Planar Graph Perfect Matching is in NC

Nima Anari[1] and Vijay V. Vazirani[2]

[1]Simons Institute for the Theory of Computing
[2]University of California, Irvine*

Abstract

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 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 paper, we give an NC algorithm for finding a perfect matching in a planar graph. Our algorithm uses the above-stated fact about counting matchings in a crucial way. Several new ideas are needed, such as finding a point in the interior of the minimum weight face of the perfect matching polytope and finding a balanced odd tight set in NC.

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 [KUW86; MVV87]. Within this question, the case of planar graphs has remained an enigma: For general graphs, counting the number of perfect matchings is far harder than finding one: the former is #P-complete [Val79] and the latter is in P [Edm65b]. However, for planar graphs, a polynomial time algorithm for counting perfect matchings was found by Kasteleyn, a physicist, in 1967 [Kas], and was observed to be in NC once a fast parallel algorithm was discovered, in 1976, for computing the determinant of a matrix [Csa76]. On the other hand, finding a perfect matching in a planar graph in NC has resisted a solution. In this paper, we provide such an algorithm.

An RNC algorithm for the decision problem, of determining if a graph has a perfect matching, was obtained by Lovasz [Lov79], 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, Upfal, and Wigderson [KUW86]. This was followed by a somewhat simpler algorithm due to Mulmuley,

*On leave from Georgia Tech.

Vazirani, and Vazirani [MVV87]. It is well known that matching has played a central role in the development of the theory of algorithms: Its study, from various computational viewpoints, has led to powerful tools and quintessential paradigms for the entire theory. The parallel perspective has also led to such gains: The first RNC matching algorithm led to a fundamental understanding of the computational relationship between search and decision problems [KUW85] and the second algorithm yielded the Isolation Lemma [MVV87], which has found several applications in complexity theory and algorithms.

The first substantial progress on answering the opening question of this paper was made by Miller and Naor in 1989 [MN89]. They gave an NC algorithm for finding a perfect matching in bipartite planar graphs using a flow-based approach. In 2000, Mahajan and Varadarajan gave an elegant way of using the NC algorithm for counting perfect matchings to finding one, hence giving a different NC algorithm for bipartite planar graphs [MV00]. Our algorithm is inspired by their approach.

In the last few years, several researchers have obtained quasi-NC algorithms for matching and its generalizations; such algorithms run in polylogarithmic time though they require $O(n^{\log^{O(1)} n})$ processors. These algorithms achieve a partial derandomization of the Isolation Lemma for the specific problem of perfect matching. Several nice algorithmic ideas have been discovered in these works and our algorithm has benefited from some of these. First, Fenner, Gurjar, and Thierauf gave a quasi-NC algorithm for perfect matching in bipartite graphs [FGT16], followed by the algorithm of Svensson and Tarnawski for general graphs [ST17]. Algorithms were also found for the generalization of bipartite matching to the linear matroid intersection problem by Gurjar and Thierauf [GT16], and to a further generalization of finding a vertex of a polytope with totally unimodular constraints, by Gurjar, Thierauf, and Vishnoi [GTV17].

Our main theorem is the following.

Theorem 1. *There is an* NC *algorithm that given a planar graph G, returns a perfect matching in G, if it has one.*

2 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.

2.1 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. 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 space in general.

Next assume that G has a perfect matching. A simple cycle C in G is said to be *nice* if removal of its vertices leave 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 edge 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.

Of course, G may not have a Pfaffian orientation. A key fact underlying our algorithm is that every planar graph has a Pfaffian orientation and moreover, such an oriantation can be found in NC and the determinant can be computed in NC by Csanky's algorithm [Csa76]. Hence we can answer the decision question of whether G has a perfect matching in NC.

We will assume that G is connected and is matching covered, i.e., each edge of G is in some perfect matching. Note that since counting perfect matchings is in NC, we can remove edges that are not in any perfect matching.

2.2 The perfect matching polytope, its faces, and odd tight sets

The perfect matching polytope for $G = (V, E)$ is defined in $\mathbb{R}^E$ and is given by the following set of linear equalities and inequalities [Edm65a].

$$\begin{aligned} \langle \mathbb{1}_{\delta(v)}, x \rangle &= 1 \quad \forall v \in V, \\ \langle \mathbb{1}_{\delta(S)}, x \rangle &\geq 1 \quad \forall S \subset V, \text{ with } |S| \text{ odd}, \\ x_e &\geq 0 \quad \forall e \in E. \end{aligned} \tag{1}$$

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 $\mathbb{PM}(G)$:

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

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 $\mathbb{PM}(G)$ and will be denoted by $\mathbb{PM}(G, w)$.

One of the key steps needed by our algorithm is finding a point in the interior of the face $\mathbb{PM}(G, w)$ in NC. We show how to do this in section 3. This requires computing a Pfaffian orientation for G and then evaluating the Tutte matrix for appropriate substitutions of the variables.

