Planar Graph Perfect Matching Is in NC

NIMA ANARI, Stanford University, Stanford, California
VIJAY V. VAZIRANI, University of California, Irvine, California

Is perfect matching in NC? That is, is there a deterministic fast parallel algorithm for it? This has been an outstanding open question in theoretical computer science for over three decades, ever since the discovery of RNC perfect matching algorithms. Within this question, the case of planar graphs has remained an enigma: On the one hand, counting the number of perfect matchings is far harder than finding one (the former is #P-complete and the latter is in P), and on the other, for planar graphs, counting has long been known to be in NC whereas finding one has resisted a solution.

In this article, we give an NC algorithm for finding a perfect matching in a planar graph. Our algorithm uses the above-stated fact about counting perfect matchings in a crucial way. Our main new idea is an NC algorithm for finding a face of the perfect matching polytope at which a set (which could be polynomially large) of conditions, involving constraints of the polytope, are simultaneously satisfied. Several other ideas are also needed, such as finding, in NC, a point in the interior of the minimum-weight face of this polytope and finding a balanced tight odd set.

21

CCS Concepts: • **Theory of computation → Parallel algorithms**; *Graph algorithms analysis*; Pseudorandomness and derandomization;

Additional Key Words and Phrases: Parallel algorithm, planar graph, perfect matching, Tutte matrix, Pfaffian

ACM Reference format:
Nima Anari and Vijay V. Vazirani. 2020. Planar Graph Perfect Matching Is in NC. *J. ACM* 67, 4, Article 21 (May 2020), 34 pages.
https://doi.org/10.1145/3397504

1 INTRODUCTION

Is perfect matching in NC? That is, is there a deterministic parallel algorithm that computes a perfect matching in a graph in polylogarithmic time using polynomially many processors? This has been an outstanding open question in theoretical computer science for over three decades, ever since the discovery of RNC matching algorithms [18, 26]. Within this question, the case of planar graphs has remained an intriguing one: For general graphs, counting the number of perfect matchings is far harder than finding one: the former is #P-complete [33] and the latter is in P [6]. However, for planar graphs, a polynomial time algorithm for counting perfect matchings

Part of this work was done while the first author was visiting the Simons Institute for the Theory of Computing. He was partially supported by the DIMACS/Simons Collaboration on Bridging Continuous and Discrete Optimization through NSF grant CCF-1740425.
The second author was supported in part by NSF grant CCF-1815901.
Authors' addresses: N. Anari, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305; email: anari@cs.stanford.edu; V. V. Vazirani, Donald Bren School of Information and Computer Sciences, University of California, Irvine 6210 Donald Bren Hall, Irvine, CA 92697-3425; email: vazirani@ics.uci.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
0004-5411/2020/05-ART21 $15.00
https://doi.org/10.1145/3397504

was found by Kasteleyn, a physicist, in 1967 [19], and an NC algorithm follows easily,[1] given an NC algorithm for computing the determinant of a matrix, which was obtained by Csanky [4] in 1976. On the other hand, an NC algorithm for finding a perfect matching in a planar graph has resisted a solution. In this article, we provide such an algorithm.

An RNC algorithm for the decision problem, of determining if a graph has a perfect matching, was obtained by Lovász [21], using the Tutte matrix of the graph. The first RNC algorithm for the search problem, of actually finding a perfect matching, was obtained by Karp et al. [18]. This was followed by a simpler algorithm due to Mulmuley et al. [26], which besides solving the cardinality matching problem also solved the generalization to weighted graphs, provided edge-weights are small.

It is worth noting that all the NC-based works on general graph matching since [26] have resorted to studying the weighted problem stated above, including the current article. For ease of exposition in the current article, and to highlight this unifying setup, we provide key definitions that follow from the work of [26]. Henceforth, by *small weights,* we will mean small edge weights and the *acronym MWPM* will be short for minimum weight perfect matching. The RNC algorithm of [26] for finding an MWMP in a graph with small weights has found several applications, e.g., it is a crucial ingredient in pseudo-deterministic RNC algorithms for finding a perfect matching in bipartite [12] and general graphs [1]; these are RNC algorithms with the additional requirement that when run on the same graph, they find the same (i.e., unique) perfect matching with high probability.

The perfect matching problem occupies an especially distinguished position in the theory of algorithms: Some of the most central notions and powerful tools within this theory were discovered in the context of an algorithmic study of this problem, including the notion of polynomial time solvability [6], the counting class #P [33] and a polynomial time equivalence between random generation and approximate counting for self-reducible problems [15], which lies at the core of the Markov chain Monte Carlo method. The parallel perspective also led to such a gain, namely the Isolation Lemma [26], which has found several applications in complexity theory and algorithms. In view of these facts, the problem of finding an NC algorithm for perfect matching has remained a premier open question ever since the 1980s.

The first substantial progress on this question was made by Miller and Naor in [1989] [25]. They gave an NC algorithm for finding a perfect matching in bipartite planar graphs using a flow-based approach. In 2000, Mahajan and Varadarajan [24] gave an elegant way of using an NC algorithm for counting perfect matchings to find one, hence giving a different NC algorithm for bipartite planar graphs; our work uses a similar approach.

Over the years, perhaps the most popular approach used for attacking the main open problem was derandomization of the Isolation Lemma. In the last few years, this approach has led to partial success, albeit via a partial derandomization: researchers have obtained quasi-NC algorithms for perfect matching and its generalizations. Such algorithms run in polylogarithmic time; however, they require a super-polynomial, in particular, $O(n^{\log^{O(1)} n})$ processors. Several nice algorithmic ideas have been discovered in these works and our algorithm has benefited from some of these; in turn, it will not be surprising if some of our ideas turn out to be useful for the resolution of the main open problem. First, Fenner et al. [10] gave a quasi-NC algorithm for perfect matching in bipartite graphs ; this was followed by the algorithm of Svensson and Tarnawski for general graphs [31]. Algorithms were also found for the generalization of bipartite perfect matching to the linear matroid intersection problem by Gurjar and Thierauf [13], and to a further generalization of finding a vertex of a polytope with faces given by totally unimodular constraints by Gurjar et al. [14].

[1]For a formal proof, in a slightly more general context, see [34].

We will first prove the following theorem, since it may be of more general interest.

THEOREM 1.1. *There is an* NC *algorithm which given a planar graph, returns a perfect matching in it, if it has one.*

In Section 7, we present the following generalization:

THEOREM 1.2. *There is an* NC *algorithm which, given a planar graph with small weights, finds an MWPM in it.*

We will also present an NC algorithm for finding a perfect matching in graphs of bounded genus; the common generalization of these results follows easily.

2 OVERVIEW AND TECHNICAL IDEAS

2.1 The Bipartite Case and Difficulties Imposed by Odd Cuts in General Graphs

We first give an outline of the NC algorithm of Mahajan and Varadarajan [24] for bipartite planar graphs. W.l.o.g. assume that the graph is matching-covered, i.e., each edge is in a perfect matching. Using an oracle for counting the number of perfect matchings, they find a point x in the interior of the perfect matching polytope and they show how to move this point to lower-dimensional faces of the polytope until a vertex is reached; this will be a perfect matching. Consider a face of the graph (in a planar embedding), which of course will be an even-length cycle. By the choice of x, each edge e in this cycle satisfies $0 < x_e < 1$. Modifying x by increasing and decreasing alternate edges by the same (small enough) amount ϵ moves the point inside the polytope; we will call this a *rotation* of the cycle. Keep increasing ϵ, starting from 0, until some edge e on this cycle attains $x_e = 0$[2]; in this case, e is dropped. When this happens, ϵ cannot be increased anymore and we will say that the cycle is *blocked.* If so, the point x moves to a lower (by at least one) dimension face.

To make substantial progress, Mahajan and Varadarajan [24] observe that, for any set of edge-disjoint cycles, this process can be executed independently (by different amounts) in parallel, thereby reaching a face of the polytope of correspondingly lower dimension. Finally, they show how to find $\Omega(n)$ edge-disjoint cycles (which will be edge-disjoint faces in a planar embedding) in NC, thereby terminating in $O(\log n)$ such iterations. We note that another way of measuring progress, which will also generalize to non-bipartite graphs, is the number of edges that get removed.

The fundamental difference between the perfect matching polytopes of bipartite and non-bipartite graphs is the additional set of constraints in the latter saying that for each odd set, S, of vertices we must have at least one edge in the cut $\delta(S)$ (see LP, Equation (1), in Section 3.2). Let us point out a new difficulty thatarises because of these constraints. Observe that in Figure 1, the marked even cycle cannot be rotated in the manner shown, even though the starting point x (namely $x_e = 1/3$ for each edge e) is strictly in the interior of the polytope. The reason is that on rotating the cycle, odd set S, which is tight w.r.t. x, goes under-tight, hence making its constraint infeasible. In this case, we say that *cycle C is blocked by odd set S*. Hence, in general graphs, a cycle may become blocked in one of two ways (Section 5.1). However, since cycle C cannot be rotated anymore it does not lose an edge. On the other hand, observe that since one of the odd set constraints has gone tight, we are already at a face of one lower dimension!

Moving forward, how do we capitalize on this progress? The natural way (which happens to need substantially many additional ideas to get to an NC algorithm) is to shrink S. This works because at least one edge of C must have both endpoints in S, and therefore the shrunk graph has fewer edges than the original graph. The rough outline is now as follows: The shrunk graph has

[2]It is easy to see that when some edge in the cycle attains $x_e = 1$, the adjacent edges will attain $x_e = 0$.

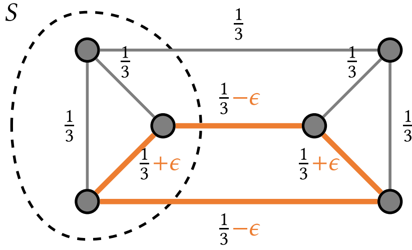

Fig. 1. Even cycle blocked by an odd set constraint. Example due to [10] and [31].

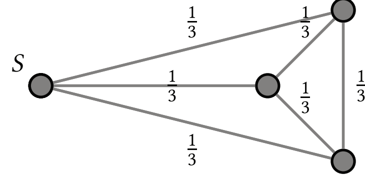

Fig. 2. Resulting graph after shrinking the blocking tight odd set.

a perfect matching, find it; expand S and remove from it the vertex that is matched via an edge in $\delta(S)$; and find a perfect matching on the remaining vertices of S. For an example of the shrunk graph, see Figure 2.

As stated above, a number of new ideas is needed to make this rough outline yield an NC algorithm. First, a small hurdle: If G is non-bipartite planar, the procedure of Mahajan and Varadarajan [24] will find $\Omega(n)$ edge-disjoint faces; however, not all of these faces may be even. In fact, there are matching-covered planar graphs having only one even face. To get around this, we define the notion of an *even walk*, first introduced in the context of parallel matching algorithms by [20]: it consists of two odd faces with a path connecting them which will be traversed in both directions; for convenience, we will call an even cycle an even walk as well (Section 3.3). We give an NC algorithm for pairing up odd cycles into even walks and thereby show how to find $\Omega(n)$ edge-disjoint even walks in G (Section 6.2). We will show that all statements made above about rotating an even cycle apply to rotating an even walk as well (this is done in Lemma 5.2). In particular, similar to a cycle, a walk is blocked either if it loses an edge or if an odd cut intersecting it goes tight; in the later case, the odd set is shrunk. In either case, at least one edge is removed.

2.2 A Central Algorithmic Issue and Its Resolution

A new algorithmic question now arises: The amount of rotation required to make a walk lose an edge is easy to compute; however, how do we find the smallest rotation to make an odd cut intersecting the walk go just tight? In particular, there may be exponentially many near-tight odd cuts intersecting the walk. Note that we need the *smallest* rotation since we do not want any odd cut to go under-tight.

We will postpone an answer to this question until we address the next hurdle, which happens to be a big one: As in the bipartite case, to make substantial progress, we need to move the point x to a face of the polytope where each of the $\Omega(n)$ even walks is blocked. The situation was far easier in bipartite graphs because a set of individually *legal rotations*—i.e, those that do not move the current point outside the polytope—on edge-disjoint cycles, if executed simultaneously in parallel are still legal. However, in the non-bipartite case, executing individually legal rotations simultaneously may take the point outside the polytope, i.e., the entire set of rotations may not legal. These two situations are illustrated in Figure 3 and Figure 4, respectively. The reason for the latter is that rotations on two different walks may be "tightening" the same odd cut. This is illustrated in Figure 5 in which the two walks can individually be rotated by ϵ_1 and ϵ_2, respectively; however, executing them both simultaneously makes the odd set S go under-tight.

The following insight, which is the central new idea in our work, helps us get around this hurdle: It suffices to find small edge=weights w so that for each of the even walks, W, the vector of motion resulting from its rotation, χ_W, satisfies $\langle w, \chi_W \rangle < 0$. Then, a minimizer of $x \mapsto \langle w, x \rangle$ in the polytope will lie in a face at which each of the walks is blocked either because it has lost an

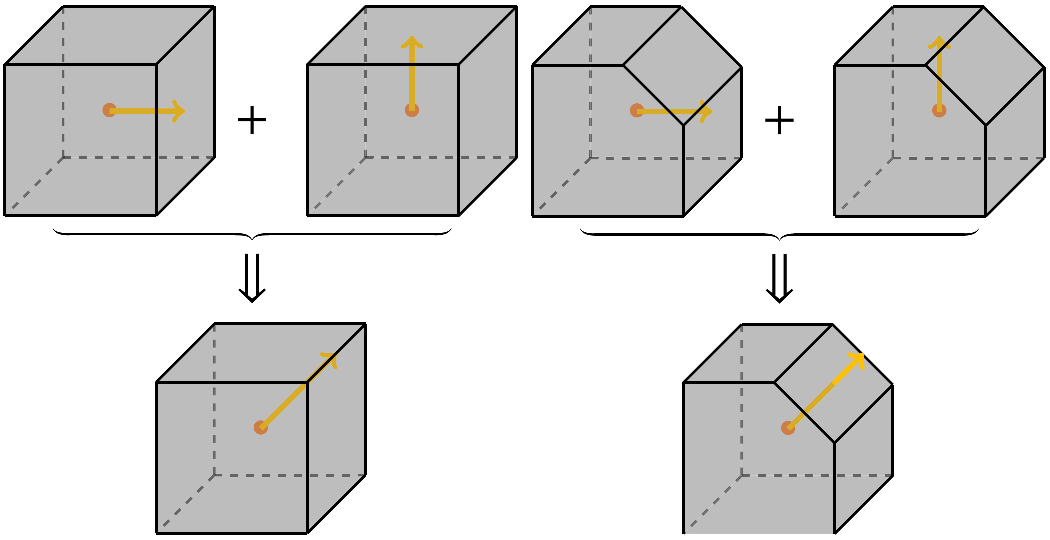

Fig. 3. Parallel moves in the bipartite perfect matching polytope.

Fig. 4. Parallel moves in the non-bipartite perfect matching polytope.

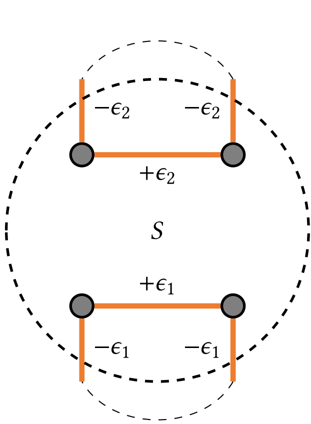

Fig. 5. Odd set constraint violated when even walks are rotated independently.

