2018 IEEE 59th Annual Symposium on Foundations of Computer Science

Planar Graph Perfect Matching is in NC

Nima Anari
Computer Science Department
Stanford University
anari@cs.stanford.edu

Vijay V. Vazirani
Computer Science Department
University of California, Irvine
vazirani@ics.uci.edu

***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. Our main new idea is an NC algorithm for finding a face of the perfect matching polytope at which many new conditions, involving constraints of the polytope, are simultaneously satisfied. Several other ideas are also needed, such as finding a point in the interior of the minimum weight face of this polytope and finding a balanced tight odd set in NC.

***Keywords*-parallel algorithm; planar graph; perfect matching; Tutte matrix; Pfaffian;**

I. 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 [11, 17]. Within this question, the case of planar graphs had remained an enigma: For general graphs, counting the number of perfect matchings is far harder than finding one: the former is #P-complete [22] and the latter is in P [4]. However, for planar graphs, a polynomial time algorithm for counting perfect matchings was found by Kasteleyn, a physicist, in 1967 [12], and an NC algorithm follows easily[1], given an NC algorithm for computing the determinant of a matrix, which was obtained by Csanky [2] in 1976. On the other hand, an NC algorithm for finding a perfect matching in a planar graph has resisted a solution. In this paper, we provide such an algorithm.

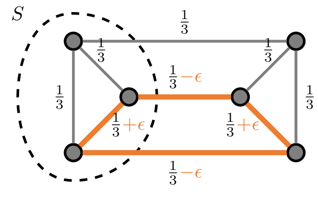
An RNC algorithm for the decision problem, of determining if a graph has a perfect matching, was obtained by Lovász [13], using the Tutte matrix of the graph. The first RNC algorithm for the search problem, of actually finding a perfect matching, was obtained by Karp, Upfal, and Wigderson [11]. This was followed by a somewhat simpler algorithm due to Mulmuley, Vazirani, and Vazirani [17].

The 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 [4] and the counting class #P [22]. 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 [10] and the second algorithm yielded the Isolation Lemma [17], which has found several applications in complexity theory and algorithms. Considering the fundamental insights gained by an algorithmic study of matching, the problem of obtaining an NC algorithm for it has remained a premier open question ever since the 1980s.

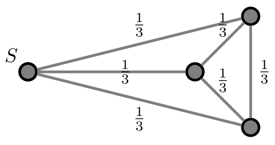
The first substantial progress on this question was made by Miller and Naor [16]. They gave an NC algorithm for finding a perfect matching in bipartite planar graphs using a flow-based approach. Later, Mahajan and Varadarajan [15] gave an elegant way of using the NC algorithm for counting perfect matchings to finding one, hence giving a different NC algorithm for bipartite planar graphs. 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 addressed. 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, Gurjar, and Thierauf [6] gave a quasi-NC algorithm for perfect matching in bipartite graphs, followed by the algorithm of Svensson and Tarnawski [21] for general graphs. Algorithms were also found for the generalization of bipartite matching to the linear matroid intersection problem by Gurjar and Thierauf [7], and to a further generalization of finding a vertex of a polytope with faces given by totally unimodular constraints, by Gurjar,

[1]For a formal proof, in a slightly more general context, see [23].



(a)



(b)

Figure 1. (a) Even cycle blocked by an odd set constraint. Example due to [6, 21]. (b) Resulting graph after shrinking the blocking tight odd set.

Thierauf, and Vishnoi [8].

Our main theorem is the following.

Theorem 1. *There is an* NC *algorithm that for any given planar graph, either returns a perfect matching in it, or reports that it has none.*

In the full version of our paper we generalize theorem 1 to finding a minimum weight perfect matching if the edge weights are polynomially bounded and to finding a perfect matching in graphs of bounded genus; their common generalization easily follows.

II. Overview and Technical Ideas

Due to space constraints, we defer many proofs to the full version of this paper. In this section we give an overview of the main ideas behind our algorithm.

A. The Mahajan-Varadarajan algorithm and difficulties imposed by odd cuts

We first give the idea behind the NC algorithm of Mahajan and Varadarajan [15] 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. In a matching covered bipartite planar graph, every planar face can be used to modify x by increasing and decreasing alternate edges by the same (small) amount ϵ, and this moves the point inside the polytope; this is called a *rotation* of the cycle. Increasing ϵ, at some point an edge e gets zero. When this happens, ϵ cannot be increased anymore and the cycle is *blocked*. If so, the point x moves to a lower (by at least one) dimension face. To make substantial progress, they observe that for any set of edge-disjoint faces, this process can be executed independently in parallel on cycles corresponding to each face, thereby reaching a face of the polytope of correspondingly lower dimension. Finally, they show how to find $\Omega(n)$ edge-disjoint faces in NC, thereby terminating in $O(\log n)$ such iterations.

