Seeking a vertex of the planar matching polytope in NC

Raghav Kulkarni[1]* and Meena Mahajan[2]

[1] Chennai Mathematical Institute, 92, G.N.Chetty Road, T. Nagar, Chennai 600 017, India. raghav@cmi.ac.in

[2] The Institute of Mathematical Sciences, Chennai 600 113, India. meena@imsc.res.in

Abstract. For planar graphs, counting the number of perfect matchings (and hence determining whether there exists a perfect matching) can be done in NC [4, 10]. For planar bipartite graphs, finding a perfect matching when one exists can also be done in NC [8, 7]. However in general planar graphs (when the bipartite condition is removed), no NC algorithm for constructing a perfect matching is known.

We address a relaxation of this problem. We consider the fractional matching polytope $\mathcal{P}(G)$ of a planar graph G. Each vertex of this polytope is either a perfect matching, or a half-integral solution: an assignment of weights from the set $\{0, 1/2, 1\}$ to each edge of G so that the weights of edges incident on each vertex of G add up to 1 [6]. We show that a vertex of this polytope can be found in NC, provided G has at least one perfect matching to begin with. If, furthermore, the graph is bipartite, then all vertices are integral, and thus our procedure actually finds a perfect matching without explicitly exploiting the bipartiteness of G.

1 Introduction

The perfect matching problem is of fundamental interest to combinatorists, algorithmists and complexity-theorists for a variety of reasons. In particular, the problems of deciding if a graph has a perfect matching, and finding such a matching if one exists, have received considerable attention in the field of parallel algorithms. Both these problems are in randomized NC [5, 2, 9] but are not known to be in deterministic NC. (NC is the class of problems with parallel algorithms running in polylogarithmic time using polynomially many processors.) For special classes of graphs, however, there are deterministic NC algorithms.

In this work, we focus on planar graphs. These graphs are special for the following reason: for planar graphs, we can count the number of perfect matchings in NC ([4, 10], see also [3]), but we do not yet know how to find one, if one exists. This counters our intuition that decision and search versions of problems

* Part of this work was done when this author was visiting the Institute of Mathematical Sciences, Chennai on a summer student programme.

are easier than their counting versions. In fact, for the perfect matching problem itself, while decision and search are both in P, counting is #P-hard and hence is believed to be much harder. But for planar graphs the situation is curiously inverted.

We consider a relaxation of the search version and show that it admits a deterministic NC algorithm. The problem we consider is the following: For a graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, consider the m-dimensional space $\mathcal{Q}^m$. A point $\langle x_1, x_2, \ldots, x_m \rangle$ in this space can be viewed as an assignment of weight x_i to the ith edge of G. A perfect matching in G is such an assignment (matched edges have weight 1, other edges have weight 0) and hence is a point in this space.

Consider the polytope $\mathcal{P}(G)$ defined by the following equations.

$$x_e \geq 0 \quad \forall\, e \in E \tag{1}$$

$$\sum_{e \text{ incident on } v} x_e = 1 \quad \forall\, v \in V \tag{2}$$

Clearly, every perfect matching of G (i.e. the corresponding point in $\mathcal{Q}^m$) lies in $\mathcal{P}(G)$. Standard matching theory (see for instance [6]) tells us that every perfect matching of G is a vertex of $\mathcal{P}(G)$. In fact, if we denote by $\mathcal{M}(G)$ the convex hull of all perfect matchings, then $\mathcal{M}(G)$ is a facet of $\mathcal{P}(G)$. In the case of bipartite graphs, the perfect matchings are in fact the only vertices of $\mathcal{P}(G)$; $\mathcal{P}(G) = \mathcal{M}(G)$. Thus for a bipartite graph it suffices to find a vertex of $\mathcal{P}(G)$ to get a perfect matching. Furthermore, for general graphs, all vertices of $\mathcal{P}(G)$ are always half-integral (in the set $\{0, 1/2, 1\}^m$). For any vertex w of $\mathcal{P}(G)$, if we pick those edges of G having non-zero weight in w, we get a subgraph which is a disjoint union of a partial matching and some odd cycles.

