Matching Is as Easy as the Decision Problem, in the NC Model

Nima Anari
Computer Science Department, Stanford University, CA, United States
https://nimaanari.com
anari@cs.stanford.edu

Vijay V. Vazirani
Computer Science Department, University of California, Irvine, CA, United States
https://www.ics.uci.edu/~vazirani/
vazirani@ics.uci.edu

Abstract

Is matching in NC, i.e., is there a deterministic fast parallel algorithm for it? This has been an outstanding open question in TCS for over three decades, ever since the discovery of randomized NC matching algorithms [17, 27]. Over the last five years, the theoretical computer science community has launched a relentless attack on this question, leading to the discovery of several powerful ideas. We give what appears to be the culmination of this line of work: An NC algorithm for finding a minimum-weight perfect matching in a general graph with polynomially bounded edge weights, provided it is given an oracle for the decision problem. Consequently, for settling the main open problem, it suffices to obtain an NC algorithm for the decision problem. We believe this new fact has qualitatively changed the nature of this open problem.

All known efficient matching algorithms for general graphs follow one of two approaches: given by [6] and [20]. Our oracle-based algorithm follows a new approach and uses many of ideas discovered in the last five years.

The difficulty of obtaining an NC perfect matching algorithm led researchers to study matching vis-a-vis clever relaxations of the class NC. In this vein, recently [10] gave a pseudo-deterministic RNC algorithm for finding a perfect matching in a bipartite graph, i.e., an RNC algorithm with the additional requirement that on the same graph, it should return the same (i.e., unique) perfect matching for almost all choices of random bits. A corollary of our reduction is an analogous algorithm for general graphs.

2012 ACM Subject Classification Theory of computation → Parallel algorithms; Theory of computation → Graph algorithms analysis; Theory of computation → Pseudorandomness and derandomization

Keywords and phrases Parallel Algorithm, Pseudo-Deterministic, Perfect Matching, Tutte Matrix

Digital Object Identifier 10.4230/LIPIcs.ITCS.2020.54

Related Version A full version of the paper is available at https://arxiv.org/abs/1901.10387.

Funding *Vijay V. Vazirani*: Supported in part by NSF grant CCF-1815901.

1 Introduction

Is matching in NC, i.e., is there a deterministic fast parallel[1] algorithm for finding a perfect or, more generally, a maximum matching in a general graph? This has been an outstanding open question in theoretical computer science for over three decades, ever since the discovery of RNC matching algorithms [17, 27]. Over the last five years, the TCS community has launched a relentless attack on this question, leading to the discovery of numerous powerful

[1] That runs in polylogarithmic time using polynomially many processors.

ideas [8, 32, 10, 1, 31]. We give what appears to be the culmination of this line of work: An NC algorithm for finding a minimum weight perfect matching in a general graph with polynomially bounded edge weights, provided it is given an oracle, say $\mathcal{O}$, for the decision problem. Consequently, for settling the main open problem, it suffices to obtain an NC algorithm for the decision problem. We believe this new fact has qualitatively changed the nature of this open problem. Henceforth, by *small weights* we will mean *polynomially bounded edge weights* and *acronym MWPM* will be short for *minimum weight perfect matching.*

The difficulty of obtaining an NC matching algorithm led researchers to study matching vis-a-vis certain clever relaxations of the class NC. One such relaxation is pseudo-deterministic RNC. This is an RNC algorithm with the additional property that on the same graph, it must return the same (i.e., unique) solution for almost all choices of random bits [9, 10]. Recently, [10] gave such an algorithm for perfect matching in bipartite graphs. A second relaxation of NC is quasi-NC, under which the algorithm must run in polylogarithmic time, though it can use $O(n^{\log^{O(1)} n})$ processors; see Section 1.1 for results obtained for this model.

A corollary of our result extends [10] to general graphs as follows: The precise decision problem for our result is: Given a graph G with small weights and a number W, is there a perfect matching of weight at most W in G. Clearly, this is NC equivalent to: Find the weight of a minimum weight perfect matching in G. This question is easy to answer in RNC with inverse-polynomial probability of error using the algorithm of [27]. Therefore, using this RNC algorithm in place of the oracle we get an RNC matching algorithm with the property that in a run, all queries to the decision problem will be answered correctly with overwhelming probability, i.e., this is a pseudo-deterministic RNC matching algorithm.

All known efficient matching algorithms for general graphs follow one of two approaches: given by [6] and [20]. Our oracle-based algorithm follows a new approach and uses many of ideas discovered in the last five years. In particular, it uses the overall structure of the recent NC algorithm of [1] for finding a perfect matching in planar graphs. Since oracle $\mathcal{O}$ can be implemented in NC for planar graphs, our current paper yields a simpler NC algorithm for finding a perfect matching in planar graphs. The second key ingredient which made our current result possible is an NC algorithm for finding a maximal laminar family of tight odd sets in a given face of the perfect matching polytope. This follows from the works of [4] and [31].

Our main result is:

▶ **Theorem 1.** *There is an* NC *algorithm for finding a minimum weight perfect matching in general graphs with small weights, provided the algorithm is given access to oracle $\mathcal{O}$ for the decision problem. The latter is: Given a graph G with small weights and a target weight W, is there a perfect matching of weight at most W in G?*

▶ **Corollary 2.** *There is an* NC *algorithm for finding a maximum matching in general graphs, provided the algorithm is given access to the oracle $\mathcal{O}$.*

▶ **Corollary 3.** *There is a pseudo-deterministic* RNC *algorithm for finding a minimum weight perfect matching in general graphs with small weights.*

We further show that our algorithms only need to call the decision oracle for minors of the input graph.

▶ **Theorem 4.** *Let $\mathcal{F}$ be a minor-closed family of graphs. If there is an* NC *algorithm for deciding whether a perfect matching of weight at most W exists in graphs from $\mathcal{F}$, weighted with polynomially small weights, then there is also an* NC *algorithm for* finding *a MWPM in such graphs.*

1.1 Related work and a brief history of parallel matching algorithms

The notion of a pseudo-deterministic algorithm with polynomial expected running time was given by [9]. Such an algorithm runs in expected polynomial time and is required to output the same (i.e., unique) solution on a given instance on each run with high probability. Hence, in this sense, it resembles a deterministic algorithm. [9] gave pseudo-deterministic polynomial expected running time algorithms for several number theoretic and cryptographic problems. The notion of pseudo-deterministic RNC algorithms was defined by [10].

An RNC algorithm for the decision problem, of determining if a graph has a perfect matching, was obtained by [20], using the Tutte matrix of the graph. The first RNC algorithm for the search problem, of actually finding a perfect matching, was obtained by [18]. This was followed by a simpler and more versatile algorithm due to [27]; besides perfect matching, it also yielded RNC algorithms for the problem of exact matching (see Section 8) and for finding a MWPM in a graph with small weights. The latter fact is crucially used for obtaining pseudo-deterministic RNC algorithms for bipartite graphs [10] and general graphs (current paper). The "philosophy" behind [27] will be useful for dealing with a difficulty that arises in the design of the current algorithm as well, so it is recalled below[2].

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 [6], the counting class #P [33] and a polynomial time equivalence between random generation and approximate counting for self-reducible problems [15], which lies at the core of the Markov chain Monte Carlo method. The perspective of parallel algorithms has also led to such a gain, namely the Isolation Lemma [27], which has found several applications in complexity theory and algorithms. Considering the fundamental insights gained from an algorithmic study of perfect matchings, 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 for the case of planar bipartite graphs by [26] via a flow-based approach, followed by [24] using the fact that there is an NC algorithm for counting perfect matchings in planar graphs. The long-standing problem of extending this result to non-bipartite planar graphs was resolved by [1]. Subsequently, [31] also got the same result using different ideas. [1] also extended their algorithm to constant genus graphs. Subsequently, [7] gave an NC algorithm for perfect matching in one-crossing-minor-free graphs, which include K_5-free graphs and $K_{3,3}$-free graphs; the resolution of the latter class settles a thirty-year-old open problem asked in [34].

