a blue cycle respectively.

Note that:

Proposition 4.25. *Two cycles of the same color in $\mathcal{L}^{k+1,l+1}(G)$ cannot share an edge.*

This is since neither is enclosed by the other as they belong to the same layer, and as they also have the same orientation. Cycles of different colors can share edges but we note:

Lemma 4.26. *Two cycles of a sublayer $\mathcal{L}^{k+1,l+1}(\mathsf{Col}(G))$ can only share one contiguous segment of edges.*

Proof. Let a red cycle R and a blue cycle B in a sublayer share two different contiguous segments of edges, from x to u and from v to y, where the the path $R(u,v)$ in R and the path $B(u,v)$ in B share no edges. Notice that the graph $(R \setminus R(u,v)) \cup B(u,v)$ is also a clockwise cycle that encloses the edges of $R(u,v)$, contradicting the first part of Lemma 4.20. □

We consider the strongly-connected components of a sublayer and note the following lemmas regarding them:

Lemma 4.27. *The trivial strongly-connected components of a sublayer (those that consist of a single vertex) are white vertices. Let H be a non-trivial strongly-connected component of a sublayer, and let o be the external face of H. Then*

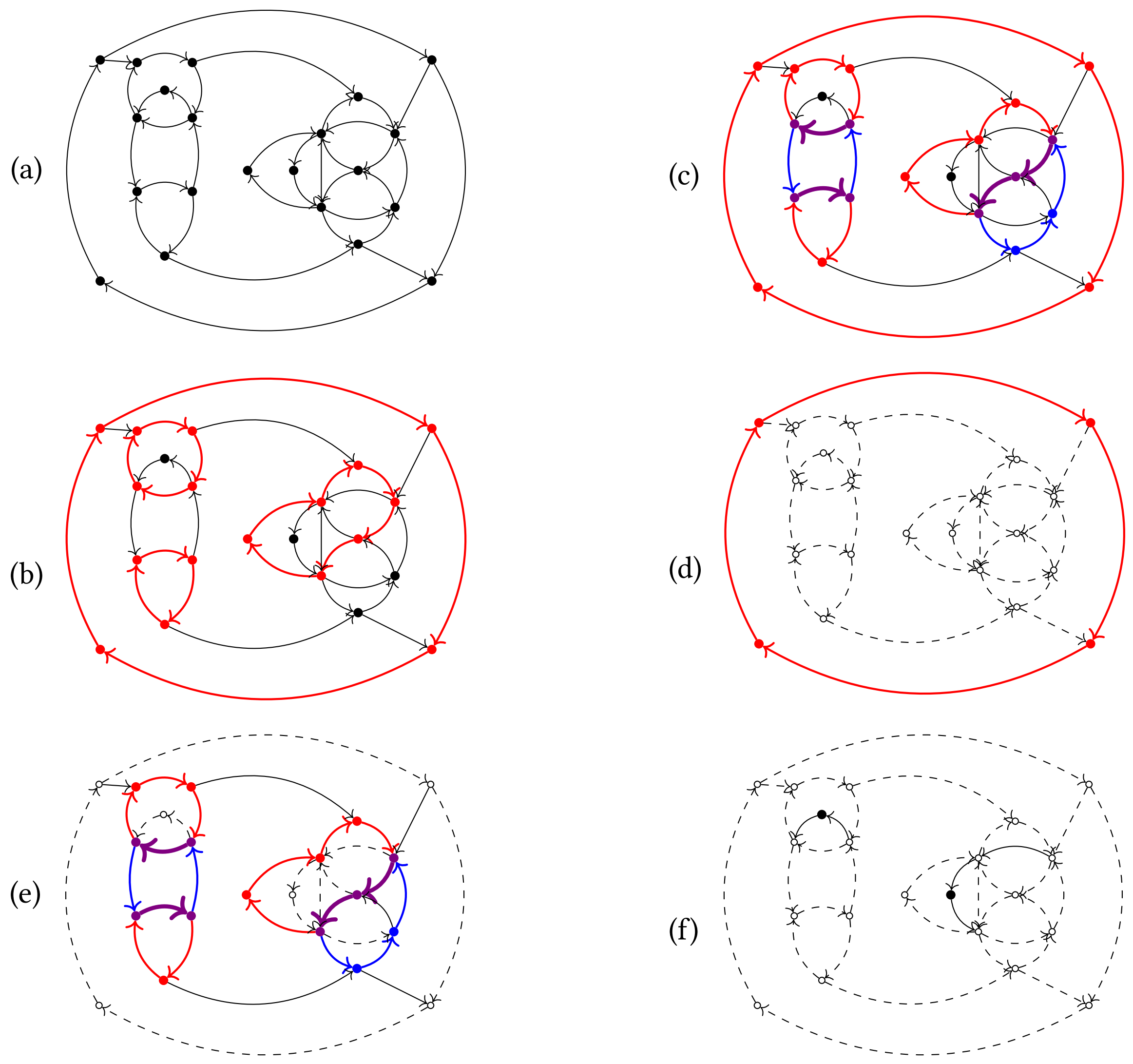


FIGURE 4.6: Figure (a) is a graph G. Figure (b) is the graph $\mathsf{Col}^{\circlearrowright}(G)$. We omit the cycle expansion and contraction procedure here. Figure (c) shows $\mathsf{Col}(G)$, which we get from G after applying blue labellings to each red layer we obtained in the previous figure. The vertices and edges colored purple are those that are red as well as blue. Figure (d) represents the sublayer $\mathcal{L}^{1,1}$. The dashed edges and empty vertices are not part of the layer. Figure (e) represents the sublayer $\mathcal{L}^{2,1}$. Figure (f) represents the sublayer $\mathcal{L}^{3,1}$.

1. *Every vertex/edge in H is blue or red (possibly both).*
2. *The boundary of every face of H, except possibly o, is a directed cycle.*
3. *Every face of H other than o has at least one edge adjacent to o.*

Proof. 1. In a non-trivial strongly-connected graph every vertex and edge lies on a cycle and therefore by Lemma 4.23 must be colored red or blue (or both).

2. Suppose there is a face f the boundary of which is not a directed cycle. Look at a directed dual (say clockwise) of H. This dual must be a DAG since the primal is strongly-connected. The vertex f^* in the dual corresponding to face f of H has in-degree at least one and

out-degree at least one since it has boundary edges of both orientations, hence the edges adjacent to f^* do not form a directed cut of the dual.

Let o^* denote the dual vertex corresponding to the outer face o of H. In order to prove the claim, it is sufficient to show the existence of a directed cut C^* that separates f^* and o^*, since it would imply by cut cycle duality that there is a directed cycle C in H that encloses the face f w.r.t the outer face. Since the boundary of f is not a directed cycle, this means that C must strictly enclose at least one edge of the boundary of f, contradicting Lemma 4.20. To see that the cut exists, consider a topological sort ordering of the dual (it is a DAG). Let the number of a dual vertex v^* in the ordering be denoted by $n(v^*)$. W.l.o.g, let $n(f^*) < n(o^*)$. Consider the partition of the dual vertices:

$$A = \{v^* \mid n(v^*) \leq n(f^*)\}, B = \{v^* \mid n(v^*) > n(f^*)\}$$

By definition of topological sort, all edges across this partition must be directed from A to B, hence it is a directed cut, and therefore it must also contain a subset which is a minimal directed cut. But clearly the minimal cut is not the set of edges adjacent to f^* since it has both out and in-degree at least one, hence proving the claim. Hence every face in H must be a directed (hence colored) cycle (by Lemma 4.23).

3. We observed from the proof above that no vertex in the dual of H, except possibly the vertex o^* corresponding to the outer face of H, can have both in-degree and out-degree more than zero (i.e. every dual vertex except o^* is a source or a sink). Therefore if any dual vertex f^* has a directed path to o^* or vice versa, then the path must be an edge and we are done. Suppose there is no directed path from f^* to o^* and w.l.o.g. let f^* be a source. Consider the trivial directed cut C_1:

$$A = \{f^*\}, B = V(H) \backslash A$$

This is a cut since there are no edges from B to A, and this cut clearly corresponds to the directed cycle which is the boundary of face f in H.

Now consider the cut C_2:

$$A' = \{v^* \mid v^* \text{ is reachable from } f^*\}, B' = V(H) \backslash A'$$

Clearly this is a f^*-o^* cut with no edge from a vertex in A' to a vertex in B' and $o^* \in B'$. But this f^*-o^* cut is different from C_1 since f^* is a source vertex and hence A' has at least one more vertex than just f^*. Hence this corresponds to a directed cycle in H that strictly encloses at least some edge of f, and we again get a contradiction of Lemma 4.20.

□

The strongly-connected components of a sublayer hence consist of intersecting red and blue facial cycles, with every face having at least one boundary edge adjacent to the outer face of the component.

Definition 4.28. We call the strongly-connected components of a sublayer $\mathcal{L}^{k,l}$ **meshes**.

4.4 Properties of the Mesh

Definition 4.29. Given a subgraph H of G embedded in the plane, we define the *closure* of H, denoted by $\widetilde{\mathsf{H}}$, to be the induced graph on the vertices of H together with the vertices of G that lie in the interior of faces of H (except for the outer face of H).

For convenience, we call a face of a graph that is not the outer face an *internal face*.

From Lemmas 4.23 and 4.27, we have a bijection: every face of a mesh, except possibly its outer face, is a directed cycle, and every directed cycle in a mesh is the boundary of a face of the mesh.

Definition 4.30. Let $0 < \alpha < 1$. An α *separator* of a digraph H that is a subgraph of a digraph G is a set of vertices of H whose removal from H separates $\widetilde{\mathsf{H}}$ into subgraphs, where no strongly-connected component has size greater than $\alpha|G|$. An (α, r) *path separator* is a sequence of vertices $\langle v_1, \dots, v_n\rangle$, that is an α separator and also is a directed path. Here $r = v_1$ is called the root of the path separator. We will have occasion to omit either or both of α, r when they are clear from the context.

Definition 4.31. Let G be a graph and let M be a mesh in a sublayer of G. For an internal face f of M, we define its weight, denoted by $w(f)$, to be $|V(\widetilde{\mathsf{f}})|$. Let $w(H)$ where H is a subgraph of M be defined as $|V(\widetilde{\mathsf{H}})|$.

Definition 4.32. For a mesh M, we call a vertex that is adjacent to the outer face of M an *external vertex*, and a vertex that is not adjacent to the outer face an *internal vertex*. We call vertices of degree more than two *junction vertices*.

If $p = \langle v_1, v_2, \dots, v_k\rangle$ is a directed path, for $k \geq 1$, such that $v_2, \dots, v_{k-1}$ are all vertices of degree two, but v_1, v_k have degree more than two [5], then we call p a segment. We call v_k the *out junction neighbour* of v_1 and v_1 the *in junction neighbour* of v_k.

We call a segment with all edges adjacent to the outer face an *external* segment, and a segment with no edge adjacent to the outer face an *internal* segment. If the end points of an *internal* segment are both internal vertices also, we call the segment an i-i-segment.

The rest of this section is devoted to a proof of the following, which asserts that we can construct a path separator in a mesh, assuming that no internal face of the mesh is too large.

Lemma 4.33. *Suppose $w(f) < w(G)/12$ holds for every internal face f of a mesh M that is a subgraph of G. Then from any external vertex r of M, we can find (in* UL ∩ co-UL*) an $\frac{11}{12}$ path separator of M, starting at r.*

The high level idea is that using a clique sum decomposition of $2, 3$-cliques (see Figure 4.12) we find a "central" vertex v in the mesh M, such that we can find a path from the external vertex r to v, and then extend the path around one of the faces adjacent to v to get the path separator (all faces are directed cycles by Lemma 4.27). Because every face touches the outer face and the weight of every face is small by the hypothesis of the lemma, we can always find a face adjacent to v to encircle such that removing the path leaves no large (weakly) connected component. The vertices of M with degree two (in-degree 1 and out-degree 1 because M is strongly-connected) are not important since they can be seen as just "subdivision" vertices.

[5] Notice that here we explicitly allow $k = 1$ so that $v_1 = v_k$.

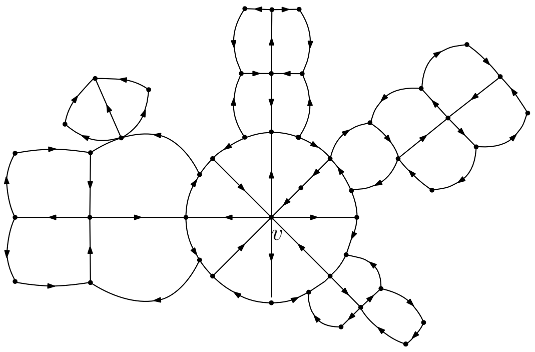


FIGURE 4.7: An example of a mesh

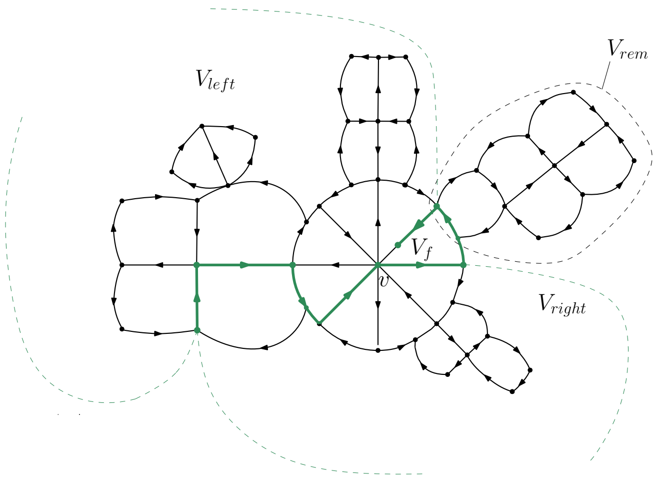


FIGURE 4.8: An example of a path separator. The vertex v is a central node, and the green path is a separator.

Now we will look at the structure of a mesh around an internal junction vertex, and the way the rest of the mesh is attached to that structure. Also, we state here that we will abuse the notion of 3-connected components by ignoring the non-junction vertices for convenience.

Lemma 4.34. *If v is an internal junction vertex of a mesh and $e_1, \dots, e_k$ are the edges adjacent to v in the cyclic order of embedding, then the edges alternate in directions i.e. if e_1 is outgoing from v, then e_2 is incoming to v and e_3 is outgoing and so on. Consequently, v has even degree (at least 4).*

Proof. Let e_i, e_{i+1} be two edges adjacent to v, that are also adjacent in the cyclic order of the drawing. Since they are adjacent in the drawing, they must enclose between them, a region, and hence a face, which is not the outer face. But, by Lemma 4.27, the boundary of every non-outer face in a mesh is a directed cycle, hence v, e_i, e_{i+1} lie on a directed cycle, with both edges adjacent to v. Hence one of them must be an out edge from v, and the other incident towards v. □

Definition 4.35. Let v be an internal junction vertex of degree $2d$ in a mesh M, and let its junction neighbours be $(u_1, w_1, u_2, w_2, \dots, u_d, w_d)$ in clockwise order starting from edge $\langle u_1, v\rangle$(the w_i's are out neighbours, and u_i's the in neighbours, since junction neighbours alternate).

Every adjacent pair of edges incident to v borders a face that is not the outer face. Let $f_{u,v,w}$ denote the face bordered by v and the junction neighbours u and w of v which are adjacent in cyclic order around v. The boundary of $f_{u,v,w}$ can be written as three disjoint parts (except for endpoints), $\mathsf{segment}(u, v)$ + $\mathsf{segment}(v, w)$ + $petal_{w,u}$, where the third part denotes a simple path from w to u along the face boundary. We will use the notation $petal_{w,u}$ to denote the corresponding boundary for any face $f_{u,v,w}$ adjacent to v. We define $\mathsf{flower}(v)$ as $\bigcup\{$vertices on the boundary of faces adjacent to $v\}$ (see Figure 4.9).

We note the following property of petals.

Proposition 4.36. *For all adjacent junction neighbour pairs w_i, u_j of an internal vertex v, $petal_{w_i,u_j}$ are disjoint, except possibly the end points.*

Proof. Petals of two faces must be internally disjoint because the corresponding faces share the vertex v and two faces cannot have a non-contiguous intersection, by Lemma 4.26. □

For an internal junction vertex v, the union of the petals around flower(v) thus form an undirected cycle around v, with at least four alternations in directions. Now we define bridges of the cycle, which (roughly) are components of M we get after removing flower(v), leaving the points of attachment intact. We use the formal definition of bridges from [Tut75]:

Definition 4.37. For a subgraph H of M, a *vertex of attachment* of H is a vertex of H that is incident with some edge of M not belonging to H. Let J be an undirected cycle of M. We define a bridge of J in M as a subgraph B of M with the following properties:

1. each vertex of attachment of B is a vertex of J.
2. B is not a subgraph of J.
3. no proper subgraph of B has both the above properties.

We denote by 2-bridge, bridges with exactly two vertices of attachment to the specified cycle, and by 3-bridge, bridges with three or more vertices of attachment.

Note that for the cycle formed by petals of flower(v), the vertex v along with paths leading to/ coming from flower(v) also form a bridge, but we call that a trivial bridge and do not take it into consideration.

Lemma 4.38. *1. The vertices of attachment of a* 2-bridge *of* flower(v) *must both lie on one* petal *of* flower(v).

2. *The vertices of attachment of a* 3-bridge *of* flower(P) *can lie on one, or at most two, adjacent petals. Moreover, in the latter case the junction neighbour of v common to both petals must be a vertex of attachment of the* 3-bridge.

3. *For an internal vertex v, and an external vertex r of M, let $p = \langle r, \ldots, u_1, v\rangle$ be a simple path from r to v, where u_1 is an in junction neighbour of v. Let the other junction neighbours of v be named as in Definition 4.35 in cyclic order from u_1. For $j \in \{i, i+1\}$, consider an extended path of p, $p_{w_i,u_j} = \langle r, \ldots, u_1, v, w_i\rangle + petal_{w_i,u_j} + \langle u_j, \ldots, v\rangle$, excluding the last edge incident to v in the sequence. That is, p_{w_i,u_j} goes from r to v, then to an out junction neighbour w_i, and then wraps around f_{u_j,v,w_i} by taking $petal_{w_i,u_j}$ and then the segment back towards v from u_j. If there is a bridge of* flower(v) *of which u_1 is a point of attachment and which also includes the edge of p incoming to u_1, we denote it by B_{in}. The set $V(\widetilde{\mathsf{M}}) \setminus V(p_{w_i,u_j})$ can be partitioned into four disconnected parts, called V_{left}, V_{right}, V_f, and V_{rem}, such that:*

$$\begin{aligned} V_{left} =& (\{\widetilde{\mathsf{f}}_{u_1,v,w_1} \cup \widetilde{\mathsf{f}}_{u_2,v,w_1} \cup \widetilde{\mathsf{f}}_{u_2,v,w_2} \ldots \cup \widetilde{\mathsf{f}}_{u_i,v,w_{i-1}}\} \cup \{\widetilde{\mathsf{f}}_{u_i,v,w_i}\, \textit{if } j = i+1\} \\ & \cup \{\textit{vertices in the closure of } \text{bridges } \textit{attached to the petals of these faces, excluding } B_{in}\} \\ & \cup \{\textit{the "left" part of } B_{in} \textit{ (see Figure 4.13) }\}) \setminus V(p_{w_i,u_j}) \end{aligned}$$

$$\begin{aligned} V_{right} =& (\{\widetilde{\mathsf{f}}_{u_i,v,w_{i+1}} \cup \widetilde{\mathsf{f}}_{u_{i+2},v,w_{i+1}} \ldots \cup \widetilde{\mathsf{f}}_{u_d,v,w_d}\} \cup \{\widetilde{\mathsf{f}}_{u_{i+1},v,w_i}\, \textit{if } j = i\} \\ & \cup \{\textit{vertices in the closure of } \text{bridges } \textit{attached to petals petals these faces, excluding } B_{in}\} \\ & \cup \{\textit{the "right" part of } B_{in} \textit{ (see Figure 4.13) }\} \setminus V(p_{w_i,u_j}) \end{aligned}$$

$$V_f = \widetilde{\mathsf{f}}_{u_j,v,w_i} \setminus V(p_{w_i,u_j})$$

$$V_{rem} = (\bigcup\{\textit{vertices in the closure of all } \text{bridges} \textit{ that have vertices}$$
$$\textit{of attachment only in } petal_{w_i,u_j}\}) \setminus V(p_{w_i,u_j}).$$

There is no undirected path between any vertex of one of these four sets to any vertex of another. The path p_{w_i,u_i} is therefore a path separator that gives these components.

Proof. 1. Let x, y be the two vertices of attachment of the 2-bridge B on flower(v). Since bridges are connected graphs without the edges of the corresponding cycle (by the third property of Definition 4.37), there must be an undirected path, p in the bridge connecting x, y, without using any edge of flower(v). If x and y were *not* on the same petal, then this path along with the other petals in flower(v), must clearly enclose a junction neighbour of v, say w (see Figure 4.9). Thus w is not adjacent to the outer face. Now since w is an internal junction vertex, and two of its adjacent faces are also adjacent to v, look at another face f adjacent to w and not adjacent to v. (Internal junction vertices have at least four adjacent faces.) The boundary of this face cannot touch B since that would make it a part of B and consequently w would be a vertex of attachment of B to flower(v). Therefore the boundary of f is enclosed within the paths p and the part of flower(v) that is also enclosed by p. Therefore f has no external edge, contradicting Lemma 4.27.

2. Let $x_1, x_2, \ldots, x_k$ be the vertices of attachment of the bridge B on flower(v), in the cyclic order of boundary of flower(v). Clearly if the vertices of attachment lie on more than two petals of v, then at least one petal will be completely enclosed by B, which is not possible since every petal must have at least one external edge. Let us say they lie on two adjacent petals, and the junction neighbour common to both of them is w. By the same argument as above, w must have an edge other than those of adjacent petals of v, that connect it to B. Therefore w must be a vertex of attachment of B to flower(v).

3. First we note that $petal_{w_i,u_j}$ will have an external vertex in it since the boundary of every face has at least one external vertex (Lemma 4.27), and segments (u_j, v) and (v, w_i) are internal. Let z be an external vertex on $petal_{w_i,u_j}$. The path p starts at external vertex r, comes to u_1, v, w_i, and reaches external vertex z on its way back to v. It will clearly divide $\widetilde{\mathsf{M}}$ into at least two parts by the Jordan Curve theorem. Since p_{w_i,u_j} is just a wrap around the face f_{u_j,v,w_i} after z, it is clear that removing p puts all of the vertices of $\widetilde{\mathsf{f}}_{u,v,w}$ in one disconnected region, while $w_1, u_2, \ldots, w_{i-1}$ and everything connected to them lie in another region, which we call V_{left}, and $w_{i+1}, u_{i+2}, \ldots, w_d$ and everything connected to them lie in yet another (V_{right}).

□

We introduce another notation for an extension of a bridge:

Definition 4.39. For a bridge B of flower(v), we define B° as B along with **segments** of flower(v) that lie between consecutive vertices of attachment of B. We call this the **closed** bridge of B.

Now we will give definitions/lemmas regarding the "internal structure" of meshes, that will be useful to define the "center" of a mesh.

Definition 4.40. For a mesh M, we call its internal-skeleton, denoted by $I(M)$, the induced subgraph on the vertices of i-i-segments of M. (See Figure 4.11.)

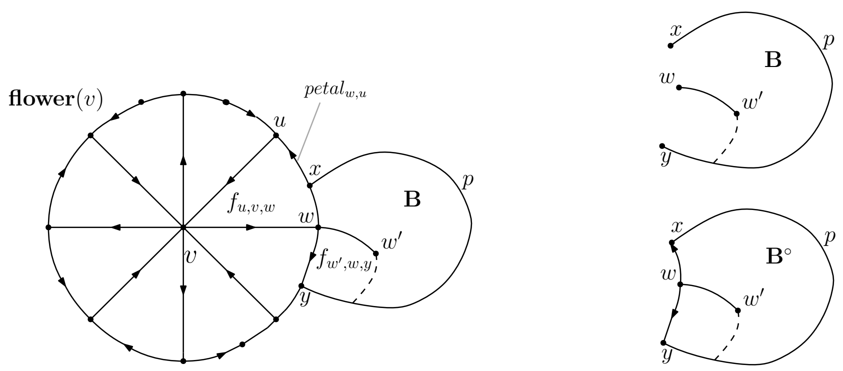


FIGURE 4.9: A vertex v and flower(v). B is a bridge with two points of attachment x, y on two different petals of flower(v). On the right are drawn the bridge B itself, and its closed version B°. The only way the boundary of $f_{w',w,y}$ can have an external edge is if it touches B, making w a point of attachment of B also.

Lemma 4.41. *1. For a* mesh M*, the graph* $I(M)$ *is a forest.*

2. *If H is a 3-connected induced subgraph of M(ignoring subdivision vertices), then $I(H)$ is a tree.*

Proof. 1. Suppose there were an undirected cycle in M of all internal segments, then this cycle must enclose a face whose boundaries are also all internal segments. This contradicts Lemma 4.27 as it states that every face must have at least one external edge, and hence an external segment. Hence there can be no cycle (directed or undirected) consisting of all internal segments, and consequently, no cycle (directed or undirected) of all internal vertices.

2. Let H be a 3-connected induced subgraph of M. By definition, $I(H)$ is obtained from M by removing all external edges and external non-junction vertices. Suppose $I(H)$ is not a tree, and hence consists of two or more disconnected trees. Let T_1 and T_2 be any two trees in $I(H)$. Let x be a vertex in T_1 and y be a vertex in T_2. Since H is 3-connected, there must be at least three disjoint paths (undirected) between x and y. Clearly in a plane graph, if there are three disjoint paths between two vertices, one of the paths must be strictly enclosed in the closed region formed by other two. Therefore there must a path between x and y that is strictly enclosed inside the boundary of H, and hence does not contain any edge or vertex adjacent to the outer face of H. Hence x and y cannot become disconnected after removing external edges and external non-junction vertices leading to a contradiction that $I(H)$ is disconnected. Therefore $I(H)$ must be a tree.

□

We will next give a procedure to define a "center" of a mesh.

Definition 4.42. For a mesh M, let T_M denote the tree obtained by the $1,2$-clique sum decomposition of M. The nodes of T_M are of two types, clique nodes (cut vertices or separating pairs), and piece nodes, which are either 3-connected parts or cycles. Every piece node is adjacent to a clique node and vice-versa. (Refer to Section 2.2.4 for background about this decomposition. For general planar graphs, we can first identify the cut vertices and find the block cut tree. For every clique node of a cut vertex v that is attached to a piece node of block

B containing a 3-connected separating pair, we replace the block B by its triconnected decomposition tree, T_B, and attach the clique node of v to a piece node of the triconnected block of T_B that contains v).

Proposition 4.43. *The tree T_M defined above is a tree decomposition.*

Proof. It is easy to see that every vertex, as well as every edge of the graph occurs in at least one piece node. To see the coherence property, we observe that the only vertices that occur in more than one node are those that are part of a 3-connected separating pair or a cut vertex. If v is such a node and is not a cut vertex then it occurs in a subtree of the biconnected block it belonged to after the block cut decomposition(since the triconnected decomposition is a tree decomposition). If it is a cut vertex, then in our construction, we have joined the subtrees in the triconnected decomposition trees to the clique node of v, which again gives a subtree. □

We will now use a modified version of the tree vertex separator theorem, to show that vertices of one of the nodes of T_M form a $\frac{1}{2}$-separator of M. We use the following fact from the proof of [CFK+15, Lemma 7.19].

Proposition 4.44. *Let T_G be a tree decomposition of a graph G. The vertices of one of the bags of T_G from a $\frac{1}{2}$ separator of G.*

Now we define the center of a mesh.

Definition 4.45. Consider the $\frac{1}{2}$ separator node of T_M as described in Proposition 4.44. If it is a separating pair, a cut vertex, or a face cycle, we call that subgraph the center of M.

If it is a 3-connected node P, look at its internal skeleton $I(P)$. We construct a new graph $I'(P)$ which is isomorphic to $I(P)$ but has edges directed differently. Let u, v be two adjacent internal junction vertices of M. To give direction to a segment(u, v) in $I'(P)$, we consider the unique bridge B of flower(u) that contains v as a point of attachment; we denote the **closed** bridge of B by $\mathsf{B}^\circ_u(v)$. $\mathsf{B}^\circ_v(u)$ is defined analogously. We orient (u, v) in the direction of the heavier of $\mathsf{B}^\circ_u(v)$ and $\mathsf{B}^\circ_v(u)$ (breaking ties arbitrarily), where the weights of $\mathsf{B}^\circ_u(v)$, $\mathsf{B}^\circ_v(u)$ are $|\wideparen{\mathsf{B}^\circ_u(v)}|$ and $|\wideparen{\mathsf{B}^\circ_v(u)}|$, respectively.

The center of M is defined to be flower(v) in this case, where v is the sink node of $I'(P)$.

We show why $I'(P)$ cannot have more than one sink.

Lemma 4.46. *The tree $I'(P)$ defined above will have exactly one sink vertex.*

Notice, while the underlying undirected graph of $I'(P)$ is a tree, a sink is defined with respect to the orientations specified in the previous definition.