In general, a face of $\mathbb{PM}(G)$ 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 *odd tight sets*. Two such odd tight 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 odd tight sets or $S_1 - S_2$ and $S_2 - S_1$ are odd tight 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 odd tight sets $\mathcal{S}$, one can successively uncross pairs to obtain a family of odd tight 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 space of equality constraints, by defining things appropriately.

2.3 An even walk, its circulation, its rotation, and blocking sets

For this paper, 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. This will be a walk that traverses an even number of edges and will also be called an even walk. Note that all of our walks start and end at the same location. Next we define the alternating vector of an even walk. 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

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

In terms of its components we have

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

Let w be a weight vector on edges of G. Then we define the circulation $\text{circ}_w(W)$, or simply $\text{circ}(W)$ if w is clear from context, as

$$\text{circ}(W) = \langle w, \chi_W \rangle = -w_{e_1} + w_{e_2} - w_{e_3} + \ldots + w_{e_k}.$$

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 can be positive or negative, but for our purposes it will always have a small, though still inverse exponentially large, magnitude. As a simple observation, note that

$$\langle w, y \rangle = \langle w, x \rangle + \epsilon \cdot \text{circ}(W).$$

So if $\text{circ}(W) \neq 0$, then by choosing the sign of ϵ appropriately we can ensure that $\langle w, y\rangle < \langle w, x\rangle$.

As stated above, in section 3 we show how to find a point in the interior of the face $\mathbb{PM}(G, w)$ in NC. In fact, the point we find is

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

where $M_1, \ldots, M_m$ are all the minimum weight perfect matchings in G. We will denote this point by $\text{avg}(\mathbb{PM}(G, w))$.

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

Lemma 1. *Let $x = \text{avg}(\mathbb{PM}(G, w))$ 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 $x_e > 0$. Let $K(n)$ 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 intuitively 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

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

and note that $\langle 1_{\delta(S)}, \mathbb{1}_{M_i}\rangle$ 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 1 almost ensures that the point y is inside the matching polytope $\mathbb{PM}(G)$. The only way that y cannot be in $\mathbb{PM}(G)$ is if there is an odd set $S \subset V$ such that $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$, i.e., an odd tight set, whose constraint gets violated by y. This leads us to the following important lemma, which enables us to extract an odd tight set *blocking* the rotation by the walk W.

Lemma 2. *Suppose that w is a weight vector, $x = \operatorname{avg}(\mathbb{PM}_w(G))$, W is a walk that satisfies the conditions of lemma 1, and furthermore $\operatorname{circ}_w(W) \neq 0$. 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 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 $\operatorname{circ}(W) \neq 0$, by choosing the sign of ϵ appropriately, we can make sure that $\langle w, y \rangle < \langle w, x \rangle$. We will further choose the magnitude of ϵ to be small enough that the conditions of lemma 1 are satisfied. Now, since x was a minimizer of the linear function $x \mapsto \langle w, x \rangle$ over the polytope $\mathbb{PM}(G)$, it must be the case that $y \notin \mathbb{PM}(G)$.

Therefore one of the constraints defining the matching polytope, eq. (1), must not be satisfied for y. But lemma 1 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 $\langle \mathbb{1}_{\delta(S)}, y \rangle < 1$.

Now, 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 1, 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. The results of section 4 tell us how to find this odd tight set S.

3 Finding a Point in the Minimum Weight Face of the Perfect Matching Polytope

In this section, we give an NC algorithm for the following problem: Given a planar graph $G = (V, E)$ and an integral weight vector on edges, $w \in \mathbb{Z}_+{}^E$, where w is specified in unary (i.e., the weights are polynomially bounded), find a point, x, in the interior of $\mathbb{PM}(G, w)$, where $\mathbb{PM}(G, w)$ 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).

Let $\#G_w$ denote the number of minimum weight perfect matchings in G w.r.t. edge weights w, and for each edge $e \in E$, let $\#G_w^e$ denote the number of such 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 minimum weight perfect matchings in G, then

$$x = \frac{\mathbb{1}_{M_1} + \ldots + \mathbb{1}_{M_m}}{m}.$$

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 $x_{ij} = y^{w_e}$, otherwise let $x_{ij} = -y^{w_e}$, where y is an indeterminate. Let B be the resulting matrix. Observe that the exponents of the entries of B are small and hence its determinant can be computed in NC [BCP83]. Consider the lowest degree term in $|B|$; 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, i.e., it is $(\#G_w)^2$.

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

Lemma 3. *Given an integral weight vector w represented in unary, there is an* NC *algorithm which returns*

$$x = \text{avg}(\mathbb{PM}(G, w)).$$

4 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 graph $G = (V, E)$ with edge weights given by $w : E \to \mathbb{R}_+$. We will crucially use the fact that an s-t max-flow and min-cut can be computed in a planar graph in NC [Joh87]. For each pair of vertices $u, v \in V$, let $f(u, v)$ denote the weight of a minimum u-v cut in G.