The quasi-NC algorithms for matching and its generalizations, mentioned above, work by achieving a partial derandomization of the Isolation Lemma. First, [8] gave a quasi-NC algorithm for perfect matching in bipartite graphs, which was followed by the algorithm of [32] for general graphs. Algorithms were also obtained for the generalization of bipartite matching to the linear matroid intersection problem by [11], and to a further generalization of isolating a vertex of a polytope with faces given by totally unimodular constraints, by [12].

[2] Under the NC model, any one processor does not even have enough time to read the entire input, and hence can perform only local computations. On the other hand, a perfect matching is a global object, unlike say, a maximal independent set. Further difficulties arise from the fact that the number of perfect matchings in a graph can vary widely, all the way from one to exponentially many (assuming it has at least one). If there were a unique perfect matching in the graph, the algorithm's task would become a lot simpler. [27] achieve uniqueness via a powerful probabilistic fact, the Isolating Lemma: under an assignment of randomly chosen small weights to the edges it claims that the MWPM will be unique with high probability.

1.2 What is the "right" decision problem?

Consider the following two decision problems for perfect matching:

- Given a graph G with small weights and a target W, is there a perfect matching of weight at most W in G?
- Does graph G have a perfect matching?

Clearly, the second can be reduced to the first and is therefore "easier". This leads to a legitimate question: why not attempt to reduce, in NC, the search problem to the second decision problem? Our experience suggests that the first problem is much more basic for the setting at hand. We next provide evidence to this effect.

Seeking a MWPM in a graph with small weights was the central problem in the work of [27]. The Isolating Lemma helped find small weights under which there was a unique MWPM. The second half of [27] gave an NC algorithm for finding this (unique) perfect matching, using the Tutte matrix of the graph and matrix inversion; the latter is known to be in NC [3]. Ever since then, perhaps the most popular avenue for obtaining an NC matching algorithm has been to derandomize the Isolating Lemma. This would deterministically yield small weights under which there is a unique MWPM, and it could be found using the second half of [27].

The question of MWPM in a graph with small weights plays a central role in NC-type approaches to all non-bipartite, and even some bipartite, perfect matching algorithms: partial derandomization leading to quasi-NC algorithms [8, 32], resolution of the open problem of non-bipartite planar graphs [1, 31], and quasi-deterministic RNC algorithms for bipartite [10] and general graphs (current paper).

In mathematics, sometimes solving the harder problem turns out to be easier than solving the easier one, if the former has a better "behavior". Our belief is that this is the case here.

1.3 Bipartite vs non-bipartite matching: An intriguing phenomenon

Decades of algorithmic work on the matching problem, from numerous perspectives, exhibits the following intriguing phenomenon: The bipartite case gets solved first. Then, using much more elaborate machinery, the general graph case also follows and yields the exact same result! This phenomenon is made all the more fascinating by the fact that the "elaborate machinery" consists not of one fact but numerous different structural properties and mathematical facts which happen to be just right for the problem at hand! We give a number of examples below.

The duality between maximum matching and minimum vertex cover for bipartite graphs extends to general graphs via the notion of an odd set cover, see [21]. The formulation of the perfect matching polytope for bipartite graphs extends by introducing constraints corresponding to odd sets [6]. Polynomial time algorithms for maximum matching and maximum weight matching in bipartite graphs generalize via the notion of blossoms [21]. The most efficient known algorithm for maximum matching in bipartite graphs [13, 19] obtained via an alternating breadth first search, extends via a much more elaborate algorithm with the same running time via the graph search procedure of double depth first search [25] and blossoms defined from the perspective of minimum length alternating paths [35]. The RNC matching algorithms [18, 27] use Tutte's theorem to extend to general graphs. The randomized matching algorithm of [30] uses Tutte's theorem and a theorem of Frobenius about ranks of sub-matrices of skew-symmetric matrices.

More recent work exhibits this phenomenon as well. The quasi-NC algorithm of [8] for bipartite graphs extends by handling tight odd cuts appropriately [32]. The NC algorithm of [24] for planar bipartite graphs was extended to non-bipartite graphs via Edmonds' formulation of the perfect matching polytope [6], an NC algorithm for max-flow in planar

graphs [16], and a result of [28] proving that the Gomory-Hu tree of a graph must contain a tight odd cut, and an elaborate NC algorithm for uncrossing tight odd cuts [1]. In the same vein, the current paper is extending the pseudo-deterministic RNC bipartite algorithm of [10] by giving a way of dealing with tight odd cuts in Edmonds' formulation of the perfect matching polytope [6] and using an NC procedure for finding a maximal laminar family of tight odd cuts [4, 31].

2 Overview and Technical Ideas

Most of this paper will concentrate on the problem of finding a perfect matching in a general graph in NC, given oracle $\mathcal{O}$. In Section 6.1 we will extend our ideas to finding a MWPM for small weights. Then, an algorithm for finding a maximum matching in a general graph in NC will easily follow. In this section, we will also give a number of key definitions which will be used throughout the paper.

2.1 The bipartite case

For ease of comprehension, we will first give an outline of a proof of Theorem 1 for the case of bipartite graphs. Such a proof can be gleaned from the paper of [10]; however, to the best of our knowledge, this important fact was not derived so far. Below, we build on the quasi-NC algorithm of [8] to obtain a somewhat simpler proof of this result.

The algorithm of [8] first starts with the perfect matching polytope and then iteratively moves to lower dimensional faces of this polytope, terminating when a vertex of the polytope is reached; this will be a perfect matching.

▶ **Definition 5.** *In a general graph $G = (V, E)$ with edge weight function w, an edge e is called an* allowed edge *if it participates in MWPM. Let $E[w]$ denote the set of all allowed edges. Edges in the complement of this set will be called* disallowed edges.

Assume w are small weights and let $\mathrm{PM}[w]$ denote the face of the polytope containing all fractional and integral MWPMs w.r.t. w. Since we are in the bipartite case, $\mathrm{PM}[w]$ has a simple description: It is defined by the set of disallowed edges, since they are set to zero, or equivalently its complement, i.e., the set of allowed edges, $E[w]$. The description of the algorithm given above can be refined to: Iteratively modify the weight vector w so that the dimension of face $\mathrm{PM}[w]$ keeps dropping, and equivalently $E[w]$ keeps getting sparser, until $E[w]$ is a perfect matching.

As argued earlier, using oracle $\mathcal{O}$, we can find the weight of a MWPM in G. Further, it is easy to see that for a given edge e, we can determine in NC if e participates in a MWPM, i.e., if $e \in E[w]$. Repeating for all edges in parallel, we can compute $E[w]$ in NC. The following is a fundamental notion in all recent NC-type matching algorithms:

▶ **Definition 6** ([5])**.** *Number the edges of an even cycle C in a general graph G with edge-weights w starting from an arbitrary edge. The* circulation of cycle C *is the absolute value of the difference of the sum of weights of odd-numbered and even-numbered edges and is denoted* $\mathrm{circ}_w(C)$.

It is easy to prove that if the MWPM in G is not unique, then any cycle in the symmetric difference of two such matchings must have zero circulation[3].

[3] Hence, if we find a weight vector w such that each cycle in G has nonzero circulation, then the MWPM must be unique and can be found in NC.

▶ **Proposition 7** ([8]). *In a bipartite graph, let cycle $C \subseteq E[w]$ have $\mathrm{circ}_w(C) = 0$. Let G denote the graph on edge set $E[w]$. Assign small weights w' to edges $E[w]$ so that $\mathrm{circ}'_w(C) > 0$. Then C will not be present in $E[w']$, i.e., at least one of its edges will be dropped in going from $E[w]$ to $E[w']$. We will say that C got* destroyed.

Hence, if we find a weight vector that destroys all cycles of G, we would be done. However, G may have exponentially many cycles, so this is non-trivial. One of the key ideas of [8] is a systematic way of destroying cycles: They iteratively destroy cycles of length $4, 8, 16, \ldots, n$; clearly, the number of iterations needed is $O(\log n)$. In the first round, G has at most $O(n^4)$ cycles of length 4. [8] show that if all cycles of length at most 2^i have already been destroyed, then there are at most $O(n^4)$ cycles of length at most 2^{i+1} left. Hence, in each iteration only $O(n^4)$ cycles need to be destroyed.