Proof. Suppose $I'(P)$ has two junction vertices x and y that are sinks. They cannot be adjacent, so consider the unique undirected path in $I'(P)$ between x and y. There must be a source z on the path. Let neighbours of z be x', y', lying on the path from x to z and from z to y respectively.

Let $\mathsf{B}^\circ_z(x')$ and $\mathsf{B}^\circ_z(y')$ denote the bridges of flower(z) with points of attachments x' and y' respectively. Then by the orientations of the edges we have: $|\wideparen{\mathsf{B}^\circ_z(x')}| \geq |\wideparen{\mathsf{B}^\circ_{x'}(z)}|$ which gives $|\wideparen{\mathsf{B}^\circ_z(x')}| > |\wideparen{\mathsf{B}^\circ_z(y')}|$ since $\mathsf{B}^\circ_z(y')$ is clearly a proper subgraph of $\mathsf{B}^\circ_{x'}(z)$ and $|\wideparen{\mathsf{B}^\circ_z(y')}| \geq |\wideparen{\mathsf{B}^\circ_{y'}(z)}|$ which gives $|\wideparen{\mathsf{B}^\circ_z(y')}| > |\wideparen{\mathsf{B}^\circ_z(x')}|$ which is clearly a contradiction. □

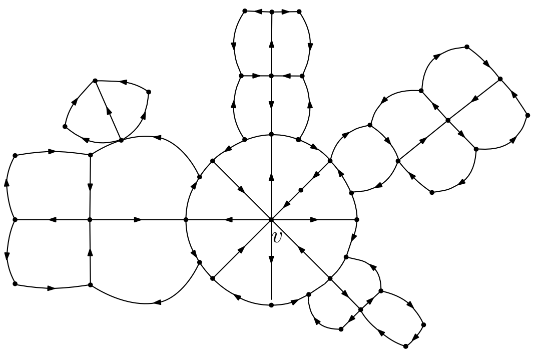


FIGURE 4.10: An example of a mesh

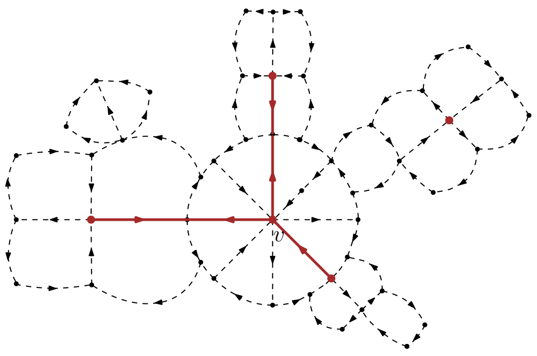


FIGURE 4.11: The internal skeleton of the mesh. One of its components is a single node.

Lemma 4.47. *If the* center *of M is* flower(v)*, and w is an out neighbour of v, then* $w(\mathsf{B}^{\circ}_{v}(w)) \leq \frac{1}{2}(w(\widetilde{\mathsf{M}} - w(V_{rem}(w,u))))$*, where u is either of the two in neighbours of v that are adjacent to w around* flower(v)*, and $V_{rem}(w,u)$ denotes bridges with all vertices of attachment in* $petal_{w,u}$.

Proof. Since the center is flower(v), we have that $w(\mathsf{B}^{\circ}_{v}(w)) \leq w(\mathsf{B}^{\circ}_{w}(v))$. But $V_{rem}(u,w)$ has empty intersection with each of $\mathsf{B}^{\circ}_{v}(w)$ and $\mathsf{B}^{\circ}_{w}(v)$. Thus $w(\mathsf{B}^{\circ}_{v}(w)) + w(\mathsf{B}^{\circ}_{w}(v)) \leq w(\widetilde{\mathsf{M}}) - w(V_{rem}(u,w))$. The lemma follows. □

Lemma 4.48. *1. If the* center *of M is not of the form* flower(v) *where v is an internal node of a 3-connected component, then removing it from $\widetilde{\mathsf{M}}$ disconnects $\widetilde{\mathsf{M}}$ into weakly-connected components, each with weight less than $\frac{1}{2}w(\widetilde{\mathsf{M}})$.*

2. If the center *of M is* flower(v) *for an internal node v of a 3-connected component P, then on removing* flower(v) *from $\widetilde{\mathsf{M}}$, no weakly-connected component has weight more than $\frac{1}{2}w(\widetilde{\mathsf{M}})$.*

Proof. 1. This follows from the vertex separator lemma for trees with weighted vertices.

2. This follows from the v being the sink node of $I'(P)$.

□

Lemma 4.49. *For every possible path p_{w_i,u_j} around v as defined in Lemma 4.38, V_{rem} consists of a disjoint union of weakly-connected components, each of which has weight $\leq \frac{1}{2}(w(M))$.*

Proof. A (weakly-connected) component of V_{rem} is a bridge, attached to $petal_{w_i,u_i}$ or to $petal_{w_i,u_{i+1}}$ via its vertices of attachment. In the clique sum decomposition, these vertices of attachment will always contain a 1 or 2 separating clique, since if a bridge is attached to a petal via three or more nodes, the first and the last vertices of attachment form a separating pair that separates the bridge from flower(v). Hence it is a branch remaining in T_M after removing the 3-connected piece node that is central in T_M. Since every branch after removal of the central piece of T_M has weight $\leq \frac{1}{2}(w(M))$, every (weakly) connected component of V_{rem} has weight $\leq \frac{1}{2}(w(M))$. □

For a path p_{w_i,u_j} (where $j \in \{i, i+1\}$) we sometimes use the notation $V_{rem}(w_i,u_j)$ to specify the petal where the bridges of V_{rem} are attached.

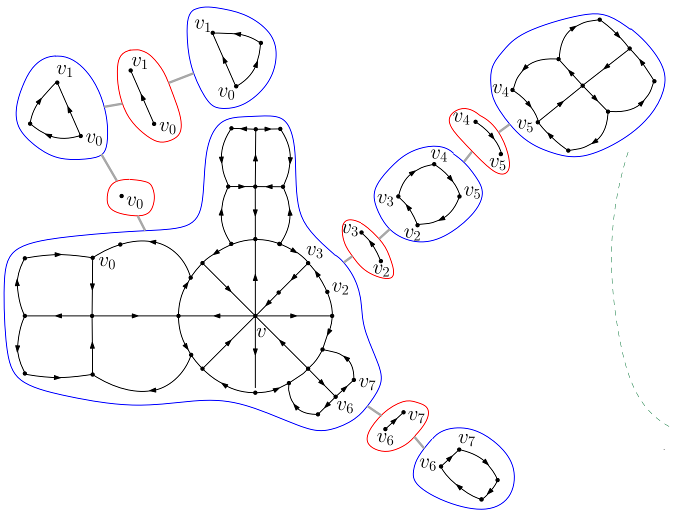


FIGURE 4.12: The tree decomposition of the mesh using 1,2-clique sums. The nodes encircled red are clique separator nodes.

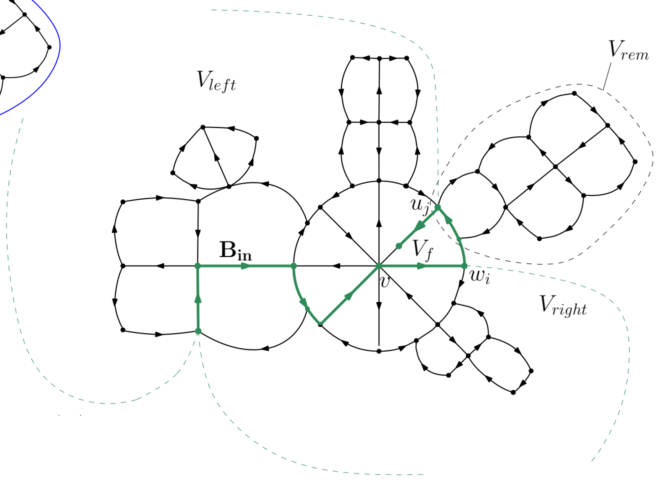


FIGURE 4.13: An example of a path separator. The vertex v is a central node, and the green path is a separator.

4.4.1 Mesh Separator Algorithm

Now we give the algorithm to find an (α, r) path separator in a mesh $M(G)$, with $r \in V(M)$, assuming the hypothesis of Lemma 4.33. Recall from Definition 4.30 an (α, r) path separator is a directed path starting at (the "root") r that is also an α separator.

1. Find the decomposition tree, T_M of M with 2-cliques and 1-cliques as the separating sets.
2. Find the center of the mesh M. It will either be a cut vertex, a separating pair, a cycle, or flower(v) for some internal vertex v.
3. If it is a cut vertex, we just find a path from the root r to it. If it is a separating pair (u, v), both the vertices must lie on a same face, which is a directed cycle. In both this case, and also the case in which the center is a cycle, find a path from the root to any vertex of the face that touches it the first time, and then extend the path by encircling the cycle.
4. If it is flower(v) for some internal vertex v, find a path $p = \langle r, \ldots, u_1, v\rangle$ to v. Let the junction neighbours of v in clockwise order starting from (u_1, v), be $w_1, u_2, w_2, \ldots, w_d$, with the w's being out junction neighbours and the u's being in junction neighbours. Starting clockwise from segment $\langle u, v\rangle$, find the first index i and $j \in \{i, i+1\}$ s.t. after removing the extended path p_{w_i,u_j}, (defined in Lemma 4.38) the remaining strongly-connected components are smaller than $\frac{11}{12}w(G)$.

The algorithm above can clearly be implemented in logspace with an oracle for planar reachability, and thus it can be implemented in UL $\cap$ co-UL.

It remains to show that the "first i" mentioned in the final step actually exists.

Lemma 4.50. *If the* center *of M is* flower(v) *for some internal vertex v, then there will always exist an adjacent face f_{u_i,v,w_i} s.t. the path p_{w_i,u_i} is an $\frac{11}{12}$-separator.*

Proof. We have the following two cases:

1. For some i and $j \in \{i, i+1\}$, $w(V_{rem}(w_i, u_j)) \geq \frac{1}{2}w(M)$.

   Then by Lemma 4.49, p_{w_i,u_j} separates $V_{rem}(w_i, u_j)$ from the rest of the graph, and also every weakly-connected component in $V_{rem}(w_i, u_j)$ has weight $\leq \frac{1}{2}w(M)$. Hence every weakly-connected component in M after removing p_{w_i,u_j} has weight $\leq \frac{1}{2}w(M)$.

2. For every p_{w_i,u_j}, $w(V_{rem}(w_i, u_j)) \leq \frac{1}{2}w(M)$.

   We know that for any index i and $j \in \{i, i+1\}$, if $f = f_{u_j,v,w_i}$, then $w(f) \leq w(G)/12$ by the hypothesis of Lemma 4.33. Starting clockwise from p_{u_1,w_1}, at first V_{left} is small, and on shifting from p_{w_i,u_i} to $p_{w_i,u_{i+1}}$ or from $p_{w_i,u_{i+1}}$ to $p_{w_{i+1},u_{i+1}}$, the increase in V_{left} is bounded above by $w(f) + w(V_{rem}(w_i, u_j)) + w(\widetilde{\mathsf{B}^\circ_\mathsf{v}(\mathsf{w_i})})$. Recall that

   (a) $w(f) \leq w(G)/12$ (by the hypothesis of Lemma 4.33).

   (b) $w(V_{rem}(w_i, u_j)) \leq \frac{1}{2}w(M)$ (by hypothesis for this case).

   (c) $w(\widetilde{\mathsf{B}^\circ_\mathsf{v}(\mathsf{w_i})}) \leq \frac{1}{2}(w(M) - w(V_{rem}(w_i, u_j)))$ (by Lemma 4.47).

   Thus the addition to V_{left} in each iteration is $\leq \frac{1}{12}w(G) + w(V_{rem}(w_i, u_j)) + \frac{1}{2}(w(M)) - \frac{1}{2}(w(V_{rem}(w_i, u_j))))$, which is equal to $\frac{1}{12}w(G) + \frac{1}{2}w(V_{rem}(w_i, u_i)) + \frac{1}{2}(w(M)) \leq \frac{1}{12}wG + \frac{3}{4}w(M)$. Thus we can stop the first time $w(V_{left})$ is greater than $w(G)/12$. So, we have $w(V_{left}) \leq \frac{2}{12}w(G) + \frac{3}{4}w(M) \leq \frac{11}{12}w(G)$, and $w(V_{right}) \leq \frac{11}{12}w(M)$, and $w(f) \leq \frac{1}{12}w(M)$, and $w(V_{rem}) \leq \frac{1}{2}w(M)$. Thus we have an upper bound of $\frac{11}{12}w(G)$ on all the disconnected components. Hence p_{x_i,w_i} is a $\frac{11}{12}$ path separator.

□

4.5 Path Separator in a Planar Digraph

Having seen how to construct a path separator in a **mesh**, we now show how to use that to construct an $(\frac{11}{12}, r)$ path separator in any planar digraph.

1. Given a graph G, first embed the graph so that the root r lies on the outer face. Through the root, draw a virtual directed cycle C_0 that encloses the entire graph, and orient it, say clockwise. Find the layering described in Section 4.3 and output it on a transducer. Cycle C_0 will be colored red and will be in the sublayer $\mathcal{L}^{0,0}$.

2. In the laminar family of red/blue cycles, find the cycle C s.t. $w(C)$ is more than $|G|/12$, but no colored cycle C' in the interior of C has the same property. Such a cycle will clearly exist (it could be the virtual cycle C_0). Let the sublayer of C be $\mathcal{L}^{k,l}$.

3. Find a path p from the root r to any vertex r_C of the cycle C such that no other vertex of C is in the path. As seen above in Lemma 4.27, the graph in the interior of C and belonging to the immediately next sublayer ($\mathcal{L}^{k+1,l}$ if C is clockwise and $\mathcal{L}^{k,l+1}$ if C is counter-clockwise) is a DAG of meshes. There are two cases possible:

   (a) The graph $\widetilde{\mathsf{C}}$ has no strongly-connected components of weight larger than $|G|/12$. In this case we simply extend the path p from r_C by encircling the cycle C till the last vertex and stop.

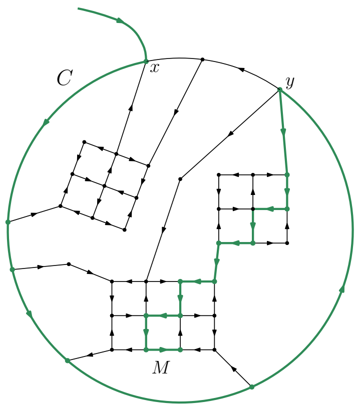


FIGURE 4.14: The cycle C is a cycle satisfying the property stated in step 2 of the algorithm. The mesh M in the next sublayer is heavy, so we find a path from the last vertex on C that can reach M (in this case y), and then apply the algorithm of previous section on M.

(b) The graph $\widetilde{C}$ has a strongly-connected component of weight more than $|G|/12$. In this case, we extend p from r_C by encircling C till the last vertex u on C that can reach any such component M_C. Then extend the path from u to any vertex of M_C and apply the mesh separator lemma (Lemma 4.33) to obtain the desired separator. (Observe that M_C satisfies the hypothesis of Lemma 4.33.)

Lemma 4.51. *The path p obtained by the above procedure is an $\frac{11}{12}$ separator.*

Proof. We look at the two cases from step 3 in the algorithm:

1. In this case it is clear that the interior and exterior of cycle C are disconnected by p. The exterior of C has size $\leq \frac{11}{12}|G|$ (by definition of C), and in its interior every strongly-connected component has weight at most $|G|/12$. Thus this satisfies the definition of an $\frac{11}{12}$ separator.

2. We took the last edge in C from r_C that can reach the mesh M_C, and extended the path to M_C. Thus after removing p, one weakly-connected component consists of the exterior of G, along with (possibly) some vertices in the interior of C that cannot reach any "large" mesh in the interior. Since M_C has weight greater than $\frac{1}{12}|G|$, no strongly-connected component embedded outside of M_C can have weight more than $\frac{11}{12}|G|$. Also, after removing path p, Lemma 4.33 guarantees that no other strongly-connected component will have weight more than $\frac{11}{12}|G|$. Thus this is an $\frac{11}{12}$ separator.

Hence overall we can guarantee an $\frac{11}{12}$ path separator in G. □

All operations involved like embedding the graph, computing the layering, computing the size of components after removing a cycle, finding a path between two vertices, and finding a separator of a mesh (using Lemma 4.33) can be done in UL $\cap$ co-UL. Therefore the entire algorithm is in UL $\cap$ co-UL. This completes the proof of Theorem 4.1.

4.6 Building a DFS Tree Using Path Separators

Given a graph G, one can determine in logspace if G is planar, and then compute a planar embedding [AM04, Rei08]. Thus it will suffice to give a give a recursive divide and conquer algorithm for DFS, assuming that G is presented embedded in the plane, and that we are given a root vertex r on the outer face.

A single phase of the algorithm starts with G and r, and creates a sequence of subgraphs, each of size at most $\frac{11}{12}$ the size of G. The algorithm then computes DFS trees for each of those graphs (recursively), and the results of (some of) the graphs are sewn together to obtain a DFS tree for G. Each phase can be computed in $\mathsf{AC}^0(\mathsf{UL} \cap \text{co-}\mathsf{UL})$, and hence the entire algorithm can be implemented in $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$.

We now describe a single phase in more detail.

1. Given a planar drawing of G and a root vertex on the outer face r, find an $\frac{11}{12}$ path separator $p = \langle r, v_1, v_2, \ldots, v_k \rangle$, as described in Section 4.5. Path p is included in the DFS tree.

2. Let $R(v)$ denote the set of vertices of G reachable from v. Now for every vertex v_i in p compute in parallel: $R'(v_i) = R(v) \backslash (\bigcup_{j=i+1}^{k} R(v_j))$ Our DFS will correspond to first traveling along p to v_k, doing DFS on $R(v_k)$, and then while backtracking on p, do DFS on $R'(v_i)$ for i from $k-1$ downto 1. Given G, the encodings of p and $R'(v_i)$ can all be computed in $\mathsf{AC}^0(\mathsf{UL} \cap \text{co-}\mathsf{UL})$.

3. For any v_i, $R'(v_i)$ can be written as a DAG of SCCs (strongly-connected components), where each SCC is smaller than $\frac{11}{12}|G|$. In $\mathsf{AC}^0(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ we can compute this DAG and we can compute an encoding of the tuple (i, M, v) where M is an SCC in $R'(v_i)$ and v is a vertex in M. Recursively, in parallel, we compute a DFS tree of M for each tuple (i, M, v), using v as the root. Now we need to show how to sew together (some of) these DFS trees, to form a DFS tree for G with root r. Namely, for each i, for each $M \in R'(v_i)$, we will select exactly one v such that the DFS tree for G will incorporate the DFS tree computed for (i, M, v), as described next.

4. Given a triple (i, M, v), let $x_0, x_1, \ldots, x_s$ be the order in which the vertices of M appear in a DFS traversal where the root $x_0 = v$. If v is such that the DFS tree for (i, M, v) is incorporated into the DFS tree that we are constructing for G, then our DFS will correspond to first following the edges from x_0 that lead to other SCCs in $R'(v_i)$. (No vertex reachable in this way can reach any x_j, or else that vertex would also be in M.) And then we will move on to x_1 and repeat the process, etc. Thus let $R''_{i,M,v}(x_j) = ((R(x_j) \cap R'(v_i)) \backslash M) \backslash (\bigcup_{k<j} R(x_k))$.

   Our DFS tree for G is composed by using Algorithm 2 of Section 4.2, on the multigraph that has a vertex for each SCC in the DAG of SCCs that makes up any $R''_{i,M',v}(x_j)$. Crucially, the ordering on the edges that leave any node M'' in this multigraph is determined by the order in which the vertices of M'' are visited in the DFS tree of M''.

   Let us see in more detail how to use the DFS trees that we computed for each (i, M, v), by considering how to process the DAG of SCCs in some $R''_{i,M',v}(x_j)$. Every SCC in this DAG is reachable from x_j. We will be using Algorithm 2 from Section 4.2 to compute the lexicographically-least path from x_j to any SCC M'' in $R''_{i,M',v}(x_j)$. We can use any ordering for the edges that leave x_j (such as the order in which the edges are presented). For the other SCCs in the DAG, the ordering must be chosen more carefully. Let us say that the first edge that leaves x_j that lies on some path to a node in M'' is (x_j, y); this edge will be in our DFS tree for G. The node y is in some SCC N in $R''_{i,M',v}(x_j)$. A DFS tree $T_{i,N,y}$ was computed for (i, N, y); the order in which the nodes of $T_{i,N,y}$ are visited imposes an order on the edges that leave N in the acyclic multigraph. That is the order that is used, in applying Algorithm 2.

More generally, when executing the **while** loop in Algorithm 2, if the variable *current* currently is set to some SCC M_1, and M_2 is the first SCC adjacent to M_1 (using the ordering on the edges of M_1) that lies on a path to M'', and this is because there is an edge (w, z) where w is the first node in the traversal of M_1 that is adjacent to any node of M_2, then on the next pass through the **while** loop, the ordering on the edges leaving M_2 is determined by the traversal order of the DFS tree that was computed for (i, M_2, z). Let us denote this node z by v_{M_2}; the edge (w, v_{M_2}) will be in the DFS tree for G.

5. The final DFS tree for G thus consists of the path $p = \langle r, v_1, v_2, \ldots, v_k\rangle$ along with the DFS trees that were computed for each (i, M, v_M) (for the unique vertex v_M identified in the preceding step).

Note that planarity is used in the algorithm only to compute the path separators at each step. If we have an oracle to compute those, the algorithm is in UL ∩ co-UL for general digraphs. This completes the proof of Theorem 4.2.

Bibliographical Remarks The results presented in this chapter are based on joint work with Eric Allender, Samir Datta which appeared in [ACD21, ACD22].

Chapter 5

More results on Parallel Complexity of Depth-First-Search

In this chapter, we continue the line of investigation on the parallel complexity of DFS and related problems in various graph classes and give NC algorithms for the problem in new graph and digraph classes.

We first look at DFS in graphs (directed, as well as undirected) of bounded genus. Our algorithm proceeds via finding a balanced path separator. To find a balanced separator, we find $O(g)$ cycles removing which leaves the remaining components planar, and then find a separator in the large planar component (if it exists). Then we apply Corollary 3.4 to combine the $O(g)$ *paths* along with the separator path in the large planar component into a single path separator.

Next we look at DFS in single-crossing-minor-free graphs. We again proceed by finding a balanced path separator of the input graph. We start by computing the clique-sum decomposition tree described in Section 2.2.4 and finding its 'center piece' (a piece such that all children subgraphs attached to it are at most half the size of input graph). Then we project the weight of all the attached children subraphs on the vertices, edges, and faces of the center piece, and find a cycle separator of this weighted center piece such that the interior and the exterior of the cycle separator are both small. Then we argue that we can lift this cycle separator from the center piece to a balanced cycle separator of the original graph. We would like to point out however that naively projecting the weights on the center piece does not work, since the cycle separator we find in this piece might use virtual edges (added while computing the decomposition to maintain connectivity) in manner such that the separator cannot be lifted to a simple cycle separator in the original graph. To deal with this, we use appropriate gadgets while projecting the weights, that mimic the connectivity of the subgraph whose weight we are projecting.

Next we look at DFS in bounded treewidth graphs (directed as well as undirected). As noted in Corollary 3.6, an $AC^1(L)$ algorithm for DFS in bounded treewidth graphs easily follows from the machinery of Kao [Kao88]. We improve this bound to L. Our technique for DFS on bounded treewidth digraphs involves Courcelle's fundamental theorem on evaluating MSO-properties on the class of bounded tree-width graphs. Specifically, we use the Logspace version of Courcelle's theorem from [EJT10]. We also use an existing result [CF12, BD15] that states that we can add a linear ordering to the vertices of the graph without blowing up the treewidth of the structure. This combined with the observation of [dlTK95] characterizeing the lex-first DFS tree in terms of lex-first paths to each vertex, allows us to express membership of an edge to the DFS tree, in MSO.

Results The results we prove can be summarized as follows:

Theorem 5.1. *We have the following bounds:*

1. *DFS in bounded genus graphs and digraphs is in* $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$.
2. *DFS in undirected single-crossing-minor-free graphs is in* NC.
3. *DFS in directed and undirected graphs of bounded tree width is in* L

We remark here that the problem of finding a perfect matching, which was the only routine requiring randomization in Aggarwal and Anderson's [AA88] RNC algorithm for DFS, was shown to be in (deterministic) NC for bounded genus graphs [MV00, DKTV12] as well as for single-crossing-minor-free graphs [EV21]. This may suggest that plugging in these results into the algorithm of [AA88] might straight away give a deterministic NC algorithm for DFS in these classes. However, the reduction in [AA88] involves adding large bi-cliques in the input graph, which does not seem to preserve the genus, nor the forbidden minors of the graph. Hence we are not aware of any method to use these results with the reduction of [AA88] to get a deterministic parallel algorithm for DFS in these classes.

We refer the reader to Chapter 2 for basic preliminaries, and Section 2.2.4 in particular for clique sum decompositions.

5.1 Path Separators in directed graphs embeddable on surfaces of bounded genus

In this section we will show how to find balanced path separators in digraphs of genus g (assuming g is fixed and given to us). We will first use Theorem 2.8 to find embeddings of bounded genus graphs in L.

Next we first find a small number of disjoint, surface non-separating cycles (we mean they are cycles if we ignore directions, they may not necessarily be directed cycles), say $C_1, C_2, \ldots C_l$ ($l \leq g$) in G, such that the (weakly) connected components left after removing them are all planar.

We use the following theorem from [MT01].

Theorem 5.2. *Let G be a graph embeddable on a surface $\mathcal{S}$, of genus g. Let C be a cycle that forms a surface non-separating curve on some embedding of G on $\mathcal{S}$. Then every connected component of $G - C$ is embeddable on a surface of genus $g - 1$.*

Therefore, if we have an algorithm to find such a (undirected) cycle in $\mathsf{UL} \cap \text{co-}\mathsf{UL}$, then we can repeatedly apply it g times to find (undirected) cycles $C_1, C_2 \ldots C_k$ such that $G - \{C_1 \cup C_2 \ldots C_k\}$ consists of planar connected components.

We will use the following lemma from [ADR05] (See also [Tho90]) to find such cycles.

Lemma 5.3. *[ADR05] Let G be a graph of genus $g > 0$, and let T be a spanning tree of G. Then there is an edge $e \in E(G)$ such that $T \cup e$ contains a surface non-separating cycle.*

The surface non-separating cycle in $T \cup e$ is naturally the fundamental cycle of the edge e. Clearly the theorem also works if we have an arborescence instead of a spanning tree. The fundamental cycle C corresponding to a non-tree edge $e = (u, v)$ would consist of the non-tree edge e and two vertex disjoint directed paths, say P, P', starting from the least common ancestor of (u, v) in the arborescence (we keep the common ancestor vertex in one of P or P', say in P), and ending at u, v respectively. We can compute an arborescence of G in UL ∩ co-UL using [BTV09, TW10] as explained in Section 3.3. We can go over each edge of G in logspace and check if removing its fundamental cycle reduces the genus of remaining components using Theorem 2.8. Hence, given an embedded digraph of genus more than zero, we can find a surface non-separating undirected cycle in the graph, the vertices of which can be partitioned into two directed paths, in UL ∩ co-UL.

Therefore we have the following algorithm to find a path separator in G:

Input : A directed graph G.

1. Decompose the graph into strongly connected components. If there is no strongly connected component of size larger than $2n/3$, then the empty set is the separator, else let G_0 be the component larger than $2n/3$.

2. Find an embedding of G_0 using Theorem 2.8. Find an arborescence of G_0 and use Lemma 5.3 to find a non-tree edge such that its fundamental cycle, C_1, is a surface non-separating cycle. Therefore every connected component of $G_0 - C_1$ has genus strictly lesser than that of G_0.

3. Let G_1 be the largest strongly connected component of $G_0 - C_1$. If it is more than $2n/3$ in size then we do the steps above on G_1. Repeat until finally we have graph $G_{rem} = G - (C_1 \cup C_2 \ldots C_l)$ (where $l < g$) such that either all strongly connected components of G_{rem} are smaller than $2n/3$ or all the components are planar. In the latter case, find a $2/3$-path separator P in the strongly connected componentin G_{rem} that is of size more than $2n/3$ using Theorem 3.8.

4. For each cycle C_i, let P_i, P'_i denote the two directed paths that form the cycle C_i as explained above. The paths $P_1, P'_1, \ldots P_l, P'_l, P$ together form a $2/3$-multipath separator of G.

5. Use Corollary 3.4 to merge the paths $P_1, P'_1, \ldots P_l, P'_l, P$ into a single path P_s which is also a $2/3$-path separator of G. Output P_s.

Each step above can be done in UL ∩ co-UL. Therefore we can find a $2/3$-path separator in directed graphs of bounded genus in UL ∩ co-UL. Using Theorem 4.2, we can find a depth first search tree in such graphs in AC^1(UL ∩ co-UL). The bound naturally also translates to DFS in undirected graphs of bounded genus.