For instance, Figure 1 shows a graph where $\mathcal{M}(G)$ is empty, while $\mathcal{P}(G)$ has one point shown by the weighted graph alongside. Figure 2 shows a graph where $\mathcal{M}(G)$ is non-empty; furthermore, the weighted graph in Figure 2(b) is a vertex of $\mathcal{P}(G)$ not in $\mathcal{M}(G)$.

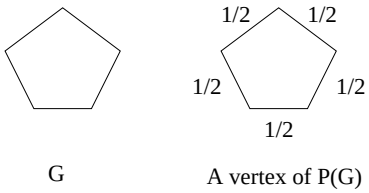

Fig. 1. An Example Graph with empty $\mathcal{M}(G)$, and a point in $\mathcal{P}(G)$

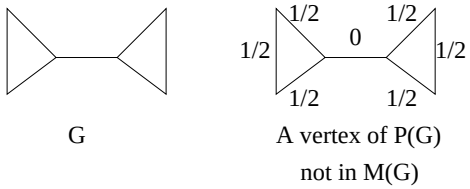

Fig. 2. An Example Graph with non-empty $\mathcal{M}(G)$, and a point in $\mathcal{P}(G) - \mathcal{M}(G)$

We show in this paper that finding a vertex of $\mathcal{P}(G)$ is in NC when G is a planar graph with at least one perfect matching. If, furthermore, the graph is

bipartite, then all vertices are integral, and thus our procedure actually finds a perfect matching, without explicitly exploiting the bipartiteness of G.

Our approach is as follows. In [7], an NC procedure is described to find a point p inside $\mathcal{P}(G)$, using the fact that counting the number of perfect matchings in a planar graph is in NC. Then, exploiting the planarity and bipartiteness of the graph, an NC procedure is described to find a vertex of $\mathcal{P}(G)$ by constructing a path, starting from p, that never leaves $\mathcal{P}(G)$ and that makes measurable progress towards a vertex. We extend the same approach for general planar graphs. In fact, we start at the same point p found in [7]. Then, using only planarity of G, we construct a path, starting from p and moving towards a vertex of $\mathcal{P}(G)$. Our main contribution, thus, is circumventing the bipartite restriction. We prove that our algorithm indeed reaches a vertex of $\mathcal{P}(G)$ and can be implemented in NC.

This paper is organised as follows. In Section 2 we briefly describe the Mahajan-Varadarajan algorithm [7]. In Section 3 we describe our generalisation of their algorithm, and in the subsequent two sections we prove the correctness and the NC implementation of our method.

2 Reviewing the Mahajan-Varadarajan algorithm

The algorithm of Mahajan & Varadarajan [7] is quite elegant and straightforward and is summarised below.

For planar graphs, the number of perfect matchings can be found in NC [4, 10]. Using this procedure as a subroutine, and viewing each perfect matching as a point in $\mathcal{Q}^m$, obtain the point p which is the average of all perfect matchings; clearly, this point is inside $\mathcal{M}(G)$. Now construct a path from p to some vertex of $\mathcal{M}(G)$, always staying inside $\mathcal{M}(G)$.

The Algorithm

- Find a point $p \in \mathcal{M}(G)$: $\forall e \in E$, assign $x_e = (\#G - \#G_e)/\#G$, where $\#G$ denotes the number of perfect matchings in G and G_e denotes the graph obtained by deleting edge e from G. Delete edges of 0 weight.
- While the graph is not acyclic,
  - Get cf edge-disjoint cycles (where f is the number of faces in a planar embedding of G) using the fact that the number of edges is less than three times the number of vertices. (Here c is a constant $c = 1/24$. We consider that subgraph of the dual containing faces with fewer than 12 bounding edges. A maximal independent set in this graph is sufficiently large, and gives the desired set of edge-disjoint cycles.) Since the graph is bipartite, all the cycles are of even length.
  - Destroy each of these cycles by removing the smallest weight edge in it. To stay inside the polytope, manipulate the weights of the remaining edges as follows: in a cycle C, if e is the smallest weight edge with weight x_e, then add x_e to the weight of edges at odd distances from e

and subtract x_e from the weights of edges at even distances from e. Thus the edge e itself gets weight 0. Delete all 0-weight edges.