Suppose the current iteration starts with small weights w under which all cycles of length at most 2^i have already been destroyed. In this iteration, the algorithm finds a weight vector w' for the edges in $E[w]$ under which all cycles of length at most 2^{i+1} are destroyed. The following fact will play a central role in the current paper as well:

▶ **Proposition 8** ([8]). *In order to destroy any set of s cycles, it suffices to try certain well-chosen $O(n^2 s)$ integral weight vectors each of which uses numbers that are $O(n^2 s)$; one of these vectors is sure to work.*

Since in the current iteration $s = O(n^4)$, at most $O(n^6)$ weight vectors suffice. The algorithm for choosing a weight vector that works is as follows. In parallel, for each of the $O(n^6)$ weight vectors, y, compute $E[y]$ and find the girth of the resulting graph; this can easily be done in NC. Pick the lexicographically first weight vector, say w', such that $E[w']$ has girth $> 2^{i+1}$. Clearly, w' destroys all cycles of length at most 2^{i+1}.

2.2 Extension to general graphs

Matching algorithms for general graphs, in different computational models, are far harder because they need to handle odd cycles in special ways. The set of constraints capturing the perfect matching polytope is also more complex: it includes exponentially many odd set constraints. An odd set $S \subset V$ which satisfies this constraint with equality is called a *tight odd set*. The description of face $\mathrm{PM}[w]$ is also much more involved: in addition to edges $E[w]$, we need a maximal laminar family of tight odd sets, say $\mathcal{L}$; see Section 3.2.

The "engine" underlying our algorithm

Analogous to the bipartite case, there is an "engine" underlying our algorithm as well – it iteratively reduces the size of the graph. This engine can be thought of as composed of three components which draw on different domains to establish structural facts and algorithms.

2.2.1 Component based on the structure of the perfect matching polytope

Proposition 7, which yielded the "engine" for the bipartite case, does not hold in general graphs. Thus a non-bipartite graph may have an even cycle $C \subseteq E[w]$ with $\mathrm{circ}_w(C) > 0$. The reason for this is the presence of a tight odd set. As a result, Proposition 7 needs to be enhanced to the fact stated below. We will say that a cycle C *crosses* a tight odd set S if C has vertices in S as well as in $(V - S)$. Similarly, edge e *crosses* S if one of its endpoints is in S and the other is in $V - S$.

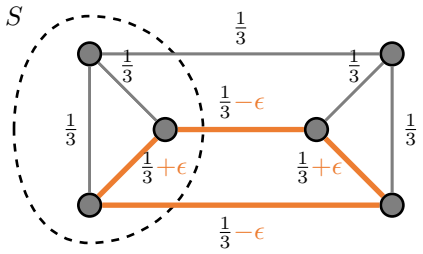

Figure 1 The orange even cycle crosses tight odd set S; example due to [8, 32].

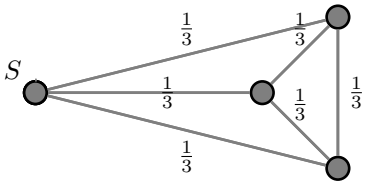

Figure 2 Resulting graph after shrinking tight odd set S.

▶ **Proposition 9** ([32]). *In a general graph G, suppose even cycle $C \subseteq E[w]$ has* $\text{circ}_w(C) > 0$. *Then, there must be a tight odd set S such that C* crosses *S.*

This is illustrated in Figure 1. In this graph, the three edges in $\delta(S)$ have weight 1 and the rest have weight 0. Observe that each edge participates in a MWPM and hence $E[w]$ consists of all edges. The cycle consisting of the four orange edges, say C, has positive circulation even though it is contained in $E[w]$. Cycle C crosses tight odd set S.

▶ **Definition 10.** *Assume that even cycle C* crosses *tight odd set S. Number the edges of C starting from an arbitrary edge. Let n_o and n_e denote the number of odd-numbered and even-numbered edges, respectively, that cross S. Then the* mismatch of C and S, *denoted* $\text{mismatch}(C, S)$, *is $|n_o - n_e|$.*

Note that in Figure 1, $\text{mismatch}(C, S) = 2$. Observe that if the MWPM is not unique and C is a cycle in the symmetric difference of two such perfect matchings then the following must hold:

- $\text{circ}_w(C) = 0$.
- If C crosses a tight odd set S, then $\text{mismatch}(C, S) = 0$; the reason is that each perfect matching crosses each tight set exactly once.

▶ **Proposition 11** (Lemma 25). *Consider a general graph G with weights w and even cycle $C \subseteq E[w]$ with* $\text{circ}_w(C) > 0$. *Let S be a tight odd set such that C crosses S. Then* $\text{mismatch}(C, S) > 0$ *and at least one edge of C has both its endpoints in S.*

Our strategy for dealing with cycle C having $\text{circ}_w(C) > 0$ is to *shrink* the tight odd set S it crosses; this is illustrated in Figure 2. By Proposition 11, this will shrink at least one edge of C, hence resulting in a smaller graph. Our overall strategy is as follow: Suppose w.r.t. weight vector w, $\text{circ}_w(C) = 0$. Let w' be a weight vector such that $\text{circ}_{w'}(C) > 0$. If so, [32] show that either C must lose an edge in going from $E[w]$ to $E[w']$ or a new odd set S goes tight w.r.t. w' such that C crosses S. In the latter case, we shrink S. In either case we will obtain a smaller graph and in both cases we will say that C is *destroyed*.

2.2.2 Component based on graph-theoretic facts

As stated in the Introduction, the overall structure of our algorithm is similar to that of [1]. Both algorithms require in each iteration a large enough number of edge-disjoint even cycles whose destruction will result in the removal of a corresponding number of edges. However, in both cases, the graph may have not such cycles. The recourse is to resort to even walks.

▶ **Definition 12** ([32]). *We call an ordered list of an even number of edges $C = (e_1, \ldots, e_{2k})$, not necessarily distinct, that start and end at the same vertex, an* even walk *if this list traverses either a simple even cycle or two odd cycles with a path joining them; in the latter case, the cycles are traversed once each and the path twice, once in each direction.*

The list C of edges of an even walk contains each edge either once or twice, and if it contains an edge e twice, then both copies will have the same parity. The notions of circulation and mismatch can be extended to even walks in a natural way by taking into consideration multiplicity of edges. Thus if e occurs twice in walk C, is odd-numbered and crosses tight odd set S, then it contributes 2 to n_o in the computation of mismatch(C, S) (see Definition 10) and it contributes $2w_e$ to the sum of odd-numbered edges in the computation of $\text{circ}_w(C)$ (see Definition 6). As shown in [32], all statements made above about destroying even cycles carry over to even walks as well.

[1] critically used Euler's formula and the planar dual of G for first finding a large number of edge-disjoint cycles in NC. If more than half were even, they sufficed. Otherwise, they paired up odd cycles and found paths connecting each pair to obtain even walks. This was done in a such a manner that the resulting even walks were edge-disjoint.

Finding edge-disjoint cycles in a general graph in NC appears to be quite difficult. Instead, we take a cue from the bipartite case, which finessed the issue of finding edge-disjoint cycles by using Proposition 7. As a result, showing the *existence* of cycles sufficed! However, there is a subtle difference: in the bipartite case, we needed to upper bound the number of cycles that needed to be destroyed in each iteration, whereas here we need to lower bound them; the latter is the case in [1] as well.

Using ideas from [2] we show that if the graph $G = (V, E)$ is not very sparse (see Definition 31), then it contains $\Omega\left(\frac{|E|}{\log^2|V|}\right)$ edge-disjoint even cycles. Then, using ideas from [1], we show how to pair up odd cycles to form walks. Unlike [1], the walks don't need to be found explicitly – establishing existence suffices.