The fundamental difference between the perfect matching polytopes of bipartite and nonbipartite graphs is the additional constraint in the latter saying that each odd subset, S, of vertices must have at least one edge in the cut $\delta(S)$; see eq. (1) in section III-B. These introduce a second way for a cycle C to be blocked, namely an odd set S, whose cut intersects C, may go tight (section V-A) and the ϵ for this cycle cannot be increased without moving outside the polytope. Thus the cycle will not lose an edge. However, notice that since one of the odd set constraints has gone tight, we are already at a face of lower dimension! In this case, we will say that *cycle C is blocked by odd set S*. For an example, see fig. 1a in which the point inside the polytope is the all $1/3$ vector and the highlighted cycle is blocked by odd set S. Observe that the rotation chosen on the highlighted cycle leads to infeasibility on cut S.

How do we capitalize on this progress? The obvious idea, which happens to need substantially many additional ideas to get to an NC algorithm, is to shrink S, find a perfect matching in the shrunk graph, expand S, remove the matched vertex from it, and find a perfect matching on the remaining vertices of S. For an example of the shrunk graph, see fig. 1b. It is easy to see that at least one edge of C must have both endpoints in S, and therefore the shrunk graph has fewer edges than the original.

As stated above, a number of new ideas are needed to make this rough outline yield an NC algorithm. First, a small hurdle: If G is nonbipartite planar, the procedure of Mahajan and Varadarajan [15] 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 use *even walks*: it consists of either an even cycle or two odd cycles with a path connecting them (section III-C). We give an NC algorithm for finding $\Omega(n)$ edge-disjoint even walks in G (section VI-B).

B. A key algorithmic issue and its resolution

A walk is blocked if it loses an edge or if an odd cut intersecting it goes tight. Either way, we reduce the dimension. However, a new challenge arises: In the first case, the amount of rotation required to make the walk lose an edge is easy to compute, similar to the bipartite case. But in the second case, how do we find the smallest rotation so an odd cut intersecting the walk goes just tight? Note that we seek the "smallest" rotation so no odd cut goes under-tight. We postpone the answer until we address the next hurdle.

Next, we state a big hurdle: 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. In the bipartite case, we could modify all cycles independently in parallel. However, in the nonbipartite case, executing these moves in parallel may take the point outside the polytope. These two cases are illustrated in figs. 2a and 2b. The reason is that whereas rotations on two different walks may be individually feasible, executing them both may make an odd set go under-tight. This is depicted in fig. 3a.

The following idea helps us get around this hurdle: It suffices to find a weight function on edges, w, so that one

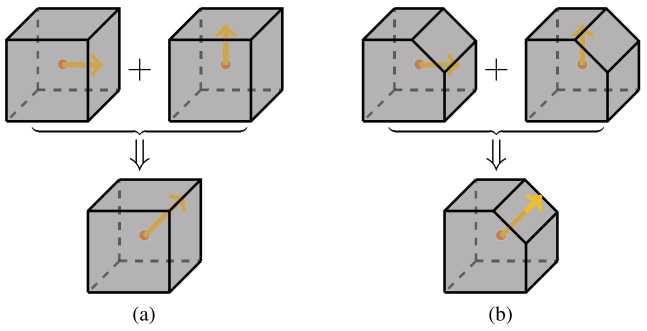


Figure 2. (a) Parallel moves in the bipartite matching polytope. (b) Parallel moves in the nonbipartite matching polytope.

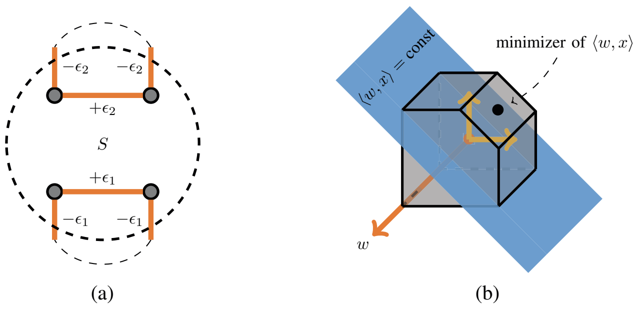


Figure 3. (a) Odd set constraint violated when even walks are rotated independently. (b) Minimizer of appropriate linear function $x \mapsto \langle w, x\rangle$ blocks even walks.

of the half-spaces defined by w contains w itself and the other contains, for each of our even walks, the direction of its rotation. Then, at the minimizer of $x \mapsto \langle w, x\rangle$ in the polytope each of the walks is blocked.Furthermore, the minimizer is still a feasible point, so no odd cut goes under-tight. This idea is illustrated in fig. 3b. Such a function w is easy to construct: in each even walk, pick the weight of any one edge to be 1 and the rest 0 (section IV-B).