5.2 Path Separators in Single-Crossing-Minor-Free-Graphs

We start with computing the 3-clique-sum decomposition by the NC algorithm of [EV21]. We will denote this decomposition tree by T_G, each node of which is either a piece node or a clique node, and the weight of each node is the number of vertices in the corresponding piece/clique. (Note that by this convention, the sum of weights of all nodes of T_G will sum up to more than n since some vertices would occur in multiple pieces).

Definition 5.4. Let t_R be a node of T_G, and R its corresponding piece/clique, such that the size of any connected component in $G - R$ is lesser than $n/2$. We call t_R a central node of T_G, and R a central piece/clique.

Such a node exists because if we traverse down the tree T_G from an arbitrary root, picking the heaviest child at every step, then at some stage the weight of the heaviest child becomes less than or equal to $n/2$ for the first time. That node must be a $central$ node. On removing the piece/clique corresponding to that node, the vertices of it that shared with its children pieces are also removed (every piece shares one, two or three vertices with its parent piece). Therefore the size of remaining components must be lesser than $n/2$.

Given the tree T_G, this procedure to find the $central$ node can clearly be done in L.

We will assume t_R to be the root of T_G hereafter. We will denote the subtrees of T_G that are children of t_R by $\{T_1, T_2, \dots T_l\}$, the subgraphs of G corresponding to these subtrees by $\{G_1, G_2, \dots G_l\}$, and the cliques by which they are attached to R by $\{c_1, c_2 \dots c_l\}$ respectively. These subgraphs might themselves consist of smaller subgraphs glued at the common clique. For example, G_1 might consist of $G_{11}, G_{12}, \dots G_{1k}$ that are glued at the shared clique c_1 (i.e. $G_1 = G_{11} \oplus_{c_1} G_{12} \dots \oplus_{c_1} G_{1k}$). See Fig. 5.1 for reference. Note that because R is a central node, $w(G_{1i}) - w(c_1) < n/2 \; \forall i \in [1..k]$.

If t_R is a clique node, then we have a $1/2$-separator of at most three vertices and we are done. Therefore there are two cases to consider. Either R is a planar piece, or a piece of bounded treewidth.

When R is of bounded treewidth In this case, we will refine the tree T_G by further decomposing R. Since R is of treewidth at most τ_H, we can compute a tree decomposition of R such that every bag is of size at most $\tau_H + 1$, in L using [EJT10]. Let the this tree be denoted by T_R. Consider the subtree T_1 described above, which has a node t_{c_1} by which it is attached to R. Since c_1 is a clique, there must be at least one bag in T_R that contains all vertices of c_1. Attach t_{c_1} to any such node of T_R. Do this proceedure for all subtrees $T_1, T_2, \dots T_l$. It is easy to see that this will result in another tree decomposition, and that at least one of the nodes of T_R will be a central node of this new tree. Hence we can use the procedure described above to find it in L. Since the bags of T_R are of size at most $\tau_H + 1$, we get a 1/2$-separator$ of G consisting of constant number of vertices, and we are done.

When R is planar We start with a simple, natural idea, but encounter an obstacle. Then we show how to fix the obstacle. Note that R (which may contain virtual edges), is biconnected.

To initialize, we assign weight 1 to all vertices of R and weight 0 to all edges and faces of R. We project the weight of subgraphs $G_1, G_2 \dots G_l$ on vertices, edges and faces of R, at the respective cliques they are attached at. Suppose the subgraph G_k is attached to R at clique c_k. Then we project the weight as follows.

1. If c_k is a single vertex v, then assign a weight equal to $|V(G_k)|$ to *vertex* v.
2. If c_k is a separating pair (u, v), then we assign a weight equal to $|V(G_k)| - 2$ to the *edge* (u, v) of R. (We subtract 2 since weights of vertices u, v are already accounted for).
3. If c_k is 3-clique of vertices (v_1, v_2, v_3), then we assign weight equal to $|V(G_k)| - 3$ to the *face* of R enclosed by c_k.

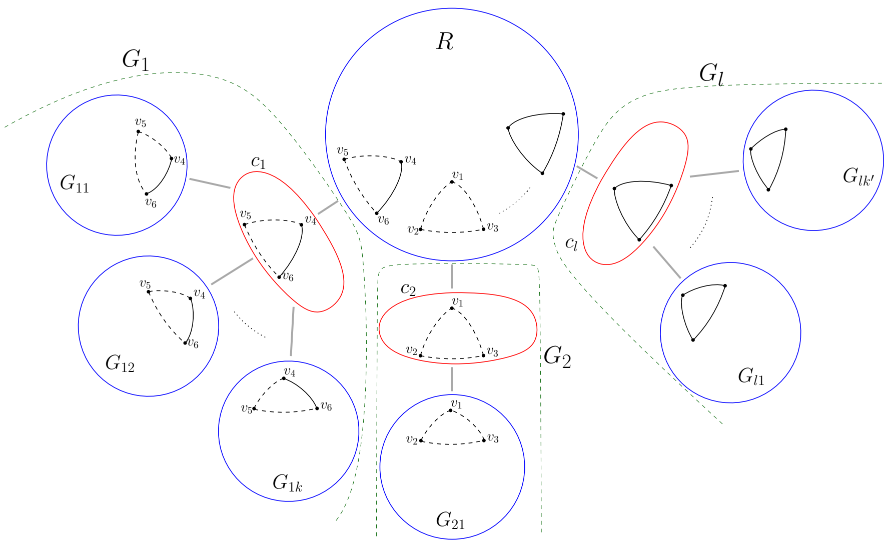


FIGURE 5.1: R is the central node with children subgraphs $G_1, G_2, \dots G_l$ attached to it via cliques $c_1, c_2, \dots c_l$ respectively. G_1 consists of subgraphs $G_{11}, G_{12} \dots G_{1k}$ glued at c_1. G_2 is same as G_{21}. Dashed edges denote virtual edges in R.

Let $\widetilde{R}$ denote the weighted version of R obtained after projecting these weights. The sum of weights of all vertices, edges, faces of R is n. There are two possibilities. Suppose there exists a clique, say c_1 (which is either a vertex, or an edge, or a face of $\widetilde{R}$), with weight at least $2n/3$. This means that total weight of subgraphs attached to R via c_1, i.e $w(G_{11} \oplus_{c_1} G_{12} \oplus_{c_1} \dots G_{1k})$, is at least $2n/3$. Since each of the graphs $G_{11}, G_{12} \dots G_{1k}$ is of size at most $n/2$ and they all get disconnected on removal of c_1, the vertices of c_1 form a 1/2$-separator$ of G, and we are done.

Therefore the remaining possibility we need to consider is one where every vertex, edge, face of $\widetilde{R}$ has weight smaller than $2n/3$. Since $\widetilde{R}$ is planar and biconnected, we can find a $\frac{2}{3}$-interior-exterior-cycle-separator of it in L using Theorem 3.2. Let $\widetilde{C}$ be such a separator. Label one of the regions $\widetilde{C}$ divides R into as the interior of $\widetilde{C}$ and the other as the exterior of $\widetilde{C}$. Observe that for any subgraph that is attached to R (Like G_{11} for example), its vertices of attachment to R (the clique via which it is attached), will either all lie in the interior of $\widetilde{C}$ (not necessarily the *strict* interior), or all in the exterior of $\widetilde{C}$. We call the attached subgraphs of the former kind as *interior components* of R with respect to $\widetilde{C}$, and the latter as *exterior components* of R with respect to $\widetilde{C}$ (we will drop the phrase with respect to $\widetilde{C}$ for brevity). On removing the vertices of $\widetilde{C}$ in G, there cannot remain any path in G from any interior component of R or any vertex in strict interior of R, to any exterior component of R or any vertex in strict interior of R. Therefore we have the following claim :

Claim 5.5. *Let $\widetilde{C}$ be an α-interior-exterior-cycle-separator of $\widetilde{R}$, where $\alpha \in [\frac{1}{2}, 1)$. Then the vertices of $\widetilde{C}$ form an α-separator of G.*

Now if we could replace the virtual edges of $\widetilde{C}$ by path segments in the attached subgraphs that are internally disjoint from each other, then we would get a cycle C in G going through the vertices of $\widetilde{C}$ which would be a 2/3$-separator$ of G, and we would be done. As we had noted, the virtual edges can be seen as capturing external connections of a component in a clique-sum decomposition. More precisely, corresponding to every virtual edge of $\widetilde{C}$, there is a path with the same end points in the subgraph attached to R via the clique that the virtual edge is part of. However there is an issue here. The virtual edges in a 3-clique need not capture *"disjointness"* of the external connections accurately. For example, suppose $c_2 = \{v_1, v_2, v_3\}$ is a 3-clique and there is only one subgraph, say G_{21}, that is attached to R at c_2. Suppose (v_1, v_2), and (v_1, v_3) are virtual edges in R, and our cycle separator $\widetilde{C}$ uses both of these edges (they must necessarily be consecutive in $\widetilde{C}$). Though there must be paths between v_1, v_2, and between v_1, v_3 in G_{21} (since it is attached via 3-clique), there might not exist such paths that are internally disjoint. Thus we cannot simply substitute the virtual edges by paths in the attached subgraphs to get a simple cycle separator of G by projecting weights in this manner. Hence instead of just projecting the weights of the entire subgraphs on faces of R, we use a more appropriate gadget to mimic the structure of G_{21}, so that any segment of $\widetilde{C}$ using the edges of the gadget can be substituted by a simple path in G_{21}, and the separator C obtained after all such substitutions remains a balanced separator of G.

Note that this issue does not occur in the case when two virtual edges belong to a clique which has at least two subgraphs attached to it. For example, suppose R has two virtual edges in clique c_1, which has G_{11}, G_{12} attached to it. If $\widetilde{C}$ uses both these edges, we can substitute one of them by a path in G_{11}, and the other one by a path in G_{12}. Since G_{11}, G_{12} are disjoint except for vertices of c_1, these paths will also be internally disjoint.

Constructing gadgets for the subgraphs attached to R. We now explain how to construct gadgets for graphs like G_{21} described above, and use them to get the correct weighted graph $\widetilde{R}$ where we will find the cycle separator $\widetilde{R}$. We can adversarially assume that all three edges $(v_1, v_2), (v_2, v_3), (v_3, v_1)$ are virtual edges.

Definition 5.6. Let G_{21} be a graph and vertices $v_1, v_2, v_3 \in V(G_{21})$ be called its terminals. The *disjoint path configuration* of G_{21} with respect to its terminals v_1, v_2, v_3 consists of three boolean variables : $DP(v_1\text{-}v_2\text{-}v_3), DP(v_2\text{-}v_3\text{-}v_1), DP(v_3\text{-}v_1\text{-}v_2)$. $DP(v_1\text{-}v_2\text{-}v_3)$ takes value *True* if there exists a path between v_1 and v_3 that goes via v_2, and *False* otherwise. $DP(v_2\text{-}v_3\text{-}v_1), DP(v_3\text{-}v_1\text{-}v_2)$ are defined similarly.

Their are four cases we need to consider for the disjoint path configuration, others are handled by symmetry. We give the gadget for each of these cases, and give the proof of correctness for one of the cases, as the others have a similar proof.

Case 1	$DP(v_1\text{-}v_2\text{-}v_3)$	$DP(v_2\text{-}v_3\text{-}v_1)$	$DP(v_3\text{-}v_1\text{-}v_2)$
	True	*True*	*True*

In this case we do not need a new gadget, we just keep the virtual edges between the clique vertices and assign weight equal to $|V(G_{21})| - 3$ to the face enclosed by them. If $\widetilde{C}$ takes two virtual edges of c_2, we can use the two disjoint path algorithm in [KMV92] to find the corresponding paths in G_{21}.

Case 2	$DP(v_1\text{-}v_2\text{-}v_3)$	$DP(v_2\text{-}v_3\text{-}v_1)$	$DP(v_3\text{-}v_1\text{-}v_2)$
	True	*True*	*False*

By Menger's theorem (see Theorem 2.3), we know that there must be at least one cut vertex in G_{21} that separates v_1 from $\{v_2, v_3\}$. Let x be a cut vertex separating these. Let the connected component of $G_{21} - x$ containing v_1, augmented with x, be H_1. Let the component containing v_2, v_3, augmented with x, be H_2 (i.e. $V(H_1) \cap V(H_2) = \{x\}$ and $V(H_1) \cup V(H_2) =$

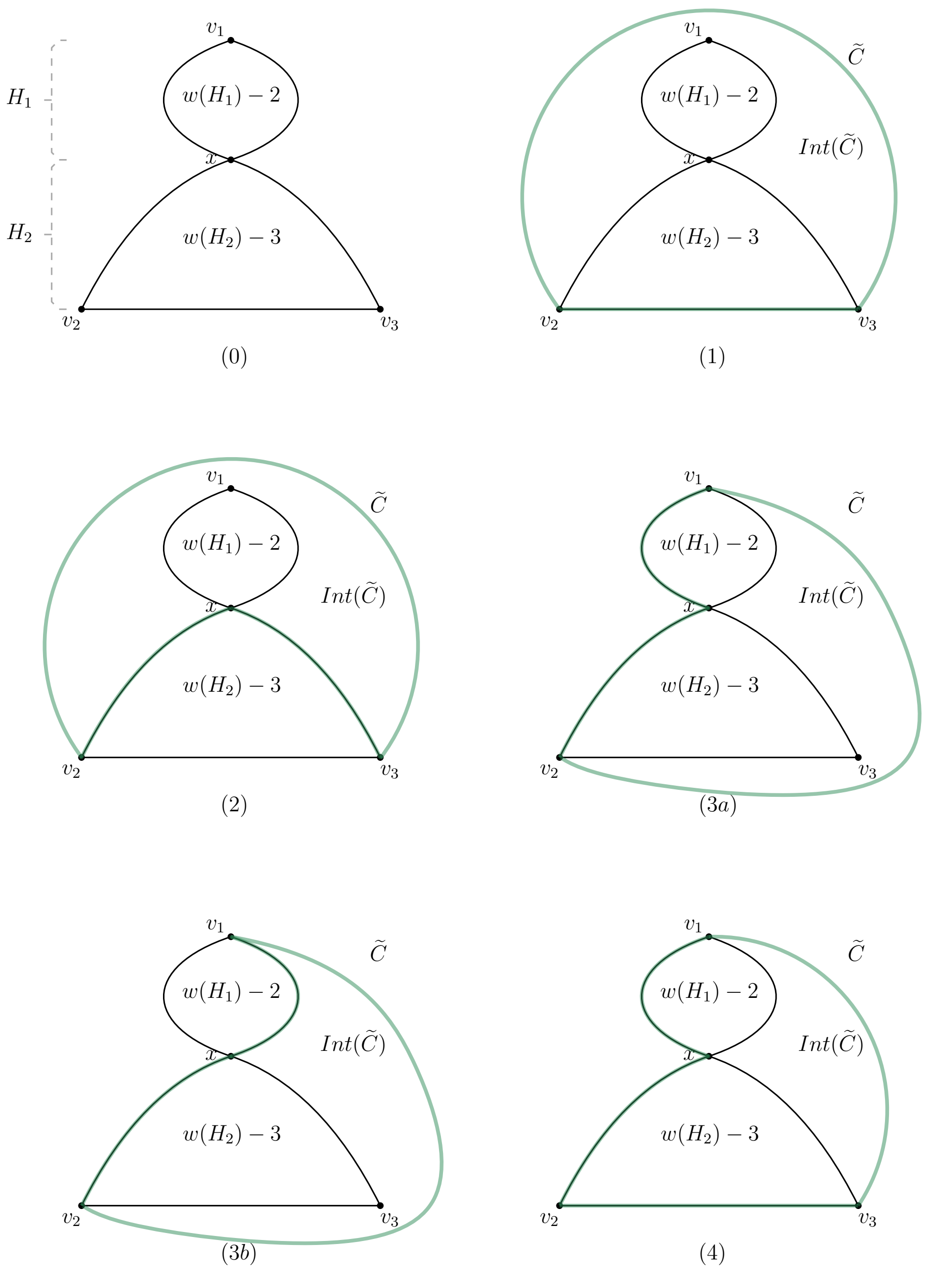


FIGURE 5.2: Figure (0) shows the gadget for the graph G_{21} for Case 2. The values $2, 3$ are subtracted from weight of the faces because the vertices v_1, v_2, v_3, x have a weight of one themselves. Note that $w(H_1) + w(H_2) = w(G_{21}) + 1$ (since x is shared by H_1 as well as H_2). The subgraph H_2 of the gadget has no cut vertices. The remaining figures show the possible ways the cycle $\widetilde{C}$ can use the segments of the gadget. They correspond to the subcases $1 - 4$ of the argument to show the correctness of the gadget in Case 2.

$V(G_{21})$). We choose x such that there is no cut vertex in H_2, that separates v_1 and $\{v_2, v_3\}$ in G_{21}. It is easy to see that x is unique. By our choice of x, there must be a path in H_2 from v_2 to v_3 via x. Thus the gadget is as shown in figure Fig. 5.2, containing vertices v_1, v_2, v_3, x, and the total weight distributed among the faces as shown. The following lemma shows the correctness of this proceedure.

Lemma 5.7. *Suppose G_{21}, which is the subgraph attached to R via c_2, is replaced in $\widetilde{R}$ by the gadget defined above for this case. Then if the α-cycle separator $\widetilde{C}$ of $\widetilde{R}$ uses any virtual edges of this gadget, we can replace them by paths in G_{21} such that they new cycle obtained is a simple cycle and it remains an α-separator in $R \oplus G_{21}$*

Proof. Suppose $\widetilde{C}$ uses some segment of this gadget. Without loss of generality, we can assume that each of v_1, v_2, v_3 lies on $\widetilde{C}$, or in the strict interior of $\widetilde{C}$. G_{21} therefore is an interior component of R with respect to $\widetilde{C}$. We need to argue two things. First, that for the segments of the gadget used by $\widetilde{C}$, we can substitute simple paths in G_{21}. Second, that after the substitution, the new cycle remains a balanced separator. For this, it is sufficient to show that the weight contribution of G_{21} to the interior components is at most the weight contribution of the gadget to $w(int(\widetilde{C}))$, and the weight contribution of G_{21} to the exterior components is at most the weight contribution of the gadget to $w(ext(\widetilde{C}))$ (we will drop the term 'weight' and just use contribution of G_{21}/gadget hereby for brevity).

This is because by the property of cycle separator, both $w(int(\widetilde{C})), w(ext(\widetilde{C}))$ are already guaranteed to be at most $2n/3$, and as explained above, there can be no path from any interior component of R in $G - V(\widetilde{C})$ to any exterior component of R in $G - V(\widetilde{C})$. Since G_{21} is an interior component, its contribution to exterior components of R is 0 regardless of which segment of the gadget is taken by $\widetilde{R}$. Therefore we only have to argue about its contribution to interior components.

Also note that in any case, if all three of v_1, v_2, v_3 lie on the cycle $\widetilde{C}$, then also the contribution of G_{21} is 0 to both interior as well as exterior components of R, as G_{21} gets disconnected from rest of the graph on removing vertices of $\widetilde{C}$. Therefore for the second part of the proof about the cycle remaining a balanced separator, we only have to consider subcases when one of v_1, v_2, v_3 lies in the strict interior of $\widetilde{C}$.

Now we look at the subcases of which segment of the gadget is used by $\widetilde{C}$. There are four possible subcases (upto symmetry):

1. $\widetilde{C}$ uses only the edge (v_2, v_3) from the gadget (see subfigure (1) in Fig. 5.2). Replace the edge (v_2, v_3) with any v_2-v_3 path in G_{21} in the new cycle C. It will clearly remain a simple cycle. By the explanation above, we can assume that v_1 lies in the *strict* interior of $\widetilde{C}$. Then the contribution of the gadget to $w(\widetilde{C})$ is $w(G_{21}) - 2$, which is at least as much as contribution of G_{21} to interior components of R. Therefore if $\widetilde{C}$ is an α interior-exterior-cycle separator of $\widetilde{R}$, then C is certainly an α cycle separator of $R \oplus G_{21}$.

2. $\widetilde{C}$ uses the segment v_2-x-v_3 from the gadget (see subfigure (2) in Fig. 5.2). We replace the segment by a path in G_{21} from v_2 to v_3 via x. We can again assume v_1 lies in the strict interior of $\widetilde{C}$. The contribution of the gadget to interior of $\widetilde{C}$ is $w(H_1) - 1$, which is at least as much as contribution of G_{21} to interior components of R (Since x also lies in C, and removing C would disconnect vertices of H_2).

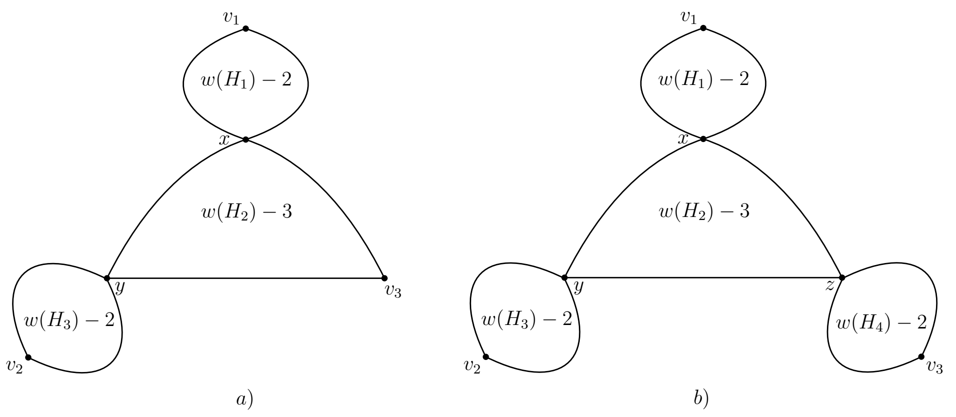


FIGURE 5.3: Figure a) shows the gadget for graph G_{21} for Case 3 and figure b) for Case 4. The subgraphs H_1, H_2, H_3, H_4 are labelled by the weight functions. The total weight in both both gadgets sums up to $w(G_{21})$. The subgraph H_2 of the gadget has no cut vertices in both the gadgets. In the gadget for Case 4, it is possible that the block H_2 is actually empty, i.e. there exists a vertex $(x = y = z)$ which on removal disconnects all of v_1, v_2, v_3 from each other. We don't draw the gadget separately since its clear what its structure is.

3. $\widetilde{C}$ uses segment v_1-x-v_2 from the gadget (see subfigures $(3a), (3b)$ in Fig. 5.2). It may use either one of the parallel edges between v_1, x. We replace the segment by a path in G_{21} from v_1 to v_2 via x. We can assume v_3 lies in the interior of $\widetilde{C}$. The contribution of the gadget to interior of $\widetilde{C}$ is at least $w(H_2) - 2$ (plus $w(H_1) - 2$ possibly , depending on which of the parallel (v_1, x) edges is taken), whereas the contribution of G_{21} to interior components of R is at most $w(H_2)$ since both x, v_1 are removed.

4. $\widetilde{C}$ uses segment v_1-x-v_2-v_3 (see subfigure (4) in Fig. 5.2). We replace the segment by a v_1-x path in H_1, concatenated with a path from x to v_3 via v_2 in H_2. There must exist such a path in H_2 since there exists a path from v_1 to v_3 via v_2 in this configuration. Since v_1, v_2, v_3 all lie in C in this case, the contribution of G_{21} is 0 to both interior and exterior components of R.

Therefore in all cases, we get a simple cycle by replacing the segments as described, and the new cycle C remains a balanced separator. □

	$DP(v_1\text{-}v_2\text{-}v_3)$	$DP(v_2\text{-}v_3\text{-}v_1)$	$DP(v_3\text{-}v_1\text{-}v_2)$
Case 3	*False*	*True*	*False*

The gadget is shown in Fig. 5.3. As in the previous case, we first find the vertex x separating $\{v_2, v_3\}$ and v_1 as described above. Note that in $G_{21} - x$, vertices v_2, v_3 must lie in the same connected component in this configuration, as otherwise there cannot exist a path from v_1 to v_2 via v_3. Let the connected component of $G_{21} - x$ containing v_1, augmented with x, be H_1, and the component containing v_2, v_3, augmented with x, be H' (i.e. $V(H_1) \cap V(H') = \{x\}$). Again by same argument, there must exist a cut vertex y in H' separating $\{x, v_3\}$ and v_2, and we choose y in a manner similar to previous step. Let H_2, H_3 be the components obtained by augmenting the components containing v_3, v_2 respectively in $H' - y$, with y. The weights are assigned similarly as shown in Fig. 5.3.

	$DP(v_1\text{-}v_2\text{-}v_3)$	$DP(v_2\text{-}v_3\text{-}v_1)$	$DP(v_3\text{-}v_1\text{-}v_2)$
Case 4	*False*	*False*	*False*

Like in previous cases, we find cut vertices that x, y, z that separate v_1, v_2, v_3 from rest of graph. There are two cases, either there is cut vertex w such that $G_{21} - w$ disconnects all of v_1, v_2, v_3. This is essentially the case when $x = y = z = w$. Other case is when there is no such cut vertex. The gadget is shown in Fig. 5.3.

The gadgets can be constructed in NC. We can in parallel check if each vertex of G_{21} satisfies the properties of vertices 'x, y, z' described above. We described the proceedure for G_{21}, but the construction can be done in parallel for all attached subgraphs of R. After finding a $\frac{2}{3}$-interior-exterior-cycle separator using Theorem 3.2, we can replace the virtual edges by the paths as described above in parallel. To find a path between two vertices via a third vertex, we can use the 2-disjoint path algorithm described in Theorem 2.4. Thus we can construct a $2/3$-path-separator of G in NC.

5.3 DFS in bounded treewidth graphs

In this section we will show that for a digraph with tree width bounded by a constant t, we can construct a depth first search tree in L. For an undirected graph, it is easy to see that a DFS tree of the bidirected version of it is also a DFS tree of the original graph.

We refer the reader to Section 2.3 for prelims related to logic and Gaifman graphs.

The input graph G naturally defines a structure $\mathcal{A}$ whose universe of discourse, $|\mathcal{A}|$ is $V(G)$, and its vocabulary consists of binary relation symbol E which is interpreted as the edge relation $E(G)$ over $V(G)$. The Gaifman graph of $\mathcal{A}$, $G_{\mathcal{A}}$, is isomophic to G. In order to construct a DFS tree, we will need to add a linear order O on $|\mathcal{A}|$, which would amount to adding $|V(G)|$ many edges in $G_{\mathcal{A}}$. A known result in model theory (see [CF12, BD15]) is that we can add a linear order to a structure, such that the treewidth of the structure is increased by at most 6. It is shown in [BD15], that we can construct such a linear order in L. We reproduce the lemma here

Lemma 5.8. *[BD15] Let G be a graph of bounded treewidth. We can augment G with some new vertices and edges to yield a graph G' with a tree decomposition T' such that treewidth of T' is bounded and there exists a relation NXT on vertices of G' satisfying the following:*

1. *NXT is compatible with tree decomposition T'.*
2. *NXT is a partial order on the vertices of G'.*
3. *NXT is computable in* L.
4. *The transitive closure NXT* is a total order when restricted to the vertices of G.*
5. *NXT* is expressible as an MSO-formula over the vocabulary of G' along with NXT.*

The order relation O we need is precisely the relation NXTdescribed in the lemma, since its transitive closure is a total order (same as linear order) on vertices of G. Therefore after adding the linear order O, the new structure $\mathcal{A}'$ also has bounded treewidth. After we compute the order O and store it on a transducer, we can query for any two vertices u, v if u is "lesser than" v in the transitive closure of O in logspace. This is because the Gaifman graph induced by this structure is of bounded treewidth, and so we can compute if the vertex v is reachable from u in the graph in logspace.

In order to quantify over sets of edges of G in MSO, we will also need to add the edges $E(G)$ to our universe of discourse, as well as an incidence relations : $tail(e,v)$ which is true iff the edge $e \in E(G)$ has vertex v as its tail, and $head(e,v)$, which is true iff the edge $e \in E(G)$ has vertex v as its head. It is easy to see that this augmentation to $\mathcal{A}'$ will also not blow up the treewidth. Let the augmented structure be $\mathcal{A}''$. For an edge $e = (u,v) \in G$, its addition to $|\mathcal{A}'|$, along with addition of relation tuples $tail(e,u), head(e,v)$ to $\mathcal{A}$, results in the following augmentation to $G_{\mathcal{A}'}$: a vertex corresponding to e, say w_e is added to $V(G_{\mathcal{A}'})$ and it is joined to the existing vertices u, v in $V(G_{\mathcal{A}'})$. Consider a tree decomposition T' of $G_{\mathcal{A}'}$ of minimum treewidth. For each new vertex w_e described above, add w_e to a bag in T' that contains both the vertices u, v. This is clearly a valid tree decomposition for $G_{\mathcal{A}''}$. Therefore the treewidth of $\mathcal{A}''$ is at most square of the treewidth of $\mathcal{A}'$, which is still a constant. In order to express an element of $|\mathcal{A}''|$ being a vertex or an edge of G, we will also need unary relations for the same. These however clearly do not add to treewidth of $G_{\mathcal{A}''}$, and we use phrases like $\exists v \in V \ldots, \exists e \in E \ldots$ as syntactic sugar. Symbols like u, v will generally be used for elements of universe of discourse that correspond to vertices of G and e for elements corresponding to edges of G.