– Finally we get an acyclic graph and we are still inside the polytope. It's easy to check that any acyclic graph inside $\mathcal{M}(G)$ must be a perfect matching (integral).

Since we are destroying cf faces each time, within $\log f$ iterations of the while loop we will end up with an acyclic graph. Hence, for bipartite planar graphs finding a perfect matching is in NC.

3 Finding a half-integral solution in a planar graph in NC

The following result is a partial generalization of the previous result.

Theorem 1. *For planar graphs, a vertex of the fractional matching polytope $\mathcal{P}(G)$ (i.e. a half-integral solution to the equations defining $\mathcal{P}(G)$, with no even cycles) can be found in NC, provided that the perfect matching polytope $\mathcal{M}(G)$ is non-empty.*

Our starting point is the same point p computed in the previous section; namely, the arithmetic mean of all perfect matchings of G. Starting from p, we attempt to move towards a vertex. The basic strategy is to find a large set S of edge-disjoint faces. Each such face contains a simple cycle, which we try to destroy. Difficulties arise if the edges bounding the faces in S do not contain even length simple cycles, since the method of the previous section works only for even cycles. We describe mechanisms to be used successively in such cases.

We first describe some basic building blocks, and then describe how to put them together.

Building block 1: Simplify, or *Standardise*, the graph G.

Let G be the current graph, let $x : E \longrightarrow \mathcal{Q}$ be the current assignment of weights to edges, and let y be the partial assignment finalised so far. The final assignment is $y : E \longrightarrow \{0, 1/2, 1\}$.

Step 1.1 For each $e = (u, v) \in E(G)$, if $x_e = 0$, then set $y_e = 0$ and delete e from G.

Step 1.2 For each $e = (u, v) \in E(G)$, if $x_e = 1$, then set $y_e = 1$ and delete u and v from G.
(This step ensures that all vertices of G have degree at least 2.)

Step 1.3 Obtain connected components of G.
If a component is an odd cycle C, then every edge on C must have weight $1/2$. For each $e \in C$, set $y_e = 1/2$. Delete all the edges and vertices of C from G.
If a component is an even cycle C, then for some $0 < a < 1$, the edges on C alternately have weights a and $1 - a$. For each $e \in C$, if $x_e = a$ then set $y_e = 1$ and if $x_e = 1 - a$ then set $y_e = 0$. Delete all the edges and vertices of C from G.

Step 1.4 Let E' be the set of edges touching a vertex of degree 2 in G. Consider the subgraph of G induced by E'; this is a disjoint collection of paths. Collapse each such even path to a path of length 2 and each such odd path to a path of length 1, reassigning weights as shown in Figure 3. Again, we stay within the polytope of the new graph, and from any assignment here, a point in $\mathcal{P}(G)$ can be recovered in a straightforward way.
This step ensures that no degree 2 vertex has a degree 2 neighbour.