Next, we need to find the minimizer of $x \mapsto \langle w, x\rangle$. For this, it suffices to construct an NC oracle for computing $\#G_w^e$, the number of minimum weight matchings in G containing the edge e. This oracle is constructed by finding a Pfaffian orientation (section III-A) for G, appropriately substituting for the variables in the Tutte matrix of G and computing the determinant of the resulting matrix (section VI-A). The minimizer point, x, is the average of all corner points of the face minimizing the weight w.

Some clarifications are due at this point: First, let us answer the opening question of this section, i.e., how to rotate just one given walk so it gets blocked by one of the two ways? Interestingly enough, at present we know of no simpler method for one walk than for multiple walks (binary search comes to mind but that is not an elegant, analytic solution). Second, in the bipartite case, we could use the decrease in the dimension as a measure of progress. In the nonbipartite case, however, when we rotate k edge-disjoint walks, and none of these walks loses an edge, then the dimension of the face we end up on may not be smaller by $O(k)$. There could be very few, or even one, tight odd cut that intersects each of the walks. However, each walk will have at least one edge in a tight odd set, and therefore shrinking them will result in $O(k)$ edges being shrunk. Hence, the number of edges works as a measure of progress.

C. 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 the result of Padberg and Rao [18] stating that the Gomory-Hu tree of a graph will contain a minimum weight odd cut. We show how to find a Gomory-Hu tree in a weighted planar graph in NC (section VI-C), a result of independent interest. Next, consider a walk W which, w.r.t. the current point x in the polytope, is intersecting a tight odd cut. We rotate walk W slightly so the point x becomes infeasible. Now W crosses an under-tight cut, (lemma 10). Hence, w.r.t. x, each minimum weight odd cut must be an under-tight cut that crosses W, and the Gomory-Hu tree must contain 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 cuts may be crossing each other arbitrarily. Standard uncrossing techniques work sequentially to produce a laminar family. But how do we find it in NC? We give a divide-and-conquer-based procedure that finds the top level sets of one such laminar family; these top level sets can be shrunk simultaneously (section V-B). The procedure works as follows: We partition the family of tight odd cuts into two (almost) equal subfamilies, recursively "uncross" each subfamily to obtain its top level sets, and then merge these two into one family of top level sets. Clearly the last step is the crux of the procedure. The key to it lies in observing that two families of top level sets 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, followed by recursively finding a perfect matching in each of the shrunk sets (after removing its matched vertex). This algorithm has polylog depth; however, it does not run in polylog time because of the following inherent sequentiality: matchings in the shrunk sets have to be found *after* finding a matching in the shrunk graph.

We next observe that if we could find a *balanced* tight odd cut, we would be done by a simple divide-and-conquer strategy: match any edge in the cut and find matchings in the two sides of the cut recursively, in parallel. Notice that with this scheme, even though matchings in the two sides can be found only after finding a matching in the shrunk graph, the latter can be done in constant time (and without any recursive calls). Hence, the time taken is indeed polylog. However, finding a balanced tight odd cut is not straightforward. It

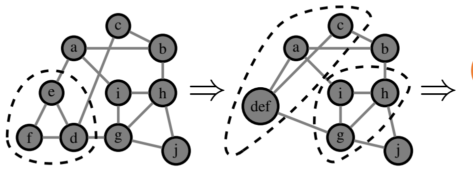


Figure 4. Shrinking repeatedly yields a balanced viable set.

involves iteratively shrinking the top level sets found, finding even walks and moving to the minimum weight face in the shrunk graph, etc. (section IV-B). This is illustrated in fig. 4. We show that $O(\log n)$ such iterations suffice for finding a balanced tight odd cut.

III. 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.

A. 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.Let A be the symmetric adjacency matrix of G Obtain matrix T from A by replacing for each edge $(i, j) \in E$, its two entries by x_{ij} and $-x_{ij}$. T is called the *Tutte matrix* for G. Its determinant is nonzero 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.

Note that for each edge edge there are two ways of completing the corresponding entry in T. These multiple ways of getting a Tutte matrix are in bijection with orientations of the graph.