We use $u <_O v$ to denote that vertex u is lesser than vertex v in the transitive closure of O. An ordering over vertices defines a unique lexicographically first DFS tree. Our algorithm will go over each edge of E, and use Theorem 2.15 to query if it belongs to the lex-first DFS tree of G, putting it on the output tape if it does. Along with outputting the edges, we also output the transitive closure of O on te output tape to specify the preference of vertices to traverse the tree. Our universe of discourse is $V \cup E$, and the relations on it given are $tail, head$, and the linear order O. To express the membership of an edge in the lex-first DFS tree with respect to O, we will use the theorem connecting the lex-first DFS tree to lex-first paths that we described in Section 4.2. We restate some of the machinery here for convenience.

We define lexicographic ordering on paths starting from a common vertex:

Definition 5.9. Let P_1, P_2 be two paths in G starting from r. We say that P_1 is lexicographically lesser than P_2 (with respect to O) if at the first point of divergence starting from r, P_1 diverges to a vertex that is lesser in the given ordering than the one P_2 diverges to. We denote this as $P_1 <_O^r P_2$.

This naturally defines the notion of the lexicographically least path starting from r and ending at a vertex v. We call it the lex-first path from r to v. We restate the following theorem of [dlTK95]:

Theorem 5.10. *[dlTK95] Let T_l be the lex-first DFS tree of G with respect to a given linear order O. Then $\forall v \in V(G)$, the unique path from r to v in T_l is exactly the lex-first path from r to v (with respect to O) in G.*

Thus in order to check if an edge (u,v) belongs to T_l (rooted at r), it suffices to check if it is the last edge of the lex-min path in G from r to v. We do this by expressing this property in MSO and using Theorem 2.15.

We will now define some formulae to construct the expression of the stated property.

For variables P, P_1, P_2 denoting sets of edges, we will define formulae with the following semantics:

$\phi_{edge}(v,u,P)$, which is true iff there exists an edge $e \in P$ such that $e = (u,v)$ and $e \in P$.

$\phi_{rpath}(P)$, which is true iff the edges in P form a single simple path with r as the starting point.

$\phi_{ep}(P, v)$, which true iff there exists exactly one in-neighbour of v in P, and no out-neighbour of v in P (i.e. v is an 'end point' in P).

$\phi_{lex}(P_1, P_2)$, which is true iff P_1, P_2 satisfy $\phi_{rpath}(P_1), \phi_{rpath}(P_2)$ respectively, and $P_1 <_O^r P_2$.

Using these formulae, we define $\phi_{DFS}(u, v)$, which is true iff the edge (u, v) is part of the lex-first DFS tree, T_l. We use the above theorem and express it as:

$$\begin{aligned}\phi_{DFS}(u, v) : \exists P_1(\phi_{edge}(v, u, P) \wedge \phi_{rpath}(P_1) \wedge \phi_{ep}(P_1, v) \wedge \\ \forall P_2(\phi_{rpath}(P_2) \wedge \phi_{ep}(P_2, v) \Rightarrow \phi_{lex}(P_1, P_2)))\end{aligned}$$

Thus the logspace algorithm is the following :
For each edge (u, v) in E, query $\phi_{DFS}(u, v)$ and add to output tape iff the query returns yes. A DFS numbering or order of traversal of the vertices of T_l can easily be obtained from T_l by doing an Euler tree traversal of T_l with respect to ordering O.

Now all that remains is to express the formulas $\phi_{edge}(v, u, P), \phi_{rpath}(P), \phi_{ep}(P, v), \phi_{lex}(P_1, P_2)$ in MSO.

For brevity, we will use some shorthands in formulae like $\exists! v \ldots$ for 'there exists exactly one v such that…', which are known to be expressible in MSO. We define some more formulae and construct ϕ_{rpath}, ϕ_{lex} with explanations below.

1. We first note that $\phi_{edge}(v, u, P)$ which is true iff edge $(u, v) \in P$, can be expressed as:

$$\phi_{edge}(v, u, P) : \exists e(tail(e, u) \wedge head(e, v) \wedge e \in P)$$

2. $\phi_{conn}(P)$, which is true iff the edges in P form a weakly connected graph. It is well known that $\phi_{conn}(P)$ as expressible in MSO so we skip the exact formula here.

3. $\phi_{sub}(P_1, P_2)$, which is true iff the set of edges of P_1 is a subset of set of edges of P_2, i.e., $\forall e(e \in P_1 \Rightarrow e \in P_2)$.

4. $\phi_{ep}(P, v)$ can be written as:

$$\phi_{ep}(P, v) : \exists! u(\phi_{edge}(v, u, P)) \wedge \nexists w(\phi_{edge}(w, v, P))$$

5. We can express $\phi_{rpath}(P)$ as :

$$\begin{aligned}\phi_{rpath}(P) :&\phi_{conn}(P) \wedge \exists! v(v \neq r \wedge \phi_{ep}(P, v)) \wedge \\ &\forall u(\exists e \in P(tail(e, u) \vee head(e, u)) \wedge \neg\phi_{ep}(P, u) \Rightarrow \\ &\exists! x, \exists! y(x \neq y \wedge \phi_{edge}(u, x, P) \wedge \phi_{edge}(y, u, P)))\end{aligned}$$

The last line says that for every vertex u in P that is not either of the end points r, v, it has exactly one in-neighbour and exactly one out-neighbour in P.

6. Using above, we can express $\phi_{lex}(P_1, P_2)$ by saying that it holds iff both $\phi_{rpath}(P_1), \phi_{rpath}(P_2)$ hold, and there exist subpaths P_1', P_2' of P_1, P_2 respectively starting from r, which diverge only in their last edges, with P_1' diverging to a vertex lesser in O than the vertex P_2' diverges to. We can express this as follows:

$$\begin{aligned}\phi_{lex}(P_1, P_2) :&\phi_{rpath}(P_1) \wedge \phi_{rpath}(P_2) \wedge \\ &\exists P', u_d, u_1, u_2(u_1 \neq u_2 \wedge \phi_{rpath}(P') \wedge \phi_{ep}(P', u_d) \wedge \phi_{sub}(P', P_1) \wedge \\ &\phi_{sub}(P', P_2) \wedge \phi_{edge}(u_1, u_d, P_1) \wedge \phi_{edge}(u_2, u_d, P_2) \wedge u_1 <_O u_2)\end{aligned}$$

This gives all expressions needed for $\phi_{DFS}(u, v)$.

Bibliographical Remarks The results presented in this chapter are based on joint work with Samir Datta, M. Praveen.

Chapter 6

The Even Path Problem in single-crossing-minor-free graphs

In this chapter we present our result of a polynomial time algorithm for the EvenPath problem in directed single-crossing-minor-free graphs. We restate the problem statement.

Definition 6.1. Let G be a digraph with vertices s, t in $V(G)$. The EvenPath problem is to check if there exists a (simple) path of even length from s to t in G, and to find one if it exists.

EvenPath was shown to be NP-complete by LaPaugh and Papadimitriou [LP84] via a reduction from an NP-complete problem, the Path-Via-A-Vertex problem. On the other hand, they also show in [LP84] that its undirected counterpart is solvable in linear time.

Nedev in 1999, showed that EvenPath in planar digraphs is polynomial-time solvable [Ned99]. More recently it was also shown in [Sha21] that EvenPath is in P for *single-crossing* digraphs.

We emphasize again that when talking about concepts like treewidth, minors of directed graphs, we intend them to apply to the underlying undirected graphs.

From here onwards, we will drop the term 'directed' and assume by default that the graphs we are referring to are directed, unless otherwise stated. Operations like clique sums, decomposing the graphs along separating triples, pairs, etc., will be applied on the underlying undirected graphs.

The following is the main theorem we prove in this chapter:

Theorem 6.2. *Given an H-minor-free graph G for any fixed single-crossing graph H, the EvenPath problem in G can be solved in polynomial time.*

We first apply the theorem of Robertson-Seymour (theorem 6.4), and decompose G using 3-clique sums into pieces that are either planar or of bounded treewidth.

Though EvenPath is tractable in planar graphs, and can also be solved in bounded treewidth graphs by Courcelle's theorem, straightforward dynamic programming does not yield a polynomial-time algorithm for the problem, as we will explain in subsequent sections. One of the technical ingredients that we develop to overcome the issues is that of *parity-mimicking networks*, which are graphs that preserve the parities of various paths between designated terminal vertices of the graph it mimics. We construct them for upto three terminal vertices.

For technical reasons, we require our parity mimicking networks to be of bounded treewidth *and* planar, with all terminals lying on a common face. One of our main contributions is to show (in Lemma 6.9) the construction of such networks, for upto three terminals.

We also come across a natural variant of another famous problem. Suppose we are given a graph G and vertices $s_1, t_1, s_2, t_2 \ldots s_k, t_k$ (we may call them terminals) in it. The problem of finding pairwise vertex disjoint paths, from each s_i to t_i is a well-studied problem called the disjoint paths problem. In planar graphs, it is known to be in P for fixed k [Sch94, CMPP13]. We consider this problem, with the additional constraint that the sum of lengths of the s_i-t_i paths must be of specified parity. We hereafter refer to the parity of the sum of lengths as total parity, and refer to the problem as DisjointPathsTotalParity. While DisjointPathsTotalParity can be solved for fixed k in bounded treewidth graphs using Courcelle's theorem [Cou90], we do not yet know if it is tractable in planar graphs, even for $k = 2$. The other main technical contribution of this chcapter is in lemma 6.11, where we show that in some special cases, i.e., when there are four terminals, three of which lie on a common face of a planar graph, DisjointPathsTotalParity can be solved in polynomial time for $k = 3$.

6.1 Notations and conventions

For a path P, and a pair of vertices u and v on P, such that u occurs before v in P, $P[u, v]$ denotes the subpath of P from u to v. If P_1's ending vertex is same as the starting vertex of P_2, then we denote the concatenation of P_1 and P_2 by $P_1.P_2$. We will use the numbers $0, 1$ to refer to *parities*, 0 for even parity and 1 for odd parity. We say a path P is of *parity* p $(p \in \{0, 1\})$, if its length modulo 2 is p. We refer the reader to Section 2.2 for pelims on graph theory, particularly to Section 2.2.4 for background on clique sum decompositions of single-crossing-minor-free graphs.

For convenience, we repeat some prelims from Chapter 2.

Definition 6.3. A k-clique-sum of two graphs G_1, G_2 can be obtained from the disjoint union of G_1, G_2 by identifying a clique in G_1 of at most k vertices with a clique of the same number of vertices in G_2, and then possibly deleting some of the edges of the merged clique.

The following is a theorem from [RS93].

Theorem 6.4 (Robertson-Seymour [RS93]). *For any single-crossing graph H, there is an integer τ_H such that every graph with no minor isomorphic to H is either*

1. *the proper* 0-, 1-, 2- *or* 3-*clique-sum of two graphs, or*
2. *planar*
3. *of treewidth* $\leq \tau_H$.

Thus, every H-minor-free graph, where H is a single-crossing graph, can be decomposed by 3-clique sums into graphs that are either planar or have treewidth at most τ_H. Polynomial time algorithms are known to compute this decomposition [DHN+04, KW11, GKR13]. The decomposition can be thought of as a two colored tree (see [DHN+04, CE13, EV21] for further details on the decomposition), where the blue colored nodes represent *pieces* (subgraphs that are either planar or have bounded treewidth), and the red nodes represent cliques at which two or more pieces are attached. We call these nodes of the tree decomposition as $piece\ nodes$ and $clique\ nodes$, respectively. The edges of the tree describe the incidence relation between pieces and cliques (see Fig. 6.2). We will denote this decomposition tree by T_G. We will sometimes abuse notation slightly and refer to a piece of T_G (and also phrases like *leaf*

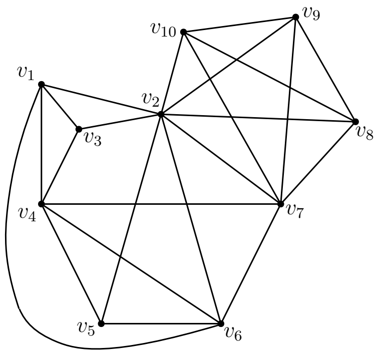


FIGURE 6.1: An example of a graph G. We ignore directions here.

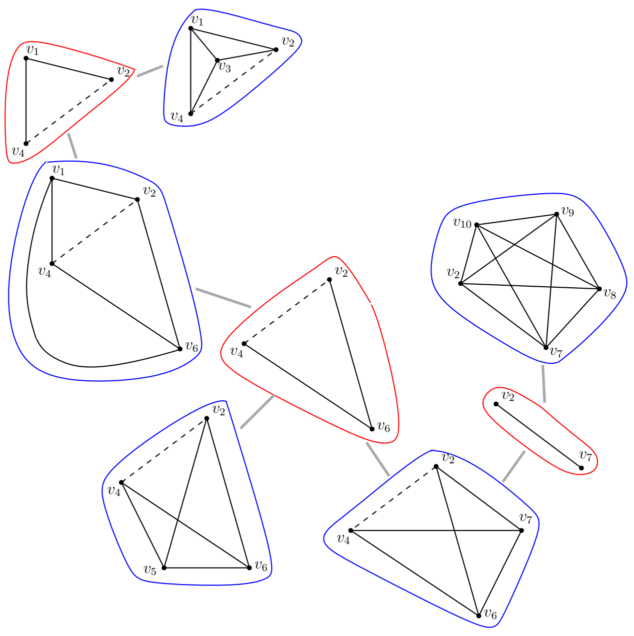


FIGURE 6.2: A clique sum decomposition of G. Red nodes are the clique nodes and blue node the piece nodes. Dashed edges denote virtual edges.

piece, *child* piece), when it is clear from the context that we mean the piece represented by the corresponding node of T_G. Note that the bounded treewidth and planarity condition on the pieces we get in the decomposition, is along with their virtual edges. As explained in [CE13, EV21], we can assume that in any planar piece of the decomposition, the vertices of a separating pair or triplet lie on a common face (Else we could decompose the graph further).

Suppose G decomposes via a 3-clique sum at clique c into G_1 and G_2. Then we write G as $G_1 \oplus_c G_2$. More generally, if $G_1, G_2, \ldots, G_\ell$ all share a common clique c, then we use $G_1 \oplus_c G_2 \oplus_c \ldots \oplus_c G_\ell$ to mean $G_1, G_2, \ldots, G_\ell$ are glued together at the shared clique. If it is clear from the context which clique we are referring to, we will sometimes drop the subscript and simply use $G_1 \oplus G_2 \oplus \ldots \oplus G_\ell$ instead. Suppose G'_2 is a graph that contains the vertices of the clique c shared by G_1 and G_2. We denote by $G[G_2 \to G'_2]$, the graph $G_1 \oplus_c G'_2$, i.e., replacing the subgraph G_2 of G, by G'_2, keeping the clique vertices intact. We will also use the notion of *snapshot* of a path in a subgraph. If G can be decomposed into G_1 and G_2 as above, and P is an s-t path in G, its snapshot in G_1 is the set of maximal subpaths of P, restricted to vertices of G_1. Within a piece, we will sometimes refer to the vertices of separating cliques, and s and t, as $terminals$.

In figures, we will generally use the convention that a single arrow denotes a path segment of odd parity and double arrow denotes a path segment of even parity, unless there is an explicit expression for the parity mentioned beside the segment.

6.2 Overview and Technical Ingredients

We first compute the 3-clique sum decomposition tree of G, T_G. We can assume that s, t, each occur in only one piece of T_G, S and T, respectively.[1] We call the pieces S and T, along with the pieces corresponding to nodes that lie in the unique path in T_G joining S and T, as the $main$ pieces of T_G, and the remaining pieces are called the $branch$ pieces of T_G. We will assume throughout that T_G is rooted at S.

The high level strategy of our algorithm follows that of [CE13]. The algorithm has two phases. In the first phase, we simplify the branch pieces of the decomposition tree. Any s-t even path P must start and end inside the main pieces S and T, respectively. However, it may take a detour into the branch pieces. Suppose L is a leaf branch piece of T_G, attached to its parent piece, say G_i, via a 3-clique c. Using Nedev's algorithm or Courcelle's theorem, we can find paths of various parities between vertices of c in L, which constitutes the *parity configuration* of L with respect to c (formally defined in next subsection). We will replace L by a parity mimicking network of L with respect to vertices of c, L'. L' will mimic the parity configuration of L and hence preserve the parities of all s-t paths of original graph. The parity mimicking networks we construct are small and planar, with the terminals (vertices of c) all lying on a common face, as decribed in lemma 6.9. Therefore, if G_i is of bounded treewidth, then $G_i \oplus L'$ will be of bounded treewidth. And if G_i is planar, then we can plug L' in the face of G_i that is common to vertices of c, and $G_i \oplus L'$ will be planar. This allows us compute the parity configurations of the merged piece, and repeat this step until a single branch, i.e. a path, remains in the decomposition tree, consisting only of the main pieces (connected by cliques), including S and T.

In the second phase, we start simplifying the main pieces, starting with the leaf piece T. Instead of a single mimicking network for T, we will store a set of small networks, each of them mimicking a particular snapshot of a solution. We call them *projection networks*. Since a snapshot of an s-t even path in T can possibly be a set of disjoint paths between the (upto) four terminals in T, we require the DisjointPathsTotalParity routine of lemma 6.11 to compute these projection networks. We combine the parent piece with each possible projection network. The merged piece will again be either planar or of bounded treewidth, allowing us to continue this operation towards the root node until a single piece containing both s and t remains. We query for an s-t even path in this piece and output yes iff there exists one. At each step, the number of projection networks used to replace the leaf piece, and their combinations with its parent piece will remain bounded by a constant number.

Once we have the decision version of EvenPath, we show a poly-time self-reduction using the decision oracle of EvenPath to construct a solution.

Necessity of a two phased approach We mention why we have two phases and different technical ingredients for each.

- Instead of a single parity mimicking network, we need a set of projection networks for the leaf piece in the second phase because it can have upto four terminals (three vertices of the separating clique and the vertex t), and we do not yet know how to find (or even the

[1]If they are part of a separating vertex/pair/triplet then they may occur in multiple pieces of T_G. Say s is a part of many pieces in T_G. To handle that case, we can introduce a dummy s' and add an edge from s' to s and reduce the problem to finding an odd length path from s' to t. The vertex s' now will occur in a unique piece in T_G. Vertex t can be handled similarly.

existence of) parity mimicking networks with the constraints we desire, for graphs with four terminals.

- We cannot however use a set of networks for each piece in phase I because of the unbounded degree of T_G. Suppose a branch piece G_i is connected to its parent piece by clique c, and suppose G_i has child pieces $L_1, L_2, \ldots, L_\ell$, attached to G_i via disjoint cliques $c_1, c_2, \ldots, c_\ell$, respectively. An even s-t path can enter G_i via a vertex of c, then visit any of $L_1, L_2, \ldots, L_\ell$ in any order and go back to the parent of G_i via another vertex of c. If we store information regarding parity configurations of $L_1, L_2, \ldots, L_\ell$ as sets of projection networks, we could have to try exponentially many combinations to compute information of parity configurations between vertices of c in the subtree rooted at G_i (note that ℓ could be $O(n)$). Therefore, we compress the information related to parity configurations of L_i into a single parity mimicking network L'_i, while preserving solutions, so that the combined graph $(((G_i \oplus_{c_1} L'_1) \oplus_{c_2} L'_2 \ldots) \oplus_{c_\ell} L'_\ell)$ is either planar or of bounded treewidth.

We will now describe these ingredients formally in the remaining part of this section.

6.2.1 Parity Mimicking Networks

We first define the parity configuration of a graph, which consists of subsets of $\{0, 1\}$ for each pair, triplet of terminals, depending on whether there exists 'direct' or 'via' paths of parity even, odd, or both (we use 0 for even parity and 1 for odd). We formalise this below.

Definition 6.5. Let L be a directed graph and $T(L) = \{v_1, v_2, v_3\}$ be the set of terminal vertices of L. Then, $\forall i, j, k \in \{1, 2, 3\}$, such that i, j, k are distinct, we define the sets *Dir*$_L(v_i, v_j)$, and *Via*$_L(v_i, v_k, v_j)$ as:

- *Dir*$_L(v_i, v_j)$ $=\{p \mid$ there exists a path of parity p from v_i to v_j in $L - v_k$ $\}$
- *Via*$_L(v_i, v_k, v_j)$ $=\{p \mid$ there exists a path of parity p from v_i to v_j via v_k *and* there does not exist a path of parity p from v_i to v_j in $L - v_k\}$

We say that the *Dir*$_L$, *Via*$_L$ sets constitute the *parity configuration* of the graph L with respect to $T(L)$. We call the paths corresponding to elements in *Dir*$_L$, *Via*$_L$ sets as *Direct paths* and *Via paths*, respectively.

The parity configuration of a graph can be visualised as a table. We have defined it for three terminals, it can be defined in a similar way for two terminals. It is natural to ask the question that given a parity configuration $\mathcal{P}$ independently with respect to some terminal vertices, does there exist a graph with those terminal vertices, realising that parity configuration. If not, we say that $\mathcal{P}$ is unrealisable. It is easy to see that the number of parity configurations for a set of three terminals is bounded by 4^{12}, many of which are unrealizable. We now define parity mimicking networks.

Definition 6.6. A graph L' is a parity mimicking network of a another graph L (and vica versa), if they share a common set of terminals, and have the same parity configuration, $\mathcal{P}$, w.r.t. the terminals. We also call them parity mimicking networks of parity configuration $\mathcal{P}$.

The reason we differentiate between direct paths and via paths while defining parity configurations is to ensure that no false solutions are introduced on replacing a leaf piece of T_G

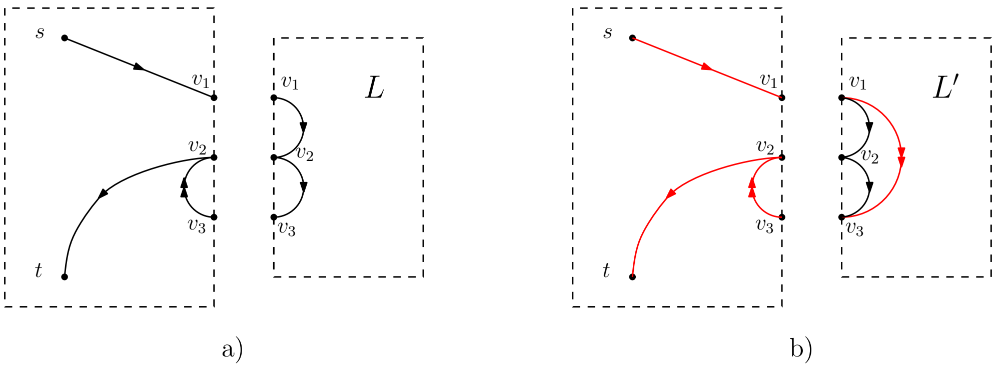


FIGURE 6.3: Figure a) shows the input graph and b) shows the graph with L replaced by an erroneous mimicking network L'. Suppose the original graph in a) has no s-t even path but does have an s-t even walk as shown in the figure, using vertex v_2 twice. If we query for a path from v_1 to v_3 in L, and add a direct v_1 to v_3 path of that parity in L', we end up creating a false solution since v_2 is freed up to be used outside L'. Hence there must be equality between corresponding direct sets.

by its mimicking network (see Fig. 6.3). Note that in our definition of *Via* sets, we exclude parity entries of via paths between two terminals if that parity is already present in *Dir* set between the same terminals. We do so because this makes the parity configurations easier to enumerate in our construction of parity mimicking networks. In Fig. 6.4, we describe why doing this will still preserve solutions.

We also need to consider the case where multiple leaf pieces, in T_G are attached to a common parent piece via a shared clique (as seen in Fig. 6.2). In this case, we will replace the entire subgraph corresponding to the clique sum of the sibling leaf pieces by one parity mimicking network. To compute the parity configuration of the combined subgraph of leaf pieces, we make the following observation:

Observation 6.7. *Let $L_1, L_2, \ldots, L_\ell$ be leaf branch pieces that are pairwise disjoint except for a common set of terminal vertices, say $\{v_1, v_2, v_3\}$. Let $L = L_1 \oplus L_2 \oplus \ldots \oplus L_\ell$. Then the parity configuration of L with respect to $\{v_1, v_2, v_3\}$ can be computed by:*

$$\mathrm{Dir}_L(v_i, v_j) = \bigcup_{a=1}^{\ell} \mathrm{Dir}_{L_a}(v_i, v_j) \tag{6.1}$$

$$\mathrm{Via}_L(v_i, v_k, v_j) = \left(\bigcup_{a=1}^{\ell} \mathrm{Via}_{L_a}(v_i, v_k, v_j) \cup \bigcup_{a,b=1}^{\ell} (\mathrm{Dir}_{L_a}(v_i, v_k) \boxplus \mathrm{Dir}_{L_b}(v_k, v_j)) \right) \setminus \mathrm{Dir}_L(v_i, v_j) \tag{6.2}$$

where $A \boxplus B$ denotes the set formed by addition modulo 2 between all pairs of elements in sets A, B, and $i, j, k \in \{1, 2, 3\}$ are distinct.

The intuition behind the observation is simple. Any direct path in L from v_i to v_j must occur as a direct path in one of $L_1, L_2 \ldots L_\ell$ since they are disjoint except for terminal vertices. Any via path in L from v_i to v_j via v_k can occur in two ways, either as a v_i-v_k-v_j via path in one of $L_1, L_2 \ldots L_\ell$, or as a concatenation of two direct paths, one from v_i to v_k in some piece L_i, and another from v_k to v_j in another piece L_j. Note that although the observation is for the case when all $L_1, L_2, \ldots, L_\ell$ share a common 3-clique $\{v_1, v_2, v_3\}$, it is easy to see it can

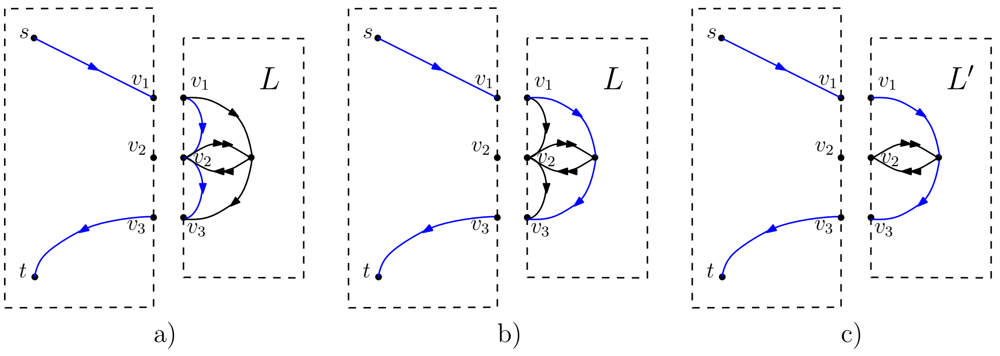


FIGURE 6.4: Figure a) denotes the original graph which has both a direct path, as well as a via path of even parity from v_1 to v_3. Suppose the via path is part of an even s-t path solution, as marked by blue. Then in L itself, we could replace the via path by the direct path and it would still be a valid even s-t path, as marked in blue in b). Hence in the mimicking network L', too (shown in c)), we could use the direct v_1 to v_3 path of the same parity. Therefore we do not need to put the parity of the v_1-v_2-v_3 path in $Via_L(v_1, v_2, v_3)$, since the same parity is already present in $Dir_L(v_1, v_3)$, and $Dir_L(v_1, v_3) = Dir_{L'}(v_1, v_3)$.

be tweaked easily to handle the cases when some of the $L_i's$ are attached via a 2-clique that is a subset of the 3-clique.

The next lemma states that replacing leaf piece nodes in T_G by parity mimicking networks obeying some planarity conditions, will preserve the existence of s-t paths of any particular parity, and also preserve conditions on treewidth and planarity for the combined piece.

Lemma 6.8. *Let G be a graph with clique sum decomposition tree T_G, and let $L_1, L_2 \dots, L_\ell$ be set of leaf branch pieces of T_G, attached to their parent piece G_1 via a common clique c. Let L' be a parity mimicking network of $L_1 \oplus L_2 \oplus \dots L_\ell$ with respect to c, such that L' is planar, and vertices of c lie on a common face in L'. Then:*

1. *There is a path of parity p from s to t in G iff there is a path of parity p from s to t in $G[L_1 \oplus L_2 \oplus \dots L_\ell \to L']$.*
2. *If G_1 is planar, then $G_1 \oplus L'$ is also planar.*
3. *If G_1 has treewidth τ_H, and L' has treewidth $\tau_{L'}$, then $G_1 \oplus L'$ has treewidth max($\tau_H, \tau_{L'}$)*

Proof. 1. The proof essentially follows from the definition of parity mimicking networks and observation 6.7, since we can replace the snapshot of any s-t path P in L_i by a path of corresponding parity in L'_i and vice-versa.