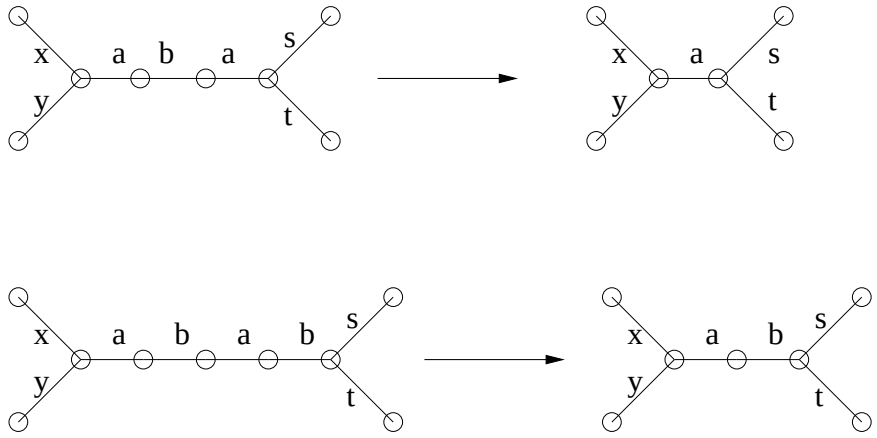

Fig. 3. Transformation assuring that no two consecutive vertices are of degree 2

Step 1.5 For each $v \in V(G)$, if v has degree more than 3, then introduce some new vertices and edges, rearrange the edges touching v, and assign weights as shown in Figure 4. This assignment in the new graph is in the corresponding polytope of the new graph, and from any assignment here, a point in $\mathcal{P}(G)$ can be recovered in a straightforward way. (This gadget construction was in fact first used in [1].)
This step ensures that all vertices have degree 2 or 3.

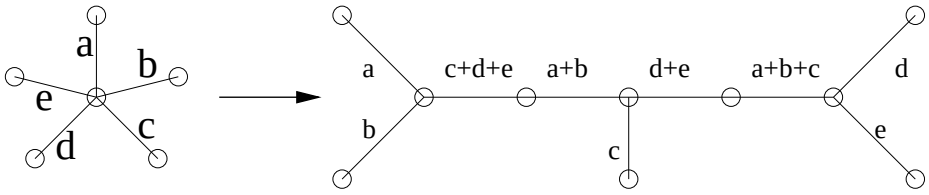

Fig. 4. Transformation to remove vertices of degree greater than 3

Note that Steps 1.4 and 1.5 above change the underlying graph. To recover the point in $\mathcal{P}(G)$ from a point in the new graph's polytope, we can initially allocate one processor per edge. This processor will keep track of which edge in the modified graph dictates the assignment to this edge. Whenever any transformation is done on the graph, these processors update their respective data, so that recovery at the end is possible.

We will call a graph on which the transformations of building block 1 have been done a *standardised graph*.

Building Block 2: Process an even cycle. This is as in [7], and is described in Section 2.

Building Block 3: Process an odd cycle connected to itself by a path. Let C be such an odd cycle, with path P connecting C to itself. We first consider the case when P is a single edge, i.e. a chord. The chord (u, v) cuts the cycle into paths P_1, P_2. Let C_i denote the cycle formed by P_i along with the chord (u, v). Exactly one of C_1, C_2 is even; process it as in Building Block 2.

If instead of a chord, there is some path $P_{u,v}$ connecting u and v on C, the same reasoning holds and so this step can still be performed.

Building Block 4: Process a pair of edge-disjoint odd cycles connected by a path.

Let C_1 and C_2 be the odd cycles and P the path connecting them. Note that if G is standardised, then P cannot be of length 0. Let P connect to C_1 at u and to C_2 at v. Then the traversal of C_1 beginning at u, followed by path P going from u to v, then the traversal of C_2 beginning at v, followed by the path P going from v to u, is a closed walk of even length. We make two copies of P, one for each direction of traversal. For edge e on P, assign weight $x_e/2$ to each copy. Now treating the two copies as separate, we have an even cycle which can be processed according to building Block 2. For each edge $e \in P$, its two copies are at even distance from each other, so either both increase or both decrease in weight. It can be seen that after this adjustment, the weights of the copies put together is still between 0 and 1.

This step is illustrated in Figure 5. The edge on the path has weight a is split into two copies with weight $a/2$ each. The dotted edge is the minimum weight edge; thus $w \leq a/2$.

The Algorithm The idea is to repeatedly identify large sets of edge-disjoint faces, and then manipulate them, in the process destroying them. The faces are identified as in [7], and a simple cycle is extracted from each face. Even cycles are processed using Building Block 2. By the building blocks 3 and 4, odd cycles can also be processed provided we identify paths connecting the cycles to themselves or other cycles. However, to achieve polylogarithmic time, we need to process several odd cycles simultaneously, and this requires that the odd cycles and the connecting paths be edge-disjoint.