If in an iteration the graph is very sparse, it will not have the required number of edge-disjoint cycles. For this case, we define the notion of a *triad* in Definition 21; this is a tight odd set consisting of three vertices. We show that the graph has sufficiently many disjoint triads, and a maximal independent set algorithm can find a large enough subset of these in NC. These can be shrunk simultaneously.

2.2.3 Component based on facts from matching theory

Suppose that in a certain iteration our algorithm is trying weight function w, as per Proposition 7. We will need to find in NC a description of face PM$[w]$, which involves, in addition to edges $E[w]$, a maximal laminar family of tight odd sets, say $\mathcal{L}$. Computing $E[w]$ using oracle $\mathcal{O}$ is straightforward. However, finding family $\mathcal{L}$ in NC is a difficult question. The difficulty is similar to that of finding a perfect matching in a graph, i.e., the presence of a plethora of solutions. Recall the "philosophy" of [27] given in Section 1.1, for dealing with this issue for perfect matching, namely attempt to narrow down the choices to one. Clearly unlike [27], randomization is not a resource we can use for this purpose. The solution involves imposing more and more restrictions on the family of tight odd sets until it becomes unique! These restrictions arise from deep structural facts from matching theory. Additional facts lead to an NC algorithm for computing $\mathcal{L}$ with the help of $\mathcal{O}$. These ideas are from [4] and [31] and are given in Section 3.2.

For the "correct" weight function, say w, among the set of even walks being handled in this iteration, some will be destroyed by losing an edge and some by crossing a tight odd set. By updating the edge set to $E[w]$, we can accrue the advantage from the first set of walks. For obtaining advantage from the second set of walks, for each such walk, say C, we need to shrink a tight odd set, say S, that it crosses. A major obstacle is that our algorithm does not "know" *any* of the walks! The way we finesse this difficulty is to shrink all outermost sets of L, which are clearly be disjoint, in the graph on edge set $E[w]$.

Finally, among all weight functions, we will pick the one, say w, that yields a graph with the smallest number of edges. There is no guarantee that w would have destroyed all s walks which we had established the existence of up-front. However, at least one of the weight functions must have done so and therefore led to a decrease of at least s edges. Hence, w must also decrease at least s edges, and that suffices for making progress. As shown in Lemma 42, the number of non-isolated edges gets reduced by a factor of $1 - \Omega(1/\log^2|V|)$ in each iteration.

2.3 The final idea: balanced viable set

Our current strategy is to iteratively reduce the number of edges until a perfect matching remains. After picking its edges, we need to recursively find a perfect matching in each of the shrunk sets (after removing its matched vertex). The resulting algorithm would have polylogarithmic depth; however, it does not run in polylogarithmic time because of the following inherent sequentiality: Perfect matchings in shrunk sets can be found only *after* finding a perfect matching in the shrunk graph, because the algorithm needs to know the vertex in S that is matched outside S. Moreover, perfect matchings in the shrunk graph and the shrunk sets need to be found via a recursive application of the full algorithm described so far.

The exact same issue arose in [1] as well. The solution proposed there was meant for general graphs and hence it works here as well. The solution is quite elaborate and hence is not repeated here; instead, we direct the reader to Section 4.2 in [1]. We note that the task is somewhat easier here because we have recourse to oracle $\mathcal{O}$; [1] had to resort to computing Pfaffians orientations, etc. We give a short, high-level summary below.

An odd set S is *viable* if there is at least one perfect matching in G which picks exactly one edge from $\delta(S)$. A set S is *balanced* if both S and its complement contain a constant fraction of the vertices. [1] show how to find in NC a balanced viable odd set. Let S be such a set. Clearly, using oracle $\mathcal{O}$, we can find an edge $e \in \delta(S)$ which is the unique edge in a perfect matching from this cut. Now we are done by a simple divide-and-conquer strategy: match e, remove its end-points and find perfect matchings in the two sides of the cut recursively, in parallel. Observe that even though perfect matchings in the two sides can be found only after finding the matched edge e, the latter can be done without any recursive calls, hence, leading to a polylogarithmic running time.

3 Preliminaries

We represent undirected graphs by $G = (V, E)$, where V is the set of vertices and E is the set of edges. Unless otherwise specified, we only work with graphs that have no loops, i.e., an edge from a vertex to itself. An edge between vertices u and v is represented as $\{u, v\}$. For a set $S \subseteq V$, we use $\delta(S)$ to denote the cut between S and its complement, i.e., $\delta(S) = \{\{u, v\} \in E \mid u \in S, v \notin S\}$. When S is a singleton, i.e., $\{v\}$ for some $v \in V$, we use the shorthand $\delta(v) = \delta(\{v\})$. A perfect matching is a subset of edges $M \subseteq E$ such that for all $v \in V$ we have $|M \cap \delta(v)| = 1$.

▶ **Definition 13.** *We call an edge* $e = \{u, v\}$ isolated *if* $\deg(u) = \deg(v) = 1$.

By this definition a graph is a perfect matching if it has no isolated vertices and all of its edges are isolated.

For a set $S \subseteq E$ of edges we use $\mathbb{1}_S \in \mathbb{R}^E$ to denote the indicator of S. We use the shorthand $\mathbb{1}_e$ to denote the e-th element of the standard basis for $\mathbb{R}^E$, where $e \in E$. We denote the standard inner product between vectors $w, x \in \mathbb{R}^E$ by $\langle w, x \rangle$.

Given a convex polytope $P \subseteq \mathbb{R}^E$, and a weight vector $w \in \mathbb{R}^E$, we use $P[w]$ to denote the set of points minimizing the weight function $x \mapsto \langle w, x \rangle$:

$$P[w] = \{x \in P \mid \forall y \in P : \langle w, x \rangle \leq \langle w, y \rangle\}.$$

Note that $P[w]$ is a face of P; all faces of P can be obtained as $P[w]$ for appropriately chosen w.

3.1 The perfect matching polytope

Given a graph $G = (V, E)$, we call a subset of edges $M \subseteq E$ a perfect matching if it contains exactly one edge in every degree cut, i.e., $|M \cap \delta(v)| = 1$ for all v. We call a graph matching-covered if any of its edges can be extended to a perfect matching.

▶ **Definition 14.** *A graph* $G = (V, E)$ *is* matching-covered *if for every edge* $e \in E$*, there exists a perfect matching* M *such that* $e \in M$*.*

The perfect matching polytope for $G = (V, E)$ is the convex hull of all perfect matchings of G in $\mathbb{R}^E$. Thus,

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

Clearly the perfect matchings of G are in one-to-one correspondence with the vertices of this polytope.

When G is clear from context, we simply use PM to refer to this polytope. PM is alternatively described by the following set of linear equalities and inequalities [6]:

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

Any face F of PM can be either described by a weight vector w, i.e., $F = \mathrm{PM}[w]$, or it can be alternatively described by the set of inequalities turned into equalities in Equation (1). These correspond to odd sets S and edges e. When face F is clear from context, we call odd sets whose inequalities have been turned into equalities, *tight* odd sets. We call an edge e *allowed* if $x_e > 0$ for some $x \in F$, i.e., if the inequality corresponding to e in Equation (1) has not been turned into equality. We use $E[w]$ or $E[F]$ to denote the set of allowed edges in the face $F = \mathrm{PM}[w]$. Putting it all together, to describe a face F it is enough to describe the set of allowed edges as well as tight odd sets.

3.2 Finding a description of a face

A key step in our oracle-based algorithm is: given small weights w, compute a description of the face $F = \mathrm{PM}[w]$. As stated before, using oracle $\mathcal{O}$, $E[w]$ can be computed in NC. However, as far as tight odd sets go, there are typically exponentially many choices of a

family of such sets that suffice. At this point, it will be useful to recall the "philosophy" of [27] given in Section 1.1, namely when designing an NC algorithm, faced with a plethora of solutions, one should attempt to narrow down the choices to one. Clearly unlike [27], randomization is not a resource we can use for this purpose. The solution to this puzzle is indeed one of the keys that enables our result and is described below. It involves imposing more and more structure on the family of tight odd sets we seek until it becomes unique! It turns out that the latter can be computed in NC with the help of $\mathcal{O}$.