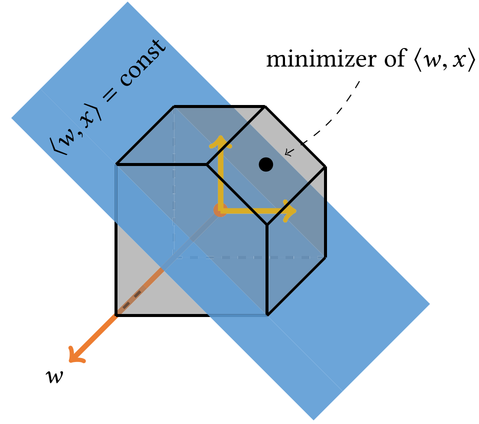

Fig. 6. Minimizer of appropriate linear function $x \mapsto \langle w, x \rangle$ blocks even walks.

edge or it intersects a tight odd cut; obviously, the minimizer is a feasible point. This is illustrated in Figure 6. The two shorter arrows indicate independent rotations of two edge-disjoint walks which lead to two different faces of the polytope. Executing them both simultaneously would take the point outside the polytope, as was illustrated in Figure 4. However, a minimizer of $\langle w, x \rangle$ lies on a face of the polytope at which both the walks are blocked.

The small edge-weights w are obtained as follows: The traversal of an even walk gives an ordered list of edges, possibly with repetition, of even-length. W.r.t. a weight function w, define the *circulation* of an even walk to be the absolute value of the difference of sum of weights of even- and odd-numbered edges in the traversal of this walk (Section 5.1); observe that the circulation of a walk is independent of the starting edge of the traversal of the walk. We show that any

weight function w that makes the circulation of each of the even walks non-zero suffices in the following sense: Given such a function w, we can pick a direction of rotation for each of the even walks so that one of the half-spaces defined by w contains the vector w and the other contains the vector of motion of each of the even walks (Lemma 5.3). Interestingly enough, such a function w is very easy to construct: in each walk, pick the weight of any one edge to be 1 and the rest 0 (Section 4.2).

Next, we need to find a minimizer, say x, of w in the polytope. Clearly, taking x to be the average of all MWPMs works. It is easy to see that this point can be specified as follows: for each edge e, $x_e = \#G_w^e/\#G_w$, where $\#G_w$ and $\#G_w^e$ are the number of MWPMs in G and the number of MWPMs in G containing the edge e, respectively. Clearly, an NC procedure for $\#G_w$ suffices for finding x. This is achieved by finding a Pfaffian orientation (Section 3.1) for G, appropriately substituting for the variables in the Tutte matrix of G and computing the determinant of the resulting matrix (Section 6.1).

Some clarifications are due at this point: First, let us answer the opening question of this section, i.e., how do we find the smallest rotation that blocks a given walk via an odd cut? Interestingly enough, at present we know of no simpler method for one walk than for multiple walks (binary search on the amount of rotation comes to mind but that is not an elegant, analytic solution). Second, let us justify using decrease in the number of edges, rather than dimension of the current point, as our measure of progress. For a single walk, both measures work. However, it could be that on rotating k edge-disjoint walks, only one odd set, S, goes tight and blocks all k walks, hence leading to a decrease in dimension of only one. On the other hand, on shrinking S, each of these k walks will lose at least one edge.

2.3 The Rest of the Ideas

A number of ideas are still needed to get to an NC algorithm. First, for each walk that does not lose an edge, we need to find a tight odd cut intersecting it. For this, we use a result of Padberg and Rao [27] stating that one of the cuts in the Gomory-Hu tree of a graph is a minimum-weight odd cut. We show how to find a Gomory-Hu tree in a weighted planar graph in NC (Section 6.3), a result of independent interest. Next, consider a walk W which is intersecting a tight odd cut w.r.t. the current point x. We rotate walk W further slightly so point x becomes infeasible, i.e., W now crosses an under-tight cut, or perhaps several of them (Lemma 5.3). Observe that rotating a walk leaves each singleton cut tight. Hence, w.r.t. x, each minimum-weight odd cut must be an under-tight odd cut that crosses W, and the Gomory-Hu tree will reveal one of them. Repeating this for all walks in parallel, we obtain a set of tight odd cuts that intersect each of the walks.

However, these odd sets cannot be shrunk simultaneously because of the following reason. Recall that two sets $S_1, S_2 \subset V$ are said to *cross* if the four sets $S_1 \cap S_2,\ S_1 - S_2,\ S_2 - S_1,\ V - (S_1 \cup S_2)$ are all non-empty. Thus, these sets are *non-crossing* if either they are disjoint or one is contained in the other. If S_1 and S_2 are crossing tight odd sets, then on shrinking them both, we will get a graph in which matching will not have the properties stated in Section 2.1. A family of subsets of V is said to be *laminar* if each pair is non-crossing. It is well known that there exists a laminar family of tight odd cuts, but how do we find it in NC?

We next give a divide-and-conquer-based procedure that removes the crossings and finds pairwise-disjoint odd sets, each corresponding to one side of a tight odd cut. These uncrossed sets correspond to the top level of a laminar family and can be shrunk simultaneously (Section 5.2). The uncrossing procedure works as follows: Partition the family of tight odd cuts into two (almost) equal subfamilies; recursively "uncross" each subfamily to obtain its top-level sets; finally, merge the two families of top-level sets into one family. Clearly, the last step is the crux of the procedure. In Section 5.2, we present a number of structural properties of the intersections of tight odd

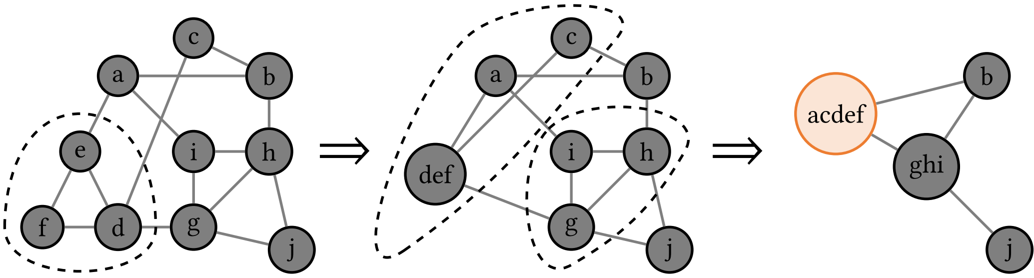

Fig. 7. Shrinking repeatedly yields a balanced viable set.

sets which eventually go to showing that they have a simple intersection structure which can be exploited appropriately.

The proposed algorithm has now evolved to the following: Shrink all top-level sets and recursively find a perfect matching in the shrunk graph; then recursively finding a perfect matching in each of the shrunk sets (after removing its matched vertex). This algorithm has polylogarithmic depth; however, it does not run in polylogarithmic time because of the following inherent sequentiality: perfect matchings in the shrunk sets have to be found *after* finding a perfect matching in the shrunk graph. The reason is that a perfect matching in a shrunk set S can be found only after knowing the vertex in S that is matched outside S. Moreover, the perfect matchings in the shrunk graph and the shrunk sets need to be found via a recursive application of the full algorithm described so far.

Let us say that odd set $S \subset V$ is *viable* if there is at least one perfect matching in G which picks exactly one edge from $\delta(S)$ and a set $S \subseteq V$ is *balanced* if both S and its complement contain a constant fraction of the vertices. Our final idea is to show how to find in NC a balanced viable odd set, and that finding such a set suffices. We give a reason for the latter claim, assuming we already found a balanced viable odd set, S. Using similar techniques to those that enabled us to find a minimizer of weight vector w in the polytope (which used the fact that counting the number of perfect matchings in planar graphs is in NC), we show how to find an edge $e \in \delta(S)$ which is the unique edge in a perfect matching from this cut. Now we are done by a simple divide-and-conquer strategy: match e, remove its end-points and find perfect matchings in the two sides of the cut recursively, in parallel. Notice that with this scheme, even though perfect matchings in the two sides can be found only after finding the matched edge e, the latter can be done without any recursive calls, hence, leading to a polylogarithmic running time. However, the task of finding a balanced viable odd set is not straightforward. It involves executing $O(\log n)$ iterations of the procedure described above, keeping track of the number of original vertices in each shrunk node, until the number of nodes is a certain constant. Finally, the node with the largest number of vertices is the required set (Section 4.2). This is illustrated in Figure 7.

3 PRELIMINARIES

In this section, we will state several notions and algorithmic primitives we need for our NC algorithm for finding a perfect matching in a planar graph.

3.1 The Tutte Matrix and Pfaffian Orientations

A key fact underlying our algorithm is that computing the number of perfect matchings in a planar graph lies in NC, see [22] and [34].

Let $G = (V, E)$ be an arbitrary graph (not necessarily planar). Let A be the symmetric adjacency matrix of G, i.e., corresponding to each edge $(i, j) \in E$, $A(i, j) = A(j, i) = 1$, and the entries

corresponding to non-edges are zero. Obtain matrix T from A by replacing for each edge $(i, j) \in E$, its two entries by x_{ij} and $-x_{ij}$, so the entries below the diagonal are positive; clearly, T is skew-symmetric. T is called the *Tutte matrix* for G. Its significance lies in that its determinant is non-zero as a polynomial iff G has a perfect matching. However, computing this determinant is not easy: Simply writing it will require exponential time in general.

Next assume that G has a perfect matching. A simple even cycle C in G is said to be *nice* if the removal of its vertices leaves a graph having a perfect matching. If so, clearly, C lies in the symmetric difference of two perfect matchings in G. Direct the edges of G to obtain $\overrightarrow{G}$. We will say that $\overrightarrow{G}$ is a *Pfaffian orientation* for G if each nice cycle C has an odd number of edges oriented in each way of traversing C. Its significance lies in the following: Let $(i, j) \in E$, with $i < j$. If in the Pfaffian orientation, this edge is directed from i to j, then let $x_{ij} = 1$, otherwise let $x_{ij} = -1$. Then the determinant of the resulting matrix is the square of the number of perfect matchings in G.

Next, we use the important fact that every planar graph has a Pfaffian orientation [19] and moreover, such an orientation can be found in NC and the determinant can be computed in NC by Csanky's algorithm [4]. Hence, we can answer the decision question of whether G has a perfect matching in NC.

3.2 The Perfect Matching Polytope, Its Faces, and Tight Odd Sets

We use the notation $\mathbb{1}_F$ to denote the indicator vector of a subset of edges $F \subseteq E$. For a subset of vertices S, we let $\delta(S)$ denote the edges that cross S, and by a slight abuse of notation, we let $\delta(v) = \delta(\{v\})$ denote the set of edges adjacent to vertex v.

The perfect matching polytope is the convex hull of indicator vectors of all perfect matchings in G and will be denoted by $PM(G)$:

$$\mathrm{PM}(G) = \mathrm{conv}\,\{\mathbb{1}_M \mid M \text{ is a perfect matching of } G\}.$$

The perfect matching polytope lives in $\mathbb{R}^E$ and is alternatively described by the following set of linear equalities and inequalities [5]:

$$\left\{ x \in \mathbb{R}^E \;\middle|\; \begin{array}{ll} \langle \mathbb{1}_{\delta(v)}, x\rangle = 1 & \forall v \in V, \\ \langle \mathbb{1}_{\delta(S)}, x\rangle \geq 1 & \forall S \subset V, \text{ with } |S| \text{ odd}, \\ x_e \geq 0 & \forall e \in E. \end{array} \right\} \tag{1}$$

For a given weight vector $w \in \mathbb{R}^E$ on edges, we can obtain minimum-weight fractional and integral perfect matchings by minimizing the linear function $x \mapsto \langle w, x\rangle = \sum_e w_e x_e$ subject to the above-stated constraints. This set of fractional and integral perfect matchings form a face of $\mathrm{PM}(G)$ and will be denoted by $\mathrm{PM}(G, w)$.

An important step needed by our algorithm is finding a point in the relative interior of the face $\mathrm{PM}(G, w)$ in NC. The algorithm of [24] obtained such a point for the case of bipartite planar graphs by using the fact that counting the number of perfect matchings in planar graphs is in NC. The current article requires a weighted-extension, using ideas from [24] and [26]. We do this by computing a Pfaffian orientation for G and then evaluating the Tutte matrix for appropriate substitutions of the variables. The point we find will be exactly the average of the vertices, i.e., $\mathbb{1}_M$ for perfect matchings M, lying on the face $\mathrm{PM}(G, w)$. We denote this average by $\mathrm{avg}(\mathrm{PM}(G, w))$. As is well known,

$$\mathrm{avg}(\mathrm{PM}(G, w)) = \frac{1}{|\{M \mid \mathbb{1}_M \in \mathrm{PM}(G, w)\}|} \sum_{M : \mathbb{1}_M \in \mathrm{PM}(G, w)} \mathbb{1}_M.$$

Lemma 3.1. *Given a planar graph $G = (V, E)$ and small edge-weights $w \in \mathbb{Z}^E$, there is an* NC *algorithm which returns* $\mathrm{avg}(\mathrm{PM}(G, w))$.

We prove Lemma 3.1 in Section 6.1.

In general, a face of PM(G) is defined by setting a particular set of inequalities to equalities. Let $\mathcal{S}$ be the family of odd sets whose inequalities are set to equality. These will be called *tight odd sets.* Two such tight odd sets $S_1, S_2 \in \mathcal{S}$ are said to *cross* if they are not disjoint and neither is a subset of the other. If so, one can prove that either $S_1 \cap S_2$ and $S_1 \cup S_2$ are also tight odd sets or $S_1 - S_2$ and $S_2 - S_1$ are tight odd sets. In the former case one can remove the equality constraint for S_1 and replace it by the equality constraints for $S_1 \cap S_2$ and $S_1 \cup S_2$, and the face would not change. In the latter case S_1 can be replaced by $S_1 - S_2$ and $S_2 - S_1$ and still the face remains invariant. In either case, the new sets do not cross. The family $\mathcal{S}$ is said to be *laminar* if no pair of sets in it cross. Given a family of tight odd sets $\mathcal{S}$, one can successively uncross pairs to obtain a family of tight odd sets defining the same face of the polytope. This operation will result in a laminar family. However, for our purposes, we only need to work with the maximal sets in the laminar family. We define a similar notion of uncrossing for such top-level sets and show how they give us the set of equality constraints, by defining things appropriately.

3.3 Finding Maximal Independent Sets and Even Walks

One of the ingredients we use in multiple ways to design our algorithm is that a maximal independent set in a graph can be found in NC.

Lemma 3.2 ([23]). *There is an* NC *algorithm for finding some maximal independent set in an input graph $G = (V, E)$.*

Kulkarni et al. [20] used Lemma 3.2 to find linearly many edge-disjoint cycles in bipartite planar graphs. We use a similar step, but instead of cycles we have to work with even walks, i.e., cycles with possibly repeated edges. We only deal with special types of even walks that are standard in graph theory; we will make them explicit next. We note that, in the past, several articles on parallel matching algorithms have used such walks.

Throughout this article, an *even walk* is used to refer to either a simple even-length cycle in G or the following structure: Let C_1 and C_2 be two odd-length edge-disjoint cycles in G and let P be a path, edge-disjoint from C_1, C_2, connecting vertex a vertex, say v_1, of C_1 to a vertex, say v_2, of C_2; if $v_1 = v_2$, P will be the empty path. Starting from v_1, traverse C_1, then P from v_1 to v_2, then traverse C_2, followed by P from v_2 to v_1.

Note that all of our walks start and end at the same location. We use Lemma 3.2 to derive the following. We prove Lemma 3.3 in Section 6.2.