We use the following definition: A path P is said to be a *3-bounded* path if the number of internal vertices of P with degree 3 is at most 1. Note that in a standardised graph, a 3-bounded path can have at most 3 internal vertices.

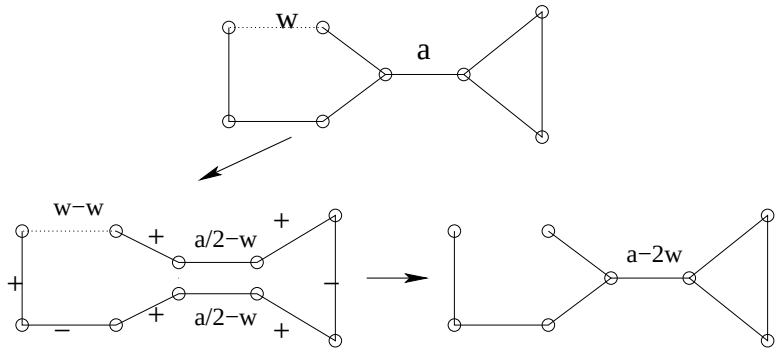

Fig. 5. Manipulating a closed walk of even length

The algorithm can be described as follows:

1. Find a point $p \in \mathcal{M}(G)$ and initialise x_e accordingly.
2. Standardise G (Building block 1; this builds a partial solution in y).
3. While G is not empty, repeat the following steps, working in parallel on each connected component of G:
   (a) Find a collection S of edge-disjoint faces in G, including at least $1/24$ of the faces from each component. Extract a simple cycle from the edges bounding each face of S, to obtain a collection of simple cycles T.
   (b) Process all even cycles (Building block 2). Remove these cycles from T. Re-standardise.
   (c) Define each surviving cycle in T to be a cluster. (At later stages, a cluster will be a set of vertices and the subgraph induced by this subset.)
   While T is non-empty, repeat the following steps:
      i. Construct an auxiliary graph H with clusters as vertices. H has an edge between clusters D_1 and D_2 if there is a 3-bounded path between some vertex of D_1 and some vertex of D_2 in G.
      ii. Process all clusters having a self-loop in H. (Building Block 3). Remove these clusters from T. Re-standardise.
      iii. Recompute H. In H, find a maximal matching. Each matched edge pairs two clusters, between which there is a 3-bounded path in G. In parallel, process all these pairs along with the connecting path using Building Block 4. Remove the processed clusters from T and re-standardise.
      iv. "Grow" each cluster: if D is the set of vertices in a cluster, then first add to D all degree-2 neighbours of D, then add to D all degree-3 neighbours of D. That is, $D = D \cup \{v \mid d(v) = 2 \wedge \exists u \in D, (u, v) \in E\}$, $D = D \cup \{v \mid d(v) = 3 \wedge \exists u \in D, (u, v) \in E\}$.
4. Return the edge weights stored in y.

4 Correctness

The correctness of the algorithm follows from the following series of lemmas:

Lemma 1. *The clusters and 3-bounded paths processed in Step 3(c) are vertex-disjoint.*

Proof. Each iteration of the while loop in Step 3(c) operates on different parts of G. We show that these parts are edge-disjoint, and in fact even vertex-disjoint. Clearly, this is true when we enter Step 3(c); the cycles are edge-disjoint by our choice in Step 3(a), and since G has maximum degree 3, no vertex can be on two edge-disjoint cycles. Changes happen only in steps 3(c)(ii) and 3(c)(iii); we analyze these separately.

Consider Step 3(c)(ii). If two clusters D_1 and D_2 have self-loops, then there are 3-bounded paths ρ_i from D_i to D_i, $i = 1, 2$. If these paths share a vertex v, it can only be an internal vertex of ρ_1 and ρ_2, since the clusters were vertex-disjoint before this step. In particular, v cannot be a degree-2 vertex. But since G is standardized, $\deg(v)$ is then 3, which does not allow it to participate in two such paths. So ρ_1 and ρ_2 must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing clusters with self-loops merely removes them from T; thus the clusters surviving after Step 3(c)(ii) continue to be vertex-disjoint.