A key fact underlying our algorithm is that when G is planar, there is an orientation which makes all of the nonzero terms in the determinant of T appear with the positive sign, and this orientation can be found in NC [2]. This yields an NC algorithm to solve the decision problem of checking whether G has a perfect matching.

B. The perfect matching polytope and tight odd 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 [3].

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

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, $\mathrm{PM}(G)$ is the convex hull of indicator vectors of all perfect matchings:

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

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

A key step needed by our algorithm is finding a point on the face $\mathrm{PM}(G, w)$ in NC. This is accomplished by computing a Pfaffian orientation for G and evaluating the Tutte matrix for appropriate substitutions of the variables. The point we find will be exactly the average of the vertices on the face $\mathrm{PM}(G, w)$. We denote this average by $\mathrm{avg}(\mathrm{PM}(G, w))$:

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

Lemma 1. *Given a planar graph $G = (V, E)$ and an integral weight vector $w \in \mathbb{Z}^E$ represented in unary, there is an* NC *algorithm which returns* $\mathrm{avg}(\mathrm{PM}(G, w))$.

We prove lemma 1 in section VI-A.

In general, a face of $\mathrm{PM}(G)$ 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 either case, we can *uncross* the initial sets and replace them by new ones that 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 laminar family. For our purposes, we only need to work with the maximal sets in the laminar family. We define a notion of uncrossing for top-level sets and show how they give us the space of equality constraints, by defining things appropriately.

C. 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 2 ([14])**.** *There is an* NC *algorithm for finding some maximal independent set in an input graph $G = (V, E)$.*

Mahajan and Varadarajan [15] used lemma 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.

Definition 1. 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

```
PerfectMatching($G = (V, E)$)
if $|V| = 0$ then
    return $\emptyset$.
else
    Find a viable set $S$ with $|S|/|V| \in [c_1, 1 - c_1]$.
    Let $w \leftarrow \mathbb{1}_{\delta(S)}$.
    Let $x \leftarrow \text{avg}(\text{PM}(G, w))$.
    Select an arbitrary edge $e \in \delta(S)$ 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
        $M_1 \leftarrow$ PerfectMatching($G_1$).
        $M_2 \leftarrow$ PerfectMatching($G_2$).
return $M_1 \cup M_2 \cup \{e\}$
```

Figure 5. Divide-and-conquer algorithm for finding a perfect matching.

P be a path, edge-disjoint from C_1, C_2, 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 closed 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. We use lemma 2 to derive the following. We prove lemma 3 in section VI-B.

Lemma 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 we can find $\Omega(|E|)$ edge-disjoint even walks in G by an NC algorithm.*

IV. Main Algorithm

A. Divide-and-conquer procedure

In this section we will describe the algorithm we use to prove theorem 1. W.l.o.g. assume that the input graph has a perfect matching. We can easily check whether a perfect matching exists first, by counting the number of perfect matchings in NC; see section III-A.

We use a divide-and-conquer approach. The pseudocode is given in fig. 5. 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 $\delta(S)$.

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 2. 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 $c_1 > 0$. Throughout the paper 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.

Proof of theorem 1: Since the set S found by fig. 5 is feasible, 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 minimum weight perfect matchings 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 minimum weight perfect matching $N \ni e$. 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 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 1, we can compute the point x 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 recursive level. Since recursive calls are executed in parallel, this shows that the entire algorithm runs in polylog time. ■

All that remains is finding a balanced viable set by an NC algorithm. This is done in section IV-B.

B. Finding a balanced viable set

In this section we describe how to find a balanced viable set S in a graph $G = (V, E)$ in NC. Notice that a single vertex is, by definition, a viable set, but is not balanced unless $|V| \leq \frac{1}{c_1}$. So w.l.o.g. we can assume $|V| > \frac{1}{c_1}$.

The main idea behind our algorithm is the following: Suppose that we reduce the size of the graph G by either removing edges not participating in perfect matchings from it, or shrinking tight odd sets (both w.r.t. some weight vector w). Any vertex in the shrunk graph corresponds to an odd set in the original graph G. This odd set is always viable. So if we manage to reduce the size of the shrunk graph enough so that it contains at most $1/c_1$ vertices, then the largest of the viable sets we get this way would have size at least $c_1|V|$. By being careful when we remove edges or shrink pieces, we can also make sure the size is not larger than

```
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($G, f$).
Find $v \in V$ for which $|f^{-1}(v)| \geq c_1|V_0|$.
return $f^{-1}(v)$.
```

Figure 6. Finding a balanced viable set.

$(1 - c_1)|V|$; so the end result is a balanced viable set. See fig. 4 for a depiction.