Lemma 3.3. *Suppose that $G = (V, E)$ is a connected planar graph with no vertices of degree 1 and at most $|V|/2$ vertices of degree 2. Then, there is an* NC *algorithm for finding $\Omega(|E|)$ edge-disjoint even walks in G.*

4 MAIN ALGORITHM

4.1 Divide-and-Conquer Procedure

In this section, we will describe the algorithm we use to prove Theorem 1.1. W.l.o.g. assume that the input graph has a perfect matching. We can easily check whether a perfect matching exists by counting the number of perfect matchings in NC; see Section 3.1.

We use a divide-and-conquer approach. The pseudocode is given in Algorithm 1. Given a graph $G = (V, E)$, our algorithm finds an odd set $S \subset V$, selects an edge $e \in \delta(S)$ as the first edge of the perfect matching, and then recursively extends this to a perfect matching in S and $V - S$, without using any other edge of the cut $\delta(S)$.

ALGORITHM 1: Divide-and-conquer algorithm for finding a perfect matching.

```
PerfectMatching(G = (V, E))
if |V| = 0 then
    return ∅.
else
    Find a viable set S with |S|/|V| ∈ [c_1, 1 − c_1].
    Let w ← 𝟙_δ(S).
    Let x ← avg(PM(G, w)).
    Select an arbitrary edge e ∈ δ(S) with x_e > 0.
    Let G_1 be the induced graph on S with the endpoint of e removed.
    Let G_2 be the induced graph on V − S with the endpoint of e removed.
    in parallel do
        M_1 ← PerfectMatching(G_1).
        M_2 ← PerfectMatching(G_2).
    end
end
return M_1 ∪ M_2 ∪ {e}
```

Note that if M is the output of our algorithm, by definition, $|M \cap \delta(S)| = 1$. This prevents us from using an arbitrary odd set $S \subset V$ in the first step and motivates the following definition.

Definition 4.1. Given a graph $G = (V, E)$, an odd set S is called *viable* if there exists at least one perfect matching $M \subseteq E$ with $|M \cap \delta(S)| = 1$.

In order for a step of the algorithm to make significant progress, i.e., reduce the size of the graph by a constant factor, we also require the viable set to be *balanced*. That is, we require

$$c_1 \leq \frac{|S|}{|V|} \leq (1 - c_1)$$

for some small constant $0 < c_1 < 1/2$. Throughout this article, we will assume several constant upper bounds for c_1. At the end c_1 can be set to the lowest of these upper bounds.

Assuming that we are able to find a balanced viable set S in NC, we can prove Theorem 1.1.

Proof of Theorem 1.1. Since the set S found by Algorithm 1 is viable, there is at least one perfect matching N with $|N \cap \delta(S)| = 1$. On the other hand, for the weight vector $w = \mathbb{1}_{\delta(S)}$ and any perfect matching N, we have $\langle w, \mathbb{1}_N \rangle = |N \cap \delta(S)|$, which is always at least one. So the MWPMs N are exactly those that have a single edge in the cut $\delta(S)$. The point x is the average of these perfect matchings, so for any edge e with $x_e > 0$, there is at least one MWPM N containing e. This shows that $\{e\}$ can be extended to a perfect matching without using any other edge of $\delta(S)$ and therefore proves that G_1 and G_2 both have a perfect matching, an assumption we need in order to be able to recursively call the algorithm. This shows the correctness of the algorithm.

We finish the proof by showing that the algorithm is in NC. By Lemma 3.1, we can compute a point x in the polytope in NC, and we assumed the viable set S was found by an NC algorithm. So all of the steps of each recursive call can be executed in polylogarithmic time with a polynomially bounded number of processors. Notice that the recursion depth of the algorithm is at most $\log_{1/(1-c_1)}(|V|)$ which is logarithmic in the input size. This is because the size of the graph gets reduced by a factor of $1 - c_1$ in each recursion level. Since recursive calls are executed in parallel, this shows that the entire algorithm runs in polylogarithmic time. □

All that remains is finding a balanced viable set by an NC algorithm. This is done in Section 4.2.

4.2 Finding a Balanced Viable Set

In this section, we describe how to find a balanced viable set S in a graph $G = (V, E)$ by an NC algorithm; we note that the procedure works for a non-planar graph as well. We can assume w.l.o.g. that $|V| > \frac{1}{c_1}$ and $\frac{1}{1-c_1} < 2$. Therefore, a single vertex and V are not balanced viable sets.

Let us reduce the size of the graph G by either removing edges not participating in perfect matchings or shrinking tight odd sets. Any vertex in the shrunk graph corresponds to an odd set in the original graph G. This odd set is always viable. Therefore, if we manage to reduce the size of the shrunk graph sufficiently so it contains at most $1/c_1$ vertices, then the largest of the viable sets we get would have size at least $c_1|V|$. When we remove edges or shrink odd sets, we can also make sure the size of a viable set is not larger than $(1 - c_1)|V|$. Hence, in the end, we obtain a balanced viable set. See Figure 7 for a depiction.

The pseudocode is given in Algorithm 2; the procedures `Reduce(G, f)` and `Preprocess(G, f)` called from this algorithm are given in Algorithm 3 and Algorithm 4, respectively. Throughout the algorithm we maintain a mapping f from the original vertices to the vertices of the current shrunk graph and at termination, we return the pre-image of the vertex that contains c_1 fraction of the original vertices.

ALGORITHM 2: Finding a balanced viable set.

`BalancedViableSet`$(G_0 = (V_0, E_0))$
Let $G = (V, E)$ be a copy of G_0 and let $f : V_0 \to V$ be the identity map.
while $|f^{-1}(v)| < c_1|V_0|$ *for all* $v \in V$ **do**
    $G, f \leftarrow$ `Preprocess`(G, f).
    $G, f \leftarrow$ `Reduce`(G, f).
end
Find $v \in V$ for which $|f^{-1}(v)| \geq c_1|V_0|$.
return $f^{-1}(v)$.

Lemma 4.2. *The while loop in Algorithm 2 finishes as soon as* $|V| \leq \frac{1}{c_1}$.

Proof. At any point in the algorithm, we have

$$\sum_{v \in V} \frac{|f^{-1}(v)|}{|V_0|} = 1.$$

When the number of terms in the sum is less than $\frac{1}{c_1}$, one of them will be larger than c_1. □

We maintain the invariant that our graph G is planar, and has a perfect matching. This invariant is satisfied because we restrict ourselves to only the following two operations:

(1) Remove an edge e from G, where e does not participate in any MWPM for some weight vector w.
(2) Shrink a set S of vertices that is a tight odd set w.r.t. some point x in the perfect matching polytope.

Lemma 4.3. *If a graph G is planar, has a perfect matching and is matching-covered, then after the removal of an edge or shrinking of a tight odd set as described above, it continues to remain planar and have a perfect matching.*

Proof. The lemma is obvious in the case of removing an edge. Planarity is automatically satisfied, and since the edge did not participate in any MWPM, the remaining graph still has a MWPM.

In the case of shrinking a tight odd set, first note that the resulting graph would still have a perfect matching. More precisely, let x be a point in the perfect matching polytope of G and let x' be the vector obtained after shrinking S. Then, x' is a valid point in the perfect matching polytope of the shrunk graph; the degree constraint of the shrunk vertex is satisfied because the odd set constraint of S was originally tight.

We next show that, after shrinking S, the graph remains planar. If the graph induced on S is connected, this follows from the fact that contracting edges preserves planarity. So assume that the graph induced on S is not connected. Observe that the latter graph cannot contain two or more odd components because each of these components would need a matched edge from $\delta(S)$, contradicting the assumption that S is a tight odd set. On the other hand, a simple parity argument shows that there must be at least one odd component. So there is exactly one odd component, that we call S_1.

Next assume that the induced graph on S has some component S_2, other than S_1. We have already shown that S_2 must be even. Note that $\langle \mathbb{1}_{\delta(S_1)}, x\rangle \leq \langle \mathbb{1}_{\delta(S)}, x\rangle$ and equality holds if and only if S_2 does not have any outgoing edges in the entire graph. But $\langle \mathbb{1}_{\delta(S)}, x\rangle = 1$, and $\langle \mathbb{1}_{\delta(S_1)}, x\rangle \geq 1$, so there must be equality. This shows that all edges in $\delta(S)$ belong to $\delta(S_1)$, and that S_2 is a connected component in the entire graph. We see that the induced graph on S must consist of one odd component, and some number of even components that are connected components in the entire graph as well. Shrinking each of the latter results in a vertex. Shrinking these vertices with the odd component preserves planarity. □

It is also easy to see that the pre-image of any viable set at any point in the resulting graph is a viable set in the original graph, because any perfect matching in G can be extended to a perfect matching in G_0. Therefore we can return $f^{-1}(v)$ at any point if it is a balanced set.

The main loop in Algorithm 2 has two steps, `Preprocess` and `Reduce`. Although not explicitly stated in the pseudocode, at any point in the execution of either step, we can terminate the whole procedure by finding a balanced viable set and directly returning it.

First, we preprocess the graph. Below, we state the properties we expect to hold after preprocessing. We postpone the description of the procedure `Preprocess` and the proof of Lemma 4.4 to Section 4.3.

Lemma 4.4. *The procedure* `Preprocess` *either finds a balanced viable set or after it returns the following conditions hold.*

(1) G is connected.
(2) No vertex $v \in V$ has degree 1 and at most half of the vertices have degree 2.
(3) For all $v \in V$, we have $|f^{-1}(v)| < c_1|V_0|$.

Now we describe the main step, i.e., `Reduce`. The pseudocode is given in Algorithm 3.

Assuming Lemma 4.4, our goal is to either remove a constant fraction of the edges of G or shrink pieces of G so that a constant fraction of the edges get shrunk. The conditions satisfied after the preprocessing step, Lemma 4.4, ensure that we can apply Lemma 3.3 and find $\Omega(|E|)$ edge-disjoint even walks, as we do in the first step of Algorithm 3.

Next, we construct a weight vector which is 0 everywhere except for the first edge of every even walk, and find a point x in the relative interior of $\mathrm{PM}(G, w)$ by applying Lemma 3.1. By our choice of weight vector, each even walk either loses an edge or gets blocked by a tight odd set as we will prove in Section 5. Our last step consists of finding a number of disjoint odd sets $S_1, \ldots, S_l$, such that each even walk W_i, that did not lose an edge, has an edge with both endpoints in one S_j. We describe the procedure `DisjointOddSets` and prove these properties in Section 5.

Now we can prove the following:

ALGORITHM 3: Reducing the size of a graph by shrinking vertices and removing edges.

```
Reduce (G = (V, E), f : V_0 → V)
Find Ω(|E|) edge-disjoint even walks W_1, ..., W_k.
Let w ← 0, the zero weight vector.
for W ∈ {W_1, ..., W_k} in parallel do
 |  Set w_e ← 1 for the first edge e of W.
end
Let x ← avg(PM(G, w)).
for e ∈ E with x_e = 0 in parallel do
 |  Remove edge e from G.
end
Let 𝒲 = {W_i | W_i did not lose an edge}.
Let S_1, ..., S_l ← DisjointOddSets(G, f, x, 𝒲).
Shrink each S_i into a single vertex and update f on f^{-1}(S_i) to point to the new vertex.
```

Lemma 4.5. *After running* Reduce *we either find a balanced viable set, or* $|E|$ *gets reduced by a constant factor.*

Proof. We find k even walks where $k = \Omega(|E|)$. Every walk either loses an edge in the edge removal step, or loses an edge after shrinking $S_1, \ldots, S_l$. So the number of edges gets reduced by at least k, which is a constant fraction of $|E|$ as long as $|E|$ is large enough (larger than a large enough constant). Note that we never encounter graphs with $|V| < 1/c_1$ by Lemma 4.2, so by setting c_1 small enough we can assume that $|E|$ is larger than a desired constant. □

By Lemma 4.5, our measure of progress, namely $|E|$, gets reduced by a constant factor each time until we find a balanced viable set. Therefore, the number of times Reduce is called is at most $O(\log(|E_0|))$, so as long as DisjointOddSets can be run in NC, the whole algorithm is in NC.

We describe the remaining pieces, Preprocess in Section 4.3, and DisjointOddSets in Section 5.

4.3 Preprocessing

Here we describe the procedure Preprocess and prove Lemma 4.4. The pseudocode is given in Algorithm 4. Throughout the process, we make sure that $|f^{-1}(v)| < c_1|V_0|$ for every v or we find a balanced viable set.

In the first step, we remove any edge of G that does not participate in a perfect matching. Next, we make the graph connected. We arrive at a connected graph where every edge participates in a perfect matching. This ensures that there are no vertices of degree 1, unless the entire graph is a single edge; but in that case, we return $f^{-1}(v)$ as a balanced viable set for one of the two vertices.

After having a connected graph with no vertices of degree 1, while half of the vertices have degree 2, we shrink them into other vertices by finding appropriate tight odd sets. The while loop can be run at most a logarithmic number of times, because each time the number of vertices gets reduced by a factor of 2.

It remains to describe the procedures MakeConnected and ShrinkDegreeTwos. Both of these procedures work by shrinking tight odd sets w.r.t. x. In both, we have to be slightly careful to avoid shrinking a large piece of the original graph causing a violation of the condition $|f^{-1}(v)| < c_1|V_0|$.

First, let us describe MakeConnected. We first find the connected components $C_1, \ldots, C_k$ of G. We sort them to make sure $|f^{-1}(C_1)| \leq \cdots \leq |f^{-1}(C_k)|$. Let v be an arbitrary vertex of C_k. For any $i < k$, the set $S_i = \{v\} \cup C_1 \cup \ldots C_i$ is a tight odd set, because $\{v\}$ is a tight odd set and adding

ALGORITHM 4: The preprocessing step.

```
Preprocess (G = (V, E), f : V_0 → V)
Let x ← avg(PM(G)).
for e ∈ E with x_e = 0 in parallel do
  | Remove e from G.
end
Let G, f ← MakeConnected(G, f).
while |{v ∈ V | deg(v) = 2}| > |V|/2 do
  | Let G, f ← ShrinkDegreeTwos(G, f).
end
```

entire connected components does not change the cut value. If $|f^{-1}(S_{k-1})| < c_1|V_0|$, then we can simply shrink S_{k-1} into a single vertex and make the graph connected. Otherwise, let j be the first index where $|f^{-1}(S_j)| \geq c_1|V_0|$. Then, $f^{-1}(S_j)$ is a viable set, because it is a tight odd set. We claim that it is balanced as well; for this we need to show that $|f^{-1}(S_j)| \leq (1 - c_1)|V_0|$. We have

$$|f^{-1}(S_j)| = |f^{-1}(S_{j-1})| + |f^{-1}(C_j)| \leq c_1|V_0| + \frac{1}{2}|V_0|,$$

where we used the fact that C_j is not the largest component in terms of $f^{-1}(C_j)$. So as long as $c_1 + 1/2 < 1 - c_1$, we are done. This is clearly satisfied for small enough c_1.

Now let us describe ShrinkDegreeTwos, which is based on a similar step in [20]. First we identify all vertices of degree 2. Some of these vertices might be connected to each other, in which case we get paths formed by these vertices. We can extend these paths, by the doubling trick in polylogarithmic time to find maximal paths consisting of degree 2 vertices. Then, in parallel, for each such maximal path we do the following: Let the vertices of the path be $(v_1, \ldots, v_k)$. Further, let v_0 be the vertex we would get if we extended this path from the v_1 side and v_{k+1} the one we would get from the v_k side. Note that $\deg(v_i) = 2$ for $i = 1, \ldots, k$ but not for $i = 0, k+1$.