Note that if $(S, \overline{S})$ is a minimum $u - v$ cut, then S must consist of a number of connected components of G together with an internally connected subset of vertices. This is because if S contains two disjoint sets S_1, S_2 that have no edges to each other, we can find a smaller $u - v$ cut by either taking $S - S_1$ or $S - S_2$ depending on which one still contains u. In any case, the graph obtained by shrinking S in G will always 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 removal of edge (S_i, S_j) from T disconnects T. This splits the partitions into two sets, and naturally defines a cut, say $(S, \overline{S})$ in G. This cut 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 mimimal minimum r-v cut in the graph G' defined above after shrinking subtrees incident to S_i. Let S_v denote 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 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. The resulting graph, say T', will still be connected, since the finer partitions of S_i always form a connected subtree of the tree on partitions at any stage of the algorithm. 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. The following fact is straightforward:

Lemma 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.*

Corollary 1. *Let $r, v_1, \ldots v_k \in V$ and let $S_{v_1}, \ldots, S_{v_k}$ be minimal minimum v_1-r, ... v_k-r cuts in G, respectively. Then $S_{v_1}, \ldots, S_{v_k}$ form a laminar family.*

Let r denote a central vertex for S_i that is found by the algorithm. By Corollary 1, the cuts S_v, for each vertex $v \in S_i$, $v \neq r$ form a laminar family. Let $M_1, \ldots, M_l$ be the maximal sets of this laminar family. Clearly, we can split S_i into the l sets $M_1 \cap S_i, \ldots, M_l \cap S_i$ and attach subtrees to appropriate sets as given by $M_1, \ldots, M_l$. 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 2. *There is an* NC *algorithm for obtaining a Gomory-Hu tree for an edge-weighted planar graph.*

Let W be a walk and y be obtained by rotating it. Assume that there is an odd set that blocks W. Then, there is an odd cut S, with $y(\delta(S)) < 1$ and we need to find one such cut. Recall that the process of rotating W does not change the value of cuts of single vertices, i.e., for each vertex v, $y(\delta(v)) = 1$. Therefore, the cut we are interested in is a minimum odd cut in G. 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 [PR82].

Remark: An alternative way of finding an odd tight set S is to use the Pickard-Queyranne structure of minimum s-t cuts [PQ80]. However, that method is more cumbersome to describe.

5 Finding Even Walks by Pairing Up Faces

In this section, we show how to find $\Omega(n)$ edge-disjoint even walks in G in NC. We first need to "clean up" the graph. We will assume that G is connected, has no parallel edges, and that it is matching-covered, i.e., there is a point $x \in \mathbb{PM}(G)$ with $x_e > 0$ for all $e \in E$. We will see in section 7 how this assumption is automatically satisfied. But we need a further "clean up". Let $(v_1, \ldots v_4)$ be a three length path in G with v_2 and v_3 having degree two. Replace this path by the direct edge (v_1, v_4); if the edge was already present, there is no need to add the edge. Similarly, if $(v_1, \ldots, v_k)$ is a path in G with $v_2, \ldots v_{k-1}$ of degree two and v_1 and v_2 not of degree two, then, if k is even, replace the path by the edge (v_1, v_k), and if k is odd, replace it by (v_1, v_{k-1}). In section 7 we will show how this operation does not hurt us. For this section we assume that the graph G has already been cleaned up, i.e., no vertices of degree 2 are adjacent to each other. We will further assume that G does not have any vertices of degree 1. This is because the edges of G are matching-covered and G is connected. So if v is a vertex of degree 1 then the neighbor of G must also have degree 1 (only the edge going to v can be in the matching), which means that u is in a connected component of size 2.

After this clean up operation, it is easy to see that the graph has $\Omega(n)$ faces. By Euler's formula we have

$$n - |E| + f = 2,$$

where f denotes the number of faces. This tells us

$$f \geq |E| - n.$$

Now let m be the number of vertices of degree 2. Since no two such vertices are adjacent to each other, it is immediate that $|E| \geq 2m$. This means

$$f \geq 2m - n.$$

On the other hand, we have

$$f \geq |E| - n = \sum_{v \in V} \frac{\deg(v) - 2}{2} \geq \frac{1}{2}(n - m).$$

The last inequality follows because for every vertex of degree higher than 2, the term in the sum is at least 1/2. Now multiplying this inequality by 4 and adding it to the previous one, we get

$$5f \geq n \implies f = \Omega(n).$$

Consider the dual G^* of G. Corresponding to each face in G, the dual has a vertex. Let f denote the number of faces in G, or equivalently, the number of vertices in G^*. [MV00] argue that at most $f/2$ of the vertices on G^* are of degree 12 or more. On the remaining vertices, called *low degree vertices*, find a maximal independent set. The inclusion of any low degree vertex in the independent set can prevent at most 11 other such vertices from entering the set. All together, the maximal independent set must contain at least $f/24$ vertices. Now, since $f = \Omega(n)$, the independent set found has cardinality $\Omega(n)$. The faces corresponding to these vertices in G are edge-disjoint. Hence we get $\Omega(n)$ edge-disjoint faces. Note that a maximal independent set can be found in NC [Lub86].

If at least half these faces 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 $\Omega(n)$ even walks of the second type.