Two tight odd sets $S_1, S_2 \subseteq V$ are said to *cross* if they are not disjoint and neither is a subset of the other. A family of these sets $\mathcal{L} \subseteq 2^V$ is said to be *laminar* if no pair of sets in it cross. It is well-known that each face F of the perfect matching polytope can be described by the set of allowed edges and a laminar family of tight odd sets $\mathcal{L}$:

$$F = \left\{ x \in \text{PM} \;\middle|\; \begin{array}{ll} \langle \mathbb{1}_{\delta(S)}, x \rangle = 1 & \forall S \in \mathcal{L}, \\ \langle 1_e, x \rangle = 0 & \forall e \notin E[F]. \end{array} \right\}.$$

In fact, $\mathcal{L}$ can be taken to be any *maximal* laminar family of tight odd sets for the given face F (see, e.g., [32, Lemma 2.2]). Note that we will always include all singletons $\{v\}$ in the laminar family $\mathcal{L}$ since the equalities $\langle \mathbb{1}_{\delta(v)}, x \rangle = 1$ are automatically satisfied over all of PM. However, there are still potentially many choices for the laminar family $\mathcal{L}$ describing face F, so we impose more conditions on $\mathcal{L}$.

▶ **Definition 15.** *Suppose we are given a face $F = \text{PM}[w]$. A* laminar optimal dual solution *is a laminar family $\mathcal{L}$ of tight odd sets, including all singletons, together with a function $\pi : \mathcal{L} \to \mathbb{R}$ such that for $S \in \mathcal{L}$, $\pi(S) > 0$ whenever $|S| > 1$ and for all edges e*

$$w_e \geq \sum_{S \in \mathcal{L} : e \in \delta(S)} \pi(S),$$

with equality for allowed edges.

This definition gives dual solutions for the linear program $\min\{\langle w, x \rangle \mid x \in \text{PM}\}$ that satisfy complimentary slackness and are in *laminar* form. By complimentary slackness, for any such solution, $\sum_{S \in \mathcal{L}} \pi(S)$ is equal to the weight of a MWPM. Laminar optimal dual solutions exist but are still not unique.

[4] showed that extra conditions can be imposed on laminar optimal dual solution to make it unique. They studied the notion of *balanaced critical dual solutions* and they showed how this unique $\mathcal{L}$ can be found by computing *primal* solutions to the MWPM problem. [31] used this procedure to design an alternative NC algorithm for planar graph perfect matching. We describe this procedure below. For more details see the work of [4]. Note that we will not use these rather complex and elaborate extra conditions in any other context, so we will not state them explicitly.

The following was shown by [4, Lemma 28].

▶ **Lemma 16** ([4])**.** *If $E[w]$ is connected, then a balanced critical dual is unique and Algorithm 1 finds its support, the laminar family $\mathcal{L}$.*

It was observed by [31] that all steps of Algorithm 1 can be performed in NC except for finding allowed edges $E[w]$ and the computation of $\mu(v)$'s. We note that using oracle $\mathcal{O}$, both these steps can be also be performed in NC.

▶ Remark 17. When $E[w]$ is not connected, Algorithm 1 still works but should be run in parallel for *each connected component* of $E[w]$.

Algorithm 1 Finding a balanced critical dual.

$\mathcal{L} \leftarrow \{\{v\} \mid v \in V\}$.
for $v \in V$ **in parallel do**
  $\mu(v) \leftarrow \min\{\langle w, \mathbb{1}_M \rangle \mid M \subseteq$
    $E[w]$ is a perfect matching on $V \setminus \{u, v\}$ for some vertex $u\}$.
end
Let $w'_e \leftarrow w_e + \mu(u) + \mu(v)$ for each $e \in E[w]$.
for $t \in \{w'_e \mid e \in E[w]\}$ **in parallel do**
  Find the connected components of the graph $(V, \{e \in E[w] \mid w'_e \leq t\})$.
  Add each nontrivial connected component to $\mathcal{L}$.
end
return $\mathcal{L}$.

3.3 Contraction of tight odd sets, matching minors, and triads

[6] observed that if a collection of tight odd sets are disjoint, one can shrink each one to a single node and obtain a smaller graph whose perfect matchings can be extended to perfect matchings in the original graph. For the sake of completeness we state and prove this fact here.

▶ **Proposition 18.** *Suppose that $F = \mathrm{PM}[w]$ is a face of the matching polytope for $G = (V, E)$ and $S_1, \ldots, S_k$ are disjoint tight odd sets w.r.t. F. Let H be obtained from G by removing disallowed edges and contracting each S_i to a single node. Then any perfect matching in H can be extended to a perfect matching in G.*

Proof. Suppose that M is a perfect matching in H. We can think of edges in M as edges in E as well; in fact $M \subseteq E[w]$, because we remove disallowed edges to obtain H. Because M is a perfect matching in H, for each S_i, there is a unique $e_i \in M \cap \delta_G(S_i)$. Now since e_i is an allowed edge, there must be some perfect matching M_i of G such that $e_i \in M_i$ and $\mathbb{1}_{M_i} \in F$. Since S_i is a tight odd set, M_i cannot have any other edge in $\delta(S_i)$, except for e_i. So if we look at $\{\{u, v\} \in M_i \mid u, v \in S_i\}$, we must have a matching covering all vertices of S_i except for the endpoint of e_i. Combining all of these matchings for $i = 1, \ldots, k$ together with M will give us a perfect matching in G as desired. ◀

Note that the graph H obtained above is a minor of the graph G. But it is not an arbitrary minor. It has the additional property that every perfect matching of it can be extended back to a perfect matching of the original graph. For convenience we name these minors, matching minors.

▶ **Definition 19.** *A matching minor H of a graph G, is a graph that can be obtained by a sequence of the following operations: Pick a face of the matching polytope and a collection of disjoint tight odd sets. Remove disallowed edges, and contract each tight odd set into a single node.*

The following statement follows directly from Proposition 18.

▶ **Lemma 20.** *If H is a matching minor of the graph G, then every perfect matching in H can be extended to a perfect matching in G.*

In our algorithms, we use the simple observation that a path of length 2 on vertices of degree 2 yields a tight odd set for the entire matching polytope. We call these paths triads.

▶ **Definition 21.** *A* triad *in graph $G = (V, E)$ is a set of three vertices $\{a, b, c\}$ such that* $\deg(a) = \deg(b) = \deg(c) = 2$*, and* $\{a, b\}, \{b, c\} \in E$*.*

▶ **Lemma 22.** *A triad $\{a, b, c\}$ is a tight odd set for the matching polytope and all of its faces.*

Proof. The only two neighbors of b are a, c. So in every perfect matching, b must be matched to one of them. The other vertex must have an edge to an outside vertex, and in fact that is the only possible edge in $\delta(\{a, b, c\})$. ◀

▶ Remark 23. Note that the proof of Lemma 22 does not use the assumptions $\deg(a) = \deg(c) = 2$ and only uses $\deg(b) = 2$. We will use these extra assumptions elsewhere, to prove that in certain situations, we can find many triads in our graph.