Now consider Step 3(c)(iii). Suppose cluster D_1 is matched to D_2 via 3-bounded path ρ, D_3 to D_4 via 3-bounded path η. Note that $D_i \neq D_j$ for $i \neq j$, since we are considering a matching in H. Thus, by the same argument as above, the paths ρ and η must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing matched clusters removes them from T; the remaining clusters continue to be vertex-disjoint.

Since Step 3(c)(iii) considers a maximal matching, the clusters surviving are not only vertex-disjoint but also not connected to each other by any 3-bounded path. □

Lemma 2. *Each invocation of the while loop inside Step 3(c) terminates in finite time.*

Proof. To establish this statement, we will show that clusters which survive Steps 3(c)(ii) and 3(c)(iii) grow appreciably in size. In particular, they double in each iteration of the while loop. Clearly, clusters cannot double indefinitely while remaining vertex-disjoint, so the statement follows. In fact, our proof establishes that the while loop in Step 3(c) executes $O(\log n)$ times on each invocation.

Let G denote the graph at the beginning of Step 3(c). Consider a cluster at this point. Let D_0 be the set of vertices in the cluster. Consider the induced subgraph GD_0 on D_0. Notice that each such GD_0 contains exactly one cycle, which is an odd cycle extracted in Step 3(a).

We trace the progress of cluster D_0. Let D_i denote the cluster (or the associated vertex set; we use this notation interchangeably to mean both) resulting from D_0 after i iterations of the while loop of Step 3(c). If D_0 does not survive i iterations, then D_i is empty.

For any cluster D, let 3-size(D) denote the number of vertices in D whose degree in G is 3. Let D' denote the vertices of D whose degree in G is 3 but degree in D is 1.

We establish the following claim:

Claim. For a cluster D_0 surviving $i+1$ iterations of the while loop of Step 3(c), GD_i contains exactly one cycle, and furthermore,

$$|D_i'| \geq \lfloor 2^{i-1} \rfloor$$
$$\text{3-size}(D_i) \geq 2^i$$

Proof of Claim: As mentioned earlier, GD_0 contains exactly one cycle. Thus 3-size$|D_0'| = 0$ In fact, each GD_j, $j \leq i$ contains just this one cycle, because if any other cycle were present in GD_j, then a self-loop would be found at the $(j+1)$th stage and the cluster would have been processed and deleted from T in Step 3(c)(ii); it would not grow $(i+1)$ times.

It remains to establish the claims on the sizes of D_i and D_i'. We establish these claims explicitly for $i \leq 1$, and by induction for $i > 1$.

Consider $i = 0$. Clearly, 3-size$(D_0) \geq 2^0 = 1$, and $\lfloor 2^{-1} \rfloor = 0$.

Now consider $i = 1$. We know that D_0 has gone through two "Grow" phases, and that GD_1 has only one cycle. Notice that each degree 3 vertex in D_0 contributes one vertex *outside* D_0; if its third non-cycle neighbour were also on the cycle, then the cycle has a chord detected in Step 3(c)(ii) and D_0 does not grow even once. In fact, since D_0 grows twice, the neighbours are not only outside the cycle but are disjoint from each other. Thus for each vertex contributing to 3-size(D_0), one degree-3 vertex is added to D_1 and these vertices are distinct. Thus all these vertices are in D_1' giving $|D_1'| = \text{3-size}(D_0) \geq 1$, and 3-size$(D_1) = \text{3-size}(D_0) + |D_1'| \geq 2$.

To complete the induction, assume that the claim holds for $i-1$, where $i > 1$. In this case, $\lfloor 2^{i-2} \rfloor = 2^{i-2}$. Thus 3-size$(D_{i-1}) \geq 2^{i-1}$, and $|D_{i-1}'| \geq 2^{i-2}$.