We claim that for any even i, the set $S_i = \{v_0, v_1, \ldots, v_i\}$ is a tight odd set. To see this, let $t = x_{(v_0, v_1)}$. Then, because v_1 has degree 2, it must be that $x_{(v_1, v_2)} = 1 - t$. Then, this means that $x_{(v_2, v_3)} = t$, and so on. In the end, we get that $x_{(v_{i-1}, v_i)} = 1 - t$. Now, look at the edges in $\delta(S)$. They are either adjacent to v_0 or v_i. Those adjacent to v_0 have a total x value of $1 - t$ and those adjacent to v_i have a total x value of t. So $\langle \mathbb{1}_{\delta(S_i)}, x \rangle = t + (1 - t) = 1$.

Now, let j be the first even index such that $|f^{-1}(S_j)| \geq c_1|V_0|$. If no such index exists, we can simply shrink S_k or S_{k+1} (depending on the parity of k). Else, we claim that S_j is a balanced viable set. Viability follows from being a tight odd set. Being balanced follows because

$$|f^{-1}(S_j)| = |f^{-1}(S_{j-2})| + |f^{-1}(v_{j-1})| + |f^{-1}(v_j)| \leq c_1|V_0| + c_1|V_0| + c_1|V_0|.$$

So as long as $3c_1 \leq (1 - c_1)$, the set S_j is balanced and we can simply return it.

Having all of the ingredients, we now finish the proof of Lemma 4.4.

Proof of Lemma 4.4. It is easy to see that the point $x \in \mathrm{PM}(G)$ remains a valid point throughout, i.e., it remains in the perfect matching polytope even after shrinking sets. This is because we only shrink tight odd sets w.r.t. x. Assume that the algorithm does not find a balanced viable set.

After MakeConnected the graph becomes connected, and from then on it remains connected. Since x remains a valid point in the perfect matching polytope until the end, every edge at the end participates in a perfect matching. But in a connected graph, this means that there are no vertices of degree 1.

Note that each time `ShrinkDegreeTwos` is called, the number of vertices in the graph gets reduced. It is easy to see that for each path of length k consisting of degree 2 vertices, the number of vertices that disappear during shrinking is at least k. So `ShrinkDegreeTwos` reduced the number of vertices by at least the number of degree 2 vertices.

Finally, note that by the stopping condition of the while loop, the algorithm terminates only when at most half of the remaining vertices have degree 2. So until then, the number of vertices in the graph gets at least halved in each iteration. Therefore, the number of iterations is at most logarithmic. □

5 TIGHT ODD SETS

In this section, we describe the main remaining piece of the algorithm, namely the procedure `DisjointOddSets`. The input to this procedure is a graph $G = (V, E)$ and a map $f : V_0 \to V$, a number of edge-disjoint even walks $W_1, \ldots, W_m$ in G, the point $x = \text{avg}(\text{PM}(G, w))$, where w is the weight vector constructed in Algorithm 3. Note that $x_e > 0$ for all $e \in E$, since we removed all edges e with $x_e = 0$. We will prove the following:

Lemma 5.1. *There is an* NC *algorithm* `DisjointOddSets`*, that either finds a balanced viable set, or finds disjoint tight odd sets $S_1, \ldots, S_l$ satisfying the following: In any W_i there is an edge e both of whose endpoints belong to some S_j. Furthermore, $|f^{-1}(S_j)| < c_1|V_0|$ for all j.*

At a high level, the procedure works as follows:

(1) First, for each even walk W_i, we find a tight odd set blocking it.
(2) The resulting tight odd sets might cross each other in arbitrary ways. We uncross them to obtain $S_1, \ldots, S_l$, being careful not to produce sets with $|f^{-1}(S_i)| \geq c_1|V_0|$.

In Section 5.1, we describe the procedure for finding a tight odd set blocking an even walk. Then, in Section 5.2, we describe how to uncross these and produce disjoint odd sets.

5.1 Finding a Tight Odd Set Blocking an Even Walk

In this section, we describe how to find a tight odd set blocking a given even walk W. At a high level, we first move slightly outside of the polytope by moving along a direction defined by W. Then, we find one of the violated constraints defining the perfect matching polytope. This must be the tight odd set we were after.

Recall that an even walk is either a simple even-length cycle in G or the following structure: Let C_1 and C_2 be two odd-length edge-disjoint cycles in G and let P be a path connecting vertex v_1 of C_1 to vertex v_2 of C_2; if $v_1 = v_2$, P will be the empty path. Starting from v_1, traverse C_1, then P from v_1 to v_2, then traverse C_2, followed by P from v_2 to v_1.

Next, we define the alternating vector of an even walk W. For this purpose, write W as a list of edges $W = (e_1, \ldots, e_k)$, where k is even and if the walk contains a path, then the edges of the path will be repeated twice in this list. We define the alternating vector associated to W as the vector χ_W given by

$$\chi_W = -\mathbb{1}_{e_1} + \mathbb{1}_{e_2} - \mathbb{1}_{e_3} + \cdots + \mathbb{1}_{e_k} = \sum_i (-1)^i \mathbb{1}_{e_i}.$$

In terms of its components, we have

$$(\chi_W)_e = \begin{cases} (-1)^i & \text{if } e = e_i \text{ and } e \text{ is not on the path in } W, \\ 2(-1)^i & \text{if } e = e_i \text{ and } e \text{ is on the path in } W, \\ 0 & \text{if } e \neq e_i \text{ for any } i. \end{cases}$$

Note that for a weight vector w, we have

$$\langle w, \chi_W \rangle = -w_{e_1} + w_{e_2} - w_{e_3} + \cdots + w_{e_k}.$$

In particular for the weight vector chosen in Algorithm 3, we have $\langle w, \chi_W \rangle < 0$.

We next define the notion of *rotation of an even walk*. For a given reference point $x \in \mathbb{R}^E$ an ϵ-rotation by W is simply the point $y = x + \epsilon \chi_W$. We remark that ϵ will always have a small, though still inverse exponentially large, magnitude. As a simple observation, note that

$$\langle w, y \rangle = \langle w, x \rangle + \epsilon \cdot \langle w, \chi_W \rangle < \langle w, x \rangle.$$

Note that the point x is avg(PM(G, w)), i.e., we have

$$x = \frac{\mathbb{1}_{M_1} + \cdots + \mathbb{1}_{M_m}}{m},$$

where $M_1, \ldots, M_m$ are all the MWPMs in G.

We will now see what happens to an ϵ-rotation of this point if ϵ is small enough.

Lemma 5.2. *Let $x = \text{avg}(\text{PM}(G, w))$ for some weight vector w. Let W be an even walk whose edges are in the support of x, i.e., for every $e \in W$, we have $x_e > 0$, and let $\langle w, \chi_W \rangle < 0$. Let $K(n) \leq n^n$ denote the number of perfect matchings in the complete graph K_n, and let y be an ϵ-rotation of x with the walk W for some $\epsilon < 1/(2nK(n))$. Then, the following hold:*

(1) For every vertex v, we have $\langle \mathbb{1}_{\delta(v)}, y \rangle = 1$.
(2) For every odd set $S \subset V$, if $\langle \mathbb{1}_{\delta(S)}, x \rangle > 1$, then $\langle \mathbb{1}_{\delta(S)}, y \rangle \geq 1$.
(3) For every edge $e \in E$, we have $y_e \geq 0$.

Proof. Condition (1) holds because $\langle \mathbb{1}_{\delta(v)}, \chi_W \rangle = 0$. This identity holds, because the walk W enters and exits each vertex v the same number of times, and the entries and exits have alternating signs, cancelling each other.

Condition (2) holds, because when $\langle \mathbb{1}_{\delta(S)}, x \rangle > 1$, then it is larger than 1 by a margin; choosing ϵ small enough will not let us erase more than this margin. Formally, we have

$$\langle \mathbb{1}_{\delta(S)}, x \rangle = \frac{\langle \mathbb{1}_{\delta(S)}, \mathbb{1}_{M_1} \rangle + \cdots + \langle \mathbb{1}_{\delta(S)}, \mathbb{1}_{M_m} \rangle}{m},$$

and note that $\langle \mathbb{1}_{\delta(S)}, \mathbb{1}_{M_i} \rangle$ is at least 1 and must be greater than 1 for some i. For that particular i, this value must be at least 2 (in fact, at least 3), which gives us

$$\langle \mathbb{1}_{\delta(S)}, x \rangle \geq 1 + \frac{1}{m}.$$

Now, note that $\|\chi_W\|_1 \leq 2n$ and $\|\mathbb{1}_{\delta(S)}\|_\infty \leq 1$ which together imply that

$$|\langle \mathbb{1}_{\delta(S)}, \chi_W \rangle| \leq 2n.$$

Finally, piecing things together, we have

$$\langle \mathbb{1}_{\delta(S)}, y \rangle = \langle \mathbb{1}_{\delta(S)}, x \rangle + \epsilon \langle \mathbb{1}_{\delta(S)}, \chi_W \rangle \geq 1 + \frac{1}{m} - 2n\epsilon \geq 1 + \frac{1}{m} - \frac{2n}{2nK(n)} \geq 1.$$

Condition (3) holds, because again, $x_e > 0$ implies that x_e is positive by a margin. We have

$$x_e = \frac{(\mathbb{1}_{M_1})_e + \cdots + (\mathbb{1}_{M_m})_e}{m} \geq \frac{1}{m},$$

which implies that $y_e \geq 1/m - 2\epsilon \geq 0$. □

Lemma 5.2 almost ensures that the point y is inside the perfect matching polytope PM(G) if the starting point x was in PM(G). The only way that y cannot be in PM(G) is if there is an odd set $S \subset V$ such that $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$, i.e., a tight odd set, whose constraint gets violated by y. This

leads us to the following important lemma, which enables us to extract a tight odd set *blocking* the rotation of the walk W.

Lemma 5.3. *Suppose w is a weight vector, $x = \text{avg}(\text{PM}_w(G))$, W is a walk that satisfies the conditions of Lemma 5.2, and furthermore $\langle w, \chi_W \rangle < 0$. Then there must be an odd set $S \subset V$ such that $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, \chi_W \rangle \neq 0$. Furthermore, such an S can be found by first obtaining y as an ϵ-rotation of x by W, for a small but inverse exponentially large ϵ, and then finding a minimum odd cut in y:*

$$\operatorname*{argmin}_{S \subset V, |S| \text{ is odd}} \langle \mathbb{1}_{\delta(S)}, y \rangle.$$

Proof. Since $\langle w, \chi_W \rangle < 0$, we have $\langle w, y \rangle < \langle w, x \rangle$. We choose the magnitude of ϵ to be small enough that the conditions of Lemma 5.2 are satisfied. Now, since x was a minimizer of the linear function $x \mapsto \langle w, x \rangle$ over the polytope PM(G), it must be the case that $y \notin \text{PM}(G)$.

Therefore, one of the constraints defining the perfect matching polytope, Equation (1), must not be satisfied for y. But Lemma 5.2 ensures that almost all of these constraints are satisfied; the only possible constraint being violated would be an odd set S such that $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$. Take any such set S where $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$. We have

$$\langle \mathbb{1}_{\delta(S)}, y \rangle = \langle \mathbb{1}_{\delta(S)}, x \rangle + \epsilon \langle \mathbb{1}_{\delta(S)}, \chi_W \rangle,$$

which means that $\langle \mathbb{1}_{\delta(S)}, \chi_W \rangle \neq 0$. In other words, S satisfies the statement of the lemma.

It only remains to show that if we take S to be a minimum odd cut in y, then S satisfies $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$. We know that the only possible constraint being violated by y is an odd set constraint, so for the minimum odd cut it must be true that $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$. On the other hand, if $\langle \mathbb{1}_{\delta(S)}, x \rangle > 1$, then we would get a contradiction from condition 2 of Lemma 5.2, because that would imply $\langle \mathbb{1}_{\delta(S)}, y \rangle \geq 1$. So such a set must satisfy $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$. □

We will say that an odd set S such that $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$ and $\langle \mathbb{1}_{\delta(S)}, \chi_W \rangle \neq 0$, is a set that *blocks* the walk W. By combining the following lemma with Lemma 5.3, we get that we can find a tight odd set blocking each of our even walks.

Lemma 5.4. *There is an* NC *algorithm that given a weight planar graph G, outputs the minimum odd cut of G.*

We will prove Lemma 5.4 in Section 6.3.

5.2 Uncrossing Tight Odd Sets

Suppose we are given a list of tight odd sets $S_1, \ldots, S_m$ that could cross each other arbitrarily.

Note that we can assume from the beginning that for each i, $|f^{-1}(S_i)| \leq \frac{1}{2}|V_0|$. If not, we simply replace S_i by $V - S_i$. We can even further assume that $|f^{-1}(S_i)| < c_1|V_0|$; otherwise, we would return $f^{-1}(S_i)$ as a balanced viable set and end the procedure. Throughout the algorithm, we maintain this property.

Our goal is to *uncross* the sets $S_1, \ldots, S_m$, so that we can shrink all of them at the same time. We make progress from shrinking these sets by making sure that each of our even walks has an edge inside at least one of the shrunk sets, so that shrinking reduces the number of edges by at least the number of walks.

Unfortunately, having an edge inside an S_i is not a property that is preserved by uncrossing. Instead, we require a stronger property that implies having an edge in one S_i, and show that this stronger property is preserved by uncrossing. Throughout this section, we assume that x is some fixed point in PM(G) with $x_e > 0$ for all $e \in E$.

Definition 5.5. For a set $S \subseteq V$, define $\Lambda(S) \subseteq \mathbb{R}^E$ to be the linear subspace defined as the span of cut indicators of all tight odd sets contained in S:

$$\Lambda(S) := \text{span}\{\mathbb{1}_{\delta(T)} \mid T \subseteq S, |T| \text{ is odd}, \langle \mathbb{1}_{\delta(T)}, x\rangle = 1\}.$$

We extend this definition to more than one set $S_1, \ldots, S_m$ by letting

$$\Lambda(S_1, \ldots, S_m) := \Lambda(S_1) + \cdots + \Lambda(S_m).$$

We also use the notation $\Lambda^\perp(S_1, \ldots, S_m)$ to denote the subspace of $\mathbb{R}^E$ orthogonal to $\Lambda(S_1, \ldots, S_m)$.

Next, we will show that χ_W not being orthogonal to $\Lambda(S_1, \ldots, S_m)$ implies that W has an edge in one $E(S_i)$.

Lemma 5.6. *Let W be an even walk, and assume that $\chi_W \notin \Lambda^\perp(S_1, \ldots, S_m)$. Then, there is at least one edge $e \in W$ and at least one i such that $e \in E(S_i)$.*

Proof. It is easy to see that $\chi_W \notin \Lambda^\perp(S_1, \ldots, S_m)$ implies that there is at least one i such that $\chi_W \notin \Lambda^\perp(S_i)$. It follows from Definition 5.5 that there must be some tight odd set $T \subseteq S_i$ such that $\langle \mathbb{1}_{\delta(T)}, \chi_W\rangle \neq 0$. We will show that $\langle \mathbb{1}_{\delta(T)}, \chi_W\rangle \neq 0$ implies that there is some $e \in W$ such that $e \in E(T) \subseteq E(S_i)$.