The pseudocode is given in fig. 6. Throughout the algorithm we maintain a mapping f from the original vertices to the vertices of the current graph. We iteratively reduce the size of the graph by removing edges and/or contracting odd sets of vertices until one vertex contains c_1 fraction of the original vertices. We then return the preimage of this vertex.

Lemma 4. *The loop in fig. 6 finishes as soon as* $|V| \leq \frac{1}{c_1}$.

We further maintain the invariant that our graph G is at all times planar, and has a perfect matching. This invariant is satisfied, because we restrict ourselves to manipulate G in only one of the two following ways: 1) We either remove an edge e from G, where e does not participate in any minimum weight perfect matching for some weight vector w. 2) Or we shrink a set of vertices S that is a tight odd set w.r.t. some point x in the matching polytope.

Lemma 5. *If a graph G is planar and has a perfect matching, after the removal of an edge or shrinking of a set as described above, it continues to be planar and have a perfect matching.*

Any viable set in the resulting graph is a viable set in the original graph at any point, because any perfect matching in G can be extended to a perfect matching in G_0. So we can return $f^{-1}(v)$ at any point if it is a balanced set.

The main loop in fig. 6 has two steps, Preprocess and Reduce. Although not explicitly stated in the pseudocode, at any point, 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 6 to section IV-C.

Lemma 6. *The procedure* Preprocess *either finds a balanced viable set or after it returns we have: 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 fig. 7. Assuming lemma 6, 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

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

Figure 7. Reducing the graph size by shrinking and/or removing edges.

get shrunk. The conditions satisfied after the preprocessing step, lemma 6, ensure that we can apply lemma 3 and find $\Omega(|E|)$ edge-disjoint even walks, as we do in fig. 7.

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 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 V. 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 V.

Now we can prove the following.

Lemma 7. *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, so by setting c_1 small enough we can assume that $|E|$ is larger than a desired constant. ■

By lemma 7, our measure of progress, $|E|$ 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 IV-C, and DisjointOddSets in section V.

C. Preprocessing

Here we describe the procedure Preprocess and prove lemma 6. The pseudocode is given in fig. 8. 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

```
PREPROCESS ($G = (V, E)$, $f : V_0 \to V$)
Let $x \leftarrow \text{avg}(\text{PM}(G))$.
for $e \in E$ with $x_e = 0$ in parallel do
    Remove $e$ from $G$.
Let $G, f \leftarrow$ MAKECONNECTED$(G, f)$.
while $|\{v \in V \mid \deg(v) = 2\}| > |V|/2$ do
    Let $G, f \leftarrow$ SHRINKDEGREETWOS$(G, f)$.
```

Figure 8. The preprocessing step.

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 any of the two vertices.

After having a connected graph with no vertices of degree 1, and no more than half of the vertices of 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 $C_1, \dots, C_k$ 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 $S_i = \{v\} \cup C_1 \cup \dots C_i$ is a tight odd set, because $\{v\}$ is a tight odd set and adding 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 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. 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 polylog 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 $(v_1, \dots, v_k)$. 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 $i = 1, \dots, k$ but not for $i = 0, k + 1$.

We claim that for any even i, the set $S_i = \{v_0, v_1, \dots, v_i\}$ 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

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

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

Having all ingredients, we finish the proof of lemma 6.

Proof of lemma 6: It is easy to see that the point $x \in \text{PM}(G)$ remains a valid point throughout, i.e., it remains in the 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 it remains connected until the end. Since x remains a valid point in the matching polytope until the end, every edge at the end participates in a perfect matching. But by connectivity, this means that there are no degree-1 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. ■

V. 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 $W_1, \dots, W_m$ in G, the point $x = \text{avg}(\text{PM}(G, w))$, where w is the weight vector constructed in fig. 7. 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 8. *There is an* NC *algorithm* DISJOINTODDSETS, *that either finds a balanced viable set, or finds disjoint tight odd sets $S_1, \dots, S_l$ 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) For each walk W_i, we find a tight odd set blocking it. 2) The resulting tight odd sets might cross each other arbitrarily. We uncross them to obtain $S_1, \dots, S_l$, being careful not to produce sets with $|f^{-1}(S_i)| \geq c_1|V_0|$.

In section V-A we describe the procedure for finding a tight odd set blocking an even walk. Then in section V-B, we describe how to uncross these to get disjoint odd sets.

A. *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 matching polytope. This must be the tight odd set we were after.

Recall that an even walk is either a simple even 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. 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 of W as $\chi_W = -\mathbb{1}_{e_1} + \mathbb{1}_{e_2} - \mathbb{1}_{e_3} + \ldots + \mathbb{1}_{e_k} = \sum_i (-1)^i \mathbb{1}_{e_i}$.