2. This follows since in the decomposition, vertices of separting cliques in every piece lie on the same face, and so is the case for L' by assumption. Therefore we can embed L' inside the face in G_1, on the boundary of which v_1, v_2, v_3 lie.

3. This follows since we can merge tree decompositons of G_1, L' along bags consisting of the common clique.

□

Now we will show how to compute parity mimicking networks that are small in size (and hence of bounded treewidth), and also planar, with terminal vertices lying on the same face, for a given parity configuration of a graph L.

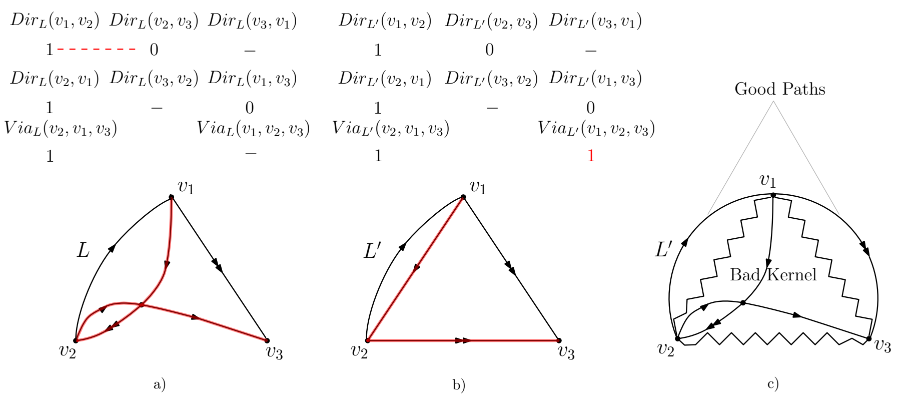

$Dir_L(v_1,v_2)$ $Dir_L(v_2,v_3)$ $Dir_L(v_3,v_1)$
1 0 –
$Dir_L(v_2,v_1)$ $Dir_L(v_3,v_2)$ $Dir_L(v_1,v_3)$
1 – 0
$Via_L(v_2,v_1,v_3)$ $Via_L(v_1,v_2,v_3)$
1 –
$Dir_{L'}(v_1,v_2)$ $Dir_{L'}(v_2,v_3)$ $Dir_{L'}(v_3,v_1)$
1 0 –
$Dir_{L'}(v_2,v_1)$ $Dir_{L'}(v_3,v_2)$ $Dir_{L'}(v_1,v_3)$
1 – 0
$Via_{L'}(v_2,v_1,v_3)$ $Via_{L'}(v_1,v_2,v_3)$
1 1
Good Paths
Bad Kernel
v_1
v_2
v_3
L
L'
a)
b)
c)

FIGURE 6.5: Fig a) denotes a graph L with its parity configuration table (only relevant sets). Fig b) denotes a 'parity mimicking network', if for each pair of terminals, we just independently put paths of correct parity, disjoint from each other. It leads to an extra path (highlighted in red) from v_1 to v_3 via v_2 in L', of odd parity. Pairs of such entries, for which we cannot add disjoint paths are called bad entries as marked by the dashed red line in the parity configuration table in a). Fig c) outlines the approach used to construct the correct mimicking network. The two paths corresponding to bad pair entries, form the bad kernel, for which we construct a mimicking network by enumerating cases. The remaining paths can be added iteratively, disjoint from all existing paths, on the outer face.

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
0	0	0
1	1	1

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
1	–	0

$Via_{\mathcal{P}}(v_3,v_1,v_2)$
0
1

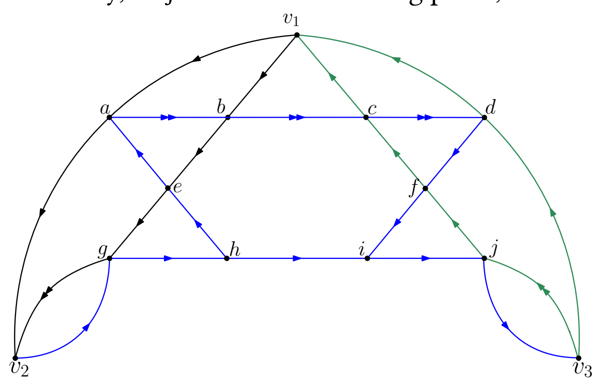


FIGURE 6.6: An example of a more non-trivial bad kernel, and a mimicking network realising its closure. This is a subcase of case $(4,0)$ described in the full proof. We give a list of paths along with their lengths, for ease of reader to check that the network obeys the parity configuration.

Lemma 6.9. *Suppose L is a graph with terminals $T(L) = \{v_1, v_2, v_3\}$, and suppose we know the parity configuration of L with respect to $\{v_1, v_2, v_3\}$. We can in polynomial-time, find a parity mimicking network L' of L, with respect to $\{v_1, v_2, v_3\}$ which consists of at most 18 vertices, and is also planar, with v_1, v_2, v_3 lying on a common face.*

Proof. We give a brief idea of the proof and defer the full proof to Appendix A. As noted above, the number of possible parity configurations are finite (bounded by 4^{12} for three terminals), but the number is too large to enumerate over all of them and individually construct the mimicking networks. We use some observations to make the case analysis tractable. We refer to elements of sets $Dir_L(v_i, v_j)$ as *entries*. But we abuse notation slightly and distinguish them from the boolean values $0, 1$. For example, we always distinguish between an entry of $Dir_L(v_1, v_2)$, and an entry of $Dir_L(v_2, v_3)$, even if they have the same *value* (0 or 1). A natural constructive approach would be to iteratively do the following step for all i, j, p : add a path of length $2 - p$ from v_i to v_j in L', disjoint from existing paths of L', if there is an entry of parity p present in $\text{Dir}_{L'}(v_i, v_j)$. Its easy to check that this will result in a planar L' with terminals on a common face. However, this could lead to wrong parity configurations in L'. For example, L could have a direct paths of parity 1 from v_1 to v_2, and a direct path of parity 0 from v_2 to v_3, but no path of parity 1 from v_1 to v_3, either direct or via v_2 (see Fig. 6.5). We will call pairs of such entries as *bad pairs*. The entries that are part of *any* bad pair are called *bad entries*. Though the example in Fig. 6.5 has a simple fix for the bad pair, it becomes more complicated to maintain the planarity conditions as the number of bad pairs increase. Let $\mathcal{P}_L$ be the parity configuration of L. The idea of the proof is to define a *bad kernel* of $\mathcal{P}_L$, as the *sub-configuration* consisting of all the *bad entries* of $\mathcal{P}_L$. The *closure* of the bad kernel is defined as the parity configuration obtained from it by adding 'minimal' number of entiries to make it realizable. We observe that the closure remains a sub-configuration of $\mathcal{P}_L$. Suppose that we can somehow construct a planar mimicking network for the *closure* of the *bad kernel* of $\mathcal{P}_L$, with terminals lying on a common face. Then we show that paths corresponding to leftover parity entries of L can be safely added using the constructive approach described above. Hence it suffices to construct parity mimicking networks for closures of all possible bad kernels. We use some observations to show that the number of possible types of bad kernels cannot be too large, and enumerate over each type, explicitly constructing the parity mimicking networks of their closures.

□

6.2.2 Disjoint Paths with Parity Problem

In this section, we will define and solve the DisjointPathsTotalParity problem for some special cases and types of graphs. We define the problem for three paths between four terminals.

Definition 6.10. Given a graph G and four distinct terminals v_1, v_2, v_3, and v_4 in $V(G)$, the DisjointPathsTotalParity problem is to find a set of three pairwise disjoint paths, from v_1 to v_2, v_2 to v_3, and from v_3 to v_4, such that the total parity is even, if such a set of paths exist, and output no otherwise.

The problem where total parity must be odd can be easily reduced to this by adding a dummy neighbour to v_4. The problem is NP-hard in general graphs since the even path problem trivially reduces to this. We show that the above problem can be solved in polynomial time in following two cases:

Lemma 6.11. *Let G be a graph, and v_1, v_2, v_3, v_4 be four vertices of G. Both decision as well as search versions of DisjointPathsTotalParity for these vertices as defined above can be solved in polynomial time in the following cases:*

1. *If G has constant treewidth.*
2. *If G is planar and v_1, v_2, v_3 lie on a common face of G.*

Proof. (Of Part 1)
Courcelle's theorem ([Cou90]) states that every graph property definable in the monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth. The vocabulary of our logic consists of a unary relation symbol V for the set of vertices, a binary relation symbol E for the set of edges, and some constant symbols, like s, t used to denote source, destination vertices. The universe of discourse consists of vertices. We use the fragment of monadic second order logic called MSO_2 which allows quantification over sets of

vertices and edges but not more complex relations. We use variables A, B, P, P_1, P_2 to denote sets of vertices, and variable M to denote set of edges. For brevity, we use some shorthand notation for phrases like $\exists! x$ for 'there exists a unique x', $x \in P$ for 'a vertex x which is in vertex set P', $x \in M$, for 'a vertex x which is in the set of vertices consisting of end points of the edges in set M.' It is well known that these phrases can be easily expressed in MSO_2. We also use $V(M)$ to denote the set of vertices consisting of all end points of the edges in M. Consider the formula, $\phi_{disj}(P_1, P_2)$ which is true iff the sets P_1, P_2 are disjoint. We can write it in MSO as:

$$\phi_{disj}(P_1, P_2) : \forall v, v \in P_1 \Rightarrow v \notin P_2$$

We define a formula $\phi_{part}(P, A, B)$, which is true iff the vertex sets A, B form a partition of the vertex set P, as:

$$\phi_{part}(P, A, B) : (\forall v(v \in P \Leftrightarrow (v \in A \vee v \in B))) \wedge (\phi_{disj}(A, B))$$

The definition $\phi_{part}(P, A, B)$ can be tweaked to get $\phi_{part}(M, A, B)$ which is true iff vertex sets A, B for a partition of the vertex consisting of endpoints of edge set M.

Now we define a formula $\phi^e_{s,t}(P)$ which is true iff the graph induced on vertex set P has a path from s to t of total length even. It is shown in [hh17] that this can be expressed in MSO_2. The idea is to express that there exists a set of edges M in the graph induced by P, whose vertices can be partitioned into vertex sets (A, B), such that:

- Every edge of M has one end point in A, and other in B. Both s, t lie in A.
- s has only one neighbour in M, an out-neighbour, and t has only one neighbour in M, an in-neighbour.
- Every other vertex of M has exactly one out-neighbour and one in-neighbour.

It is clear that the graph induced by set of edges M in P must be a collection of disjoint cycles and an s-t path. Since the graph induced by M is also bipartite, with s, t lying in the same partition, it follows that the disjoint cycles, as well as the s-t path must be of even length.

Thus we can write the formula for $\phi^e_{s,t}(P)$ as:

$$\begin{aligned}
&\exists M((V(M) \subseteq P), \exists A, B(\phi_{part}(M, A, B) \wedge \\
&\quad (((u, v) \in M) \Rightarrow ((u \in A \wedge v \in B) \vee (u \in B \wedge v \in A))) \wedge (s, t \in A)) \ \wedge \\
&\ ((\exists v(s, v) \in M) \wedge (\nexists v(v, s) \in M) \wedge (\exists v((v, t) \in M)) \wedge (\nexists v((t, v) \in M))) \ \wedge \\
&\ (\forall u \in M, (\exists! u \in M, (u, v) \in M) \wedge (\exists! v \in M, (v, u) \in M)))
\end{aligned}$$

We can similarly define $\phi^o_{s,t}(P)$ which is true iff graph induced on P has an s-t path of odd length. With $\phi^e_{s,t}(), \phi^o_{s,t}()$ and $\phi_{disj}(,)$, we can easily define a formula $\phi^e_{s_1,t_1,s_2,t_2}$, which is true iff the graph has two disjoint paths from s_1 to t_1, and from s_2 to t_2, with total parity even as follows (we skip argument for brevity):

$$\begin{aligned}
\phi^e_{s_1,t_1,s_2,t_2} : \exists P_1, P_2(\phi_{disj}(P_1, P_2) \wedge ((\phi^e_{s,t}(P_1) \wedge \phi^e_{s,t}(P_2)) \vee \\
(\phi^o_{s,t}(P_1) \wedge \phi^o_{s,t}(P_2))))
\end{aligned}$$

We can define a similar formula as above for the case when total parity must be odd, as well as extend it for three disjoint paths with total parity constraints.

□

Proof. (Of part 2) We first give a high level idea of the proof of the second part. The argument of Nedev for EvenPath uses two main lemmas. One lemma states that if there are two paths P_1, P_2, of different parities from s to t, then their union forms a (at least one) structure, which they call an *odd list superface.* It (roughly) consists of two internally disjoint paths of different parities, with a common starting vertex, say b and a common ending vertex, say e. Let F denote such a superface. They show that there exist two disjoint paths in $P_1 \cup P_2 - F$, one from s to b, and one from e to t. This provides a 'switch' in $P_1 \cup P_2$, and if we can find this switch efficiently, then we can use existing 2-disjoint path algorithms to connect s and t via this switch. But the number of odd list superfaces in a graph can be exponential. The second lemma of Nedev says that we can exploit the structure of planarity and show that each of the odd list superfaces formed by $P_1 \cup P_2$, 'contain' a 'minimal' odd list superface, which they call a *simple* odd list superface, that obeys the same conditions. The set of simple odd list superfaces is small and can be enumerated in polynomial time. In our setting, we start from the case that two instances of three disjoint paths between the specified terminals exist, such that they have different total parity. Say the instances are P_1, P_2, P_3, and P_1', P_2', P_3'. At least one of P_i, P_i' must be of different parity. We show that using the constraints of three terminals on a face, and using ideas of *leftmost* (and rightmost) paths of $P_i \cup P_i'$, for each case of $i \in \{1, 2, 3\}$, there does exist an analogous structure: a simple odd list super face, and four disjoint path segments connecting the required vertices. A point to note is that in Nedev's argument, *any* odd list superface formed by P_1, P_2 could be trimmed to a simple odd list superface that would give a valid solution. That does not hold true here. We generalise their lemma, and argue that there does exist *at least one* odd list superface between P_i, P_i' that will work in our setting. Now we formally prove the lemma.

We first reiterate some of the machinery developed in [Ned99]. We give the definitions of *list superface* and *simple list superface* in planar graphs as defined in [Ned99].

Definition 6.12. Let G be a planar graph and G' be its planar embedding. Let P and P' be two paths in G, such that (1) both P and P' start from the same vertex, say b, and end at the same vertex, say e, and (2) P and P' are vertex disjoint except at b and e. Then, we call P and P' together with their interior region (the finite region enclosed by P and P') or exterior region (the region on the plane apart from the interior), a list superface.

Note that P and P' as described above can form two list superfaces, one with the interior region on the plane and one with the exterior region on the plane with respect to the boundary formed by them.

Definition 6.13. A list superface F is called a simple list superface if for every directed path $Q = (q_1, q_2, \ldots, q_k)$, where q_1 and q_k lie on the boundary of F and $q_2, q_3, \ldots, q_{k-1}$ lie on the region of F, there exists a directed path from q_k to q_1 using only a subset of the edges of the boundary of F.

As noted earlier, if we can solve the above problem for total parity even, we can also find if such paths of total parity odd exist, by adding a dummy out neighbour v_4' of v_4 and finding disjoint paths of total parity even from $v_1$to v_2, v_2 to v_3 and from v_3 to v_4'. Hence we assume that we want to find disjoint paths with total parity even.

We first find a set of pairwise disjoint paths from v_1 to v_2, v_2 to v_3 and from v_3 to v_4(if it exists) using [Sch94], but the total parity might be odd (If no such paths exist, we output negative, and if we get an instance of paths of total parity even then we output that instance). Suppose a solution exists, and let P_1, P_2, P_3, be one set of pairwise disjoint paths from $v_1$to

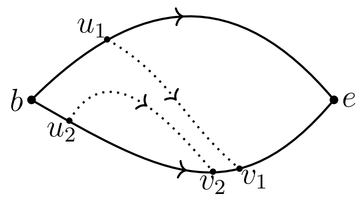


FIGURE 6.7: A list superface

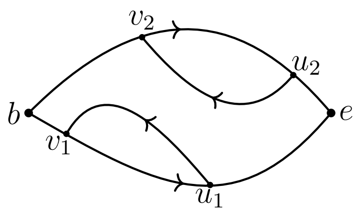


FIGURE 6.8: A simple list superface

FIGURE 6.9: List superface in **(a)** is non-simple because of paths from u_2 to v_2 and u_1 to v_1. On the other hand, **(b)** is a simple list superface.

v_2, v_2 to v_3 and from v_3 to v_4 respectively, of total parity even. Let P_1', P_2', P_3' be another set of pairwise disjoint paths of total parity odd between same corresponding vertices. Let i, j, k refer to distinct integers from the set $\{1, 2, 3\}$ hereafter. At least one of the pairs P_i, P_i' must have different parities, otherwise the total parities of both sets would be the same. Suppose P_1, P_1' are of different parities.

Using this we are going to extend the ideas of [Ned99] and show the following lemma:

Lemma 6.14. *Let G be a plane graph and v_1, v_2, v_3, v_4 vertices such that v_1, v_2, v_3 lie on the same face. Consider two instances of disjoint paths (P_1, P_2, P_3) and (P_1', P_2', P_3') between v_1, v_2, v_3, v_4 as described above, of different total parities, and let P_1, P_1' have different parities. Such a pair of instances exists iff a structure consisting of the following exists :*

- *A simple odd list superface F with starting and ending vertices b, e and paths Q_F, Q_F' (of different parities) as boundaries of F.*
- *Paths Q_b from v_1 to b, Q_e from e to v_2, Q_2 from v_2 to v_3, and Q_3 from v_3 to v_4.*
- *$Q_F, Q_F', Q_b, Q_e, Q_2, Q_3$ are all pairwise disjoint (except at vertices b, e).*

Similar claim with appropriate changes also holds if paths P_2, P_2' are of different parities or if paths P_3, P_3' are of different parities.

The odd list superface will act as a switch for parities. This gives a recipe, similar to [Ned99] to find a solution in polynomial time. We can enumerate over all simple odd list superfaces as done in [Ned99] and use the algorithm of [Sch94] to find the four pairwise disjoint paths described above in G, after removing boundary vertices of the simple odd list superface. If such an odd list superface and the corresponding paths exist, it gives a solution, else we output that no solution exists. One direction of the claim is straightforward: If there exists a simple odd list super face F and paths Q_b, Q_e, Q_2, Q_3 as described above, then the set of paths $(Q_e.(\text{one of } Q_F, Q_F').Q_b, Q_2, Q_3)$ form two instances of disjoint paths of different parities. In order to prove the other direction of the claim, we note some definitions and observations.

Let G be a plane graph and v_1, v_2 be two vertices on the outer face, and P_1, P_1' be two paths from v_1 to v_2. By Jordan curve theorem, each path P_1, P_2 divides G into two regions. If P is a directed path starting and ending on a closed boundary, we call the region on the left of P as $left(P)$ and the region along the right as $right(P)$.

Let $P_1 \cup P_1'$ denote the subgraph of G formed by vertices and edges of paths P_1, P_1'. The leftmost undirected walk or the of subgraph $P_1 \cup P_1'$, denote by ${P_1}^L$, is defined as the undirected walk obtained by traversing the clockwise first edge(ignoring direction) of $P_1 \cup P_1'$ with respect to the parent edge at every step, until it reaches v_2 or starts repeating. For the first step, we can use an imaginary dart from v_1 into the outer face as parent edge for reference. The rightmost walk of $P_1 \cup P_1'$, denoted by ${P_1}^R$ is defined similarly by taking the counter-clockwise first edge at every step. We show some lemmas using these paths.

Claim 6.15. *Suppose v_1, v_2 lie on outer face of G and P_1, P_1' are paths from v_1 to v_2.*

1. *The undirected walks ${P_1}^L, {P_1}^R$ are simple, directed paths from v_1 to v_2.*

2. *Let P_2 be a path disjoint from P_1 and lying (strictly) in $right(P_1)$. Let P_2' be a path disjoint from P_1', lying (strictly) in $right(P_1')$. Then the path ${P_1}^L$ is disjoint from both P_2, P_2', and lies in $left(P_1) \cap left(P_1')$.*

Proof. 1. We first show that ${P_1}^L$ is a simple path ignoring directions.
Let ${P_1}^L = \langle v_1, \dots y, x, z \dots w, x, \dots \rangle$, where x is the first vertex on ${P_1}^L$ that repeats in the walk. Let C denote the subcycle $\langle x, z \dots w, x \rangle$ of ${P_1}^L$. Suppose the edge (y, x) belongs to P_1 (See Fig. 6.10). The path $P_1[x, v_2]$ cannot touch C at any point other than x, since v_2 lies on the exterior of C and any edge touching C from exterior at any point other than x would condradict the assumption that C was obtained in the leftmost walk. But if it touches C at x only, then edges $(x, z), (w, x)$ must both belong to P_1' and by similar argument they would have to leave C contradicting the assumption that C occured in a leftmost walk. Therefore ${P_1}^L$ is a simple undirected path. Next we show that ${P_1}^L$ is actually a directed path from v_1 to v_2. Suppose that is not the case, which means it consists of segments that are alternately directed towards or away from v_2. Let x, y be the first maximal segment directed away from v_2(i.e. ${P_1}^L = \langle v_1, \dots y, \dots x_1, x, x_2 \dots v_2 \rangle$, where all arcs in the segment from v_1 till y are directed towards v_2, all arcs in segment from y till x are directed away from v_2, and the arc (x, x_2) is directed towards v_2). Without loss of generality, let the arc (x, x_1) be a part of P_1, which implies the incoming arc to x in P_1 is not on ${P_1}^L$ and lies in $right({P_1}^L)$. Consider the closed loop formed by segments $P_1[v_1, x], {P_1}^L[x, y], {P_1}^L[v_1, y]$. Since P_1 goes to x_1 after x, and v_2 lies on the exterior of the loop, P_1 must exit the loop after x_1 to reach v_2. But cannot exit via that segment since that would contradict that ${P_1}^L[v_1, y]$ is a subpath of the lefmost undirected path ${P_1}^L$. Therefore every arc in ${P_1}^L$ must be directed from v_1 to v_2.

2. This follows easily from definition of ${P_1}^L$.

□

Now we show a tweaked version of Lemma 2, and restate Lemma 1 of [Ned99].

Lemma 6.16. 1. *(Extension of Lemma 2 of [Ned99]) Let P_1, P_1' be paths as defined above, of different parity. Let ${P_1}^L$ and P_1 be of different parities. Then there exists an odd list superface F in ${P_1}^L \cup P_1$ with begining and ending vetices b, e respectively with boundaries ${P_1}^L[b, e], P_1[b, e]$, and paths Q_b, Q_e from v_1 to b and e to v_2 respectively such that Q_b, Q_e and F are pairwise disjoint(except at some end points). Moreover $Q_b = {P_1}^L[v_1, b], Q_e = {P_1}^L[e, v_2]$.*

2. *(Reiteration of Lemma 1 of [Ned99])If the odd list superface F found above in G is not simple, we can find a simple odd list superface F_s which is contained inside the region of F, and extend Q_b to Q_b', from v_1 to beginning of F_s, and Q_e to Q_e', from ending of F_s to v_2, such that the extensions are also contained inside F and F_s, Q_b', Q_e' are all pairwise disjoint.*

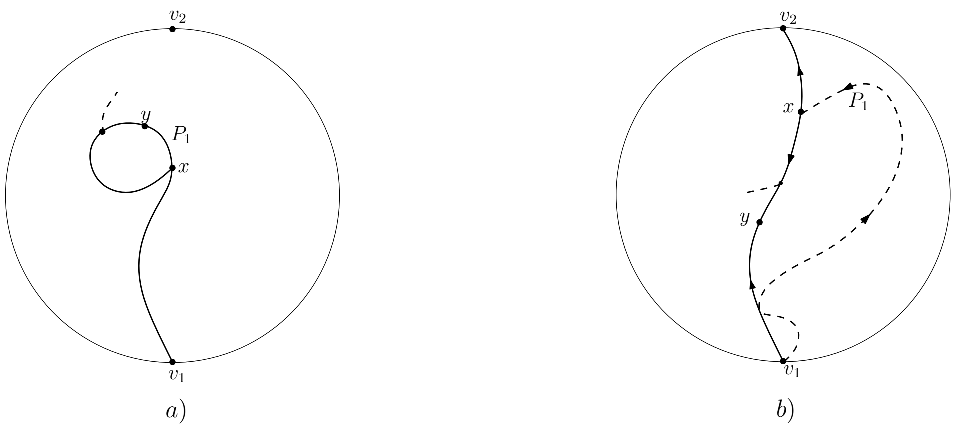


FIGURE 6.10: Suppose the undirected leftmost walk from v_1 in $P_1 \cup P_1'$ ends up in the cycle shown in a). Suppose the edge (x, y) (undirected) belongs to P_1. Then one of the (undirected) segments $P_1[v_1, x], P_1[x, v_2]$ must have an edge touching the cycle from its exterior as shown by the dashed path. This would contradict the cycle being part of the leftmost walk. Therefore the leftmost walk ${P_1}^L$ must be a path. In figure b), suppose the leftmost path ${P_1}^L$, drawn in bold, has a segment from x to y directed away from v_2. Suppose the outgoing edge of x in this segment belongs to P_1. Then $P_1[v_1, x]$ is a subpath of P_1 lying in $right({P_1}^L)$, forming a closed loop with the undirected segment ${P_1}^L[v_1, x]$. Therefore the only way for P_1 to reach v_2 is to exit the segment ${P_1}^L[x, y]$ into $left({P_1}^L)$, contradicting that ${P_1}^L$ is the leftmost path.

Proof. 1. Let x_1 denote the first vertex after v_1 where ${P_1}^L$ and say, P_1 diverge. Let y_1 be the first vertex of ${P_1}^L$ after x_1 which also is in P_1. Let y_1' be the first vertex of P_1 after x_1 which is also in ${P_1}^L$ and $y_1 \neq y_1'$. Then y_1' appears after y_1 in ${P_1}^L$. The segments ${P_1}^L[x_1, y_1'] \cup (P_1[x_1, y_1'])$ forms a closed loop. Now the vertex y_1 lies on the segment ${P_1}^L[x_1, y_1']$ of the loop ${P_1}^L[x_1, y_1'] \cup P_1[x_1, y_1']$. The simple path $P_1[y_1', v_2]$ must continue from y_1 to v_2 which is outside the loop, for which it must exit the loop via the segment ${P_1}^L[x_1, y_1']$. But any edge going out of the loop from a vertex of ${P_1}^L[x_1, y_1')$ will contradict the assumption that ${P_1}^L$ is the leftmost walk. Therefore $y_1' = y_1$. Therefore we have a list superface in ${P_1}^L \cup P_1$ with beginning and ending vertices x_1, y_1 respectively and boundaries consisting of ${P_1}^L[x_1, y_1], P_1[x_1, y_1]$. Moreover, since x_1 was the first point of divergence and y_1 the first point after x_1 where both ${P_1}^L, P_1$ meet each other, the segments $Q_b = {P_1}^L[v_1, x_1], Q_e = {P_1}^L[y_1, v_2]$ are mutually disjoint and disjoint from boundaries of the list superface. If both boundaries are of different parity, then we are done. Else we keep the segment ${P_1}^L[v_1, y_1]$ as part of our initial segment, and repeat the same argument starting from y_1 instead of v_1. At some point we must get a list superface satisfying all the conditions and with different boundaries since the parities of ${P_1}^L, P_1$ are different. Therefore the lemma holds.

2. The proof of this is the same as that of Lemma 2 of [Ned99].

□

Now we prove the other direction of Lemma 6.14

Proof. We assume that v_1, v_2, v_3 lie on the outer face, and hence can consider them on an imaginary boundary that encloses the region R where the graph is embedded.(see Fig. 6.12). We consider the cases of which of the pairs $P_i, P_i', i \in \{1, 2, 3\}$, consists of paths of different parity. In each case, we assume without loss of generality that ${P_i}^L$ is of same parity as P_i', and hence of parity different from that of P_i

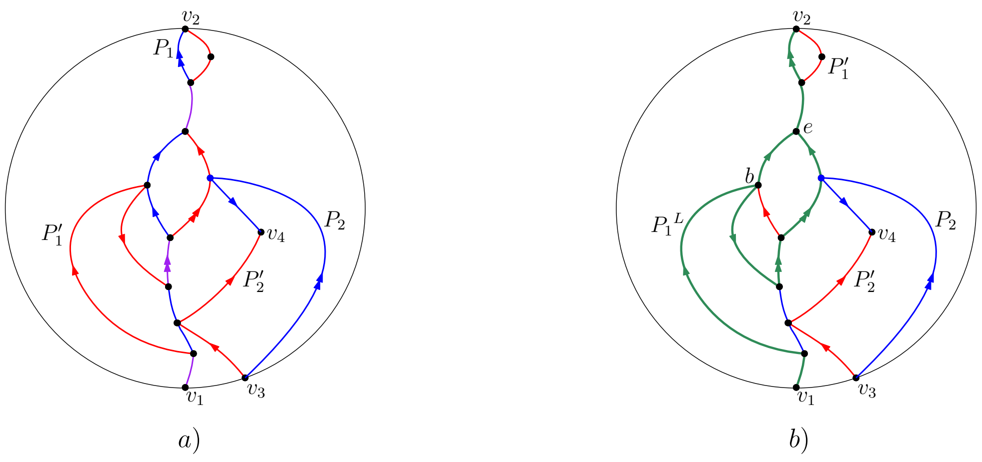


FIGURE 6.11: The red colored paths in Fig a) are P_1', P_2', and the blue colored paths are P_1, P_2(We use purple segments where they share edges). Single arrows denote odd length segments and double arrows even length segments. Segments without arrows can be assumed to be of any parity. The paths highlighted in green in Fig b) denote the odd list superface and the segments Q_b, Q_e, P_2' that we get in lemma 1.