Suppose the contrary, that no edge $e \in W$ is in $E(T)$. Let $W = (e_1, \ldots, e_k)$ and note that

$$\langle \mathbb{1}_{\delta(T)}, \chi_W\rangle = \sum_{j=1}^{k} (-1)^j \langle \mathbb{1}_{\delta(T)}, \mathbb{1}_{e_j}\rangle.$$

Every time that W enters a vertex $v \in T$, it must leave immediately from T, or else we would find an edge $e \in E(T) \cap W$. Therefore, we can pair up the nonzero $\langle \mathbb{1}_{\delta(T)}, \mathbb{1}_{e_j}\rangle$s into consecutive pairs, possibly pairing up the last edge with the first. Since these pairs appear in the sum with alternating signs, they cancel each other, giving us

$$\langle \mathbb{1}_{\delta(T)}, \chi_W\rangle = 0,$$

which is a contradiction. Therefore, W must have at least one edge in $E(T) \subseteq E(S_i)$. □

Next we will define our basic *uncrossing* operations and show that they preserve this nonorthogonality property. Whenever we have two tight odd sets S_1 and S_2, we will show that we can uncross them, i.e., replace them by new tight odd sets without shrinking the subspace $\Lambda(S_1) + \Lambda(S_2)$. We will use the following uncrossing lemma, which is standard in the literature. We will prove it for the sake of completeness.

Lemma 5.7. *If S_1 and S_2 are tight odd sets, then either $S_1 \cap S_2, S_1 \cup S_2$ are tight odd sets and*

$$\mathbb{1}_{\delta(S_1)} + \mathbb{1}_{\delta(S_2)} = \mathbb{1}_{\delta(S_1 \cap S_2)} + \mathbb{1}_{\delta(S_1 \cup S_2)},$$

or $S_1 - S_2$ and $S_2 - S_1$ are tight odd sets and

$$\mathbb{1}_{\delta(S_1)} + \mathbb{1}_{\delta(S_2)} = \mathbb{1}_{\delta(S_1 - S_2)} + \mathbb{1}_{\delta(S_2 - S_1)}.$$

Proof. The following identity holds for any S_1 and S_2 and can be easily checked by considering all possible configurations of the endpoints of an arbitrary edge:

$$\mathbb{1}_{\delta(S_1)} + \mathbb{1}_{\delta(S_2)} = \mathbb{1}_{\delta(S_1 \cap S_2)} + \mathbb{1}_{\delta(S_1 \cup S_2)} + 2\mathbb{1}_{\delta(S_1 - S_2, S_2 - S_1)}.$$

We have two cases: Either $|S_1 \cap S_2|$ is odd or it is even.

Case 1. Assume that $|S_1 \cap S_2|$ is odd. It follows that $|S_1 \cup S_2|$ is also odd. Then by taking the dot product with x, we get

$$1 + 1 = \langle \mathbb{1}_{\delta(S_1)}, x\rangle + \langle \mathbb{1}_{\delta(S_2)}, x\rangle \geq \langle \mathbb{1}_{\delta(S_1 \cap S_2)}, x\rangle + \langle \mathbb{1}_{\delta(S_1 \cup S_2)}, x\rangle \geq 1 + 1,$$

where the last inequality follows from the fact that $x \in \text{PM}(G)$ and that $S_1 \cap S_2$ and $S_1 \cup S_2$ are odd sets. Since this inequality is tight, it must be the case that $\langle \mathbb{1}_{\delta(S_1 \cap S_2)}, x \rangle = \langle \mathbb{1}_{\delta(S_1 \cup S_2)}, x \rangle = 1$, which proves that $S_1 \cap S_2$ and $S_1 \cup S_2$ are tight odd sets. It further follows that

$$\langle \mathbb{1}_{\delta(S_1 - S_2, S_2 - S_1)}, x \rangle = 0,$$

which implies that $\mathbb{1}_{\delta(S_1 - S_2, S_2 - S_1)} = 0$, i.e., $\delta(S_1 - S_2, S_2 - S_1) = 0$; this is because x has strictly positive entries. Now we have the desired identity

$$\mathbb{1}_{\delta(S_1)} + \mathbb{1}_{\delta(S_2)} = \mathbb{1}_{\delta(S_1 \cap S_2)} + \mathbb{1}_{\delta(S_1 \cup S_2)}.$$

Case 2. Now assume that $|S_1 \cap S_2|$ is even. We can replace S_2 by $V - S_2$, since $V - S_2$ is also a tight odd set. But now $S_1 \cap (V - S_2) = S_1 - S_2$ which is an odd set. So it follows from the proof of Case 1 that $S_1 \cap (V - S_2)$ and $S_1 \cup (V - S_2)$ are both tight odd sets and we have

$$\mathbb{1}_{\delta(S_1)} + \mathbb{1}_{\delta(S_2)} = \mathbb{1}_{\delta(S_1 \cap (V - S_2))} + \mathbb{1}_{\delta(S_1 \cup (V - S_2))}.$$

Now observe that $S_1 \cap (V - S_2) = S_1 - S_2$ and $S_1 \cup (V - S_2) = V - (S_2 - S_1)$. Since taking complements does not change either $\delta(\cdot)$ or being a tight odd set, the claim follows. □

Now we use Lemma 5.7 to prove the claim that tight odd sets can be uncrossed without shrinking $\Lambda(S_1) + \Lambda(S_2)$.

Lemma 5.8. *Suppose that S_1, S_2 are tight odd sets, i.e., $|S_1|, |S_2|$ are odd and $\langle \mathbb{1}_{\delta(S_1)}, x \rangle = \langle \mathbb{1}_{\delta(S_2)}, x \rangle = 1$. Then exactly one of the following two conditions holds:*

(1) $S_1 \cup S_2$ is a tight odd set and

$$\Lambda(S_1) + \Lambda(S_2) \subseteq \Lambda(S_1 \cup S_2),$$

(2) S_1 and $S_2 - S_1$ are both tight odd sets and

$$\Lambda(S_1) + \Lambda(S_2) \subseteq \Lambda(S_1) + \Lambda(S_2 - S_1).$$

Proof. Look at the parity of $|S_1 \cup S_2|$. If $|S_1 \cup S_2|$ is odd, then we claim that Case 1 happens. Otherwise, we will show that Case 2 happens.

Case 1. $|S_1 \cup S_2|$ is odd. In this case, $|S_1 \cap S_2|$ is also odd and it follows by Lemma 5.7 that $S_1 \cup S_2$ is a tight odd set. It is trivial from Definition 5.5 that $\Lambda(S_1), \Lambda(S_2) \subseteq \Lambda(S_1 \cup S_2)$ which immediately yields

$$\Lambda(S_1) + \Lambda(S_2) \subseteq \Lambda(S_1 \cup S_2).$$

Case 2. $|S_1 \cup S_2|$ is even. In this case, $|S_1 - S_2|$ and $|S_2 - S_1|$ are both odd. Again, from Lemma 5.7 it follows that $S_2 - S_1$ is a tight odd set. It remains to prove that $\Lambda(S_1) + \Lambda(S_2) \subseteq \Lambda(S_1) + \Lambda(S_2 - S_1)$. It is enough to prove that $\Lambda(S_2) \subseteq \Lambda(S_1) + \Lambda(S_2 - S_1)$.

It is enough to show that for any tight odd set $T \subseteq S_2$, we have the inclusion $\mathbb{1}_{\delta(T)} \in \Lambda(S_1) + \Lambda(S_2 - S_1)$. We again have two cases: Either $|T \cap S_1|$ is odd or even.

If $|T \cap S_1|$ is even, it follows from Lemma 5.7 that $T - S_1$ and $S_1 - T$ are tight odd sets and

$$\mathbb{1}_{\delta(T)} = \mathbb{1}_{\delta(T - S_1)} + \mathbb{1}_{\delta(S_1 - T)} - \mathbb{1}_{\delta(S_1)}.$$

We have $\mathbb{1}_{\delta(S_1 - T)}, \mathbb{1}_{\delta(S_1)} \in \Lambda(S_1)$ and $\mathbb{1}_{\delta(T - S_1)} \in \Lambda(S_2 - S_1)$. So $\mathbb{1}_{\delta(T)} \in \Lambda(S_1) + \Lambda(S_2 - S_1)$ as desired.

The only case that remains is when $|T \cap S_1|$ is odd. In this case, we apply Lemma 5.7 to the sets T and $S_2 - S_1$, both of which are tight odd sets. Note that $T \cap (S_2 - S_1) = T - S_1$ which has even size by assumption. Therefore, by Lemma 5.7, $(S_2 - S_1) - T$ and $T - (S_2 - S_1) = S_1 \cap T$ are also tight odd sets and

$$\mathbb{1}_{\delta(T)} = \mathbb{1}_{\delta(S_2 - S_1 - T)} + \mathbb{1}_{\delta(S_1 \cap T)} - \mathbb{1}_{\delta(S_2 - S_1)}.$$

We have $\mathbb{1}_{\delta(S_1 \cap T)} \in \Lambda(S_1)$ and $\mathbb{1}_{\delta(S_2 - S_1 - T)}, \mathbb{1}_{\delta(S_2 - S_1)} \in \Lambda(S_2 - S_1)$ which proves that $\mathbb{1}_{\delta(T)} \in \Lambda(S_1) + \Lambda(S_2 - S_1)$ as desired. □

Given tight odd sets $S_1, \ldots, S_m$, repeated applications of Lemma 5.8 allow us to uncross them, i.e., replace them by pairwise disjoint tight odd sets $S'_1, \ldots, S'_{m'}$ such that $\Lambda(S_1, \ldots, S_m) \subseteq \Lambda(S'_1, \ldots, S'_{m'})$. However, naively applying Lemma 5.8 would result in a sequential algorithm which is not in NC. We will next show how we can do the uncrossing in NC.

We will use a divide-and-conquer approach to uncross a given list of tight odd sets $S_1, \ldots, S_m$. The high-level description of our procedure, `Uncross`, is given in Algorithm 5. We roughly divide the given sets into two parts, and recursively uncross each part. Then we call the procedure `MergeUncross` in order to merge the resulting sets.

ALGORITHM 5: Divide-and-conquer algorithm for uncrossing tight odd sets

```
Uncross(S_1, ..., S_m)
if m=1 then
  | return S_1
else
  | in parallel do
  |   | R_1, ..., R_p ← Uncross(S_1, ..., S_⌈m/2⌉)
  |   | C_1, ..., C_q ← Uncross(S_⌈m/2⌉+1, ..., S_m)
  | end
  | return MergeUncross(R_1, ..., R_p, C_1, ..., C_q)
end
```

Next, we will describe the merging procedure `MergeUncross`. The procedure `MergeUncross`, similarly to `Uncross`, accepts a list of tight odd sets and returns a list of pairwise disjoint tight odd sets whose Λ is not smaller. With some abuse of notation, we still name the inputs to `MergeUncross` as $S_1, \ldots, S_m$. The difference between `MergeUncross` and `Uncross` is that the input sets to `MergeUncross` satisfy certain properties highlighted below.

Lemma 5.9. *Suppose that $\{S_1, \ldots, S_m\} = \{R_1, \ldots, R_p, C_1, \ldots, C_q\}$, where $m = p + q$ and $R_1, \ldots, R_p$ are pairwise disjoint tight odd sets and $C_1, \ldots, C_q$ are also pairwise disjoint tight odd sets. Then $S_1, \ldots, S_m$ have no 3-wise intersections. Furthermore, the intersection graph of $S_1, \ldots, S_m$, where two S_i's are connected if they have a nonempty intersection, is bipartite.*

Proof. If we select any three sets S_i, S_j, S_k, then either two of them are from $R_1, \ldots, R_p$ or two of them are from $C_1, \ldots, C_q$. In either case, those two sets would not have any intersection. It is also easy to see that the intersection graph is bipartite, since $R_1, \ldots, R_p$ naturally form one part and $C_1, \ldots, C_q$ the other; by assumption, no two sets from the same part have any intersection. □

Having no 3-way intersections means that we can compute the parity of any union of $S_1, \ldots, S_m$ from their pairwise intersections. This is more handily captured by the notion of an intersection parity graph.

Definition 5.10. For tight odd sets $S_1, \ldots, S_m$ satisfying the conditions of Lemma 5.9, define the *intersection parity graph* $H = (V_H, E_H)$, as follows: Let V_H, the nodes of H, be $S_1, \ldots, S_m$ and, for $i \neq j$, let there be an edge between S_i and S_j if and only if $|S_i \cap S_j|$ is odd.

An immediate corollary of Lemma 5.9 is that H is bipartite. Another corollary is that the parity of $|\cup_i S_i|$ is the same as the parity of $|V_H| + |E_H|$ which we simply denote by $|H|$; this is because the inclusion-exclusion formula stops at pairwise intersections for our sets. We use the notation

$H(S_{i_1}, \dots, S_{i_k})$ to denote the induced subgraph on nodes $S_{i_1}, \dots, S_{i_k}$. With this notation, we have

$$|S_{i_1} \cup \dots S_{i_k}| \overset{2}{\equiv} |H(S_{i_1}, \dots, S_{i_k})|,$$

where $\overset{2}{\equiv}$ represents having the same parity.

By Lemma 5.8, if S_1, S_2 have an edge between them in H, then the union $S_1 \cup S_2$ will also be a tight odd set. If there is a third set S_3 connected to S_2, we can again include S_3 in this union, i.e., $S_1 \cup S_2 \cup S_3$ will be a tight odd set.

Can we repeatedly apply this procedure and obtain $S_1 \cup \cdots \cup S_m$ as a tight odd set? There seem to be two barriers to this. If the graph H is not connected, we can never take the union of two sets from different connected components. Another natural barrier is that $|S_1 \cup \cdots \cup S_m|$ could possibly be even; so it will never emerge out of this process, because Lemma 5.8 only produces tight *odd* sets. For simplicity of notation, we use $\cup H$ to denote $S_1 \cup \cdots \cup S_m$.

Surprisingly, the two mentioned barriers are really the only barriers, as we will show next.

Lemma 5.11. *Assume that $H = H(S_1, \dots, S_m)$ is connected and that $|H| \overset{2}{\equiv} 1$. Then, $\cup H = S_1 \cup \cdots \cup S_m$ is a tight odd set, and $\Lambda(S_1, \dots, S_m) \subseteq \Lambda(\cup H)$.*

Proof. We just need to show that $\cup H$ is a tight odd set. The fact that $\Lambda(S_1, \dots, S_m) \subseteq \Lambda(\cup H)$ is trivial from Definition 5.5.

We will use induction on $|V_H|$ to prove this fact. It is trivial to check this for $|V_H| \leq 2$. Even if $|V_H| = 3$, the only graph that is connected and bipartite on 3 nodes would be the path of length 2 and we have already described that in this case we can take the union by two applications of case 1 from Lemma 5.8.

Now consider a depth-first-search (DFS) tree started from an arbitrary node of H. If S is any leaf of this tree with $\deg_H(S) \overset{2}{\equiv} 1$, then we can proceed as follows: The graph $H - \{S\}$ will have one fewer node and odd many fewer edges. Therefore, $|H - \{S\}| \overset{2}{\equiv} 1$, and obviously $H - \{S\}$ is connected, since S was a leaf. By induction, $\cup(H - \{S\})$ is a tight odd set. But S is also a tight set, and by assumption the union of the two, $\cup(H - \{S\}) \cup S = \cup H$, is also odd. So, by Lemma 5.8, we get that $\cup H$ is a tight odd set. So from now on, assume that for any leaf node S, $\deg_H(S) \overset{2}{\equiv} 0$. More generally, if S is any node whose removal does not disconnect the graph, we can assume that $\deg_H(S) \overset{2}{\equiv} 0$, or else we can proceed as before. Note that this implies that any leaf in the tree has at least one back edge, i.e., an edge going to an ancestor other than its parent. This is true because any leaf must have at least one edge in addition to the one going to its parent. Since an undirected DFS tree has no cross edges, this edge must be a back edge.