Each $u \in D_{i-1}'$ has two neighbours, u_1 and u_2, not in D_{i-1}. These vertices contributed by each member of D_{i-1}' must be disjoint, since otherwise D_{i-1} would have a 3-bounded path to itself and would be processed at the ith stage; it would not grow the ith time. Furthermore, if u_l is of degree 2, let u_l' denote its degree-3 neighbour other than u; otherwise let $u_l' = u_l$. By the same reasoning, the vertices u_1', u_2' contributed by each $u \in D_{i-1}'$ must also be disjoint. So $2|D_{i-1}'|$ vertices are added to D_{i-1} in obtaining D_i. All these new vertices must be in D_i' as well, since otherwise D_i would have a 3-bounded path to itself and would be processed at the $(i+1)$th stage; it would not grow the $(i+1)$th time. Hence $|D_i'| = 2|D_{i-1}'| \geq 2 \cdot 2^{i-2} = 2^{i-1}$.

Every degree-3 vertex of D_{i-1} continues to be in D_i and contributes to 3-size(D_i). Furthermore, all the vertices of D_i' are not in D_{i-1} and also contribute to 3-size(D_i). Thus 3-size$(D_i) = \text{3-size}(D_{i-1}) + |D_i'| \geq 2^{i-1} + 2^{i-1} = 2^i$. □

□

Lemma 3. *The while loop of Step 3 terminates in finite time.*

Proof. Suppose some iteration of the while loop in Step 3 does not delete any edge. This means that Step 3(b) does nothing, so S has no even cycles, and Step 3(c) deletes nothing, so the clusters keep growing. But by the preceding claim, the clusters can grow at most $O(\log n)$ times; beyond that, either Step 3(c)(ii) or Step 3(c)(iii) must get executed.

Thus each iteration of the while loop of Step 3 deletes at least one edge from G, so the while loop terminates in finite time.

Lemma 4. *After step 2, and after each iteration of the while loop of Step 3, we have a point inside $\mathcal{P}(G)$.*

Proof. It is easy to see that all the building blocks described in Section 3 preserve membership in $\mathcal{P}(G)$. Hence the point obtained after Step 2 is clearly inside $\mathcal{P}(G)$. During Step 3, various edges are deleted by processing even closed walks. By our choice of S, the even cycles processed simultaneously in Step 3(b) are edge-disjoint. By lemma 1, the even closed walks processed simultaneously in Steps 3(c)(ii) and 3(c)(iii) are edge-disjoint. Now all the processing involves applying one of the building blocks, and these blocks preserve membership in $\mathcal{P}(G)$ even if applied simultaneously to edge-disjoint even closed walks. The statement follows. □

Lemma 5. *When the algorithm terminates, we have a vertex of $\mathcal{P}(G)$.*

Proof. When G is empty, all edges of the original graph have edge weights in the set $\{0, 1/2, 1\}$. Consider the graph H induced by non-zero edge weights y_e. From the description of Building Block 1, it follows that H is a disjoint union of a partial matching (with edge weights 1) and odd cycles (with edge weights 1/2). Such a graph must be a vertex of $\mathcal{P}(G)$ (see, for instance, [6]). □

5 Analysis

It is clear that each of the basic steps of the algorithm runs in NC. The proof of Lemma 2 establishes that the while loop inside Step 3(c) runs $O(\log n)$ times. To show that the overall algorithm is in NC, it thus suffices to establish the following:

Lemma 6. *The while loop of Step 3 runs $O(\log n)$ times.*

Proof. Let F be the maximum number of faces per component of G at the beginning of step 3. We show that F decreases by a constant fraction after each iteration of the while loop of step 3. Since $F = O(n)$ for planar graphs, it will follow that the while loop executes at most $O(\log n)$ times.

At the start of Step 3, connected components of G are obtained, and they are all handled in parallel. Let us concentrate on any one component. Within each component, unless the component size (and hence number of faces f in the

embedding of this component) is very small, $O(1)$, a set of $\Omega(f)$ edge-disjoint faces (in fact, $f/24$ faces) and $\Omega(f)$ edge-disjoint simple cycles can be found in NC. This is established in Lemma 3 of [7], which basically shows that a maximal independent set amongst the low-degree vertices of the dual is the required set of faces.