1. Case 1. P_1, P_1' are paths of different parities. The vertices v_3, v_4 must lie on the same side of both P_1, P_1', since they are connected by a path disjoint from P_1 and a path disjoint from P_1'. W.l.o.g., we can assume that they lie strictly on the right side of P_1, P_1'. By our assumption, ${P_1}^L, P_1$ are of different parities. By claim 6.15, the sets (P_1, P_2, P_3) and $({P_1}^L, P_2', P_3')$ are also instances of disjoint paths of different total parities. Let F be the odd list superface formed by $P_1, {P_1}^L$, as described in Lemma 6.16. The boundary ${P_1}^L[b, e]$ of F, as well as segments $Q_b = {P_1}^L[v_1, b], Q_e = {P_1}^L[e, v_2]$ are disjoint from P_2, P_2', P_3, P_3' by claim 6.15. Since other boundary of F is $P_1[b, e]$, both the boundaries of F, and segments Q_b, Q_e are disjoint from P_2, P_3. Therefore we have the odd list superface F, and segments Q_b, Q_e that are all mutually disjoint, satisfying all requirements of lemma 6.14 except possibly for F being simple. Now if F is not a simple odd list superface, then by lemma 6.16, there is a simple odd list superface F_s contained inside F and Q_b, Q_e can be extended to beginning and ending of F_s respectively. Since F_s and the extensions of Q_b, Q_e lie inside F, they are also disjoint from P_2, P_3 and therefore all requirements of lemma 6.14 are satisfied.

2. Case 2. P_2, P_2' are paths of different parities. We have two subcases:

   a) Both v_1, v_4 lie on the same side of P_2, P_2', say right side. In this case the same argument as above works.

   b) Vertex v_1 lies on, say, the left side of P_2, P_2', and v_4 on their right side. Since ${P_2}^L, P_2$ are of different parities, let F be the odd list superface formed by ${P_2}^L, P_2$, with $Q_b = {P_2}^L[v_2, b], Q_e = {P_2}^L[e, v_3]$ as described in the lemma above. Then consider the paths ${P_1}^L, {P_2}^L[v_2, b], {P_2}^L[e, v_3], P_3$ and F. By claim 6.15, ${P_1}^L$ is disjoint from both ${P_2}^L, P_2, P_3$ and ${P_2}^L$ is disjoint from P_3. Now we can proceed as in above argument and satisfy conditions of the lemma.

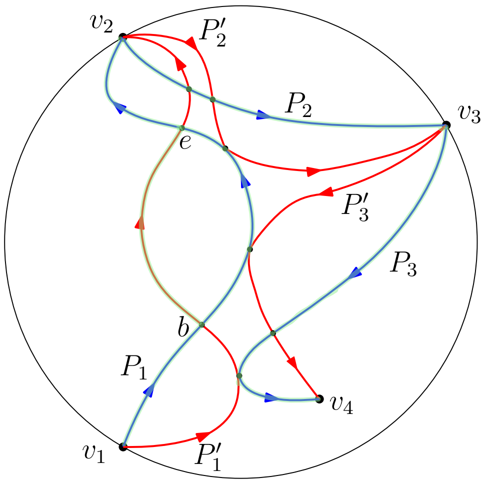
3. Case 3. P_3, P_3' are paths of different parities. Since v_4 need not lie on the outer face, ${P_3}^L$ or ${P_3}^R$ need not have the properties we showed in above lemma. Let F be an odd list superface formed by P_3, P_3', with b, e as beginning, ending vertices, and Q_b, Q_e paths from

v_3 to b and e to v_4 as described by lemma 2 of [Ned99] (Note that each of Q_b, Q_e here can have edged from both P_3, P_3'). There are two subcases (others are symmetrical).

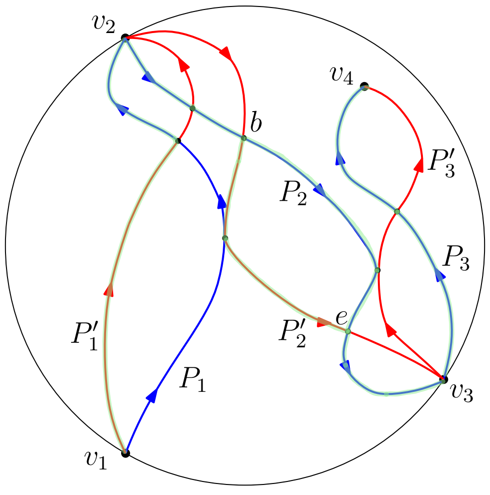
a) Vertices v_3, v_4 lie on right side of P_1, P_1' and also on the right side of P_2, P_2'. In this case consider the paths ${P_1}^L, {P_2}^L$. They are both mutually disjoint as well as disjoint from F, Q_b, Q_e by claim 6.15. Then we can proceed as above.

b) Vertices v_3, v_4 lie on right side of P_1, P_1' and on left side of P_2, P_2'. Then consider the paths ${P_1}^L, {P_2}^R$. They are both mutually disjoint and also disjoint from F, Q_b, Q_e by claim 6.15. Now we can again proceed as above.

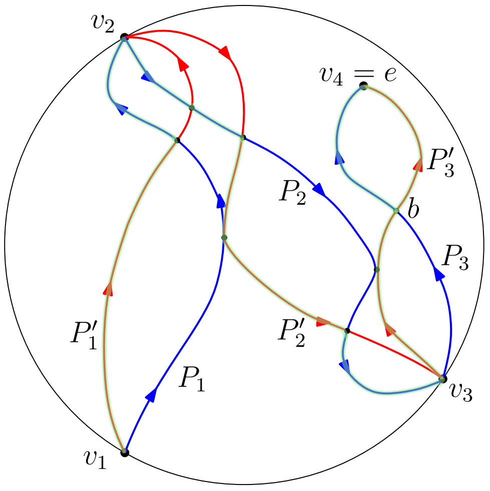
Hence the claim holds. □



Case 1



Case 2 b)



Case 3 b)

FIGURE 6.12: Some cases of Lemma 6.14. The red colored paths are P_1', P_2', and the blue colored paths are P_1, P_2. The paths highlighted in green denote the odd list superface and the segments Q_b, Q_e, P_2' that we get in lemma 6.14.

□

6.3 Main Algorithm

We now explain the two phases of the algorithm.

6.3.1 Phase 1

1. Find the 3-clique sum decomposition tree T_G. Mark the piece that contains the vertex s as the root of T_G.

2. Pick any maximal set of leaf branch pieces of T_G, say $L_1, L_2, \ldots, L_\ell$, which are attached to a parent piece G_i via a common clique. Compute their parity configurations using Nedev's algorithm, or using Courcelle's theorem.Then compute the parity configuration of $L_1 \oplus L_2 \oplus \ldots \oplus L_\ell$ using observation 6.7.

3. Compute the parity mimicking network, L', of $L_1 \oplus L_2 \oplus \ldots \oplus L_\ell$ using lemma 6.9. Replace $L_1 \oplus L_2 \oplus \ldots \oplus L_\ell$ by L' and merge it with G_i.

4. Since $G_i \oplus L'$ is either of bounded treewidth or is planar by lemma 6.8, we can repeat this step until no branch pieces remain.

6.3.2 Phase II

Let G' denote the graph after phase I. After phase I, the modified tree T'_G looks like a path of pieces, $G_1, G_2, \ldots, G_m$, joined at cliques $c_1, c_2 \ldots c_{m-1}$.[2] The vertex s is in root piece G_1, and t in leaf piece G_m (we use G_1, G_m instead of S, T here for notational convenience). We can write $G' = G_1 \oplus_{c_1} G_2 \oplus_{c_2} \ldots \oplus_{c_{m-1}} G_m$. Since it is clear in this phase that c_i is the clique joining G_i, G_{i+1}, we will omit the subscript for notational convenience and just write $G_1 + G_2 + \ldots + G_m$ instead. Let $c_{m-1} = \{v_1, v_2, v_3\}$ and let $i, j, k \in \{1, 2, 3\}$ be distinct. The snapshot of any even s-t path P in G_m, can be one of the following four types (see figure 6.13):

- Type 1 : A path from v_i to t without using v_j, v_k.
- Type 2 : A path from v_i to t via v_j, without using v_k.
- Type 3 : A path from v_i to t via v_j, v_k.
- Type 4 : A path from v_i to v_j and a path from v_k to t, both disjoint from each other.

We call any path/set of paths in G_m of one of the above types as a *potential snapshot* of G_m. We now construct the *projection networks* of potential snapshots of G_m.

Definition 6.17. Let G_m be the leaf piece as described above with clique $c_{m-1} = \{v_1, v_2, v_3\}$, and vertex t present in G_m.

- For each of the types described above, for all $i \in \{1, 2, 3\}$, and for all $p \in \{0, 1\}$, find a potential snapshot (if it exists) in G_m from v_i to t, of total parity p, using lemma 6.11.

[2]Note that the vertices of a clique, say c_i need no longer lie on the same face of G_i after phase I, since we might have merged the parity mimicking network of the branch pieces incident at c_i into the face corresponding to c_i.

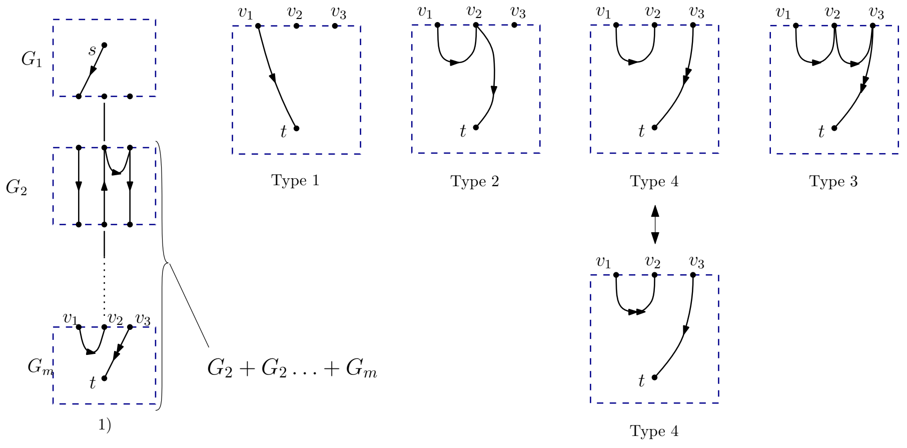


FIGURE 6.13: Fig 1) Denotes the decomposition tree after phase 1, with $G_1, G_2 \ldots, G_m$ denoting the pieces. We skip drawing clique nodes here. On the right are examples of projection networks of different types. In fig 1), the snapshot of the s-t path in G_m is of type 4. The two projection networks of type 4 drawn on the right have same total parity, but different parities of individual segments. It is sufficient for our purpose to find any one of them since they are interchangeable.

- Let J be a potential snapshot found in the previous step, Its *projection network*, is defined as the *graph* obtained from J by keeping terminal vertices intact, and replacing every terminal to terminal path in J by a path of length $2 - p$.

The type of the projection network is the type of the corresponding potential snapshot.
The *set of projection networks* of G_m, denoted by $\mathcal{N}(G_m)$, is the set of all projection networks obtained for G_m by the above procedure.

See Fig. 6.13 for an example. Since the total number of terminals is at most 4 (with one fixed as t), it is easy to see that the number of possible projections networks for G_m is bounded. Therefore $\mathcal{N}(G_m)$ can be computed in polynomial time. Note that $\mathcal{N}(G_m)$ is not uniquely defined. But it is sufficient for our purpose, to compute any one of the various possible choices of the set $\mathcal{N}(G_m)$ as explained in Fig. 6.13. The next lemma shows that the projection networks of G_m preserve solutions, and also maintain invariants on planarity and treewidth, when merged with the parent piece.

Lemma 6.18. *Given $G' = G_1 + \ldots + G_m$ as described above.*

1. *Given a $\mathcal{N}(G_m)$, there is an s-t path in G' of parity p iff $\exists N \in \mathcal{N}(G_m)$ such that $G'[G_m \to N]$ has an s-t path of parity p.*

2. *If G_{m-1} is planar/of bounded treewidth, then for any projection network $N \in \mathcal{N}(G_m)$, $G_{m-1} + N$ is planar/of bounded treewidth, respectively.*

Proof. 1. This follows from the definition of projection networks. The only minor technical point to note is that $\mathcal{N}(G_m)$ is not unique. For example, suppose there are two potential snaphots in G_m of type 4. One is J_1, consisting of a path P_1 from v_1 to v_2 of even parity, and a path P_2 from v_3 to t of odd parity. The other is J_2, consisting of a path P_1' from v_1 to v_2 of odd parity, and a path P_2' from v_3 to t, of even parity. Since the total parity of J_1 and J_2 is same, Lemma 6.11 could output either one of them. We don't have control over it to find both. But finding any one of them is sufficient for us, since if J_1 is a snapshot of an actual solution, then replacing J_1 by J_2 would also give a valid solution and vice versa.(See Fig. 6.13)

2. Suppose $c_{m_1} = \{v_1, v_2, v_3\}$ is the clique where G_{m-1}, G_m are attached. The argument of treewidth bound is same as that of Lemma 6.8 in previous phase, when we attached mimicking networks to parent pieces. However if G_{m-1} is planar, there could have been a parity mimicking network L' attached to G_{m-1} via c_{m-1} during phase I. Hence v_1, v_2, v_3 might not lie on a common face in G_{m-1} after phase I. We observe however, since L' was attached at a 3-clique, c_{m-1}, every pair v_i, v_j of vertices of c_{m-1}, must share a common face in G_{m-1}. Now, the projection networks consist of at most three paths, two between v_1, v_2, v_3, and one from them to t. For any v_i, v_j, we can embed the path between v_i, v_j in N, in the face in G_{m-1} shared by v_i, v_j, and finally just add the path leading to t. Therefore if G_{m-1} is planar, all projection networks of G_m can be embedded in their parent nodes. □

We make the following observation to compute a $\mathcal{N}(G_i + \dots G_m)$ recursively:

$$\mathcal{N}(G_i + \dots G_m) = \bigcup_{N \in \mathcal{N}(G_{i+1} + \dots G_m)} \mathcal{N}(G_i + N) \tag{6.3}$$

Thus we can proceed as follows:

1. Compute $\mathcal{N}(G_m)$ using lemma 6.11
2. For all $N \in \mathcal{N}(G_m)$, compute $G_{m-1} + N$, and hence compute $\mathcal{N}(G_m + G_{m-1})$ using the observation above.
3. For all $N \in \mathcal{N}(G_m + G_{m-1})$, compute $G_{m-2} + N$, and hence compute $\mathcal{N}(G_m + G_{m-1} + G_{m-2})$. Repeat until we reach G_1.

This completes the algorithm for decision version of EvenPath.

6.3.3 Reducing even path and disjoint path with parity search to decision

We now give a reduction from search version of even path problem to decision version. Our algorithm will output the s-t even path which is lexicographically least according to vertex indices. A path P_1 is lexicographically lesser then P_2 if the among the first vertices in P_1, P_2 where they diverge (assuming they start at the same vertex), the one in P_1 has a lesser index than the one in P_2.

We give an algorithm below.

```
Procedure findParityPath($G, s, t, p$):
    if $s == t$ then
        return <>;
    end
    foreach $v_{out}$ = EACH OUT NEIGHBOUR OF $s$ (in increasing index order) do
        if searchParityPath($G - \{s\}, v_{out}, t, 1 - p$) then
            return $\langle s, v_{out} \rangle$ + findParityPath($G - \{s\}, v_{out}, t, 1 - p$);
        end
    end
    return NULL;
```

Algorithm 3: Routine to find a path of parity p between given vertices

The correctness of the algorithm can be proved by inducting on length of the unique lexicographically least path of parity from s to t. The algorithm can be optimized to makes $\mathcal{O}(n)$ queries to searchParityPath routine, and so the time complexity is $\mathcal{O}(n.T(n))$ where $T(n)$ is the complexity of decision version of parity path in G.

The same algorithm can be extended to find disjoint paths between (s_i, t_i) with total parity even with an oracle to decision version of the same problem. In that case, we will output the lexicographically least instance of disjoint paths with total parity even, which consists of solution obtained by considering solutions with lex-least (s_1, t_1) path, and then amongst those, the solutions with lex-least (s_2, t_2) path and so on. Hence we can also find disjoint paths with parity in time $\mathcal{O}(n.T(n))$ where $T(n)$ is the complexity of decision version of disjoint paths with parity in G.

Bibliographical Remarks The results presented in this chapter are based on joint work with Samir Datta, Chetan Gupta, and Vimal Raj Sharma, which appeared in [CDGS24].

Chapter 7

Conclusions

We summarize and conclude the thesis in this chapter.

We started with our results on DFS in planar digraphs in Chapter 4, giving a bound of $\mathsf{AC}^1(\mathsf{UL} \cap \text{co-}\mathsf{UL})$ for the problem. However the goal we had in mind while starting this problem remains open, that is to show that DFS in planar digraphs reduces to finding distances between vertices in planar digraphs, which would lead to a $\mathsf{UL} \cap \text{co-}\mathsf{UL}$ bound for the problem. We believe that this does hold, though we do not yet have a proof for it. For undirected graphs, we saw in Chapter 3 that DFS does reduce to computing distances. One approach to attack the directed version would be to use the decomposition we develop in Chapter 4 more optimally. Though we are able to compute the layering for planar digraphs in $\mathsf{UL} \cap \text{co-}\mathsf{UL}$, we are not able to use it to compute consistent DFS trees for each layer and patch them up. Using it to find a separator and then going by divide and conquer discards all the information regarding the layers that we had computed. It would be interesting to see if we can somehow reuse the layerings to compute DFS traversals more efficiently. It would also be interesting to see if the decomposition can be used in some other problems.

In Chapter 5 we further explored results on DFS in other graph classes like bounded genus graphs, bounded treewidth graphs, single-crossing-minor-free graphs. We gave gave an NC bound for DFS in single-crossing-minor-free graphs, without going into finer analysis. We believe that the problem is within the class AC^2, perhaps even $\mathsf{AC}^1(\mathsf{L})$ if we can show how to compute path separators in single-crossing-minor-free graphs in L. The main things to check would be the complexity of 3-clique sum decompositions (note that 2-clique sum decompositions are already known to computable in L by [DLN+22]), the complexity of finding 2-disjoint paths in planar graphs, and finding balanced interior-exterior-cycle separators in weighted planar graphs (we believe Shannon's algorithm [Sha88] can be tweaked to achieve that in L). It would also be interesting to see if we can give an NC algorithm for DFS in more general minor-closed graph classes, like apex-minor free graphs or general H-minor-free graphs. There are decomposition theorems known for these classes by the work of Robertson and Seymour and even polynomial time algorithms to compute them [RS03, GKR13]. It would be quite interesting and useful if we could have NC algorithms for computing these decompositions, as it could open the doors to attack many problems in these graph classes in the parallel setting.

Finally in Chapter 6 we study the EvenPath problem in directed single-crossing-minor-free graphs and show that it is in P in that class. One question that naturally arises again is to see if the problem is more tractable in more general minor-closed classes. The gap between the classes where EvenPath is known to be tractable (planar graphs, single-crossing-minor-free graphs, single-crossing graphs) and where it is known to be NP-hard is quite large, and it would be interesting to get some dichotomy results on its complexity. The two problems we encountered while solving EvenPath in single-crossing-minor-free raise interesting questions too. It would be nice to see if there exist (and if we can efficiently consstruct them) small, planar parity mimicking networks for more than three terminals. We also do not know the status of DisjointPathsTotalParity problem in planar directed graphs even for 2-paths (without any constraints on placement of terminals), and it would be interesting to see if the problem is tractable or is NP-hard.

Appendix A

Proof of lemma 6.9

We restate the lemma here:

Lemma A.1. *Suppose L is a graph with terminals $T(L)$, $|T(L)| \leq 3$, and suppose we know the parity configuration of L with respect to $T(L)$. We can in polynomial-time, find a parity mimicking network L' with respect to $T(L)$ which consists of at most 13 vertices, and is also planar, with all vertices of $T(L')$ on a common face.*

Lemma 6.9. *Suppose L is a graph with terminals $T(L) = \{v_1, v_2, v_3\}$, and suppose we know the parity configuration of L with respect to $\{v_1, v_2, v_3\}$. We can in polynomial-time, find a parity mimicking network L' of L, with respect to $\{v_1, v_2, v_3\}$ which consists of at most 18 vertices, and is also planar, with v_1, v_2, v_3 lying on a common face.*

We will now formalise the proof idea using some definitions and lemmas. Let $\mathcal{P}$ denote the parity configuration of L with respect to terminals $\{v_1, v_2, v_3\}$. We will throughout assume that variables i, j, k take distinct values in $\{1, 2, 3\}$. We will generally use $x_{ij}, x_{ij} + 1$, where $x_{ij} \in \{0, 1\}$ and addition is modulo 2, to denote entries of sets $Dir_{\mathcal{P}}(v_i, v_j)$, $Via_{\mathcal{P}}(v_i, v_k, v_j)$. We will refer to the entries in the *Dir* sets as direct set entries. Given two parity configurations $\mathcal{P}, \mathcal{P}'$, we say $\mathcal{P}'$ is a *sub-configuration* of $\mathcal{P}$, if all *Dir,Via* sets of $\mathcal{P}'$ are subsets of the corresponding *Dir,Via* sets of $\mathcal{P}$, i.e. $Dir_{\mathcal{P}'}(v_i, v_j) \subseteq Dir_{\mathcal{P}}(v_i, v_j)$, and $Via_{\mathcal{P}'}(v_i, v_k, v_j) \subseteq Via_{\mathcal{P}}(v_i, v_k, v_j), \forall i, j, k$. We denote this as $\mathcal{P}' \subseteq_c \mathcal{P}$.

We now define *bad pairs* of $\mathcal{P}$.

Definition A.2. Given the parity configuration $\mathcal{P}$, a pair of entries, $x_{ij} \in Dir_{\mathcal{P}}(v_i, v_j)$ and $x_{jk} \in Dir_{\mathcal{P}}(v_j, v_k)$, are called a *bad pair* if $x_{ij} + x_{jk} \notin (Dir_{\mathcal{P}}(v_i, v_k) \bigcup Via_{\mathcal{P}}(v_i, v_j, v_k))$. A direct set entry x_{ij} is called a *bad* entry if it is part of at least one bad pair in $\mathcal{P}$. We call direct set entries that are not part of *any* bad pair as *good* entries. We use the phrase *bad pairs between* $Dir_{\mathcal{P}}(v_i, v_j), Dir_{\mathcal{P}}(v_j, v_k)$, to refer to the bad pairs formed by entries of $Dir_{\mathcal{P}}(v_i, v_j), Dir_{\mathcal{P}}(v_j, v_k)$

Next we define the *bad kernel*, $\mathcal{P}_{bad}$ of $\mathcal{P}$, which is the sub-configuration of $\mathcal{P}$ consisting of all the bad entries of $\mathcal{P}$. Its closure, $\widetilde{\mathcal{P}}_{bad}$ adds a minimal number of entries to make it realisable.

Definition A.3. Given a parity configuration, $\mathcal{P}$, its *bad kernel*, denoted by $\mathcal{P}_{bad}$, is the sub-configuration defined as:

- For every i, j and $x_{ij} \in \{0, 1\}$, $x_{ij} \in Dir_{\mathcal{P}_{bad}}(v_i, v_j)$ iff x_{ij} is a bad entry in $Dir_{\mathcal{P}}(v_i, v_j)$.

The *closure* of $\mathcal{P}_{bad}$, denoted by $\widetilde{\mathcal{P}}_{bad}$, is obtained from $\mathcal{P}_{bad}$ by augmenting it as follows:

1. For every bad pair (x_{ij}, x_{jk}) in $\mathcal{P}_{bad}$, if $x_{ij} + x_{jk} + 1 \notin Dir_{\mathcal{P}_{bad}}(v_i, v_k)$, then add $x_{ij} + x_{jk} + 1$ to $Dir_{\mathcal{P}_{bad}}(v_i, v_k)$.

2. For every good pair (x_{ij}, x_{jk}) in $\mathcal{P}_{bad}$, if $x_{ij} + x_{jk} \notin Dir_{\mathcal{P}_{bad}}(v_i, v_k)$, then add $x_{ij} + x_{jk}$ to $Via_{\mathcal{P}_{bad}}(v_i, v_j, v_k)$.

The following is a simple claim:

Claim A.4. *Let $\mathcal{P}$ be a realizable parity configuration. Let $\mathcal{P}_{bad}$ be the bad kernel of $\mathcal{P}$, and $\widetilde{\mathcal{P}}_{bad}$, its closure. Then $\mathcal{P}_{bad} \subseteq_c \widetilde{\mathcal{P}}_{bad} \subseteq_c \mathcal{P}$.*

Proof. It is clear from definition of $\widetilde{\mathcal{P}}_{bad}$ that $\mathcal{P}_{bad} \subseteq_c \widetilde{\mathcal{P}}_{bad}$. Suppose x_{ij}, x_{jk} form a bad pair in $\mathcal{P}$, and hence are present in $\mathcal{P}_{bad}$. Since, $\mathcal{P}$ is realizable, there must a path from v_i to v_k in any graph with parity configuration $\mathcal{P}$. Since by definition of a bad pair, the parity of that path cannot be $x_{ij} + x_{jk}$, it must be $x_{ij} + x_{jk} + 1$. Therefore any entry added in step 1 of construction of $\widetilde{\mathcal{P}}_{bad}$ must also be present in the corresponding set in $\mathcal{P}$. Similar argument also holds for entries added in step 2 of construction of $\widetilde{\mathcal{P}}_{bad}$. Therefore the claim holds. □

We show that in order to find a parity mimicking network of any realizable parity configuration that satisfies the required planarity conditions, it is sufficient to give a parity mimicking network obeying those conditions for the *closure* of its *bad kernel*.