Since an undirected DFS tree has no cross edges, its leaf nodes are never connected to each other. Therefore, if S_1, S_2 are two leaves, by the definition of H, $S_1 \cap S_2$ is even and hence so is $S_1 \cup S_2$. Hence, $|H - \{S_1, S_2\}| \overset{2}{\equiv} 1$. Note that $H - \{S_1, S_2\}$ is also connected, so by induction $\cup(H - \{S_1, S_2\})$ is a tight odd set.

Now, if the DFS tree has at least four leaves S_1, S_2, S_3, S_4, we can proceed as follows: Consider the graphs $H - \{S_1, S_2\}$ and $H - \{S_3, S_4\}$. They both satisfy the assumptions of the induction and therefore $\cup(H - \{S_1, S_2\})$ and $\cup(H - \{S_3, S_4\})$ are both tight odd sets. Their union is again $\cup H$ which has an odd parity. So again by Lemma 5.8 we get that $\cup H$ is a tight odd set. From now on we assume that there are at most 3 leaves in the tree.

If there are any two leaves S_1, S_2 that share a parent P, we can proceed as follows: The graph $H - \{S_1, S_2\}$ again satisfies the assumptions of induction. We also have that $S_1 \cup P \cup S_2$ is a tight odd set; this follows by applying the base case to the subgraph $H(S_1, S_2, P)$ which is a path of length 2. Again we have two tight odd sets $\cup(H - \{S_1, S_2\})$ and $S_1 \cup P \cup S_2$ whose union $\cup H$ is

odd. Therefore, $\cup H$ is a tight odd set. So from now on, we assume that no two leaves share a parent.

Now assume that the DFS tree has three leaves S_1, S_2, S_3. Without loss of generality, assume that S_1 is the deepest leaf. Let P be the parent of S_1. Note that P does not have any other children in the tree, because S_1 was the deepest leaf and no two leaves share a parent. Note that the removal of P does not disconnect the graph because S_1 has a back edge. Therefore, it must be that $\deg_H(P) \stackrel{2}{\equiv} 0$. Note also that P is not connected to S_2 or S_3, because a DFS tree does not have cross edges. All of this implies that $H - \{S_3, P\}$ is connected, and also has odd parity. As before, $H - \{S_1, S_2\}$ also satisfies the assumptions of the induction. So again, we get two tight odd sets whose union is $\cup H$ and therefore $\cup H$ is a tight odd set.

Now assume that the DFS tree has only two leaves S_1, S_2. Let P_1 be the parent of S_1 and P_2 the parent of S_2. Let Q be the lowest common ancestor of S_1 and S_2 in the tree. If $P_1, P_2 \neq Q$, then we can proceed similarly to the previous case: Both P_1 and P_2 must have an even degree, since their removal does not disconnect the graph. Now $H - \{P_1, S_2\}$ and $H - \{P_2, S_1\}$ are both connected and have an odd parity. We use induction and the fact that their union is $\cup H$ to again show that $\cup H$ is a tight odd set. So assume that one of P_1, P_2 is the same as Q. Without loss of generality, assume that $P_2 = Q$. Note that $P_1 \neq Q$, or else we would have two leaves sharing a parent, which is already a resolved case. Now let R be the parent of $P_2 = Q$. Note that S_2 has a back edge, but its back edge cannot be to R because that would create a triangle between S_2, P_2, R which is forbidden in our bipartite graph. So the back edge must be to some ancestor of R. This means that removing R or even removing both R, S_1 does not disconnect the graph. Since removing R does not disconnect the graph, we have $\deg_H(R) \stackrel{2}{\equiv} 0$. Now we have two cases:

(1) If S_1 does not have an edge to R, that would imply $H - \{S_1, R\}$ is odd and connected. Similar to the case of three leaves, we would get that $H - \{P_1, S_2\}$ is odd and connected as well. But then $H - \{S_1, R\}$ and $H - \{P_1, S_2\}$ are two connected and odd subgraphs whose union is H which implies that $\cup H$ is a tight odd set.

(2) Now assume that S_1 does have an edge to R. Note that $Q = P_2$ is a parent of S_2 and an ancestor of S_1. So it must have some other child, which we will call C. Note that $C \neq S_1$, or else S_1, S_2 would be two leaves sharing a parent, which has already been resolved. Now, the removal of C does not disconnect the graph because of the edge between S_1 and R. So it must be that $\deg_H(C) \stackrel{2}{\equiv} 0$. On the other hand, the removal of both S_2, C also does not disconnect the graph. Also note that there is no edge between S_2 and C because such an edge would create a triangle S_2, C, Q which is forbidden in our bipartite graph. All of these mean that $H - \{S_2, C\}$ is odd and connected and by induction $\cup(H - \{S_2, C\})$ is a tight odd set. On the other hand, $S_2 \cup Q \cup C$ is also a tight odd set because the induced graph on these three sets is a path of length 2. Again, we have found two tight odd sets $\cup(H - \{S_2, C\})$ and $S_2 \cup Q \cup C$ whose union gives us $\cup H$ and we are done.

The only remaining case is when the DFS tree has only one leaf, i.e., when the DFS tree is a Hamiltonian path. If the root and the leaf are not connected to each other, we can find another DFS tree such that it has more than one leaf and reduce the problem to the previous cases considered. Consider starting the DFS from the child of the current root and going down the Hamiltonian path until we reach the current child. Since this child was not connected to the root, the DFS procedure cannot continue and has to back up. Eventually, the original root will be connected somewhere along the tree as a leaf, but we now have two leaves, and we have already considered this case.

So the only case that remains is if the DFS tree is a Hamiltonian path and that the root is connected to the leaf. This tree with the extra edge gives us a Hamiltonian cycle. Since the removal of

any node in this graph does not disconnect the graph, all of the degrees must be even. Note that the entire graph cannot be simply this Hamiltonian cycle, because otherwise $|H| \overset{2}{\equiv} m + m \overset{2}{\equiv} 0$. So there must be some edge, other than those of the cycle, between two vertices P and Q. Let the two neighbors of P on the Hamiltonian cycle be A, B. Note that removing both A, B does not disconnect the graph. There is also no edge between A and B, because otherwise we would have a triangle A, B, P which is forbidden in bipartite graphs. So $H - \{A, B\}$ is odd and connected and by induction $\cup(H - \{A, B\})$ is a tight odd cut. Note that $A \cup B \cup P$ is also a tight odd cut, because the induced graph on A, B, P is a path of length 2. Again we have written H as the union of two connected and odd subgraphs; this implies that $\cup H$ is a tight odd set. □

Lemma 5.11 is the powerful pillar we use to create the method `MergeUncross`. If the intersection parity graph H has multiple connected components, we can deal with each one separately and then uncross the results using Case 2 of Lemma 5.8. If all of the connected components have odd parity, then we can take the union in each one and proceed. The only case we still need to show how to handle is when a connected component of H has even parity. We will show next that the even parity case can also be handled very easily.

Lemma 5.12. *Assume that $H = H(S_1, \ldots, S_m)$ is connected and $|H| \overset{2}{\equiv} 0$. Then there are two induced subgraphs of H, which are both odd and connected, and which together cover every node. Furthermore, these two subgraphs can be found in* NC.

Proof. We will be working with the biconnected components of H and the corresponding block-cut tree. A biconnected component is simply a maximal subgraph such that the removal of any vertex from it does not disconnect the subgraph. The block-cut tree is formed by introducing a node for each biconnected component and a node for every cut vertex, a vertex whose removal disconnects the graph, and connecting a cut vertex to all biconnected components to which it belongs. Finding biconnected components and forming the block-cut tree can be easily done in NC. For example, in parallel for every pair of edges, and every vertex, one can check whether the removal of that vertex disconnects the pair of edges; then one can form equivalence classes out of the edges and obtain the biconnected components. For more efficient and elegant algorithms in NC, see [32].

For an induced subgraph $B = (V_B, E_B)$, let us define its inverse parity as the parity of $|V_B| + |E_B| + 1$ and denote this by $\overline{[B]}$. Note that we have $\overline{[B]} \overset{2}{\equiv} 1 + |B|$. We regard biconnected components as induced subgraphs, unless otherwise stated. Inverse parity has a certain additivity property. Namely, if B_1 and B_2 are induced subgraphs that share only a single vertex and have no edges to each other, then $\overline{[B_1 \cup B_2]} = \overline{[B_1]} + \overline{[B_2]}$.

Using this, one can easily compute the inverse parity of any subtree of the block-cut tree. In the block-cut tree, to each biconnected component assign its inverse parity, and to each cut vertex assign 0. Then it is easy to see, by the additivity property, that for any subtree of the block-cut tree, the inverse parity of the union of all blocks in the subtree is simply the parity of the sum of assigned numbers.

In particular, since $|H| \overset{2}{\equiv} 0$, or in other words, $\overline{[H]} \overset{2}{\equiv} 1$, there must be an odd number of 1s in the block-cut tree.

We will first solve the problem when there are at least three 1s in the tree. In this case, we can find two subtrees whose union is the entire tree, each having an even number of 1s. This suffices, because the union of all biconnected components in each subtree would be an odd connected graph, and by Lemma 5.11, we can merge all of the nodes in it. Each subtree will be obtained by simply partitioning the block-cut tree by removing an edge and looking at one of the resulting

two subtrees. Clearly, we can try all such partitions in NC. So it remains to show that at least two of them, whose union is the entire tree, have an even internal sum. For this, look at the 1 nodes in the tree whose distance, in the tree, is the largest. Let them be B_1 and B_2. Look at the path on the block-cut tree connecting B_1 to B_2 and let the edge adjacent to B_1 be e_1 and the one adjacent to B_2 be e_2. Now if we partition the block-cut tree by removing e_1, we get two parts, one of which contains B_2, and the other part can only contain one 1 node, namely B_1. Otherwise, the distance between B_1 and B_2 would not have been maximal. So the subtree containing B_2 has an even sum. Similarly, if we remove e_2 from the block-cut tree, the part containing B_1 will have an even sum. It is not hard to see that these two subtrees cover the whole tree.

So the only remaining case is when the block-cut tree has only one 1 node. In that case, let B be the biconnected component with $\overline{[B]} \overset{2}{\equiv} 1$.

First, consider the case where B is the entire graph H. In this case, we will show that either there is a vertex S where B and $B - \{S\}$ are both odd and connected, or there are two vertices S_1, S_2 connected by an edge such that $B - \{S_1, S_2\}$ and $H(S_1, S_2)$ are both odd and connected. First, note that if any node in B has an even degree, then this condition is automatically satisfied. Because if S_1 is such a node, $B - \{S_1\}$ is connected since B is biconnected. It is also odd because $B - \{S_1\}$ has one fewer node and an even number of fewer edges. So assume from now on that the degree of every node in B is odd. Now we want to obtain the nodes S_1, S_2 as described before. This is easy to derive from an open ear decomposition of B. Note that $\overline{[B]} \overset{2}{\equiv} 1$ implies that B cannot be simply a single edge, so it must have an open ear decomposition. Look at this ear decomposition, and add the ears one by one. Look at the last ear added that was not a single edge. Suppose that this ear was some path $(S_1, \ldots, S_k)$. Then, note that S_2 is a new node added by this ear, and since no new nodes are added after this ear, the removal of S_1, S_2 leaves B connected; even if this ear was the initial cycle, this is still true. So $B - \{S_1, S_2\}$ is connected and since the degrees of S_1, S_2 are both odd and they are connected to each other, it must be that $B - \{S_1, S_2\}$ is odd. Since S_1, S_2 are connected to each other as well $H(S_1, S_2)$ is also connected and odd as desired. Note that the vertex S_1 or the pair of vertices S_1, S_2 can be found in NC by simply checking all possibilities in parallel.

Now consider the case where B is not the entire graph H. In this case we proceed as before, and by looking at the induced subgraph B, we find either a node S_1 or two connected nodes S_1, S_2 such that $B - \{S_1\}$ or $B - \{S_1, S_2\}$ is connected and odd. So we have a partition of B into a single or a pair of vertices and the rest of B. We simply attach the biconnected components other than B to one of the partitions, based on the block-cut tree. This ensures that connectivity is preserved, and further, the parity of the partitions is not changed because every biconnected component other than B has inverse parity 0. Again this operation can be done in NC, since the partition inside B can be found in NC, and connecting the rest of the biconnected components is simply a matter of partitioning the block-cut tree into two or three parts. □

Now, armed with Lemmas 5.11 and 5.12, we can describe the procedure `MergeUncross`. We will first make sure that even intersections are completely removed, i.e., made empty. This is easy to do in parallel, because there are no 3-wise intersections. Then we apply Lemma 5.11 or Lemma 5.12 to each connected component of H. To avoid creating sets S with $|f^{-1}(S)| \geq c_1|V_0|$, we always pass our new sets through the procedure `CheckBalancedViable`, which will potentially find a balanced viable set and end the procedure.

We just have to describe `CheckBalancedViable`. The input to this procedure is an odd connected subset H of the intersection parity graph. If $|f^{-1}(\cup H)| < c_1|V_0|$, then this procedure simply does nothing. Otherwise, it outputs a balanced viable set as follows:

If $|f^{-1}(\cup H)| \leq (1 - c_1)|V_0|$, then it simply outputs $f^{-1}(\cup H)$ as the balanced viable set. Otherwise, we order the vertices of H as $S'_1, \ldots, S'_k$ so that for any i, the induced subgraph on

ALGORITHM 6: Algorithm for uncrossing partially uncrossed sets

```
MergeUncross(S_1, ..., S_m)
for i = 1 ... m in parallel do
 | S_i ← S_i − ⋃_{j<i, |S_i ∩ S_j| even} S_j
end
H ← H(S_1, ..., S_m)
H_1, ..., H_k ← ConnectedComponents(H)
F ← ∅
for i = 1 ... k in parallel do
 | if |H_i| = |V_{H_i}| + |E_{H_i}| is odd then
 | | CheckBalancedViable (H_i).
 | | Add ∪H_i to F.
 | else
 | | Let H'_i, H''_i be the two induced subgraphs promised by Lemma 5.12.
 | | CheckBalancedViable (H'_i).
 | | CheckBalancedViable (H''_i).
 | | Add ∪H'_i to F.
 | | Add ∪H''_i − ∪H'_i to F.
 | end
end
return F
```

$U_i = \{S'_1, \dots, S'_i\}$ is connected. For example, sorting according to shortest distance (in H) to an arbitrary initial vertex S'_1 would satisfy this property. Now let j be the first index for which $|f^{-1}(\cup U_j)| \geq 2c_1|V_0|$. Then, $|f^{-1}(\cup U_j)| \leq 3c_1|V_0| < (1-c_1)|V_0|$. So if $|\cup U_j| \overset{2}{\equiv} 1$, then we can return $f^{-1}(\cup U_j)$ as a balanced viable set (it is a tight odd set by Lemma 5.11). Otherwise, by Lemma 5.12, we can find two subsets of U_j whose union covers U_j and that are odd. We simply return the subset with the larger value of $|f^{-1}(\cdot)|$ as the balanced viable set.

All together, we get the following result:

THEOREM 5.13. *Given tight odd sets $S_1, \dots, S_m$, there is an* NC *algorithm that either finds a viable set or outputs pairwise disjoint tight odd sets $S'_1, \dots, S'_{m'}$ such that*

$$\Lambda(S_1, \dots, S_m) \subseteq \Lambda(S'_1, \dots, S'_{m'}),$$

and $|f^{-1}(S'_i)| < c_1|V_0|$.

6 OTHER ALGORITHMIC INGREDIENTS

In this section, we describe the remaining algorithmic ingredients we used in Section 4 and 5.