Let us assume that the number of such odd faces is even number (possibly by discarding one face) and that each one has length bounded by $O(1)$ (in fact 12). For each odd face O, pick an arbitrary vertex v of O and put a "token" on that vertex v; note that multiple tokens may be placed on the same vertex. Now find a spanning tree T in the graph. Then we can pair up our tokens by edge-disjoint paths from the tree T.

Lemma 6. *Given a tree T with an even number of tokens placed on its vertices, we can pair up the tokens in such a way that the paths connecting the pairs are edge disjoint. Furthermore, these paths can be found in* NC*.*

Proof. For each edge $e \in T$, we will count the number of tokens on either side of T when e is removed. Since there are an even number of tokens, this count must either be odd on both sides or even on both sides. We can clearly 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 in the tree must have the same parity as the number of tokens on it. This is because the parity of the total number of tokens is the same as the parity of the degree of this vertex plus the number of tokens on it.

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 be done using the doubling trick to get an NC algorithm. Alternatively, we will ultimately only use pairings where the length of the path is at most a constant, which can be followed even sequentially. □

For any two tokens that get paired, look at the corresponding faces F_1, F_2. We can connect F_1, F_2 by the path in T going from F_1's token to F_2's token. Let us call this path P. Note that P might share vertices or edges with F_1 or F_2, but there is a simple trimming procedure to fix this: Replace P by the subpath from the last vertex of F_1 on P until the first vertex after it from F_2. Therefore by possibly relocating the tokens, we can ensure that F_1 and F_2 can be connected to each other using a path from T, and that the paths from T are all edge-disjoint.

Now look at all pairs of faces whose connecting paths from T have length more than a large constant C. There can be at most $(n-1)/C$ such pairs, because the paths on the tree are edge-disjoint and the tree has $n-1$ edges. So if we take C to be large enough, these pairs would only constitute half of all the pairs. Now we simply ignore them.

We end up with $\Omega(n)$ paired up faces, whose connecting paths on the tree have length at most $O(1)$ each. Now connect each such pair of faces with their connecting path from the tree in order to get an even walk of the second type. This gives us $\Omega(n)$ edge-disjoint walks. Two such even walks W_1, W_2 might still not be edge-disjoint. But if W_1, W_2 share an edge, it must be a tree edge on W_1 and a face edge on W_2 or the other way around; this is because connecting paths are edge-disjoint, and we also chose the faces to be edge-disjoint. This means that W_1 can share an edge with at most $O(1)$ other even walks: For each face edge of W_1 (of which there are $O(1)$ many) there could be at most one other even walk having that edge in its connecting path, and for every edge in the connecting path of W_1 (of which there are $O(1)$ many) there could be at most one other even walk having that edge in one of its two faces.

So if we form a conflict graph between the even walks, where each edge represents having an edge in common, this graph would have maximum degree bounded by $O(1)$. We can find a maximal independent set in this graph to find edge-disjoint even walks, which can be done in NC [Lub86]. Since the maximum degree is $O(1)$, such a maximal independent set would still have $\Omega(n)$ many even walks as desired.

It is easy to see that all these operations can be done in NC via well known methods. Hence we get:

Lemma 7. *There is an* NC *algorithm for finding* $\Omega(n)$ *edge-disjoint even walks in a simple, connected, matching-covered planar graph* G *with no vertices of degree* 2 *adjacent to each other.*

6 Uncrossing Odd Tight Sets

Suppose we are given a list of odd tight sets $S_1, \ldots, S_m$ that could cross each other in arbitrary ways. Our goal is to *uncross* them, so that we can shrink all of these sets 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 $\Omega(n)$.

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 $\mathbb{PM}(G)$ with $x_e > 0$ for all $e \in E$.

Definition 1. *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 odd tight 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 8. *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 1 that there must be some odd tight 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 odd tight sets S_1 and S_2 we will show that we can uncross them, i.e., replace them by new odd tight 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 9. *If S_1 and S_2 are odd tight sets then either $S_1 \cap S_2, S_1 \cup S_2$ are odd tight 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 odd tight 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 $x \in \mathbb{PM}(G)$ 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 odd tight 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 an odd tight 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 odd tight 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 an odd tight set, the claim follows. □

Now we use lemma 9 to prove the claim that odd tight sets can be uncrossed without shrinking $\Lambda(S_1) + \Lambda(S_2)$.

Lemma 10. *Suppose that S_1, S_2 are odd tight 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 an odd tight 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 odd tight 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 9 that $S_1 \cup S_2$ is an odd tight set. It is trivial from definition 1 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 9 it follows that $S_2 - S_1$ is an odd tight 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 odd tight 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 9 that $T - S_1$ and $S_1 - T$ are odd tight 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 9 to the sets T and $S_2 - S_1$, both of which are odd tight sets. Note that $T \cap (S_2 - S_1) = T - S_1$ which has even size by assumption. Therefore by lemma 9, $(S_2 - S_1) - T$ and $T - (S_2 - S_1) = S_1 \cap T$ are also odd tight 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 odd tight sets $S_1, \ldots, S_m$, repeated applications of lemma 10 allows us to uncross them, i.e., replace them by pairwise disjoint odd tight 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 10 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 odd tight sets $S_1, \ldots, S_m$. The high-level description of our procedure, Uncross, is given in algorithm 1. 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.

```
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
```

Algorithm 1: Divide-and-conquer algorithm for uncrossing odd tight sets

Next, we will describe the merging procedure MergeUncross. The procedure MergeUncross, similarly to Uncross, accepts a list of odd tight sets and returns a list of pairwise disjoint odd tight 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 11. *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 odd tight sets and $C_1, \ldots, C_q$ are also pairwise disjoint odd tight 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 intersections.