Lemma A.5. *Let $\mathcal{P}$ be a parity configuration with respect to terminals $\{v_1, v_2, v_3\}$ and let $\widetilde{\mathcal{P}}_{bad}$ denote the closure of the bad kernel of $\mathcal{P}$. Suppose L'' is a planar parity mimicking network of the parity configuration $\widetilde{\mathcal{P}}_{bad}$, with terminals lying on outer face. We can construct a planar parity mimicking network L' for configuration $\mathcal{P}$, with terminals lying on outer face, by adding edges to L'' using the following iterative operation:*

- *For every entry $x_{ij} \in \mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$, if there is not a direct path in L'' from v_i to v_j of parity x_{ij} already, then add a path of parity x_{ij} (length one or two) from v_i to v_j, disjoint from all currently existing paths in the network.*

Proof. By hypothesis, terminals v_1, v_2, v_3 all lie on the outer face of L''. It can be seen easily that we can repeatedly add direct paths from v_i to v_j without intersecting others by drawing them on the outer face (see Fig. 6.5). Hence L' is planar with v_1, v_2, v_3 on the same face. As noted above, $\widetilde{\mathcal{P}}_{bad}$ is a sub-configuration of $\mathcal{P}$. Since all paths of parity corresponding to bad entries of $\mathcal{P}$ have already been added in L'', the remaining entries for which paths are yet to be added are good entries. Now, at any step in the above procedure, if we add a path of parity x_{ij} from v_i to v_j disjoint from all existing paths in the network, the only possible extra terminal to terminal paths that can be formed are *Via* paths from v_i to v_k via v_j, of parity $x_{ij}+x_{jk}$, and from v_k to v_j via v_i, of parity $x_{ki}+x_{ij}$. By definition of a good entry, both $x_{ij}+x_{jk}, x_{ki}+x_{ij}$ must exist in $(Dir_{\mathcal{P}}(v_i, v_k) \bigcup Via_{\mathcal{P}}(v_i, v_j, v_k))$ and $(Dir_{\mathcal{P}}(v_k, v_j) \bigcup Via_{\mathcal{P}}(v_k, v_i, v_j))$ respectively. Therefore this does not create any paths of unwanted parities in L', and hence we can safely construct L' by this operation. □

Now all that remains to show is how to construct parity mimicking networks for closures of all possible *bad kernels*. For visual aid in figures, we will call the direct sets $Dir_{\mathcal{P}}(v_1, v_2)$, $Dir_{\mathcal{P}}(v_2, v_3)$, $Dir_{\mathcal{P}}(v_3, v_1)$ as sets in *upper row* and the sets $Dir_{\mathcal{P}}(v_2, v_1)$, $Dir_{\mathcal{P}}(v_3, v_2)$, $Dir_{\mathcal{P}}(v_1, v_3)$ as the sets in *lower row*. We will make a few observations regarding bad pairs of a realizable parity configuration $\mathcal{P}$ which follow easily from definition of a *bad pairs*. We observe that:

Observation A.6. *1. There can be at most two bad pairs between* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$ *and* $\mathrm{Dir}_{\mathcal{P}}(v_j, v_k)$. *This follows from the observation that at least one of* $\{0, 1\}$ *must be present in* $(\mathrm{Dir}_{\mathcal{P}}(v_i, v_k) \bigcup \mathrm{Via}_{\mathcal{P}}(v_i, v_j, v_k))$*, since there is some path from* v_i *to* v_k *if* $\mathcal{P}$ *is realizable.*

2. *If there are two bad pairs between* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$ *and* $\mathrm{Dir}_{\mathcal{P}}(v_j, v_k)$*, then each of* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$ *and* $\mathrm{Dir}_{\mathcal{P}}(v_j, v_k)$ *has both* $0, 1$ *as entries, and the bad pairs are disjoint. For example, if* (x_{ij}, x_{jk}) *form a bad pair bewteen sets* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$*, and* $\mathrm{Dir}_{\mathcal{P}}(v_j, v_k)$*, then the other bad pair between these sets, if it exists, must be* $(x_{ij} + 1, x_{jk} + 1)$*.*

3. *Suppose a direct set* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_j)$ *has both* $0, 1$ *as entries. Then there can be no bad pairs formed between* $\mathrm{Dir}_{\mathcal{P}}(v_i, v_k)$ *and* $\mathrm{Dir}_{\mathcal{P}}(v_k, v_j)$*.*

4. *Bad pairs can be formed only between* Dir *sets within upper row, or* Dir *sets within lower row, not across.*

To enumerate on the bad kernels, we can adopt without loss of generality, the following two conventions:

- Number of bad pairs between upper row sets $\geq$ Number of bad pairs between lower row sets.
- Number of bad pairs between $Dir_{\mathcal{P}}(v_1, v_2), Dir_{\mathcal{P}}(v_2, v_3) \geq$ Number of bad pairs between $Dir_{\mathcal{P}}(v_2, v_3), Dir_{\mathcal{P}}(v_3, v_1) \geq$ Number of bad pairs between $Dir_{\mathcal{P}}(v_3, v_1), Dir_{\mathcal{P}}(v_1, v_2)$,

The other cases are handled by symmetry. We make the following claim:

Claim A.7. *Any realisable parity configuration* $\mathcal{P}$ *can have at most* 6 *bad pairs.*

Proof. Let the number of bad pairs in $\mathcal{P}$ be more than 6. We can then assume using our conventions that the upper row sets have at least 4 bad pairs, and two of them must be between $Dir_{\mathcal{P}}(v_1, v_2), Dir_{\mathcal{P}}(v_2, v_3)$. This implies, by parts 2 and 3 of observation A.6 above, that $Dir_{\mathcal{P}}(v_1, v_2)$ and $Dir_{\mathcal{P}}(v_2, v_3)$ each have both $0, 1$ as entries, and hence there cannot be any bad pairs between $Dir_{\mathcal{P}}(v_1, v_3), Dir_{\mathcal{P}}(v_3, v_2)$ and between $Dir_{\mathcal{P}}(v_2, v_1), Dir_{\mathcal{P}}(v_1, v_3)$ in the lower row. Now there are two cases:

- If $Dir_{\mathcal{P}}(v_3, v_1)$ has two bad entries, each forming bad pairs with other upper row sets, then there cannot be any bad pairs between the lower row sets. Since the total number of bad pairs cannot be more than 6 in the upper row, this leads to a contradiction.
- If $Dir_{\mathcal{P}}(v_3, v_1)$ has lesser than two bad entries, they can form at most two bad pairs. Then the total number of bad pairs in upper row is at most four, and in lower row at most two, which also leads to a contradiction.

Hence total number of bad pairs in $\mathcal{P}$ cannot be more than 6. □

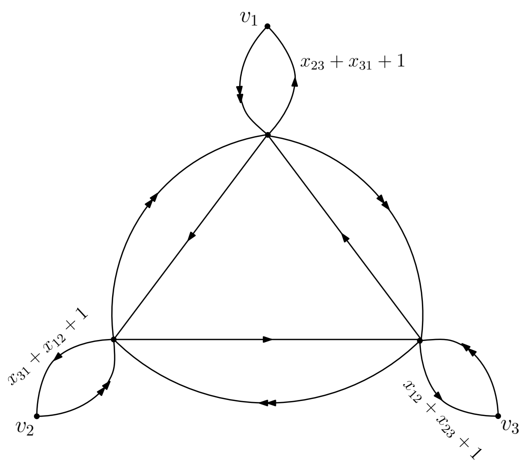
This gives a bound on number of types of bad kernels we need to consider. We can write the number of bad pairs of $\mathcal{P}$ as (n_u, n_l), where n_u is the number of bad pairs in upper row and n_l the number of bad pairs in the lower row. Since $n_l \leq n_u \leq 6$ and $n_l + n_u \leq 6$, we can lexicographically enumerate all cases from $(6, 0)$ to $(1, 0)$, and construct explicitly, the required mimicking networks for closure of each case. We list the cases below starting from $(6, 0)$ to $(1, 1)$. The cases lying in between $(6, 0)$ and $(1, 1)$ that are not drawn are those which cannot occur as bad kernels. We give some examples of such a cases, others have a similar argument. Case remaining after $(1,1)$ is $(1,0)$, which is shown in proof idea, so we do not draw it here.

In all of the figures below, the entries of the parity configuration tables joined by the red lines denote bad pairs. In the corresponding gadgets, if no parity expression is written beside an edge, then it is a path of length one or two according to if it has a single or a double arrow respectively. If a parity expression is written then the length is according to the parity expression. The variables can take values in $\{0,1\}$. We only show Via sets when required. Otherwise we assume them to be empty by default. The only case that is a bit tedious to verify is that of $(4,0)$, as shown in Fig. A.1. For ease of verification, we have listed all the paths from v_1 to v_3 along with their lengths (counting single arrows as length 1, double arrows as length 2.) All of these must be of same parity in that particular case. Paths from v_2 to v_1 follow a symmetric pattern.

Case: $(6,0)$

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	$x_{31}+1$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$



Case: $(5,1)$

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	$x_{31}+1$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
	x_{23}	x_{31}

Not realizable.

No bad pair possible in the lower row since every column in upper row has both $0,1$ parity entries.

Case: $(4,2)$

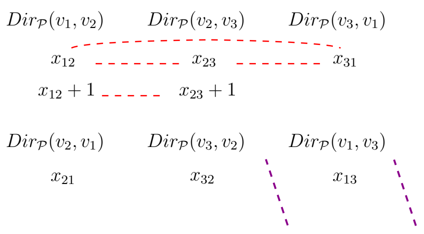
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	$x_{31}+1$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	x_{13}
$x_{12}+1$	$x_{32}+1$	$x_{13}+1$

Not realizable.

Lower row cannot contain any bad pair.

Case: $(4,2)$ *b*)

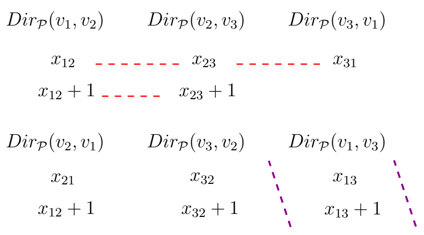


$Dir_{\mathcal{P}}(v_1,v_2)$ $Dir_{\mathcal{P}}(v_2,v_3)$ $Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} x_{23} x_{31}
$x_{12}+1$ $x_{23}+1$
$Dir_{\mathcal{P}}(v_2,v_1)$ $Dir_{\mathcal{P}}(v_3,v_2)$ $Dir_{\mathcal{P}}(v_1,v_3)$
x_{21} x_{32} x_{13}

Not realizable.

No bad pair possible between $Dir_{\mathcal{P}}(v_3,v_2)$, $Dir_{\mathcal{P}}(v_1,v_3)$, and between $Dir_{\mathcal{P}}(v_1,v_3)$, $Dir_{\mathcal{P}}(v_2,v_1)$. Moreover, no direct set in lower row can contain more than one element.

Case: $(3,3)$ *b*)

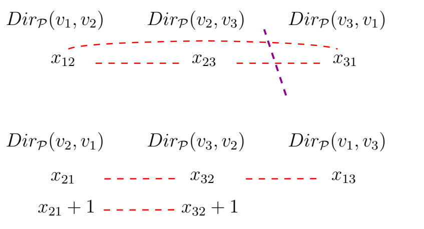


$Dir_{\mathcal{P}}(v_1,v_2)$ $Dir_{\mathcal{P}}(v_2,v_3)$ $Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} x_{23} x_{31}
$x_{12}+1$ $x_{23}+1$
$Dir_{\mathcal{P}}(v_2,v_1)$ $Dir_{\mathcal{P}}(v_3,v_2)$ $Dir_{\mathcal{P}}(v_1,v_3)$
x_{21} x_{32} x_{13}
$x_{12}+1$ $x_{32}+1$ $x_{13}+1$

Not realizable.

Lower row can contain at most two bad pairs.

Case: $(3,3)$ *b*)

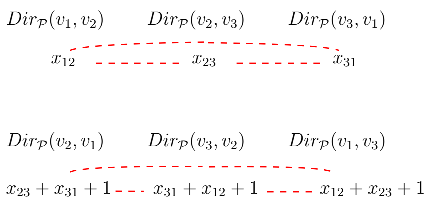


$Dir_{\mathcal{P}}(v_1,v_2)$ $Dir_{\mathcal{P}}(v_2,v_3)$ $Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} x_{23} x_{31}
$Dir_{\mathcal{P}}(v_2,v_1)$ $Dir_{\mathcal{P}}(v_3,v_2)$ $Dir_{\mathcal{P}}(v_1,v_3)$
x_{21} x_{32} x_{13}
$x_{21}+1$ $x_{32}+1$

Not realizable.

There cannot be bad pairs between all three *Dir* sets in upper row if a *Dir* set in lower row has elements of both parities.

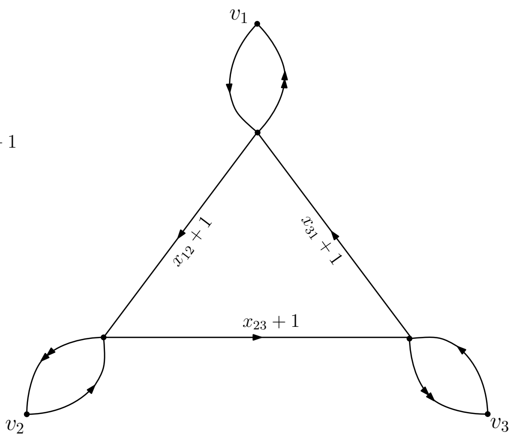
Case: $(3,3)$ *a*)



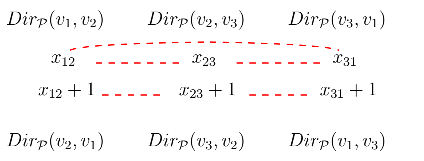
$Dir_{\mathcal{P}}(v_1,v_2)$ $Dir_{\mathcal{P}}(v_2,v_3)$ $Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} x_{23} x_{31}
$Dir_{\mathcal{P}}(v_2,v_1)$ $Dir_{\mathcal{P}}(v_3,v_2)$ $Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$ $x_{31}+x_{12}+1$ $x_{12}+x_{23}+1$

The following constraints follow from bad pairs in lower row :

$$x_{12}+x_{23}+x_{31}=1$$



Case: $(5,0)$



$Dir_{\mathcal{P}}(v_1,v_2)$ $Dir_{\mathcal{P}}(v_2,v_3)$ $Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} x_{23} x_{31}
$x_{12}+1$ $x_{23}+1$ $x_{31}+1$
$Dir_{\mathcal{P}}(v_2,v_1)$ $Dir_{\mathcal{P}}(v_3,v_2)$ $Dir_{\mathcal{P}}(v_1,v_3)$

Not realizable.

If x_{31}, x_{12} form a bad pair, then $x_{31}+1$, $x_{12}+1$ *must* also form a bad pair.

Case: $(4,1)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	$x_{31}+1$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	x_{13}
$x_{12}+1$	$x_{32}+1$	$x_{13}+1$

Not realizable.

Lower row cannot contain any bad pair.

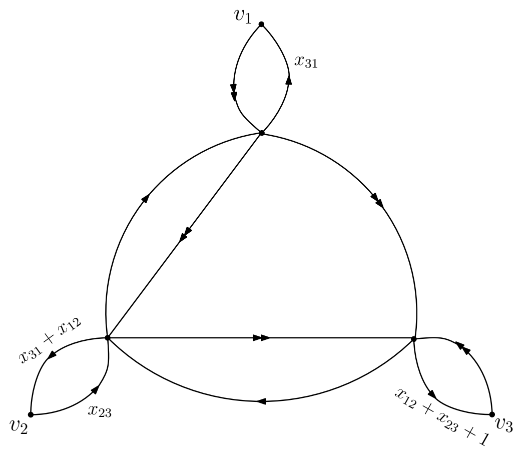
Case: $(4,1)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

The following constraints follow from bad pairs in lower row :

$$x_{12}+x_{23}+x_{31}=1$$



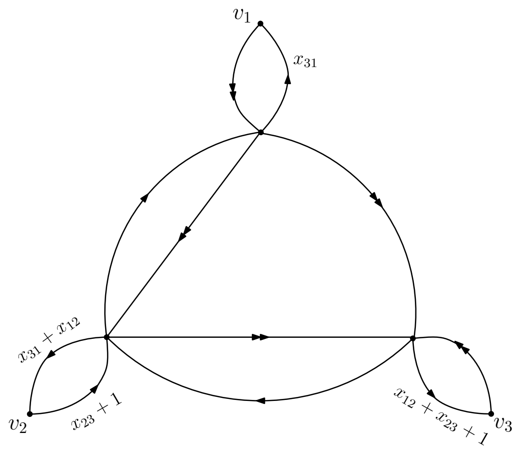
Case: $(4,1)$ *c)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

The following constraints follow from bad pairs in lower row :

$$x_{12}+x_{23}+x_{31}=0$$



Case: $(3,2)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	x_{13}
	$x_{32}+1$	

Not realizable.

No bad pair possible between $Dir_{\mathcal{P}}(v_3,v_2)$, $Dir_{\mathcal{P}}(v_1,v_3)$, and between $Dir_{\mathcal{P}}(v_1,v_3)$, $Dir_{\mathcal{P}}(v_2,v_1)$. Moreover, $Dir_{\mathcal{P}}(v_2,v_1)$ cannot cannot contain more than one element.

Case: $(3,2)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	x_{13}
$x_{21}+1$	$x_{32}+1$	

Not realizable.

The bad pair structure in upper row fixes the parity values of lower rows, and then there must exist a bad pair between all three sets of lower row.

Case: $(3,2)$ *c)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

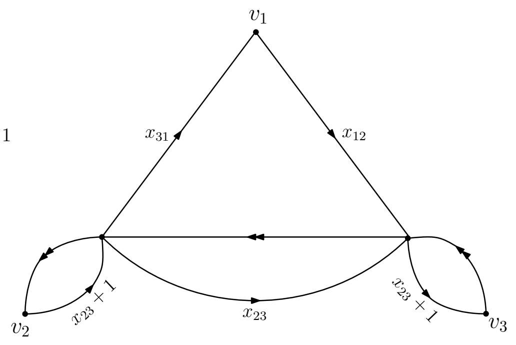
$Via_{\mathcal{P}}(v_2,v_1,v_3)$

$x_{12}+x_{31}$

The good pair between $Dir_{\mathcal{P}}(v_1,v_3)$ and $Dir_{\mathcal{P}}(v_2,v_1)$ enforces this Via entry in the closure of this bad kernel.

The following constraints follow from bad pairs in lower row:

$$x_{12}+x_{31}+x_{23}=1$$



Case: $(4,0)$

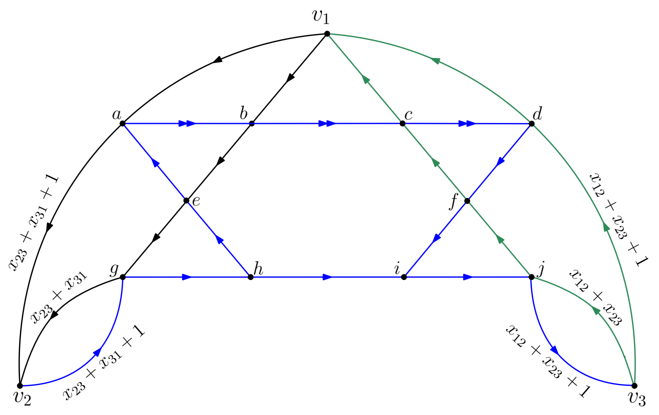
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	$x_{31}+1$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$		$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_3,v_1,v_2)$

x_{23}

$x_{23}+1$



List of all direct v_1 to v_3 paths with lengths:

v_1,b,c,d,f,i,j,v_3	$-$	$x_{12}+x_{23}+9$
v_1,b,e,g,h,i,j,v_3	$-$	$x_{12}+x_{23}+7$
v_1,a,b,c,d,f,i,j,v_3	$-$	$x_{12}+x_{23}+11$
v_1,a,b,e,g,h,i,j,v_3	$-$	$x_{12}+x_{23}+9$

List of all via v_1 to v_3 paths with lengths:

v_1,a,v_2,g,h,i,j,v_3	$-$	$x_{12}+x_{23}+7$
$v_1,b,e,a,v_2,g,h,i,j,v_3$	$-$	$x_{12}+x_{23}+9$

FIGURE A.1: Case $(4,0)$, with list of all paths from v_1 to v_3 for ease of verification.

Case: $(4,0)$ $b)$

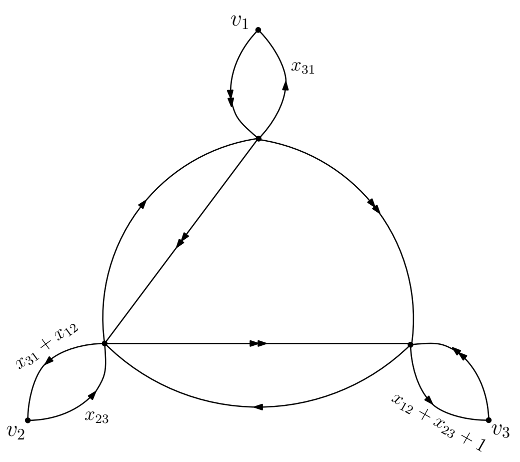
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

There are two subcases based on parity of $x_{12}+x_{23}+x_{31}$:

Subcase 1) : $x_{12}+x_{23}+x_{31}=0$:

In this subcase, the elements of lower row all form good pairs with each other already and we do not need to add any *via* entries in the closeure. The same gadget as that in case $(4,1)$ $b)$ works here. (Reproduced on the right).



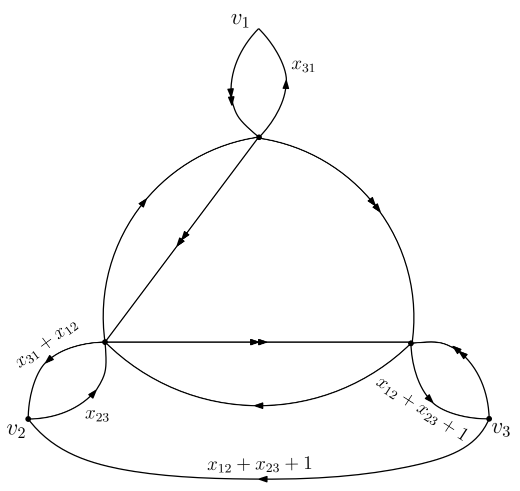
Subcase 1) : $x_{12}+x_{23}+x_{31}=1$:

In this subcase, we need to add the following *via* entry to ensure that elements of $Dir_{\mathcal{P}}(v_2,v_1), Dir_{\mathcal{P}}(v_3,v_2)$ form a good pair.

$Via_{\mathcal{P}}(v_3,v_2,v_1)$

$x_{31}+1$

(Same as $x_{12}+x_{23}$)



Case: $(4,0)$ $c)$

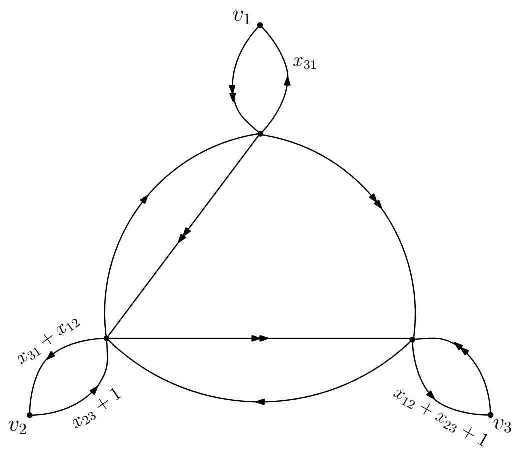
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

There are two subcases based on parity of $x_{12}+x_{23}+x_{31}$:

Subcase 1) : $x_{12}+x_{23}+x_{31}=1$

In this subcase, the elements of lower row all form good pairs with each other already and we do not need to add any *via* entries in the closure. The same gadget as that in case $(4,1)$ $c)$ works here. (Reproduced on the right).



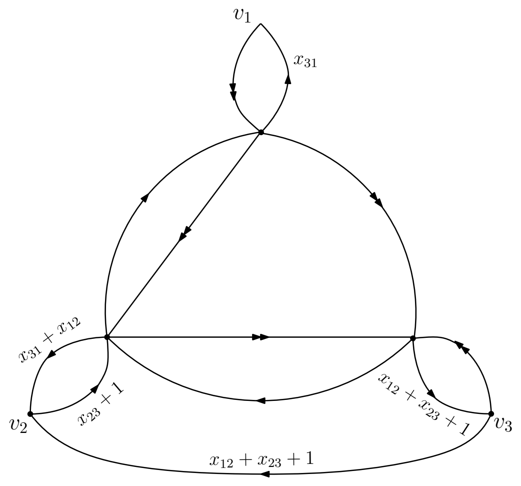
Subcase 1) : $x_{12}+x_{23}+x_{31}=0$:

In this subcase, we need to add the following *via* entry to ensure that elements of $Dir_{\mathcal{P}}(v_2,v_1), Dir_{\mathcal{P}}(v_3,v_2)$ form a good pair.

$Via_{\mathcal{P}}(v_3,v_2,v_1)$

$x_{31}+1$

(Same as $x_{12}+x_{23}+1$)



Case: $(3,1)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

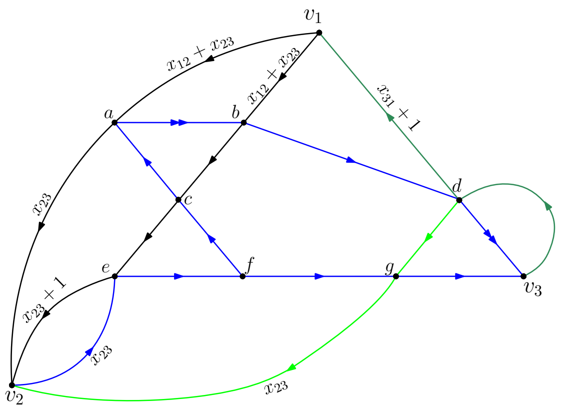
$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	x_{32}	$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_3,v_1,v_2)$

$x_{23}+1$

The following constraints follow from bad pairs of lower row:

$x_{32}=x_{23}$



List of all direct v_2 to v_1 paths with lengths:	v_2,e,f,c,a,b,d,v_1	–	$x_{23}+x_{31}+7$
List of all via v_2 to v_1 paths with lengths:	v_2,e,f,g,v_3,d,v_1	–	$x_{23}+x_{31}+5$
List of all direct v_3 to v_2 paths with lengths:	v_3,d,g,v_2	–	$x_{23}+2$
List of all via v_3 to v_2 paths with lengths:	v_3,d,v_1,a,v_2	–	$x_{12}+2x_{23}+x_{31}+2$
	v_3,d,v_1,b,c,e,v_2	–	$x_{12}+2x_{23}+x_{31}+5$
	v_3,d,v_1,b,c,a,v_2	–	$x_{12}+2x_{23}+x_{31}+4$
List of all direct v_1 to v_3 paths with lengths:	v_1,b,d,v_3	–	$x_{12}+x_{23}+3$
	v_1,b,c,e,f,g,v_3	–	$x_{12}+x_{23}+5$
	v_1,a,b,d,v_3	–	$x_{12}+x_{23}+5$
	v_1,a,b,c,e,f,g,v_3	–	$x_{12}+x_{23}+7$
List of all via v_1 to v_3 paths with lengths:	v_1,a,v_2,e,f,g,v_3	–	$x_{12}+x_{23}+3$
	v_1,b,c,a,v_2,e,f,g,v_3	–	$x_{12}+x_{23}+5$

Case: $(3,1)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

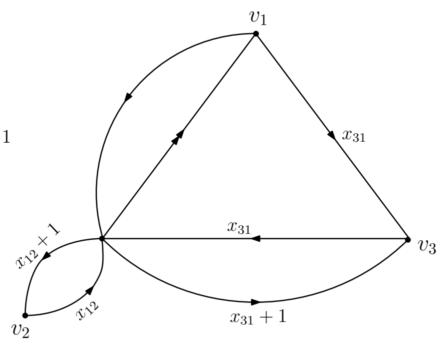
$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_1,v_3,v_2)$	$Via_{\mathcal{P}}(v_2,v_1,v_3)$
$x_{23}+x_{31}$	$x_{31}+x_{12}$

Via paths of these parity must exist because of good pairs in lower row formed with element of $Dir_{\mathcal{P}}(v_1,v_3)$.

The following constraints follow from bad pairs of lower row:

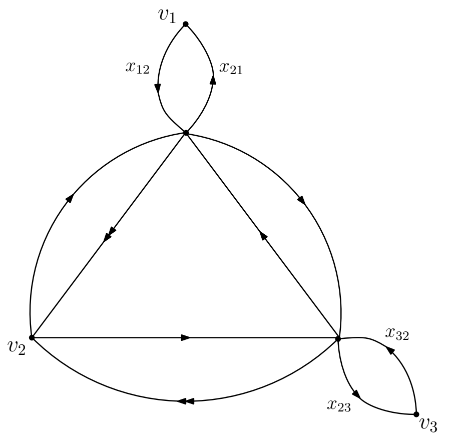
$x_{12}+x_{31}+x_{23}=1$



Case: $(2,2)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	$x_{21}+x_{32}+1$
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	$x_{12}+x_{23}+1$
$x_{21}+1$	$x_{32}+1$	



Case: $(2,2)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

The following constraints follow from bad pairs in lower row:

$$x_{32} = x_{23}$$

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	x_{32}	$x_{12}+x_{23}+1$

There are two subcases base on parity of $x_{12}+x_{23}+x_{31}$:

Subcase 1) : $x_{12}+x_{23}+x_{31}=1$

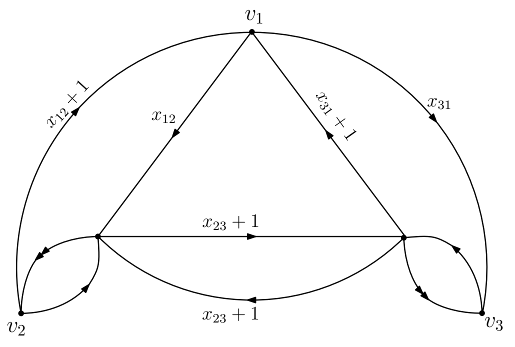
The good pair between $Dir_{\mathcal{P}}(v_3,v_1)$, $Dir_{\mathcal{P}}(v_1,v_2)$ enforce the following *via* entries in the closure, in this subcase.