6.1 Finding a Point in the Relative Interior of a Face of the Perfect Matching Polytope

In this section, we prove Lemma 3.1 by giving an NC algorithm for the following problem: Given a planar graph $G = (V, E)$ and a weight vector on edges $w \in \mathbb{Z}^E$ given in unary, find a point, x, in the interior of $\mathrm{PM}(G, w)$, where $\mathrm{PM}(G, w)$ denotes the face of the perfect matching polytope of G containing all minimum-weight perfect matchings in G and their convex combinations (clearly, the corner points of this face are precisely the set of minimum-weight perfect matchings in G).

PROOF OF LEMMA 3.1. Let $\#G_w$ denote the number of MWPMs in G w.r.t. edge weights w, and for each edge $e \in E$, let $\#G_w^e$ denote the number of such perfect matchings which contain the edge

e. The point x we will find will have coordinate

$$x_e = \frac{\#G_w^e}{\#G_w}.$$

Clearly, x satisfies all required conditions. Additionally, observe that if $M_1, \ldots, M_m$ are all the MWPMs in G, then

$$x = \frac{\mathbb{1}_{M_1} + \cdots + \mathbb{1}_{M_m}}{m} = \mathrm{avg}(\mathrm{PM}(G, w)).$$

We will crucially use the fact that a Pfaffian orientation of G can be computed in NC. Let $(i, j) \in E$, with $i < j$. If in the Pfaffian orientation this edge is directed from i to j, then let $B_{ij} = y^{w_e}$; otherwise, let $B_{ij} = -y^{w_e}$, where y is an indeterminate. Let B be the resulting matrix. Observe that the exponents of the entries of B are small in the input size and hence its determinant can be computed in NC [2]. Consider the lowest degree term in $\det(B)$; let its degree be d. Then, the coefficient of y^d is the square of the number of perfect matchings of minimum-weight in G (see [26]), i.e., it is $(\#G_w)^2$.

Next, for each edge $e \in E$, we will compute $\#G_w^e$, the number of MWPMs that edge e participates in. Zero out the two entries in B corresponding to e to obtain matrix B_e and compute $\det(B_e)$. Then the coefficient of y^d will be $(\#G_w - \#G_w^e)^2$. Hence, $\#G_w^e$ as well as $\#G_w$ can be computed. Clearly, this can be done in parallel for all edges. □

6.2 Finding Linearly Many Edge-Disjoint Even Walks

In this section, we prove Lemma 3.3 by showing how to find $\Omega(|E|)$ many edge-disjoint even walks in a given graph $G = (V, E)$ in NC. By assumption, G is a connected planar graph that does not have any vertices of degree 1 and at most $|V|/2$ vertices of degree 2. We first find linearly many edge-disjoint planar faces in G.

Lemma 6.1 (Adapted from [20]). *Suppose that $G = (V, E)$ is a connected planar graph with no vertices of degree 1 and at most $|V|/2$ vertices of degree 2. Then, there is an* NC *algorithm that returns $|E|/300$ edge-disjoint planar faces of G.*

Proof. It is easy to see that the graph has $\Omega(|E|)$ faces. By Euler's formula, we have

$$|V| - |E| + |F| = 2,$$

where F denotes the set of (planar) faces. By rearranging and using the fact that $\deg(v) \geq 2$ for every v, we get

$$|F| \geq |E| - |V| = \sum_{v \in V} \frac{\deg(v) - 2}{2} \geq \sum_{v \in V : \deg(v) \geq 3} \frac{\deg(v)}{6} = \frac{1}{3}(|E| - |\{v : \deg(v) = 2\}|)$$
$$\geq \frac{1}{3}(|E| - |V|/2) \geq \frac{1}{6}|E|.$$

Consider the planar dual G^* of G. Corresponding to each face in G, the dual has a vertex, and corresponding to each edge in the primal, there is an edge in the dual. The sum of the degrees in the dual graph is $2|E| \leq 12|F|$. In other words, the average degree in the dual graph is at most 12. By Markov's inequality, at least half of the dual vertices must have degree at most 24. We simply drop the dual vertices of degree more than 24 from the dual graph and find a maximal independent set in the remaining dual. This can be done in NC, by Lemma 3.2. The remaining dual has maximum degree at most 24, so its maximal independent set has at least 1/25 of its vertices, i.e., at least $|F|/50 \geq |E|/300$. □

If at least half of the faces found by Lemma 6.1 are even, we work with these as our even walks. Else, we need to pair up odd faces together with an edge-disjoint path connecting each pair to get $\Omega(|E|)$ even walks of the second type.

Lemma 6.2. *Given an even number f of edge-disjoint odd faces in a planar graph, we can find, in* NC, $f^2/(4f + 16|E|)$ *edge-disjoint even walks, each formed by joining two of the given faces.*

Proof. First, we find a spanning tree T of G. We will only use paths on the spanning tree to pair up odd faces. For each given odd face, place a token at one of its vertices, arbitrarily. Now we have f tokens on the spanning tree T. In Lemma 6.3, we will prove that these tokens can be paired up by edge-disjoint paths from the tree in NC.

We use this pairing of tokens and the paths from the tree T to pair the given odd faces. When we connect two odd faces O_1 and O_2 by a path P, the path P might intersect or even use the edges of O_1 and O_2. We fix this by replacing P with a subpath of P. More precisely, we find the last intersection of P with O_1, and the first intersection after that point with O_2, and replace P by the subpath between these two intersections. Now the path is edge-disjoint from O_1, O_2.

So far, we have created $f/2$ even walks; but they are not necessarily edge-disjoint. The only way that two of these walks can intersect each other is if the connecting path from one shares an edge with an odd face of the other. Because of this, any given edge e can appear in at most 2 of the walks. So the average number of edges in an even walk is at most $2|E|/f$. Markov's inequality implies that at least $f/4$ of the even walks have at most $4|E|/f$ edges. Any of these even walks shares an edge with at most $4|E|/f$ other walks, since an edge appears in at most two walks.

By Lemma 3.2, we can find, in NC, a maximal independent set of these short even walks (an independent set is a just a set of walks that are pairwise edge-disjoint). The number of walks in this independent set will be at least

$$\frac{f/4}{1 + 4|E|/f} = \frac{f^2}{4f + 16|E|}. \qquad \square$$

We now describe the missing part from the above proof.

Lemma 6.3. *Consider a tree T and an even number of tokens $o_1, \ldots, o_f$ placed on the vertices of the tree. We can find, in* NC*, a pairing of the tokens using the shortest path on the tree, so that no two paths share an edge.*

Proof. For each edge $e \in T$, we will count the number of tokens on either side of T when e is removed; this follows from an NC tree traversal algorithm [17]. Since there are an even number of tokens, this count must either be odd on both sides or even on both sides. We will do this in parallel for every edge. We then remove all of the edges whose token counts were even-even.

After this operation, the degree of every vertex v must have the same parity as the number of tokens on it. This is because for each edge e adjacent to v, the number of tokens on the other side of e is odd. So the number of tokens not on v has the same parity as $\deg(v)$. But the total number of tokens on the entire tree is even, so this parity is also shared by the number of tokens on v.

Now we do the following in parallel for each vertex v: We pair up all the tokens on v in an arbitrary way until there is at most one token left. We will then pair the remaining token, if any, with one of the remaining edges; there must be at least one edge if there is at least one token. Now there are an even number of edges adjacent to v that remain. We pair them up in an arbitrary way, so that whenever we use an edge in a pair to enter v, we exit using the other edge.

Now, by following the paths from each token to the edge it is assigned to, we will get to another token, and this gives us a pairing between tokens. Note that this path following does not have to be done sequentially, but can instead be done using the doubling trick to get an NC algorithm. □

We now have the ingredients needed to finish the proof of Lemma 3.3.

Proof of Lemma 3.3. We first find $|E|/300$ edge-disjoint faces by invoking Lemma 6.1. If at least half of these faces are even, we return this half. Otherwise, we invoke Lemma 6.2. We have $|E|/600$ odd faces, and by possibly dropping one of them we can supply an even number $f \geq (|E|/600) - 1$ of odd faces to the algorithm described by Lemma 6.2 and obtain

$$\frac{(|E|/600 - 1)^2}{4(|E|/600 - 1) + 16|E|} = \Omega(|E|)$$

edge-disjoint even walks. This finishes the proof. □

6.3 Finding Gomory-Hu Trees and Minimum Odd Cuts

In this section, we will give an NC algorithm for constructing a Gomory-Hu tree for a planar connected graph $G = (V, E)$ with edge weights given by $w : E \to \mathbb{R}_{\geq 0}$ and finding a minimum odd cut. We will crucially use the fact that an s-t max-flow and min-cut can be computed in a planar graph in NC [16].

We first define the notion of a Gomory-Hu tree. For each pair of vertices $u, v \in V$, let $f(u, v)$ denote the weight of a minimum u-v cut in G. Let T be a spanning tree on vertex set V, not necessarily a subset of E. On removing an edge e from T, we get two subtrees. Let S_1, S_2 be the sets of vertices spanned by these subtrees; clearly, this is a cut in G. We will say that (S_1, S_2) is the *cut defined by edge e*.

A Gomory-Hu tree, say T, is a spanning tree of V, together with a function w' giving weights to the edges of T, such that:

(1) For $u, v \in V$, let (s, t) be the minimum-weight edge, under w', on the unique path in T between u and v. Then, we require that $w'(s, t) = f(u, v)$.
(2) The cut defined by (s, t) is a minimum u-v cut in G.

Note that if $(S, \overline{S})$ is a minimum $u - v$ cut, then the graph induced on S must be one connected component; if not, the component not containing u can be moved to the other side to obtain a smaller u-v cut. Hence, the graph obtained by shrinking S in G will remain planar.

The sequential algorithm for constructing a Gomory-Hu tree has, at any point, a tree T defined on a partition $S_1, \ldots, S_k$ of V, and a weight function w' defined on the edges of T. The starting partition is simply V, with T having no edges. The partition and T satisfy:

- For each edge $(S_i, S_j) \in T$, $\exists\, u \in S_i,\ v \in S_j$ such that $w'(S_i, S_j) = f(u, v)$.
- The cut defined by edge (S_i, S_j) must be a minimum u-v cut in G.

In each iteration, the sequential algorithm refines the tree by *splitting* one of the partitions into two as follows. It picks a partition having at least two vertices, say S_i. Let $u, v \in S_i$. Let $T_1, \ldots T_l$ be the subtrees of T incident at node S_i. By *shrinking* subtree T_j, we mean identifying all vertices in T_j and replacing it by single vertex t_j. All edges incident at vertices in T_j from outside T_j are now incident at t_j, with the same weight as before. Shrinking $T_1, \ldots T_l$ gives a graph on $S_i \cup \{t_1, \ldots, t_l\}$. Let this graph be G'; clearly it will be planar. In G', find a minimum u-v cut. It is easy to show that the weight of this cut will also be $f(u, v)$.

This cut will partition S_i into two sets, say S' and S'', with $u \in S'$ and $v \in S''$. Replace S_i by these two sets to obtain a partition on $k + 1$ sets. The new tree will contain the edge (S', S'') with weight $w'(S', S'') = f(u, v)$. Next, among the subtrees $T_1, \ldots T_l$ take the ones on the u side (v side) of the cut and let them be incident at S' (S''). The algorithm ends when each partition is a singleton vertex. The tree so found will be a Gomory-Hu tree.

We now give our NC algorithm. The main difference lies in the way set S_i is split. We first define the notion of a *central vertex* for S_i. Pick a vertex $r \in S_i$ and for each remaining vertex $v \in S_i$, find a minimal minimum v-r cut in the graph G' defined above after shrinking subtrees incident to S_i; by a minimal minimum v-r cut we mean that the side containing v is smallest among all minimum v-r cuts. Let S_v denote the side containing v in this cut and let $S'_v = S_v \cap S_i$. We will say that r is a *central vertex* for S_i if for each $v \in S_i,\ v \neq r$, $|S'_v| \leq |S_i|/2$. Let us first show that such a vertex exists.

Lemma 6.4. *For any partition S_i, a central vertex r exists for S_i.*

Proof. Let T be the eventual Gomory-Hu tree found by the sequential algorithm stated above. Remove all vertices not in S_i from T. Let us argue that the resulting graph, say T', will still be connected. This follows since at any stage in the algorithm, the sets which define the current partition of S_i are connected via the set of edges which were added each time a subset of S_i was split.

It is easy to see that there is a vertex $r \in T'$ such that each subtree of T' incident at r has at most $|T'|/2$ vertices. Since T is a Gomory-Hu tree, for each $v \in S_i,\ v \neq r$, a minimum v-r cut is defined by one of the edges of T that lies in T'. It follows that each such cut satisfies $|S'_v| \leq |S_i|/2$ and hence r is a central vertex for S_i. □

A central vertex for S_i can be found in NC: For each vertex $r \in S_i$, test if it is a central vertex by finding, in parallel, a minimal minimum v-r in G' for each vertex $v \in S_i,\ v \neq r$. From now on, let r denote a central vertex for S_i.

Lemma 6.5. *Let $r, u, v \in V$ and let S_u and S_v be minimal minimum u-r and v-r cuts in G, respectively. Then S_u and S_v do not cross.*

Proof. Assume S_u and S_v cross. There are three cases:

Case 1. If $u, v \in (S_u \cap S_v)$ then it is well known that $(S_u \cap S_v)$ is also a u-r and v-r minimum cut, contradicting minimality.

Case 2. Assume $u \in (S_u - S_v)$ and $v \in (S_v - S_u)$. We will partition the edges incident at the sets S_u and S_v into six types and let a, b, c, d, e, f denote their capacities:

(1) **a:** Edges between $(S_u - S_v)$ and $\overline{(S_u \cup S_v)}$.
(2) **b:** Edges between $(S_u \cap S_v)$ and $(S_u - S_v)$.
(3) **c:** Edges between $(S_u \cap S_v)$ and $(S_v - S_u)$.
(4) **d:** Edges between $(S_u \cap S_v)$ and $\overline{(S_u \cup S_v)}$.
(5) **e:** Edges between $(S_v - S_u)$ and $\overline{(S_u \cup S_v)}$.
(6) **f:** Edges between $(S_u - S_v)$ and $(S_v - S_u)$.

Since S_u and S_v are u-r and v-r minimum cuts, we get:

$$f(u,r) = a + b + d + f.$$

$$f(v,r) = b + c + d + f.$$

Since $(S_u - S_v)$ and $(S_v - S_u)$ are u-r and v-r cuts (not necessarily minimum), we get:

$$f(u,r) \leq a + c + f.$$

$$f(v,r) \leq b + e + f.$$

These four equalities and inequalities imply that $d = 0$. Now, if either of the inequalities is strict, we will get that the other inequality is violated. Hence, both inequalities are actually equalities, giving us smaller u-r and v-r minimum cuts and a contradiction. Hence, S_u and S_v cannot cross.

Case 3. Assume $u \in (S_u - S_v)$ and $v \in (S_u \cap S_v)$. Since S_u is a u-r minimum cut, $f(u,r) = a + c + d + f$.