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 2. *For odd tight sets $S_1, \ldots, S_m$ satisfying the conditions of lemma 11, 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 11 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 $H(S_{i_1}, \ldots, S_{i_k})$ to denote the induced subgraph on nodes $S_{i_1}, \ldots, S_{i_k}$. With this notation we have

$$|S_{i_1} \cup \ldots S_{i_k}| \stackrel{2}{\equiv} |H(S_{i_1}, \ldots, S_{i_k})|,$$

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

By lemma 10, if S_1, S_2 have an edge between them in H, then the union $S_1 \cup S_2$ will also be an odd tight 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 an odd tight set.

Can we repeatedly apply this procedure and otain $S_1 \cup \cdots \cup S_m$ as an odd tight 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 10 only produces *odd* tight sets. For simplicity of notation we use $\cup H$ to denote $S_1 \cup \ldots S_m$.

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

Lemma 12. *Assume that $H = H(S_1, \ldots, S_m)$ is connected and that $|H| \stackrel{2}{\equiv} 1$. Then $\cup H = S_1 \cup \cdots \cup S_m$ is an odd tight set, and $\Lambda(S_1, \ldots, S_m) \subseteq \Lambda(\cup H)$.*

Proof. We just need to show that $\cup H$ is an odd tight set. The fact that $\Lambda(S_1, \ldots, S_m) \subseteq \Lambda(\cup H)$ is trivial from definition 1.

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 10.

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 an odd tight set. But S is also an add tight set, and by assumption the union of the two, $\cup(H - \{S\}) \cup S = \cup H$, is also odd. So by lemma 10 we get that $\cup H$ is an odd tight 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 other than the one going to its parent, and in a DFS tree there are no cross edges, which means that this edge must be a back edge.

Note that in a DFS tree, the leaf nodes are never connected to each other. This implies, by simple parity counting, that if S_1, S_2 are two leaves then $|H - \{S_1, S_2\}| \overset{2}{\equiv} 1$. Note that $H - \{S_1, S_2\}$ is also connected, so by induction $\cup(H - \{S_1, S_2\}$ is an odd tight 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 odd tight sets. Their union is again $\cup H$ which has an odd parity. So again by lemma 10 we get that $\cup H$ is an odd tight 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 an odd tight 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 odd tight 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 an odd tight 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 $\deg_H(P) \overset{2}{\equiv} 0$. 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 has satisfies the assumptions of the induction. So again, we get two odd tight sets whose union is $\cup H$ and therefore $\cup H$ is an odd tight 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 an odd tight 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 $\deg_H(R) \overset{2}{\equiv} 0$. 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 an odd tight 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 $\deg_H(C) \overset{2}{\equiv} 0$. 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 an odd tight set. On the other hand $S_2 \cup Q \cup C$ is also an odd tight 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 an odd tight cut. Note that $A \cup B \cup P$ is also an odd tight 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 an odd tight set. □

Lemma 12 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 10. 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 13. *Assume that $H = H(S_1, \ldots, S_m)$ is connected and $|H| \stackrel{2}{\equiv} 0$. 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 [TV85].

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 $\overline{|B|} \stackrel{2}{\equiv} 1 + |B|$. 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 $|H| \stackrel{2}{\equiv} 0$, or in other words, $\overline{|H|} \stackrel{2}{\equiv} 1$, 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 12 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 12 and 13, 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 12 or lemma 13 to each connected component of H.

```
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
        Add ∪H_i to F.
    else
        Let H'_i, H''_i be the two induced subgraphs promised by lemma 13.
        Add ∪H'_i to F.
        Add ∪H''_i − ∪H'_i to F.
    end
end
return F
```

Algorithm 2: Algorithm for uncrossing partially uncrossed sets

All together we get the following result:

Theorem 3. *Given odd tight sets $S_1, \ldots, S_m$, there is an* NC *algorithm that outputs pairwise disjoint odd tight sets $S'_1, \ldots, S'_{m'}$ such that*

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

7 Shrinking and Finding One Balanced Tight Cut

Having most of the elements so far, we would have ideally liked to do the following:

1. Find $\Omega(n)$ even walks.
2. Set the weight vector w so that $\text{circ}_w(W) \neq 0$ for all of our even walks W.
3. Find $x \in \mathbb{PM}(G, w)$ and delete edges of G where $x_e = 0$.
4. We either delete $\Omega(n)$ edges, in which case we can recurse, or most of our even walks get blocked by odd tight sets.
5. We find an odd tight set blocking each even walk in parallel.
6. We use theorem 3 to uncross these blocking odd tight sets to get pairwise disjoint $S_1, \ldots, S_m$.
7. We shrink each S_i. By doing so, we remove at least one edge from each W that was blocked, so we still remove $\Omega(n)$ edges.