Note that for a weight vector w, we have $\langle w, \chi_W\rangle = -w_{e_1} + w_{e_2} - w_{e_3} + \ldots + w_{e_k}$. In particular for the weight vector chosen in fig. 7, we have $\langle w, \chi_W\rangle < 0$.

We next define *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 $\operatorname{avg}(\mathrm{PM}(G, w))$, i.e., we have $x = \frac{\mathbb{1}_{M_1} + \ldots + \mathbb{1}_{M_m}}{m}$, where $M_1, \ldots, M_m$ are the minimum weight perfect matchings of G.

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

Lemma 9. *Let $x = \operatorname{avg}(\mathrm{PM}(G, w))$ 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 $y_e \geq 0$.*

Lemma 9 almost ensures that the point y is inside the matching polytope $\mathrm{PM}(G)$ if the starting point x was in $\mathrm{PM}(G)$. The only way that y cannot be in $\mathrm{PM}(G)$ 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 10. *Suppose w is a weight vector, $x = \operatorname{avg}(\mathrm{PM}_w(G))$, W is a walk that satisfies the conditions of lemma 9, 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: $\operatorname{argmin}_{S\subset V, |S| \text{ is odd}} \langle \mathbb{1}_{\delta(S)}, y\rangle$.*

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 10, we get that we can find a tight odd set blocking each of our even walks.

Lemma 11. *There is an* NC *algorithm that given a weight planar graph G, outputs the minimum odd cut of G.*

We will prove lemma 11 in section VI-C.

B. *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 at the same time. We make progress from shrinking by making sure that each even walk has an edge inside at least one shrunk set, so that the number of edges gets reduced by at least the number of walks.

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

Definition 3. 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:

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

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 12. *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 $e \in E(S_i)$.*

Next we 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 and follows from submodularity of the cut function.

Lemma 13. *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 $\mathbb{1}_{\delta(S_1)}+\mathbb{1}_{\delta(S_2)}=\mathbb{1}_{\delta(S_1-S_2)}+\mathbb{1}_{\delta(S_2-S_1)}$.*

Now we use lemma 13 to prove the claim that tight odd sets can be uncrossed without shrinking $\Lambda(S_1)+\Lambda(S_2)$.

Lemma 14. *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 13 that $S_1\cup S_2$ is a tight odd set. It is trivial from definition 3 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 13 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 13 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 13 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 13, $(S_2-S_1)-T$ 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 $S_1,\dots,S_m$, repeated applications of lemma 14 allow us to uncross them, i.e., replace them by pairwise disjoint tight odd sets $S'_1,\dots,S'_{m'}$ such that $\Lambda(S_1,\dots,S_m)\subseteq\Lambda(S'_1,\dots,S'_{m'})$. However, naively lemma 14 would result in a sequential algorithm which is not in NC. Next we show how we can do uncrossing in NC.