3.4 Even walks and weight vectors

Even walks were defined in Definition 12. For an even walk C, define the *signature* of C to be the vector:

$$\text{sign}(C) = \sum_{i=1}^{2k} (-1)^i \mathbb{1}_{e_i}.$$

The notions of circulation and mismatch can be stated in terms of signature:

$$\text{circ}_w(C) = |\langle w, \text{sign}(C)\rangle|$$

$$\text{mismatch}(C, S) = |\langle \mathbb{1}_{\delta(S)}, \text{sign}(C)\rangle|$$

Now, there cannot be two distinct points $x, y \in \text{PM}[w]$ whose difference $x - y$ is a multiple of $\text{sign}(C)$, since otherwise we would have $\langle w, x\rangle \neq \langle w, y\rangle$. Another way of stating this is that if $x \in \text{PM}[w]$, then $x + \epsilon\,\text{sign}(C) \notin \text{PM}[w]$ for any $\epsilon \neq 0$. So, some inequality or equality describing $\text{PM}[w]$ must be violated for this point. If we pick x to be in the relative interior of the face $\text{PM}[w]$ we will have some slack for non-tight inequalities describing $\text{PM}[w]$. So the violated constraint for $x + \epsilon\,\text{sign}(C)$ must be a constraint that is tight for the entire face $\text{PM}[w]$. This implies that:

▶ **Lemma 24.** *Let C be an even walk with* $\text{circ}_w(C) > 0$*. Then either there is an edge $e \in C$ that is disallowed, i.e., $e \notin E[w]$, or for any laminar dual $(\mathcal{L}, \pi)$ describing* $\text{PM}[w]$*, there is some set S such that* $\text{mismatch}(C, S) > 0$*.*

For a more detailed proof of this, see [1, 32]. Note that if $\text{mismatch}(C, S) > 0$, then C must have an edge with both endpoints inside S.

▶ **Lemma 25.** *If C is an even walk and S is a tight odd set such that* $\text{mismatch}(C, S) > 0$*, then there is an edge $e = \{u, v\} \in C$ such that $u, v \in S$.*

Proof. If this is not true, then every time C enters S it must immediately exit. So if we compute $\text{mismatch}(C, S)$ by looking at edges that cross S, we always get a $+1$ followed by a -1, and a -1 followed by a $+1$. So the entire sum would be 0 which is a contradiction. ◀

We also borrow from [8] the following important result, which is also stated in [32] and as Proposition 7 in this paper.

▶ **Lemma 26** ([8]). *There exists a polynomial sized family of polynomially bounded weight vectors $\mathcal{W}$, such that for any set of edge disjoint even walks $C_1, \ldots, C_k$, there is some $w \in \mathcal{W}$ which ensures*

$$\forall i : \mathrm{circ}_w(C) > 0.$$

Proof. This lemma is actually proved in [8, 32] for any collection of nonzero vectors, not just $\mathrm{sign}(C_i)$'s, as long as there is both a polynomial bound on the number of vectors and the absolute value of their coordinates. Edge-disjointness of even walks automatically puts a bound of $|E|$ on their number, and the coordinates of our even walks are always bounded in absolute value by 2. ◀

3.5 Maximal independent sets

Given a graph $G = (V, E)$, we call a subset $S \subseteq V$ independent if no edge $e \in E$ has both endpoints in S. We call an independent set maximal if no strict superset $T \supsetneq S$ is independent. We will crucially use the fact that maximal independent sets can be found in NC.

▶ **Theorem 27** ([22]). *There is a deterministic* NC *algorithm that on input graph $G = (V, E)$ returns a maximal independent set $S \subseteq V$.*

We usually want a large, rather than a maximal, independent set. We will use the fact that in bounded degree graphs, any maximal independent set is automatically large.

▶ **Proposition 28.** *If $G = (V, E)$ is a graph with $\deg(v) \leq \Delta$ for all $v \in V$, then any maximal independent set $S \subseteq V$ satisfies*

$$|S| \geq \frac{|V|}{\Delta + 1}.$$

4 The Decision Oracle

We will assume that our algorithm is equipped with an oracle $\mathcal{O}$ which answers the following type of queries: Given a graph $G = (V, E)$ and small weights $w \in \mathbb{Z}^E$, what is the weight of a MWPM in G? We denote the answer by

$$\mathcal{O}(G, w) = \min\{\langle w, x\rangle \mid x \in \mathrm{PM}_G\}.$$

We now list several deterministic NC primitives based on $\mathcal{O}$. Versions of these two lemmas appear implicitly, stated for planar graphs, in [31], but we prove them for the sake of completeness.

▶ **Lemma 29.** *Given access to $\mathcal{O}$, for polynomially bounded $w \in \mathbb{Z}^E$, one can find $E[w]$ in* NC.

Proof. An edge $e = \{u, v\}$ can be in a MWPM if and only if $\mathcal{O}(G, w) = w_e + \mathcal{O}(G - \{u\} - \{v\}, w)$, where $G - \{u\} - \{v\}$ is obtained from G by removing vertices u, v. This can be checked in parallel for all edges e. ◀

▶ **Lemma 30.** *Given access to $\mathcal{O}$, for polynomially bounded $w \in \mathbb{Z}^E$, one can run Algorithm 1 in* NC.

Proof. As was observed by [31], all steps of Algorithm 1 can be run in NC except for finding $E[w]$ and computing $\mu(v)$. Given access to $\mathcal{O}$, we can find $E[w]$ in NC by Lemma 29. Furthermore observe that for any $v \in V$

$$\mu(v) = \min\{\mathcal{O}(G - \{u\} - \{v\}, w) \mid u \in V - \{v\}\},$$

which can be computed by making all queries $\mathcal{O}(G - \{u\} - \{v\}, w)$ in parallel and then taking the minimum. ◀

An implementation for the oracle, in RNC with arbitrarily small inverse polynomial probability of error for general graphs, follows from [27], since they give an RNC algorithm for finding a MWPM for small weights. Since $\mathcal{O}$ is promised to be called at most polynomially many times, the probability of error over the entire run of the algorithm can be made inverse polynomially small.

5 Structural Facts

Our algorithm requires two structural facts, one for the case that the graph G is very sparse and the other for the complementary case. They are encapsulated in Lemmas 32 and 34.

▶ **Definition 31.** *A connected graph $G = (V, E)$ is said to be* very sparse *if $|E| < |V|/(1-\epsilon)$, for some constant $\epsilon < 1/9$.*

▶ **Lemma 32.** *If $G = (V, E)$ is a matching-covered, very sparse graph, then the number of triads in any maximal set of node-disjoint triads in G is at least $c_1|E|$, for some constant $c_1(\epsilon) > 0$.*

The proof of this lemma involves two steps: first, we prove that the total number of triads is large and second, that a maximal node-disjoint set of triads must also be large. The first step is accomplished in the following lemma.

▶ **Lemma 33.** *Suppose that $G = (V, E)$ is a graph with no vertices of degree 0 or 1. Then the number of triads in G is at least $9|V| - 8|E|$.*

Proof. Consider a charging scheme, where we allocate a budget of 1 to each edge, and the edge distributes its budget between its two endpoints. We then sum up the charge on all vertices and use the fact that this sum is exactly $|E|$.

Let $e = \{u, v\}$ be an edge. If neither u nor v is of degree 2, let the edge give $1/2$ to u, and $1/2$ to v. If both u and v are of degree 2, we allocate the budget the same way by splitting it equally between u and v. The only remaining case is when one of u and v has degree 2 and the other has degree at least 3; by symmetry let us assume that $\deg(u) = 2$ and $\deg(v) \geq 3$. Then we allocate $5/8$ to u and $3/8$ to v.

Now let us lower bound the charge that each vertex v receives. Note that the minimum amount v receives from any of its adjacent edges is $3/8$, so an obvious lower bound is $3\deg(v)/8$. If $\deg(v) \geq 3$, this is at least $9/8$. Now consider the case when $\deg(v) = 2$. Then v receives at least $1/2$ from each of its adjacent edges. If one of the neighbors of v is not of degree 2, then the charge that v receives will be at least $1/2 + 5/8 = 9/8$. The only possible case where v does not receive at least $9/8$ is when it is of degree 2, and both of its neighbors are also of degree 2 (the center of a triad), in which case it receives 1.