However, this is the point where more ideas are needed. Indeed we can perform all of the operations mentioned in NC. But our goal is to ultimately extract a matching. It is not hard to see that any matching in the shrunk graph can be extended to a matching in the original graph, but the problem is that this extension cannot be simply done in parallel. I.e., we have to wait for the matching in the shrunk graph to be determined, to know which edge of each $\delta(S_i)$ is chosen, and then continue to extend that edge to a matching in S_i. This results in an algorithm with higher than polylogarithmic running time.

Next we introduce the idea of a balanced viable cut. A set is viable if and only if it is a tight odd cut for some point $x \in \mathbb{PM}(G)$.

Definition 3. *An odd set S is called viable if there exists at least one matching M with $M \cap \delta(S) = 1$.*

Suppose that we had an NC algorithm for finding a balanced viable set, i.e., a viable set S such that $|S|, |V - S| = \Omega(n)$. Then the following algorithm would find a perfect matching for us.

```
PerfectMatching(G = (V, E))
if |V| = O(1) then
  | Find a perfect matching M by some sequential algorithm.
  | return M
else
  | Find a balanced viable set S.
  | w ← 1_δ(S)
  | 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}
```

Algorithm 3: Divide-and-conquer algorithm for finding a perfect matching

Let us show why algorithm 3 is correct. When S is a viable set, then there is a matching M such that $\langle \mathbb{1}_{\delta(S)}, \mathbb{1}_M \rangle = 1$. Since we set $w = \mathbb{1}_{\delta(S)}$, any minimizer x of the weight w will have $\langle \mathbb{1}_{\delta(S)}, x \rangle = 1$. Now if e is an edge of $\delta(S)$ with $x_e > 0$, then there is some perfect matching M with $\mathbb{1}_M \in \mathbb{PM}(G, w)$ such that $M \cap \delta(S) = \{e\}$. If we include the edge e in the matching, and remove all other edges in $\delta(S)$, the remaining graph that has to be matched will be a disjoint union of G_1 and G_2. Each of these graphs has a perfect matching (the restriction of M to them for example), so we can recursively, but *in parallel*, find one perfect matching in each.

If the viable set found is balanced, then n will be reduced by a constant factor each time, making the recursion depth logarithmic.

For the rest of this section we explain how we can find a balanced viable set S in NC. We will first describe the algorithm, leaving some loose ends, and at the end come back and address the loose ends.

We start from some graph G. At any point we will be working with some shrunk version of G. We keep track of how many original vertices of G are in each shrunk vertex and call that the multiplicity of a vertex. Initially each vertex has multiplicity 1.

We make sure that the multiplicity of each vertex v is always odd, and that any viable set S in our current graph is also viable in the initial graph that we started from (when unshrunk). Note that if at any point during the algorithm the multiplicity of a vertex becomes balanced, i.e., if it becomes $\geq \Omega(n)$ and $\leq n - \Omega(n)$, then we have found our viable set; a singleton is always a viable set (but not always balanced) and the singleton representing this vertex is a balanced set in the original graph. We will actually make sure that the multiplicity of every vertex is always at most cn for some constant c. If a multiplicity every tries to jump to $\geq n - \Omega(n)$, we will immediately extract a balanced viable set; we will discuss this at the end.

At any point in time we will only be working with the connected component of G that has the largest total multiplicity. If this total multiplicity falls below $n/2$, then we can easily find a viable set: We simply sort the components by total multiplicity and then partition them into two a large pieces part and a small pieces part such that each part's total multiplicity is $\geq \Omega(n)$ and $\leq n - \Omega(n)$. This can be done in NC by checking all partitions in parallel. Then we simply move one vertex of one connected component from one side to the other to get our balanced cut. This cut is viable because it is really the same as a singleton cut.

Because of the previous observation, and for simplicity, we let G refer to its largest connected component at any time, and assume that G is connected.

We "clean up" G (to be discussed at the end of this section) and then find $\Omega(|V_G|)$ many pairwise-disjoint even walks in it. We then find a weight vector w such that $\text{circ}_w(W) \neq 0$ for any of our even walks W. This is easily done by setting $w_e = 1$ for *exactly one* edge e of every even walk W and setting $w_e = 0$ for every other edge.

Next, we find a point $x = \text{avg}(\mathbb{PM}(G, w))$ and delete the edges e of G such that $x_e = 0$. We either removed an edge from half of our even walks, in which case the number of edges in G is reduced by a constant factor, or we find $\Omega(|V_G|)$ many even walks that are blocked by odd tight sets.

For each of our blocked even walks, we find an odd tight set blocking it, by rotating x by the walk an inverse-exponentially large amount and finding the minimum odd cut using the Gomory-Hu tree in the result. We can obviously do this in parallel for all of our even walks.