We will use a divide-and-conquer approach to uncross a given list of tight odd sets $S_1,\dots,S_m$. The high-level description of our procedure, UNCROSS, is given in fig. 9. 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 ($S_1,\dots,S_m$)
if $m$=1 then
    return $S_1$
else
    in parallel do
        $R_1,\dots,R_p \leftarrow$ UNCROSS ($S_1,\dots,S_{\lceil m/2\rceil}$)
        $C_1,\dots,C_q \leftarrow$ UNCROSS ($S_{\lceil m/2\rceil+1},\dots,S_m$)
    return MERGEUNCROSS ($R_1,\dots,R_p,C_1,\dots,C_q$)
```

Figure 9. Divide-and-conquer algorithm for uncrossing tight odd sets

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 $S_1,\dots,S_m$. The difference between MERGEUNCROSS and UNCROSS is that the input sets to MERGEUNCROSS satisfy certain properties highlighted below.

Lemma 15. *Suppose that $\{S_1,\dots,S_m\}=\{R_1,\dots,R_p,C_1,\dots,C_q\}$, where $m=p+q$ and $R_1,\dots,R_p$ are pairwise disjoint tight odd sets and $C_1,\dots,C_q$ are also pairwise disjoint tight odd sets. Then $S_1,\dots,S_m$ have no 3-wise intersections. Furthermore, the intersection graph of $S_1,\dots,S_m$, where two S_i's are connected if they have a nonempty intersection, is bipartite.*

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

Definition 4. For tight odd sets $S_1,\dots,S_m$ satisfying the conditions of lemma 15, define the intersection parity graph $H=(V_H,E_H)$, as follows: Let V_H, the nodes of H, be $S_1,\dots,S_m$ 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 15 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 14, 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 otain $S_1 \cup \dots \cup S_m$ 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 $|S_1 \cup \dots \cup S_m|$ could possibly be even; so it will never emerge out of this process, because lemma 14 only produces *odd* tight sets. For simplicity of notation we use $\cup H$ to denote $S_1 \cup \dots \cup S_m$.

Surprisingly, these two are really the only barriers.

Lemma 16. *Assume that $H = H(S_1, \dots, S_m)$ is connected and that $|H| \overset{2}{\equiv} 1$. Then $\cup H = S_1 \cup \dots \cup S_m$ is a tight odd set, and $\Lambda(S_1, \dots, S_m) \subseteq \Lambda(\cup H)$.*

Lemma 16 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 14. 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.

Lemma 17. *Assume that $H = H(S_1, \dots, S_m)$ is connected and $|H| \overset{2}{\equiv} 0$. There are two induced subgraphs of H, which are odd and connected, and which together cover every node. Furthermore, these two subgraphs can be found in* NC.

Now, armed with lemmas 16 and 17, 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 16 or lemma 17 to each connected component of H. To avoid creating sets S with $|f^{-1}(S)| \geq c_1|V_0|$, we always pass our sets through the procedure CHECKBALANCEDVIABLE, which can potentially find a balanced viable set and end.

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 $S'_1, \dots, S'_k$ so that for any i, the induced subgraph on $U_i = \{S'_1, \dots, S'_i\}$ 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 16). Otherwise by lemma 17, we can find two subsets of U_j whose union covers U_j and are odd. We simply return the subset with the larger value of