Now let k be the number of triads. Then, by the above argument the total charge on all the vertices is at least

$$\frac{9}{8}(|V| - k) + k \leq |E|.$$

Rearranging yields $k \geq 9|V| - 8|E|$. ◀

Proof of Lemma 32. We know that the number of triads is at least $9|V|-8|E| = (1-9\epsilon)|E|$. Now consider the conflict graph of triads, where nodes represent triads, and edges represent having an intersection. It is easy to see that any triad can only intersect at most 4 other triads. So the degrees in this conflict graph are bounded by 4. By Proposition 28, any maximal node-disjoint set of triads will contain at least $(1-9\epsilon)|E|/5$ many triads. So we can take $c_1(\epsilon) = (1-9\epsilon)/5$ which is positive for $\epsilon < 1/9$. ◀

▶ **Lemma 34.** *If $G = (V, E)$ is a matching-covered graph on $|V| > 2$ vertices that is not very sparse, then there exist $c_2|E|/\log^2|V|$ edge-disjoint even walks in G, for some constant $c_2(\epsilon) > 0$.*

We first show that there are many edge-disjoint cycles in a non-sparse graph. If at least half of them are even, we are done. Otherwise, we show how to pair up odd cycles and connect them via suitable paths to get sufficiently many edge-disjoint even walks. A proof of the next lemma can be found in [2]; however, for the sake of completeness we provide it here.

▶ **Lemma 35.** *In a graph $G = (V, E)$ there exists a collection of edge-disjoint cycles with at least the following number of cycles:*

$$\frac{|E| - |V|}{2\log_2|V|}.$$

Proof. We prove this by induction on $|V| + |E|$. We have several cases:

- **i)** If there are any loops in the graph, we extract that as one of our cycles, and remove the edge from the graph. The promised quantity goes down by $1/(2\log_2|V|)$ which is $\leq 1/2$. So from now on we assume that G has no loops.
- **ii)** If there are any two parallel edges e, e', we extract those as a cycle of length 2, and remove both from the graph. The promised number of edge-disjoint cycles goes down by $2/(2\log_2|V|) \leq 1$. So adding the cycle we extracted fulfills the promise. From now on we assume that G is simple.
- **iii)** If G has any vertices of degree 0: We can simply remove it and the promised quantity grows.
- **iv)** If G has a vertex of degree 1: We can also remove this vertex. This operation does not change the numerator but shrinks the denominator, which results in a larger promised quantity.
- **v)** If G has a vertex v of degree 2: Let e, e' be the two adjacent edges to v. Remove v, e, e' from the graph, and place a new edge e'' between the two former neighbors of v. By doing this, both $|V|$ and $|E|$ go down by 1. So now the promised number of edge-disjoint cycles becomes larger. By induction we find them, and now we replace the edge e'' if it is used at all in a cycle, by the path of length two consisting of e, e'. Since e'' appears in at most one cycle, this operation preserves edge-disjointness.
- **vi)** Finally if G is a simple graph with no vertices of degree ≤ 2, it must have a cycle of length at most $2\log_2|V|$. If we prove this, we are done by induction, because we can remove the edges of this cycle and the promised quantity goes down by at most 1. Now to prove the existence of this cycle, assume the contrary, that the length of the minimum cycle of the graph is at least $2\log_2|V| + 1$. Pick a vertex v and look at all simple paths of length at most $\log_2|V|$ going out of v. The number of paths of length i is at least twice the number of paths of length $i-1$. This is because every path of length $i-1$ ending at a vertex u can be extended in at least $\deg(u) - 1 \geq 2$ ways, and none of these extensions will intersect themselves, otherwise we would get a cycle of length $\log_2|V| + 1$.

So in the end, the total number of such paths will be $> 2^{\log_2|V|} = |V|$, which means that two of the paths must share an endpoint. But now from the union of these two paths, we can extract a cycle of length at most $\log_2|V| + \log_2|V| = 2\log_2|V|$. ◀

If at least half of the cycles guaranteed by Lemma 35 are odd, we need to pair them up and connect them with paths. We use a spanning tree to do this.

▶ **Proposition 36** ([1, Lemma 20]). *Consider a tree T with an even number of tokens placed on its vertices, with possibly multiple tokens on each vertex. There is a pairing, i.e., a partitioning of tokens into partitions of size two, such that the unique tree paths connecting each pair are all edge-disjoint.*

▶ **Lemma 37.** *Suppose that there are 2ℓ edge-disjoint cycles of odd length in a matching-covered connected graph $G = (V, E)$. Then G contains at least $\Omega(\ell^2/|E|)$ edge-disjoint even walks.*

Proof. We will pair up the odd cycles by paths connecting each pair. This will create ℓ even walks, but they might not be edge-disjoint. We will then show how to extract $\Omega(\ell^2/|E|)$ *edge-disjoint* even walks out of them.

Consider a spanning tree T of G. For each of the 2ℓ odd cycles, pick an arbitrary vertex, and put a token on that vertex. Now we have an even number of tokens on the vertices. We can pair up these tokens, so that the unique tree paths (of possibly length 0) connecting each pair are edge-disjoint, see Proposition 36.

Now for each pair of odd cycles C_1, C_2 whose tokens got paired up, we create an even walk. Let P be the tree path connecting tokens from C_1 and C_2. If P has no common edges with C_1, C_2 we can simply create our even walk, but this is not guaranteed to happen. So instead, traverse P from C_1's token to C_2's token and look at the last exit from C_1; afterwards look for the first time any vertex of C_2 is visited. This portion of P is a subpath connecting C_1 and C_2 having no common edged with either. We use C_1, C_2 and this subpath of P to create our even walk.

So far we have created ℓ even walks, but they might not be edge-disjoint. The odd cycles are edge-disjoint, as are the paths connecting them, but one of the paths might share an edge with an unrelated odd cycle. This also means that no edge e can be shared between more than two even walks; e can be used once as part of an odd cycle, and once as part of a path.

Now consider the number of edges in each even walk. If we sum this over all even walks, we get at most $2|E|$, since each edge can appear in at most two even walks. So the average number of edges in an even walk is $\leq 2|E|/\ell$. By Markov's inequality at least half of the even walks, $\ell/2$ of them, will have at most twice this average number of edges, $4|E|/\ell$. Now create a conflict graph where nodes represent these $\ell/2$ even walks, and an edge is placed when the two even walks share an edge. The degree of each node is at most $4|E|/\ell$. So if pick a maximal independent set in this conflict graph, it will consist of at least $\Omega(\ell^2/|E|)$ many even walks. ◀

We are finally ready to prove Lemma 34.

Proof of Lemma 34. First note that if our graph is not an isolated edge and is matching-covered it must contain at least one even cycle. This is so because there must be at least two perfect matchings in the graph, and in their symmetric difference, we can find one such cycle.

Because we are guaranteed to have at least 1 cycle, we can simply show that asymptotically we can extract $\Omega(|E|/\log^2|V|)$ edge-disjoint even walks. Then the asymptotic statement translates to the more concrete bound of $c_2|E|/\log^2|V|$.

If $(1-\epsilon)|E| \geq |V|$, by Lemma 35, we have at least $\epsilon|E|/2\log_2|V| = \Omega(|E|/\log|V|)$ cycles. If at least half of them are of even length, we are done. Otherwise we get $\Omega(|E|/\log|V|)$ odd cycles. Perhaps by throwing away one of them, we can assume the number of odd cycles we have is even. Then we can apply Lemma 37 to obtain $\Omega(|E|/\log^2|V|)$ edge-disjoint walks. This completes the proof. ◀

6 The Oracle-Based Algorithm

In this section we describe our oracle-based algorithm for finding a perfect matching. In Section 6.1, we will extend this to finding a *minimum weight* perfect matching for small weights.

On input $G = (V, E)$, our algorithm proceeds by finding smaller and smaller matching minors H of G, until H has a unique perfect matching, or in other words is a perfect matching. Then we pick the edges in H as a partial matching in G and extend this partial matching to a perfect matching independently and in parallel for the preimage of each node in H. That is for each node s in H, we take the set $S \subseteq V$ that got shrunk to s, remove the single endpoint of the partial matching from S, and recursively find a perfect matching in S. In the end we return the results of all these recursive calls along with the edges of H as the final answer.

We crucially make sure that the pre-image of nodes in H never contain more than a constant fraction of V. This makes sure that our recursive calls end in $O(\log|V|)$ steps.

In all of our algorithms, when we construct matching minors, we implicitly maintain the mapping from the resulting edges to the original edges, and the mapping from original vertices to the minor's vertices. These are trivial to maintain in NC, but for clarity we avoid explicitly mentioning them. We also keep node weights for matching minors, where the *weight of a node* is simply the number of original vertices that got shrunk to it.