Now, $(S_u \cap S_v)$ is a v-r cut; however, it is not a minimum v-r cut, since that would contradict the minimality of the v-r minimum cut S_v. Therefore,

$$b + c + d > b + d + e + f \text{ implying } c > e + f.$$

Combining with the previous fact, we get

$$f(u,r) = a + c + d + f > a + d + e + 2f \geq a + d + e.$$

Now, $(S_u \cup S_v)$ is also a u-r cut having capacity $a + d + e < f(u,r)$, leading to a contradiction. □

Corollary 6.6. *Let $r, v_1, \dots v_k \in V$ and let $S_{v_1}, \dots, S_{v_k}$ be minimal minimum v_1-r, … v_k-r cuts in G, respectively. Then, $S_{v_1}, \dots, S_{v_k}$ form a laminar family.*

Let r denote a central vertex for S_i that is found by the algorithm. By Corollary 6.6, the cuts S_v, for each vertex $v \in S_i,\ v \neq r$ form a laminar family. Let $M_1, \dots, M_l$ be the maximal sets of this laminar family. Clearly, we can split S_i into the l sets $M_1 \cap S_i, \dots, M_l \cap S_i$ and attach subtrees to appropriate sets as given by $M_1, \dots, M_l$. This can be done for all sets S_i of the current partition, in parallel. This defines one iteration of our parallel algorithm. Clearly, after each iteration, the cardinality of the largest set in the partition drops by a factor of 2 and therefore only $O(\log n)$ such iterations are needed. Hence, we get:

Theorem 6.7. *There is an* NC *algorithm for obtaining a Gomory-Hu tree for an edge-weighted planar graph.*

Now we use Padberg and Rao's theorem that states that the Gomory-Hu tree of a graph must contain a minimum odd cut as one of its edges [27] to finish the proof of Lemma 5.4.

Proof of Lemma 5.4. We first find a Gomory-Hu tree, then try all of the cuts obtained by removing an edge of the tree. We return the minimum among cuts that split the vertices into odd pieces. Clearly, all of this can be done in parallel, and hence the algorithm is in NC. □

Remark 6.1. An alternative way of finding a minimum odd cut S is to use some properties of the Pickard-Queyranne structure of minimum s-t cuts [29]. However, that method is more cumbersome to describe.

7 EXTENSIONS

In this section, we will build on the machinery established in the previous sections to prove two generalizations of Theorem 1.1. We first give a proof of Theorem 1.2 stated in the Introduction.

Proof of Theorem 1.2. For a fixed integer k, assume that we are given a weight function on the edges of planar graph $G = (V, E)$,

$$W : E \to \{0, \dots, n^k\},$$

and we wish to find an MWPM in G. Recall that in Section 6.1, for the purpose of finding a perfect matching in G, we had defined 0/1 weights on edges given by function w. Now, define the following composite weight function, c; its least significant $\log n$ bits correspond to w and the rest of the bits correspond to W.

$$c_e = (n \cdot W_e) + w_e.$$

Clearly, c gives small weights to edges and can be used in place of w to carry out the NC algorithm given in the previous sections. The algorithm will return an MWPM w.r.t. weights given by c. Since w is 0/1, for any perfect matching, the sum of weights according to w is at most $n/2$.

Therefore, in computing the weight of a perfect matching according to c, there will be no carry-over from the least significant $\log n$ bits to the rest. Hence, the MWPM w.r.t. c will also be a MWPM w.r.t. W. □

THEOREM 7.1. *There is an* NC *algorithm which given a bounded-genus graph, returns a perfect matching in it, if it has one.*

PROOF. In order to derive our algorithm for planar graphs, we used planarity in exactly three ways, and here we will show how one can obtain the same results for graphs embeddable on orientable surfaces of bounded genus.

(1) $\Omega(|E|)$ *Edge-Disjoint Even Walks (Lemma 3.3).* We used planarity to extract $\Omega(|E|)$ edge-disjoint faces and then argued that, by pairing up the faces, we get $\Omega(|E|)$ edge-disjoint even walks.
(2) *Counting Perfect Matchings (Lemma 3.1).* We used planarity to argue that we can count the number of MWPMs and hence get a point inside a face of the perfect matching polytope $PM(G, w)$ when w is small.
(3) *Finding the Minimum Odd Cut (using Theorem 6.7).* We used the fact that minimal minimum s-t cuts can be computed in NC for planar graphs [16], in order to prove that we can construct Gomory-Hu trees and find the minimum odd cut.

Note that given a graph, one can find an embedding onto a surface of genus $g = O(1)$ in NC if one exists [7]. So from now on, we assume this embedding is given to us. We now address how each of Lemmas 3.3 and 3.1, and Theorem 6.7 can be proved for bounded genus graphs.

For Lemma 3.3, note that we simply need to obtain $\Omega(|E|)$ edge-disjoint cycles in our graph. Pairing up odd cycles can be done as before using a spanning tree. In planar graphs, these cycles were obtained from the faces, and we used Euler's formula $|V| - |E| + |F| = 2$ to argue that in graphs without degree 1 vertices and with at most half of the vertices having degree 2, there must be $\Omega(|E|)$ faces. We still have an Euler's formula in the case of bounded genus graphs, but with 2 replaced by a (negative) constant. The proof still works as before and one can show that $|F| \geq a|E| - b$ for some $a > 0$, which implies that $|F| = \Omega(|E|)$.

For Lemma 3.1, it is enough to be able to count perfect matchings. To be more precise, given weights w over the edges of the graph, we simply need to compute the perfect matching generating function

$$\sum_{M \text{ is perfect matching}} \prod_{e \in M} w_e,$$

in NC as long as the bit complexity of w is small. Kulkarni et al. [20] showed how this can be done in NC by slightly modifying an algorithm of Galluccio and Loebl [11]. Their method reduces computing the perfect matching generating function to taking a linear combination of Pfaffians over planar graphs.

Finally, for finding minimum odd cuts in bounded genus graphs: We will show that we can use the methods of Borradaile et al. to find the minimum odd cut in NC [3]. The algorithm of Borradaile et al. allows one to find minimum cuts between all pairs of vertices in bounded genus graphs in nearly linear time, however we will show that a slight modification of it runs in NC. This almost shows that one can find the minimum odd cut in NC, because every minimum odd cut is also a minimum s-t cut for some s and t. However, one still needs to be careful about cases where there can be multiple minimum s-t cuts.

The main observation behind the algorithm of Borradaile et al. is that a minimum cut separating vertices s and t is composed of dual cycles (of which there are at most $2^{O(g)}$), all but one of which can be chosen from certain homology classes without regards to the pair s and t. They use this

observation to reduce the problem to finding minimum cuts in $2^{O(g^2)}$ planar graphs, where g is the genus of the original graph. Roughly speaking, they enumerate all possible homology classes for all but one the cycles, and one by one, from each homology class they find the shortest possible cycle in the chosen homology class and perform some surgery on the graph and its embedding. These surgeries reduce the genus, until the embedding becomes planar. The surgeries are easy to perform in NC, as long as the cycles are found in NC.

In order to define the homology classes, and also find shortest cycles given the homology class, Borradaile et al. use the results of Erikson and Nayyeri [9]. Their algorithm again can be implemented in NC; at a high level, it works as follows: construct a spanning forest in the dual graph in order to define $\mathbb{Z}_2$-homology signatures, and then construct a $\mathbb{Z}_2$-homology cover of the bounded genus graph and find shortest paths between pairs of vertices in the cover. All of these can be parallelized and are in particular in NC as long as the genus is bounded.

We now point out the main technicality needed to adapt the algorithm of Borradaile et al. Although minimum cuts between all pairs of vertices can be found from the minimum cuts in the $2^{O(g^2)}$ planarizations, it is not guaranteed that these cuts are nicely uncrossed from each other and form a Gomory-Hu tree. In fact, Borradaile et al. perturb the weights in order to have unique minimum cuts, and in order to be able to merge the Gomory-Hu trees from $2^{O(g^2)}$ planar graphs into one Gomory-Hu tree for the original graph. However, we cannot afford to perturb the weights because we do not have access to random bits. Instead, we argue in the next paragraph that a minimum odd cut can be directly found in one of the planarizations. Therefore, one can produce the planarizations in NC and then construct a Gomory-Hu tree from each and find the minimum odd cut amongst them.

Note that if the weights of the graph were slightly perturbed, then the minimum odd cut would have been the *unique* minimum s-t cut for some pair s and t, and we would have been able to find the unique cut in one of the planarizations. But this shows that even if the edges were not perturbed, a minimum odd cut must survive the surgeries performed on the graph for some sequence of fixed homology signatures. In other words, a minimum odd cut must be comprised of dual cycles, all but one of which are the minimum cycles from given homology classes. So a minimum odd cut must survive one of the sequences of surgeries performed on the graph, and found at the end by our algorithm for finding Gomory-Hu trees in planar graphs. □

We remark that all of the subroutines mentioned in the previous proof still remain in NC for genus up to $O(\sqrt{\log n})$, except for the subroutine that finds the embedding of the graph. Hence, Theorem 7.1 can be slightly strengthened to handle graphs of genus $O(\sqrt{\log n})$, as long as the input graph is given along with its embedding. Finally, observe that the common generalization of Theorems 1.2 and 7.1 easily follows.

8 DISCUSSION

The main open problem, of course, is to go beyond bounded genus graphs and obtain an NC perfect matching algorithm for general, or even bipartite, graphs. Below, we state a more easily accessible open problem.

An interesting problem defined by Papadimitriou and Yannakakis [28], called Exact Matching, is the following: Given a graph G with a subset of the edges marked red and an integer k, find a perfect matching with exactly k red edges. This problem is known to be in RNC [26]; however, it is not yet known to be in P, even for bipartite graphs. For the case of planar graphs, the decision version of this problem is known to be in NC (and hence P); this follows from the fact that a Pfaffian orientation for such a graph can be computed in NC [34] and by using the techniques of Section 3.1. Using the decision oracle to sequentially prune edges, the search version can be seen

to be in P; however, it is not known to be in NC. Can our techniques be used for obtaining an NC algorithm for the search version?

Remark 8.1. Subsequent to our work, Sankowski [30] gave an NC algorithm for computing a perfect matching in a planar graph, using somewhat different techniques.

Remark 8.2. Building on our result, recently Eppstein and Vazirani [8] resolved the open problem stated by Vazirani [34], of obtaining an algorithm for $K_{3,3}$-free graphs. Eppstein and Vazirani [8] go further to give NC algorithms for computing an MWPM for small edge-weights, counting the number of perfect matchings, and a maximum s-t flow in one-crossing-minor-free graphs.

Remark 8.3. In the Introduction of this article, and the version that appeared in FOCS 2018, we stated, "... it will not be surprising if some of our ideas turn out to be useful for the resolution of the main open problem." Our recent article [1] already justifies this remark. The article gives an NC reduction from search to decision for the problem of finding an MWMP in general graphs with small edge-weights; as remarked in [1], this new fact has qualitatively changed the nature of the main open problem. The overall structure of the oracle-based algorithm given in [1] follows that of the current article. Additionally, this article uses some key ideas from the current work, in particular, an NC algorithm for finding a balanced viable set; observe that the algorithm given in the current article works for general graphs. For completeness, it is important to remark that [1] also critically draws on several ideas discovered in other articles on matching published on the last five years.

ACKNOWLEDGMENTS

We wish to thank David Eppstein, László Lovász, and Satish Rao for valuable discussions.

REFERENCES

[1] Nima Anari and Vijay V. Vazirani. 2020. Matching is as easy as the decision problem, in the NC model. In *Innovations in Theoretical Computer Science, 2020*.

[2] Allan Borodin, Stephen Cook, and Nicholas Pippenger. 1983. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and Control* 58, 1–3 (1983), 113–136.

[3] Glencora Borradaile, David Eppstein, Amir Nayyeri, and Christian Wulff-Nilsen. 2014. All-pairs minimum cuts in near-linear time for surface-embedded graphs. *Arxiv Preprint Arxiv:1411.7055* (2014).

[4] Laszlo Csanky. 1976. Fast parallel matrix inversion algorithms. *SIAM J. Comput.* 5, 4 (1976), 618–623.

[5] J. Edmonds. 1965. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B* 69B (1965), 125–130.

[6] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of Mathematics* 17, 3 (1965), 449–467.

[7] Michael Elberfeld and Ken-ichi Kawarabayashi. 2014. Embedding and canonizing graphs of bounded genus in logspace. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. ACM, 383–392.

[8] David Eppstein and Vijay V. Vazirani. 2019. NC algorithms for computing a perfect matching, number of perfect matchings, and a maximum flow in one-crossing-minor-free graphs. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*.

[9] Jeff Erickson and Amir Nayyeri. 2011. Minimum cuts and shortest non-separating cycles via homology covers. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1166–1176.

[10] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. 2016. Bipartite perfect matching is in quasi-NC. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 754–763.

[11] Anna Galluccio and Martin Loebl. 1999. On the theory of Pfaffian orientations. I. Perfect matchings and permanents. *Electron. J. Combin* 6, 1 (1999), R6.

[12] Shafi Goldwasser and Ofer Grossman. 2015. Perfect bipartite matching in pseudo-deterministic RNC. In *Electronic Colloquium on Computational Complexity (ECCC)*, Vol. 22. 208.

[13] Rohit Gurjar and Thomas Thierauf. 2016. Linear matroid intersection is in quasi-NC. In *Electronic Colloquium on Computational Complexity (ECCC)*, Vol. 23. 182.

[14] Rohit Gurjar, Thomas Thierauf, and Nisheeth K. Vishnoi. 2017. Isolating a vertex via lattices: Polytopes with totally unimodular faces. *Arxiv Preprint Arxiv:1708.02222* (2017).

[15] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43 (1986), 169–188.

[16] Donald B Johnson. 1987. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM (JACM)* 34, 4 (1987), 950–967.

[17] N. C. Kalra and P. C. P. Bhatt. 1985. Parallel algorithms for tree traversals. *Parallel Comput.* 2, 2 (1985), 163–171.

[18] Richard M. Karp, Eli Upfal, and Avi Wigderson. 1986. Constructing a perfect matching is in random NC. *Combinatorica* 6, 1 (1986), 35–48.

[19] Pieter Kasteleyn. 1967. Graph theory and crystal physics. *Graph Theory and Theoretical Physics* (1967), 43–110.

[20] Raghav Kulkarni, Meena Mahajan, and Kasturi R, Varadarajan. 2008. Some perfect matchings and perfect half-integral matchings in NC. *Chicago Journal of Theoretical Computer Science* 4, 2008.

[21] László Lovász. 1979. On determinants, matchings, and random algorithms. In *FCT*, Vol. 79. 565–574.

[22] L. Lovász and M. D. Plummer. 1986. *Matching Theory*. North-Holland, Amsterdam–New York.

[23] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing* 15, 4 (1986), 1036–1053.

[24] Meena Mahajan and Kasturi R, Varadarajan. 2000. A new NC-algorithm for finding a perfect matching in bipartite planar and small genus graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. ACM, 351–357.

[25] Gary L. Miller and Joseph Naor. 1989. Flow in planar graphs with multiple sources and sinks. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, 1989*. IEEE, 112–117.

[26] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. 1987. Matching is as easy as matrix inversion. *Combinatorica* 7, 1 (1987), 105–113.

[27] Manfred W. Padberg and M. Ram Rao. 1982. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research* 7, 1 (1982), 67–80.

[28] Christos H. Papadimitriou and Mihalis Yannakakis. 1982. The complexity of restricted spanning tree problems. *Journal of the ACM (JACM)* 29, 2 (1982), 285–309.

[29] Jean-Claude Picard and Maurice Queyranne. 1980. On the structure of all minimum cuts in a network and applications. *Combinatorial Optimization II* (1980), 8–16.

[30] Piotr Sankowski. 2018. NC algorithms for weighted planar perfect matching and related problems. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Ola Svensson and Jakub Tarnawski. 2017. The matching problem in general graphs is in quasi-NC. In *Proceedings of the 2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. 696–707.

[32] Robert E. Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* 14, 4 (1985), 862–874.

[33] Leslie G. Valiant. 1979. The complexity of computing the permanent. *Theoretical Computer Science* 8, 2 (1979), 189–201.

[34] Vijay V. Vazirani. 1989. NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. *Information and Computation* 80, 2 (1989), 152–164.

Received July 2018; revised December 2019; accepted April 2020