$Via_{\mathcal{P}}(v_2,v_1,v_3)$

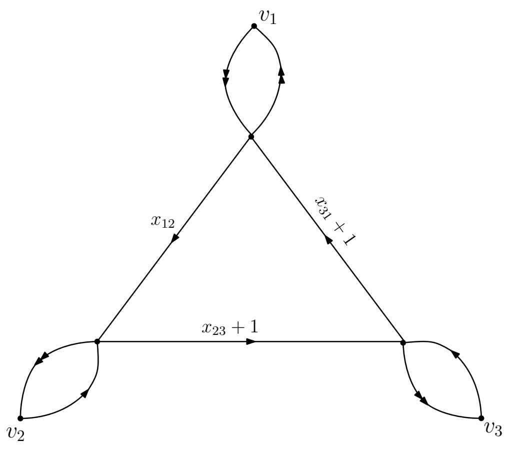
$x_{12}+x_{31}$

$Via_{\mathcal{P}}(v_3,v_1,v_2)$

$x_{12}+x_{31}$



Subcase 2) : $x_{12}+x_{23}+x_{31}=0$



Case: $(2,2)$ *c)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

The following constraints follow from the bad pairs in lower row:

$$x_{23} = x_{32}$$
$$x_{12} + x_{31} + x_{23} = 1$$

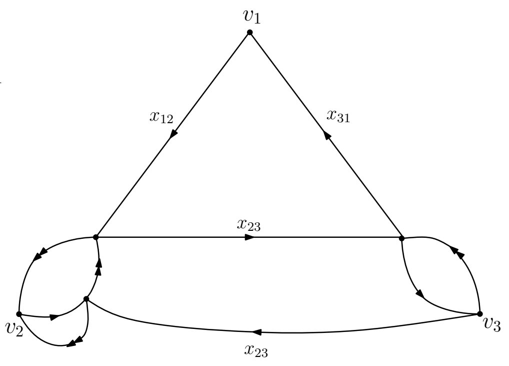
$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	x_{32}	$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_1,v_3,v_2)$

$x_{23}+x_{31}$

$Via_{\mathcal{P}}(v_3,v_1,v_2)$

$x_{12}+x_{31}$



Case: $(3,0)$ *a)*

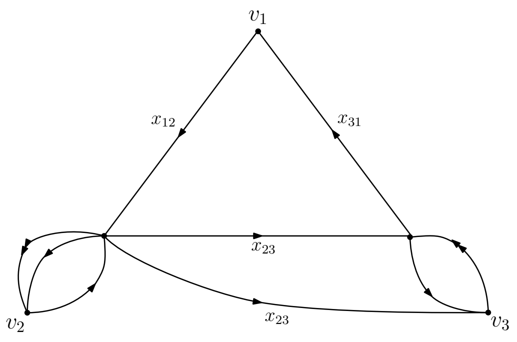
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$		$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_3,v_1,v_2)$

0

1



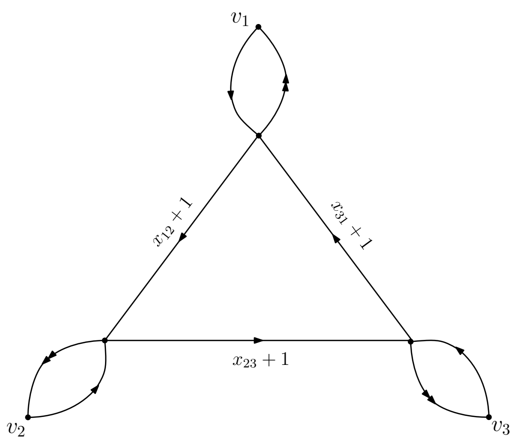
Case: $(3,0)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	$x_{31}+x_{12}+1$	$x_{12}+x_{23}+1$

There are two subcases based on parity of $x_{12}+x_{23}+x_{31}$

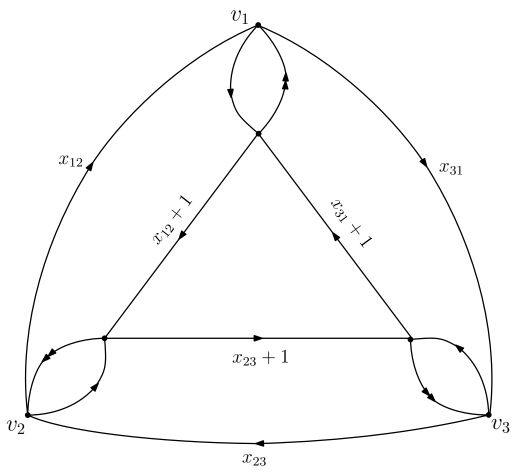
Subcase 1) : $x_{12}+x_{23}+x_{31}=0$



Subcase 2) : $x_{12}+x_{23}+x_{31}=1$

In this subcase, the following via entries must be added in the closure due to good pairs in the lower row :

$Via_{\mathcal{P}}(v_1,v_3,v_2)$	$Via_{\mathcal{P}}(v_2,v_1v_3)$	$Via_{\mathcal{P}}(v_3,v_2,v_1)$
$x_{23}+x_{31}$	$x_{31}+x_{12}$	$x_{12}+x_{23}$



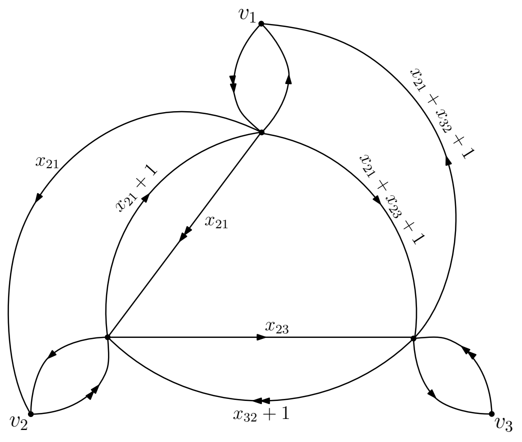
Case: $(2,1)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	$x_{21}+x_{32}+1$
$x_{12}+1$	$x_{23}+1$	

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21}	x_{32}	$x_{12}+x_{23}+1$

$Via_{\mathcal{P}}(v_2,v_3,v_1)$	$Via_{\mathcal{P}}(v_3,v_1,v_2)$
$x_{21}+1$	$x_{32}+1$

Since element of $Dir_{\mathcal{P}}(v_3,v_1)$ forms good pairs with both elements of $Dir_{\mathcal{P}}(v_1,v_2)$, there must be an element in $Via_{\mathcal{P}}(v_2,v_3,v_1)$ of parity $x_{21}+1$. Note that in the mimicking network we construct, there are v_2-v_3-v_1 paths of parity both $0,1$.



Case: $(2,1)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$	x_{32}	$x_{12}+x_{23}+1$

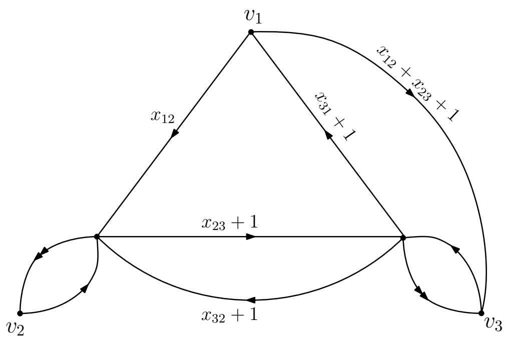
The following constraints follow from definition of bad pairs:

$$x_{23} = x_{32}$$

There are two subcases based on parity of $x_{12}+x_{23}+x_{31}$

Subcase 1) : $x_{12}+x_{31}+x_{32}=0$

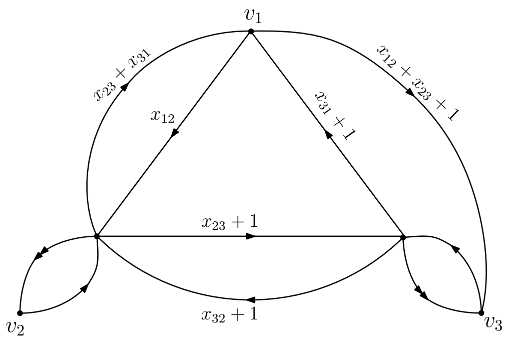
$Via_{\mathcal{P}}(v_1,v_3,v_2)$
$x_{12}+1$



Subcase 2) : $x_{12}+x_{31}+x_{32}=1$

$Via_{\mathcal{P}}(v_1,v_3,v_2)$
$x_{12}+1$

$Via_{\mathcal{P}}(v_2,v_1,v_3)$
$x_{23}+1$
$Via_{\mathcal{P}}(v_3,v_1,v_2)$
$x_{23}+1$



Case: $(2,1)$ *c)*

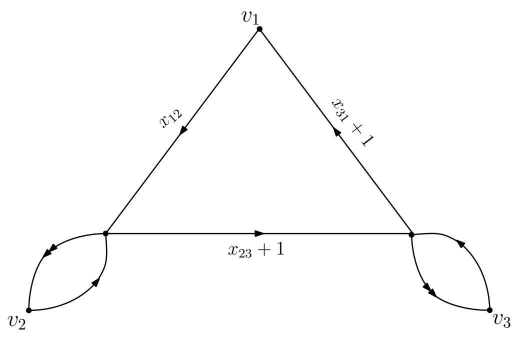
$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
$x_{23}+x_{31}+1$		$x_{12}+x_{23}+1$

The following constraints follow from definition of bad pairs:

$$x_{12}+x_{23}+x_{31}=1$$

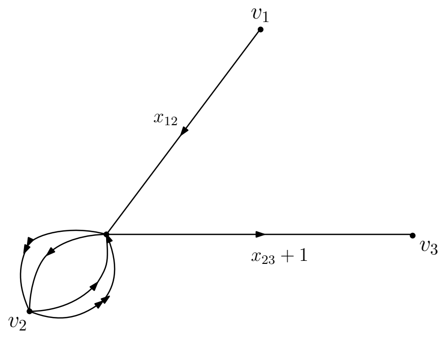
$Via_{\mathcal{P}}(v_3,v_1,v_2)$
$x_{31}+x_{12}$



Case: $(2, 0)$ *a)*

$Dir_{\mathcal{P}}(v_1, v_2)$	$Dir_{\mathcal{P}}(v_2, v_3)$	$Dir_{\mathcal{P}}(v_3, v_1)$
x_{12}	x_{23}	
$x_{12} + 1$	$x_{23} + 1$	

$Dir_{\mathcal{P}}(v_2, v_1)$	$Dir_{\mathcal{P}}(v_3, v_2)$	$Dir_{\mathcal{P}}(v_1, v_3)$
		$x_{12} + x_{23} + 1$



Case: $(2, 0)$ *b)*

$Dir_{\mathcal{P}}(v_1, v_2)$	$Dir_{\mathcal{P}}(v_2, v_3)$	$Dir_{\mathcal{P}}(v_3, v_1)$
x_{12}	x_{23}	x_{31}

$Dir_{\mathcal{P}}(v_2, v_1)$	$Dir_{\mathcal{P}}(v_3, v_2)$	$Dir_{\mathcal{P}}(v_1, v_3)$
$x_{23} + x_{31} + 1$		$x_{12} + x_{23} + 1$

The following constraints follow from definition of bad pairs:

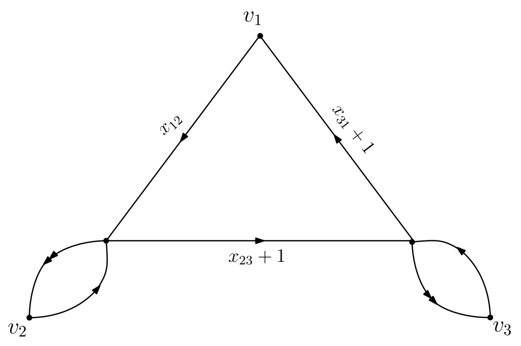
$$x_{23} = x_{32}$$

There are two subcases based on parity of $x_{12} + x_{23} + x_{31}$

$Via_{\mathcal{P}}(v_3, v_1, v_2)$

$x_{12} + x_{31}$

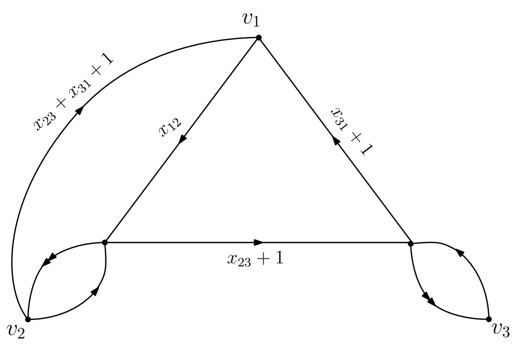
Subcase 1) : $x_{12} + x_{23} + x_{31} = 0$



Subcase 2) : $x_{12} + x_{23} + x_{31} = 1$

$Via_{\mathcal{P}}(v_2, v_1, v_3)$

$x_{12} + x_{31}$

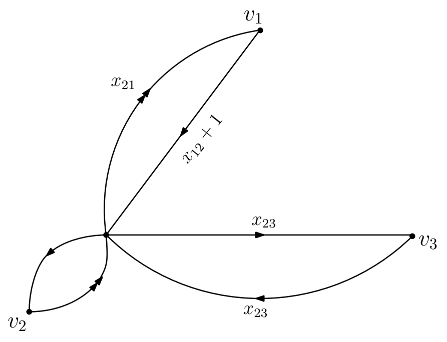


Case: $(1,1)$ *a)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{\mathcal{P}}(v_3,v_1)$
x_{12} - - - - - - -	x_{23}	$x_{21}+x_{32}+1$
$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
x_{21} - - - - - - -	x_{32}	$x_{12}+x_{23}+1$

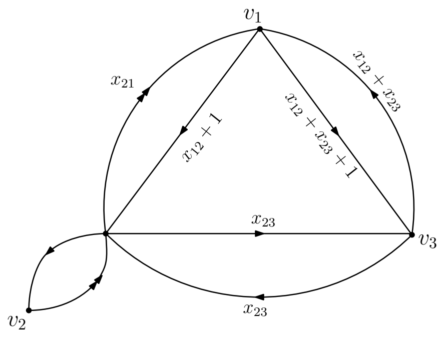
We consider three subcases based on parities of x_{21}, x_{32}. (The others are symmetric).

Subcase 1) : $x_{21} = x_{12} + 1$,
$x_{32} = x_{23} + 1$



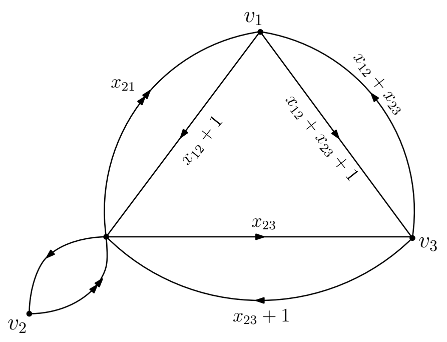
Subcase 2) : $x_{21} = x_{12}$,
$x_{32} = x_{23} + 1$

$Via_{\mathcal{P}}(v_2,v_1,v_3)$
$x_{23}+1$
$Via_{\mathcal{P}}(v_3,v_1,v_2)$
x_{23}



Subcase 3) : $x_{21} = x_{12}$,
$x_{32} = x_{23}$

$Via_{\mathcal{P}}(v_1,v_3,v_2)$	$Via_{\mathcal{P}}(v_2,v_1,v_3)$
$x_{12}+1$	$x_{23}+1$
$Via_{\mathcal{P}}(v_2,v_3,v_1)$	$Via_{\mathcal{P}}(v_3,v_1,v_2)$
$x_{12}+1$	x_{23}

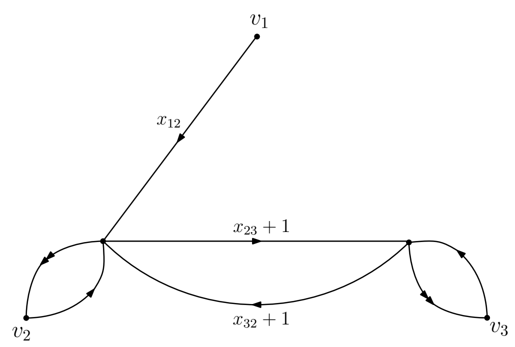


Case: $(1,1)$ *b)*

$Dir_{\mathcal{P}}(v_1,v_2)$	$Dir_{\mathcal{P}}(v_2,v_3)$	$Dir_{L}(v_3,v_1)$
x_{12} - - - - - - -	x_{23}	
$Dir_{\mathcal{P}}(v_2,v_1)$	$Dir_{\mathcal{P}}(v_3,v_2)$	$Dir_{\mathcal{P}}(v_1,v_3)$
	x_{32} - - - - -	$x_{12}+x_{23}+1$

The following constraints follow from bad pair in lower row:

$$x_{23} = x_{32}$$



Bibliography

[AA88] Alok Aggarwal and Richard J. Anderson. A random NC algorithm for depth first search. *Comb.*, 8(1):1–12, 1988.

[AAK90] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity, a modern approach.* Cambridge University Press, 2009.

[ABC$^+$09] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, 2009.

[ACD21] Eric Allender, Archit Chauhan, and Samir Datta. Depth-first search in directed planar graphs, revisited. In Filippo Bonchi and Simon J. Puglisi, editors, *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, volume 202 of *LIPIcs*, pages 7:1–7:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[ACD22] Eric Allender, Archit Chauhan, and Samir Datta. Depth-first search in directed planar graphs, revisited. *Acta Informatica*, 59(4):289–319, 2022. Preliminary version appeared in MFCS 2021.

[ACDM19] Eric Allender, Archit Chauhan, Samir Datta, and Anish Mukherjee. Planarity, exclusivity, and unambiguity. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:39, 2019.

[ADR05] Eric Allender, Samir Datta, and Sambuddha Roy. The directed planar reachability problem. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, volume 3821 of *Lecture Notes in Computer Science*, pages 238–249. Springer, 2005.

[AIK$^+$14] Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using $O(n)$ bits. In Hee-Kap Ahn and Chan-Su Shin, editors, *Proc. 25th International Symposium on Algorithms and Computation (ISAAC)*, volume 8889 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2014.

[AM04] Eric Allender and Meena Mahajan. The complexity of planarity testing. *Information and Computation*, 189(1):117–134, 2004.

[ARZ99] 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.

[Asa85] Takao Asano. An approach to the subgraph homeomorphism problem. *Theoretical Computer Science*, 38:249–267, 1985.

[BD15] Nikhil Balaji and Samir Datta. Bounded treewidth and space-efficient linear algebra. In Rahul Jain, Sanjay Jain, and Frank Stephan, editors, *Theory and Applications of Models of Computation*, pages 297–308, Cham, 2015. Springer International Publishing.

[BETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassiao, and Ioannis G. Tollis. *Graph drawing: algorithms for the visualization of graphs.* Prentice Hall, 1998.

[BHK22] Andreas Björklund, Thore Husfeldt, and Petteri Kaski. The shortest even cycle problem is tractable. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 117–130, New York, NY, USA, 2022. Association for Computing Machinery.

[BK09] Glencora Borradaile and Philip N. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, 2009.

[BTV09] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *ACM Trans. Comput. Theory*, 1(1):4:1–4:17, 2009.

[CDGS24] Archit Chauhan, Samir Datta, Chetan Gupta, and Vimal Raj Sharma. The even-path problem in directed single-crossing-minor-free graphs. In *To appear in 49th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, LIPIcs, 2024.

[CE13] Erin Wolf Chambers and David Eppstein. Flows in one-crossing-minor-free graphs. *Journal of Graph Algorithms and Applications*, 17(3):201–220, 2013.

[CF12] Yijia Chen and Jörg Flum. On the ordered conjecture. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 225–234. IEEE Computer Society, 2012.

[CFK$^+$15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[CLRS22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition.* MIT Press, 2022.

[CMPP13] Marek Cygan, Daniel Marx, Marcin Pilipczuk, and Michal Pilipczuk. The planar directed k-vertex-disjoint paths problem is fixed-parameter tractable. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 197–206, 2013.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12 – 75, 1990.

[DBT90] Giuseppe Di Battista and Roberto Tamassia. On-line graph algorithms with spqr-trees. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 598–611, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[DHN+04] Erik D Demaine, MohammadTaghi Hajiaghayi, Naomi Nishimura, Prabhakar Ragde, and Dimitrios M Thilikos. Approximation algorithms for classes of graphs excluding single-crossing graphs as minors. *Journal of Computer and System Sciences*, 69(2):166–195, 2004.

[Die16] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate texts in mathematics*. Springer, 2016.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976.

[DKTV12] Samir Datta, Raghav Kulkarni, Raghunath Tewari, and N.V. Vinodchandran. Space complexity of perfect matching in bounded genus bipartite graphs. *Journal of Computer and System Sciences*, 78(3):765–779, 2012. In Commemoration of Amir Pnueli.

[DLN+22] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Planar graph isomorphism is in log-space. *ACM Trans. Comput. Theory*, 14(2):8:1–8:33, 2022.

[dlTK95] Pilar de la Torre and Clyde P. Kruskal. Fast parallel algorithms for all-sources lexicographic search and path-algebra problems. *J. Algorithms*, 19(1):1–24, 1995.

[dlTK01] Pilar de la Torre and Clyde P. Kruskal. Polynomially improved efficiency for fast parallel single-source lexicographic depth-first search, breadth-first search, and topological-first search. *Theory Comput. Syst.*, 34(4):275–298, 2001.

[DP11] Samir Datta and Gautam Prakriya. Planarity testing revisited. In Mitsunori Ogihara and Jun Tarui, editors, *Theory and Applications of Models of Computation - 8th Annual Conference, TAMC 2011, Tokyo, Japan, May 23-25, 2011. Proceedings*, volume 6648 of *Lecture Notes in Computer Science*, pages 540–551. Springer, 2011.

[EHK15] Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient basic graph algorithms. In *Proc. 32nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[EJT10] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 143–152, 2010.

[EK14] Michael Elberfeld and Ken-ichi Kawarabayashi. Embedding and canonizing graphs of bounded genus in logspace. In *ACM Symposium on Theory of Computing (STOC)*, pages 383–392, 2014.

[End01] H.B. Enderton. *A Mathematical Introduction to Logic*. Elsevier Science, 2001.

[EV21] David Eppstein and Vijay V. Vazirani. Nc algorithms for computing a perfect matching and a maximum flow in one-crossing-minor-free graphs. *SIAM Journal on Computing*, 50(3):1014–1033, 2021.

[FGT19] Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. A deterministic parallel algorithm for bipartite perfect matching. *Commun. ACM*, 62(3):109–115, 2019.

[FHW80] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[GKR13] Martin Grohe, Ken-ichi Kawarabayashi, and Bruce Reed. A simple algorithm for the graph minor decomposition: logic meets structural graph theory. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 414–431, USA, 2013. Society for Industrial and Applied Mathematics.

[GL94] Anna Galluccio and Martin Loebl. Even/odd dipaths in planar digraphs. *Optimization Methods and Software*, 3(1-3):225–236, 1994.

[GM88] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, 1988.

[Hag90] Torben Hagerup. Planar depth-first search in O(log n) parallel time. *SIAM J. Comput.*, 19(4):678–704, June 1990.

[Hag20] Torben Hagerup. Space-efficient DFS and applications to connectivity problems: Simpler, leaner, faster. *Algorithmica*, 82(4):1033–1056, 2020.

[Hal43] Dick Wick Hall. A note on primitive skew curves. 1943.

[hh17] Tony Huynh (https://mathoverflow.net/users/2233/tony huynh). Mso2-expressible graph properties unexpressible in mso1. MathOverflow, 2017. URL:https://mathoverflow.net/q/257819 (version: 2017-01-03).

[HKNR98] Torben Hagerup, Jyrki Katajainen, Naomi Nishimura, and Prabhakar Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998.

[HT73a] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[HT73b] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, jun 1973.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, oct 1974.

[HY88] Xin He and Yaacov Yesha. A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs. *SIAM Journal on Computing*, 17(3):486–491, 1988.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[IO20] Taisuke Izumi and Yota Otachi. Sublinear-space lexicographic depth-first search for bounded treewidth graphs and planar graphs. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *Proc. 47th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 168 of *LIPIcs*, pages 67:1–67:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[JK89] B. Jenner and B. Kirsig. Alternierung und Logarithmischer Platz. Dissertation, Universität Hamburg, 1989.

[Jon75] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.

[Kao88] Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in DNC. In John H. Reif, editor, *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings*, volume 319 of *Lecture Notes in Computer Science*, pages 53–63. Springer, 1988.

[Kao93] Ming-Yang Kao. Linear-processor NC algorithms for planar directed graphs I: strongly connected components. *SIAM J. Comput.*, 22(3):431–459, 1993.

[Kao95] Ming-Yang Kao. Planar strong connectivity helps in parallel depth-first search. *SIAM J. Comput.*, 24(1):46–62, 1995.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[Khu90] Samir Khuller. Extending planar graph algorithms to k_3,3-free graphs. *Inf. Comput.*, 84(1):13–25, 1990.

[KK93] Ming-Yang Kao and Philip N. Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. *Journal of Computer and System Sciences*, 47(3):459–500, 1993.

[KMV92] Samir Khuller, Stephen G. Mitchell, and Vijay V. Vazirani. Processor efficient parallel algorithms for the two disjoint paths problem and for finding a kuratowski homeomorph. *SIAM J. Comput.*, 21(3):486–506, 1992.

[KN11] Haim Kaplan and Yahav Nussbaum. Maximum flow in directed planar graphs with vertex capacities. *Algorithmica*, 61(1):174–189, 2011.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–942. Elsevier and MIT Press, 1990.

[KR13] Robert Krauthgamer and Inbal Rika. *Mimicking Networks and Succinct Representations of Terminal Cuts*, pages 1789–1799. SIAM, 2013.

[KR14] Arindam Khan and Prasad Raghavendra. On mimicking networks representing minimum terminal cuts. *Information Processing Letters*, 114(7):365–371, 2014.

[KRW11] Ken-ichi Kawarabayashi, Bruce Reed, and Paul Wollan. The graph minor algorithm with parity conditions. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 27–36, 2011.

[KW11] Ken-ichi Kawarabayashi and Paul Wollan. A simpler algorithm and shorter proof for the graph minor decomposition. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 451–458, New York, NY, USA, 2011. Association for Computing Machinery.

[Lan37] Saunders Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460 – 472, 1937.

[Lev73] L. A. Levin. Universal sequential search problems. volume 9, page 265–266, 1973.

[LMP$^+$20] Daniel Lokshtanov, Pranabendu Misra, Michał Pilipczuk, Saket Saurabh, and Meirav Zehavi. An exponential time parameterized algorithm for planar disjoint paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 1307–1316, New York, NY, USA, 2020. Association for Computing Machinery.

[LP84] Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Lyn75] James F. Lynch. The equivalence of theorem proving and the interconnection problem. *SIGDA Newsl.*, 5(3):31–36, sep 1975.

[Men27] Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[Mil86] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.

[MIS90] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[MRST97] William McCuaig, Neil Robertson, P. D. Seymour, and Robin Thomas. Permanents, pfaffian orientations, and even directed circuits (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 402–405, New York, NY, USA, 1997. Association for Computing Machinery.

[MT01] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, 2001.

[MV00] Meena Mahajan and Kasturi R. Varadarajan. A new nc-algorithm for finding a perfect matching in bipartite planar and small genus graphs (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 351–357, New York, NY, USA, 2000. Association for Computing Machinery.

[Ned99] Zhivko Prodanov Nedev. Finding an even simple path in a directed planar graph. *SIAM J. Comput.*, 29(2):685–695, 1999.

[NVG17] Maxim Naumov, Alysson Vrielink, and Michael Garland. Parallel depth-first search for directed acyclic graphs. In *Proc. 7th Workshop on Irregular Applications: Architectures and Algorithms*, pages 4:1–4:8, 2017.

[RA00] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000.

[Rei84] John H. Reif. Symmetric complementation. *J. ACM*, 31(2):401–421, 1984.

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008.

[RRSS91] Bruce Reed, Neil Robertson, Alexander Schrijver, and Paul Seymour. Finding dsjoint trees in planar graphs in linear time. In *Graph Structure Theory, Proceedings of a AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors held June 22 to July 5, 1991, at the University of Washington, Seattle, USA*, volume 147 of *Contemporary Mathematics*, pages 295–302, 01 1991.

[RS93] N. Robertson and P.D. Seymour. Excluding a graph with one crossing. *Graph struc- ture theory (Seattle, WA, 1991)*, 1993.

[RS95] N. Robertson and P.D. Seymour. Graph minors .xiii. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[RS03] Neil Robertson and P.D Seymour. Graph minors. xvi. excluding a non-planar graph. *Journal of Combinatorial Theory, Series B*, 89(1):43–76, 2003.

[RS04] Neil Robertson and P.D. Seymour. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004. Special Issue Dedicated to Professor W.T. Tutte.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.

[Sch94] Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM J. Comput.*, 23(4):780–788, 1994.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[Sha88] Gregory E. Shannon. A linear-processor algorithm for depth-first search in planar graphs. *Information Processing Letters*, 29(3):119–123, 1988.

[Sha21] Vimal Raj Sharma. Catalytic computation and the even-path problem. In *PhD Thesis, IIT Kanpur*, 2021.

[Smi86] Justin R. Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM J. Comput.*, 15(3):814–830, 1986.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar76] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[Tho90] Carsten Thomassen. Embeddings of graphs with no short noncontractible cycles. *Journal of Combinatorial Theory, Series B*, 48(2):155–177, 1990.

[Tut75] W. T. Tutte. Separation of vertices by a circuit. *Discrete Mathematics*, 12(2):173–184, 1975.

[TV12] Raghunath Tewari and N. V. Vinodchandran. Green's theorem and isolation in planar graphs. *Inf. Comput.*, 215:1–7, 2012.

[TW10] Thomas Thierauf and Fabian Wagner. The isomorphism problem for planar 3-connected graphs is in unambiguous logspace. *Theory Comput. Syst.*, 47(3):655–673, 2010.

[Vaz89] Vijay V. Vazirani. NC algorithms for computing the number of perfect matchings in k_3,3-free graphs and related problems. *Inf. Comput.*, 80(2):152–164, 1989.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York Inc., 1999.

[Wag37] Klaus Von Wagner. Über eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114:570–590, 1937.

[YZ97] Raphael Yuster and Uri Zwick. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics*, 10(2):209–222, 1997.