```
MERGEUNCROSS ($S_1, \dots, S_m$)
for $i = 1 \dots m$ in parallel do
    $S_i \leftarrow S_i - \bigcup_{j<i, |S_i \cap S_j| \text{ even}} S_j$
$H \leftarrow H(S_1, \dots, S_m)$
$H_1, \dots, H_k \leftarrow$CONNECTEDCOMPONENTS ($H$)
$\mathcal{F} \leftarrow \emptyset$
for $i = 1 \dots k$ in parallel do
    if $|H_i| = |V_{H_i}| + |E_{H_i}|$ is odd then
        CHECKBALANCEDVIABLE ($H_i$).
        Add $\cup H_i$ to $\mathcal{F}$.
    else
        Let $H'_i, H''_i$ be the two induced subgraphs
          promised by lemma 17.
        CHECKBALANCEDVIABLE ($H'_i$).
        CHECKBALANCEDVIABLE ($H''_i$).
        Add $\cup H'_i$ to $\mathcal{F}$.
        Add $\cup H''_i - \cup H'_i$ to $\mathcal{F}$.
return $\mathcal{F}$
```

Figure 10. Algorithm for uncrossing partially uncrossed sets

$|f^{-1}(\cdot)|$ as the balanced viable set.

All together we get the following result:

Theorem 2. *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 $|f^{-1}(S'_i)| < c_1|V_0|$.*

VI. OTHER ALGORITHMIC INGREDIENTS

In this section we describe the remaining algorithmic ingredients we used in sections IV and V.

A. Finding a point on a face of the matching polytope

In this section, we prove lemma 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 $\mathrm{PM}(G, w)$.

Proof of lemma 1: 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 $x_e = \frac{\#G_w^e}{\#G_w}$. Observe that if $M_1, \dots, M_m$ are all the minimum weight perfect matchings in G, then $x = \frac{\mathbb{1}_{M_1} + \dots + \mathbb{1}_{M_m}}{m} = \mathrm{avg}(\mathrm{PM}(G, w))$.

To compute $\#G_w^e$ and $\#G_w$, we find a Pfaffian orientation of G in NC. Then we replace the entries of the Tutte matrix by $\pm y^{w_e}$, where the sign is chosen according to the orientation and w_e is the weight of the corresponding edge. The exponents of the entries of this matrix are polynomially bounded and hence its determinant can be computed in NC [1]. Consider the lowest degree term in the determinant; 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 $(\#G_w)^2$. By removing each edge e and repeating the above procedure we can get $\#G_w^e$. ■

B. Finding linearly many edge-disjoint even walks

Here we prove lemma 3 by showing how to find $\Omega(|E|)$ many edge-disjoint even walks in $G = (V, E)$ in NC. By assumption, G is a connected planar graph with no 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 18 (Adapted from [15]). *There is an* NC *algorithm that returns $|E|/288$ edge-disjoint planar faces of a graph satisfying the assumptions of lemma 3.*

If at least half of the faces found by lemma 18 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 19. *Given an even number f of edge-disjoint odd faces in a planar graph, we can find in* NC*, $f^2/16|E|$ edge-disjoint even walks, each formed by joining two given faces.*

Sketch of 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 20 we will prove that these tokens can be paired up by edge-disjoint paths from the tree in NC.

This pairing needs to be modified to form the even walks. For more details refer to the full version of this paper. ■

We now describe the missing part from the above proof.

Lemma 20. *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.*

We now have the ingredients to prove lemma 3.

Proof of lemma 3: We first find $|E|/288$ edge-disjoint faces by invoking lemma 18. If at least half of these faces are even, we return this half. Otherwise we invoke lemma 19. We have $|E|/576$ odd faces, any by possibly dropping one of them we can supply an even number $f \geq |E|/576 - 1$ of odd faces to the algorithm described by lemma 19 and obtain $(|E|/576 - 1)^2/16|E| = \Omega(|E|)$ edge-disjoint even walks. This finishes the proof. ■

C. 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 $w : E \to \mathbb{R}_{\geq 0}$ 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 [9]. For each pair of vertices $u, v \in V$, let $f(u, v)$ denote the weight of a minimum u-v cut in G.

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: 1) For each edge $(S_i, S_j) \in T$, $\exists\ u \in S_i,\ v \in S_j$ such that $w'(S_i, S_j) = f(u, v)$. 2) The removal of (S_i, S_j) from T disconnects T, and splits the partitions which naturally defines a cut, $(S, \overline{S})$ in G. This must be a minimum u-v cut.

In each iteration, the sequential algorithm refines the tree by *splitting* one of the partitions into two. It picks a partition, S_i, having at least two vertices, u, v. Let $T_1, \ldots T_l$ be the subtrees of T incident at node S_i. Each subtree T_j is *shrunk* into a single vertex t_j. This gives a graph G' on $S_i \cup \{t_1, \ldots, t_l\}$. 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''. Replace S_i by these two sets and connect them by an edge of weight $f(u, v)$. 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 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 $v \in S_i,\ v \neq r$, $|S'_v| \leq |S_i|/2$.

Lemma 21. *For any partition S_i, a central vertex r exists.*

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 22. *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 3. *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 [18] to finish the proof of lemma 11.

Proof of lemma 11: 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 VI.1. An alternative way of finding a minimum odd cut S is to use the Pickard-Queyranne structure of minimum s-t cuts [19]. However, that method is cumbersome to describe.

Acknowledgements

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

Part of this work was done while the first author was visiting the Simons Institute for the Theory of Computing. It 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.

Remark VI.2. Recent work [5] has resolved the problem for the more general $K_{3,3}$-free graphs. Eppstein and Vazirani [5] go further to give NC algorithms for finding a perfect matching, a minimum weight perfect matching if the weights are polynomially bounded, and an s-t min-cut in one-crossing-minor-free graphs. Also recent work of Sankowski [20] has given an alternative approach to finding perfect matchings in planar graphs in NC, as well as other related problems.

References

[1] 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.

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

[3] 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.

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

[5] David Eppstein and Vijay V Vazirani. "NC Algorithms for Perfect Matching and Maximum Flow in One-Crossing-Minor-Free Graphs". In: *arXiv preprint arXiv:1802.00084* (2018).

[6] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. "Bipartite perfect matching is in quasi-NC". In: *STOC*. 2016, pp. 754–763.

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

[8] 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).

[9] 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.

[10] Richard M Karp, Eli Upfal, and Avi Wigderson. "Are search and decision programs computationally equivalent?" In: *STOC*. 1985, pp. 464–475.

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

[12] Pieter Kasteleyn. "Graph theory and crystal physics". In: *Graph Theory and Theoretical Physics* (1967), pp. 43–110.

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

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

[15] Meena Mahajan and Kasturi R Varadarajan. "A new NC-algorithm for finding a perfect matching in bipartite planar and small genus graphs". In: *STOC*. 2000, pp. 351–357.

[16] Gary L Miller and Joseph Naor. "Flow in planar graphs with multiple sources and sinks". In: *FOCS*. 1989, pp. 112–117.

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

[18] 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.

[19] 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.

[20] Piotr Sankowski. "NC Algorithms for Weighted Planar Perfect Matching and Related Problems". In: *ICALP*. Vol. 107. 2018.

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

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

[23] Vijay V Vazirani. "NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems". In: *Information and Computation* 80.2 (1989), pp. 152–164.