Next, we uncross the obtained odd tight sets using theorem 3 to get pairwise disjoint odd tight sets $S_1, \ldots, S_m$. If we shrink each S_i into a single vertex, and do this shrinking with respect to x as well, the resulting x would be in the perfect matching polytope of the shrunk graph; this is because $\langle \mathbb{1}_{\delta(S_i)}, x \rangle = 1$, so each S_i satisfies the degree constraint of the perfect matching polytope for the shrunk graph. Note that any edge in each $\delta(S_i)$ can be extended to a matching in S_i itself. This is because there is a matching M containing only that edge from $\delta(S)$. So this means that any viable set in the shrunk graph is viable in the previous graph as well, because any matching in the shrunk graph will pick exactly one edge from each $\delta(S_i)$, and this can be extended back to a matching in the original graph.

Our shrinking operation however reduces the number of edges. Because for each even walk W, there is some S_i such that $W \cap E(S_i) \neq \emptyset$. So the number of edges is reduced by at least a constant factor again.

In each iteration we reduce the number of edges by a constant factor. So after at most polylogarithmically many steps, we are bound to find a viable set.

7.1 Odds and ends

We now address some of the loose ends left in the above description.

The first point is: how do we avoid getting vertices with very large multiplicities ($\geq n - \Omega(n)$). As mentioned before we always keep the multiplicities below some cn for some very small constant $c > 0$; if at any point the multiplicity of a vertex goes above cn but not all the way to $n - cn$, we automatically get a balanced viable cut from that singleton. The only way that this jump can possibly happen is if we shrink a set S_i containing a very large total multiplicity. We can ensure our sets do not have this property before uncrossing; if a set contains a very large total multiplicity we simply replace it with its complement. If the total multiplicity is still above cn, then that set gives us a balanced viable set. But still our uncrossing algorithm might create an S_i with very large multiplicity. But if we look closely at our uncrossing algorithm, we can find a simple remedy to this problem. If the multiplicity of any of the tight odd sets found at any point in uncrossing are between cn and $n - cn$, we simply return them, since they are an odd tight set and therefore viable. So we must again have a jump from odd tight sets below cn to $n - cn$ at some point. But this jump can only happen in the applications of lemmas 12 and 13 where we take the union of a number of odd tight sets. So at the point where the jump occurs, there is a connected intersection parity graph $H(S_1, \dots, S_m)$ where the multiplicities of S_i are below cn and then the multiplicity of $S_1 \cup \cdots \cup S_m$ is $\geq n - cn$. In this case we use the following trick: We sort $S_1, \dots, S_m$ in such a way that $H(S_1, \dots, S_i)$ is connected for all i; for example, one way of doing this in NC is to find a spanning tree in H and then sorting S_i's according to distance from an arbitrary root in the tree. Now look at the first index i such that the multiplicity of $S_1 \cup \cdots \cup S_i$ goes above $2cn$. Clearly it cannot go above $3cn$. Now look at $H(S_1, \dots, S_i)$. Depending on the parity of it, we can either find one or two tight odd sets that cover everything. One of these sets has to have at least cn multiplicity and clearly no more than $3cn$ multiplicity, which gives us our desired balanced viable set.

The next loose end that we did not directly mention is that when we shrink a set, we need to ensure that the shrunk graph is still planar. But this is true for any of our odd tight cuts S_i. We just need to argue that S_i is internally connected. This is true because if $S_i = S_i' \cup S_i''$ with $S_i' \cap S_i'' = \emptyset$, then one of $|S_i'|$ or $|S_i''|$ will be odd. But note that S_i was a minimum odd cut in a connected graph, i.e., the largest component of G. This cannot be, because either moving S_i' or S_i'' to the other side of the cut would reduce the cut value but still keep the cut odd.

Finally we address the issue of cleaning up G. Before finding the even walks in G, we have to make sure that it satisfies some "niceness" properties. We have to remove parallel edges from G, which is very easy to do. We also have to ensure that G is connected, but for that as mentioned before, we always restrict our attention to the largest connected component. We have to make sure that G is matching-covered. This will be true after the first iteration of the algorithm, but we can also make sure it is always true by simply finding $x = \text{avg}(\mathbb{PM}(G))$ and deleting all edges e with $x_e = 0$. Finally we have to do path reductions, where a path consisting of internal nodes of degree 2 is reduced to a simpler structure; this is essential to make sure that we can always find $\Omega(|V_G|)$ many even walks. In this operation we replace a path $(v_1, \ldots, v_k)$, where k is even, by an edge from v_1 to v_k and remove $v_2, \ldots, v_{k-1}$ from the graph. It is easy to see that this operation preserves matchings. A matching in the new graph that uses the edge (v_1, v_k) would correspond to a matching in the original graph that uses $(v_1, v_2), (v_3, v_4), \ldots, (v_{k-1}, v_k)$, and a matching that does not use this edge corresponds to a matching in the original graph that uses $(v_2, v_3), (v_4, v_5), \ldots, (v_{k-2}, v_{k-1})$. So a viable set in the cleaned up graph can be translated back to a viable set in the original graph as follows. When we delete the vertices $v_2, \ldots, v_{k-1}$, we absorb the multiplicities of $v_2, \ldots, v_i$ into v_1 and the multiplicities of $v_{i+1}, \ldots, v_{k-1}$ into v_k for some odd i. Now if a viable set S cuts (v_1, v_k) we translate it into the original graph, by putting $v_2, \ldots, v_i$ on the v_1 side and $v_{i+1}, \ldots, v_{k-1}$ on the v_k side. This is consistent with S being viable because of the way we extend matchings. We only need to ensure that by absorbing these multiplicities we do not suddenly jump from below cn to above $n - cn$, but this again can be achieved by a similar trick as before. If the path is really long, then even length subpaths can also be thought of as odd tight cuts, and then if a jump is about to occur, we can easily find a balanced subpath that is also a tight odd cut and by extension, a viable set.