So let T be the set of edge-disjoint faces obtained at the beginning of step 3. If $|T| \leq f/24$, then the component is very small, and it can be processed sequentially in $O(1)$ time. Otherwise, note that after one iteration of the while loop of Step 3, T is emptied out, so all the clusters in T get processed. A single processing step handles either one cluster (cluster with self-loop) or two (clusters matched in H), so at least $|T|/2$ processing steps are executed (not necessarily sequentially).

Each processing step deletes at least one edge. Let the number of edges deleted be $k \geq |T|/2$. Of these, k_1 are not bridges at the time when they are deleted and $k_2 = k - k_1$ are bridges when they are deleted. Each deletion of a non-bridge merges two faces in the graph. Thus if $k_1 \geq |T|/4$, then at least $|T|/4 \geq f/96$ faces are deleted; the number of faces decreases by a constant fraction. If $k_2 > |T|/4$, consider the effect of these deletions. Each bridge deleted is on a path joining two clusters. Deleting it separates these two clusters into two different components. Thus after k_2 such deletions, we have at least k_2 pairs of separated clusters. Any connected component in the resulting graph has at most one cluster from each pair, and hence *does not have* at least k_2 clusters. Since each cluster contains an odd cycle and hence a face, the number of faces in the new component is at most $f - k_2 \leq f - |T|/4 \leq f - f/96$. Hence, either way, the new F is at most $95F/96$, establishing the lemma. □

6 Discussion

We show that if the perfect matching polytope $\mathcal{M}(G)$ of a planar graph is non-empty, then finding a vertex of $\mathcal{P}(G)$ is in NC. Unfortunately, we still do not know any way of navigating from a vertex of $\mathcal{P}(G)$ to a vertex of $\mathcal{M}(G)$. Note that both our algorithm and that of [7] navigate "outwards", in a sense, from the interior of $\mathcal{P}(G)$ towards an extremal point. To use this to construct a perfect matching in NC, we need a procedure that navigates "along the surface" from a vertex of $\mathcal{P}(G)$ to a vertex of its facet $\mathcal{M}(G)$.

The work of [7] shows that a vertex of $\mathcal{M}(G)$ can be found not only for bipartite planar graphs but also for bipartite small $O(\log n)$ genus graphs. The ideas needed to extend the planar bipartite case to the small-genus bipartite case apply here as well; thus we have an NC algorithm for finding a vertex of $\mathcal{P}(G)$ for $O(\log n)$ genus graphs.

For perfect matchings in general graphs, search reduces to counting, since search is in P while counting is #P-hard even under NC reductions. We believe that this holds for many reasonable subclasses of graphs as well; searching for a perfect matching in a graph reduces, via NC reductions, to counting perfect

matchings in a related graph from the same subclass. Proving this at least for planar graphs would be an important first step.

References

1. E Dahlhaus and M Karpinski. Perfect matching for regular graphs is AC^0-hard for the general matching problem. *Journal of Computer and System Sciences*, 44(1):94–102, 1992.
2. R M Karp, E Upfal, and A Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.
3. Marek Karpinski and Wojciech Rytter. *Fast parallel algorithms for graph matching problems*. Oxford University Press, Oxford, 1998. Oxford Lecture Series in Mathematics and its Applications 9.
4. P W Kastelyn. Graph theory and crystal physics. In F Harary, editor, *Graph Theory and Theoretical Physics*, pages 43–110. Academic Press, 1967.
5. L Lovasz. On determinants, matchings and random algorithms. In L Budach, editor, *Proceedings of Conference on Fundamentals of Computing Theory*, pages 565–574. Akademia-Verlag, 1979.
6. L Lovasz and M Plummer. *Matching Theory*. North-Holland, 1986. Annals of Discrete Mathematics 29.
7. M. Mahajan and K. Varadarajan. A new NC-algorithm for finding a perfect matching in planar and bounded genus graphs. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 351–357, 2000.
8. G Miller and J Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24:1002–1017, 1995.
9. K Mulmuley, U Vazirani, and V Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–131, 1987.
10. V Vazirani. NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. *Information and Computation*, 80(2):152–164, 1989.