8 Discussion

It is quite easy to see that our algorithm can be extended to give an NC algorithm for finding a minimum weight perfect matching in a planar graph, provided the edge weights are given in unary.

We have extended our NC algorithm for perfect matching, using several additional ideas, to bounded genus graphs; more precisely, graphs of genus at most $O(\sqrt{\log n})$ [AV17].

9 Acknowledgements

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

References

[AV17] Nima Anari and Vijay V. Vazirani. "An NC Algorithm for Finding a Perfect Matching in Bounded Genus Graphs". To appear. 2017.

[BCP83] Allan Borodin, Stephen Cook, and Nicholas Pippenger. "Parallel computation for well-endowed rings and space-bounded probabilistic machines". In: *Information and Control* 58.1-3 (1983), pp. 113–136.

[Csa76] Laszlo Csanky. "Fast parallel matrix inversion algorithms". In: *SIAM Journal on Computing* 5.4 (1976), pp. 618–623.

[Edm65a] J. Edmonds. "Maximum matching and a polyhedron with 0,1-vertices". In: *Journal of Research of the National Bureau of Standards B* 69B (1965), pp. 125–130.

[Edm65b] Jack Edmonds. "Paths, trees, and flowers". In: *Canadian Journal of mathematics* 17.3 (1965), pp. 449–467.

[FGT16] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. "Bipartite perfect matching is in quasi-NC". In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*. ACM. 2016, pp. 754–763.

[GT16] Rohit Gurjar and Thomas Thierauf. "Linear Matroid Intersection is in quasi-NC". In: *Electronic Colloquium on Computational Complexity (ECCC)*. Vol. 23. 2016, p. 182.

[GTV17] Rohit Gurjar, Thomas Thierauf, and Nisheeth K Vishnoi. "Isolating a Vertex via Lattices: Polytopes with Totally Unimodular Faces". In: *arXiv preprint arXiv:1708.02222* (2017).

[Joh87] Donald B Johnson. "Parallel algorithms for minimum cuts and maximum flows in planar networks". In: *Journal of the ACM (JACM)* 34.4 (1987), pp. 950–967.

[Kas] PW Kasteleyn. *Graph theory and crystal physics. 1967 Graph Theory and Theoretical Physics pp. 43–110*.

[KUW85] Richard M Karp, Eli Upfal, and Avi Wigderson. "Are search and decision programs computationally equivalent?" In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM. 1985, pp. 464–475.

[KUW86] Richard M Karp, Eli Upfal, and Avi Wigderson. "Constructing a perfect matching is in random NC". In: *Combinatorica* 6.1 (1986), pp. 35–48.

[Lov79] László Lovász. "On determinants, matchings, and random algorithms." In: *FCT*. Vol. 79. 1979, pp. 565–574.

[Lub86] Michael Luby. "A simple parallel algorithm for the maximal independent set problem". In: *SIAM journal on computing* 15.4 (1986), pp. 1036–1053.

[MN89] Gary L Miller and Joseph Naor. "Flow in planar graphs with multiple sources and sinks". In: *Foundations of Computer Science, 1989., 30th Annual Symposium on*. IEEE. 1989, pp. 112–117.

[MV00] Meena Mahajan and Kasturi R Varadarajan. "A new NC-algorithm for finding a perfect matching in bipartite planar and small genus graphs". In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM. 2000, pp. 351–357.

[MVV87] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. "Matching is as easy as matrix inversion". In: *Combinatorica* 7.1 (1987), pp. 105–113.

[PQ80] Jean-Claude Picard and Maurice Queyranne. "On the structure of all minimum cuts in a network and applications". In: *Combinatorial Optimization II* (1980), pp. 8–16.

[PR82] Manfred W Padberg and M Ram Rao. "Odd minimum cut-sets and b-matchings". In: *Mathematics of Operations Research* 7.1 (1982), pp. 67–80.

[ST17] Ola Svensson and Jakub Tarnawski. "The Matching Problem in General Graphs is in Quasi-NC". In: *arXiv preprint arXiv:1704.01929* (2017).

[TV85] Robert E Tarjan and Uzi Vishkin. "An efficient parallel biconnectivity algorithm". In: *SIAM Journal on Computing* 14.4 (1985), pp. 862–874.

[Val79] Leslie G Valiant. "The complexity of computing the permanent". In: *Theoretical computer science* 8.2 (1979), pp. 189–201.