Algorithm 2 Divide-and-conquer algorithm for finding a perfect matching.

```
PerfectMatching(G = (V, E))
if V = ∅ then
    return ∅.
else
    Call PartialMatching(G), and let H be the matching minor returned.
    Let M ⊆ E be the edges of H.
    for each node s of H in parallel do
        Let S ⊆ V be the nodes of G that are shrunk to s.
        Let v be the unique endpoint of the unique edge of M in δ(S).
        Let G_s be the induced graph on S − {v}.
        M ← M ∪ PerfectMatching(G_s).
    end
    return M.
end
```

The pseudocode for the main algorithm PerfectMatching can be seen in Algorithm 2. On input G, the algorithm calls PartialMatching to find a matching minor H of G which itself is a perfect matching. Then the edges of H, which form a partial matching in G, are extended to a perfect matching independently and in parallel in the preimage of each node from H. Since H is a matching minor, this extension can always be performed by Lemma 20.

The pseudocode for PartialMatching can be seen in Algorithm 3. This algorithm keeps a node-weighted matching minor of the input graph G. It tries several ways of obtaining a smaller matching minor, where *size of a matching minor* is measured in terms of the number

Algorithm 3 Find a matching minor of the input graph that is itself a perfect matching.

PartialMatching($G = (V, E)$)
Assign node weight 1 to each node $v \in V$.
while *G is not a perfect matching* **do**
  if *any node v of G has at least $1/6$ of the total node weight* **then**
    Remove disallowed edges $e \notin E[0]$ from G.
    Contract the complement of $\{v\}$ to a single node. If there are parallel edges, remove all except for an arbitrary one.
    return G.
  end
  Find a maximal set of node-disjoint triads in G.
  Let H be obtained from G by removing disallowed edges and contracting each triad into a single node.
  $U \leftarrow \{H\}$.
  for $w \in \mathcal{W}$ **in parallel do**
    Call Reduce(G, w) and let the result be H.
    $U \leftarrow U \cup \{H\}$.
  end
  Find the graph $H \in U$ with the minimum number of non-isolated edges.
  $G \leftarrow H$.
end
return G.

Algorithm 4 Remove disallowed edges and contract certain tight odd sets.

Reduce($G = (V, E)$, w) ; // The graph G has node weights.
Remove disallowed edges $e \notin E[w]$ from G.
Find all connected components of G.
for *each connected component C of G* **in parallel do**
  Run Algorithm 1 on C to find a laminar family of tight odd sets $\mathcal{L}$.
  for $S \in \mathcal{L}$ **in parallel do**
    if *node weight of S is more than half of the node weight of C* **then**
      Replace S in $\mathcal{L}$ with $C - S$.
    end
  end
  Find the inclusion-wise maximal sets in $\mathcal{L}$ and shrink each one to a single node.
end
return G.

of non-isolated edges, see Definition 13. One way of obtaining a smaller matching minor is by picking a maximal node-disjoint set of triads and shrinking them simultaneously. By Lemma 22, this produces a matching minor. Also note that the maximal set of node-disjoint triads can be found in NC by enumerating all triads and using Theorem 27.

Another way of obtaining smaller matching minors is by trying weights from the set of weight vectors $\mathcal{W}$ and calling Reduce to remove disallowed edges $e \notin E[w]$ and shrinking top-level sets of a laminar family of tight odd sets w.r.t. w.

Finally. the pseudocode for Reduce can be seen in Algorithm 4. This algorithm is simply fed a graph $G = (V, E)$ and a weight vector w. It removes disallowed edges $e \notin E[w]$ and shrinks the maximal sets of a laminar family of tight odd set. The laminar family is found using Algorithm 1, but is modified to make sure that no shrunk set becomes too large; to be more precise no shrunk vertex in the end will have node weight more than half of the total node weight.

6.1 Finding a minimum weight perfect matching

We extend our algorithm so it returns not just any perfect matching, but rather a minimum weight perfect matching, for small weights.

Given an input graph $G = (V, E)$ and a weight vector w, we can remove disallowed edges $e \notin E[w]$, and find a laminar family of tight odd sets $\mathcal{L}$ w.r.t. w, by calling Algorithm 1 on each connected component of G. By complementary slackness, any perfect matching that has only one edge in $\delta(S)$ for each $S \in \mathcal{L}$ will automatically be of minimum weight, see Definition 15. We can simply contract the top level sets in $\mathcal{L}$, use Algorithm 2 to find a perfect matching in the shrunk graph, and recursively extend this to a minimum weight perfect matching in each shrunk piece. Following an almost identical argument as in the proof of Proposition 18, the perfect matching in the shrunk graph can be extended to a minimum weight perfect matching.

The only problem with this method is that the recursion depth is not guaranteed to be polylogarithmic. However we can fix that by making sure that tight odd sets $S \in \mathcal{L}$ do not have more than half of the vertices in the graph; if they do, we replace them by their complements and we will see in Lemma 40 why this operation preserves laminarity.

6.2 Minor-closed families of graphs

Throughout our algorithm we only call the decision oracle on graphs obtained from the original through a sequence of edge and vertex removals and contractions. In this section we will prove that the decision oracle is only called on minors of the original graph, that is those graphs obtained by vertex and edge removals and contractions of *connected* subgraphs.

▶ **Lemma 38.** *Algorithms 2 to 4 call the decision oracle on minors of their input graph only.*

This lemma is all we need to prove Theorem 4. Note that there are several minor-closed families of graphs where the decision problem can be solved in NC by using a counting oracle. In particular we can count perfect matchings in graphs embedded on surfaces of genus at most $O(\log n)$, and therefore solve the decision problem, all in NC. This improves upon the genus bound of $O(\sqrt{\log n})$ given by [1].

▶ **Corollary 39.** *For graphs embedded on a surface of genus at most $O(\log n)$ and weighted with polynomially bounded edge weights, there is an NC algorithm to find a minimum weight perfect matching.*

Another consequence of Theorem 4 is an alternative algorithm for $K_{3,3}$-free graphs, which was resolved earlier by [7].

Now we prove Lemma 38.

Proof of Lemma 38. First we prove this for Algorithm 4. In this algorithm, we only remove edges from the input graph, and shrink tight odd sets in connected components. We just have to show that what we shrink is already connected. Consider a tight odd set S in a connected component C. If it is not internally connected, then one of its internal connected components must have odd size; let that be S'. Since $S \subseteq C$ and C is a connected component, there is an edge $e \in \delta(S - S')$. Since S' is not internally connected to $S - S'$, it must be that $e \in \delta(S)$ too. Now since the graph is matching-covered with minimum weight perfect matchings, there must be some minimum weight perfect matching $M \ni e$. But because S' is odd, there must also be an edge $f \in M \cap \delta(S')$. But note that $e \neq f$, and both $e, f \in \delta(S)$. This is a contradiction, since S cannot have more than one edge in a perfect matching. This shows that S must be connected and Algorithm 4 only produces minors of its input graph.

Next we prove the statement for Algorithm 3. This algorithm either calls Algorithm 4, or finds triads and contracts them. The former produces minors of the input graph, and the latter also produces minors of the input graph since triads are connected.

Note that the graph returned by Algorithm 3 may not be a proper minor of the input graph; that could happen if the node weight of some v goes above $1/6$ the total node weight. In this scenario, the complement of v might not be connected and yet we contract it. However the algorithm immediately returns and the decision oracle is not called on this returned graph. So this does not contradict the statement of the lemma.

Finally we prove the statement for Algorithm 2. The only graphs produced and passed onto Algorithm 3 are obtained from the input graph by vertex removals and edge removals. So they are all minors of the input graph. The output of Algorithm 3 might not be a proper minor, but this output is only used to decide which edges and vertices to remove from the original graph to get to induced graphs on $S - \{v\}$. ◀

7 Analysis of the algorithm

First we will prove that our oracle-based algorithm returns a correct answer. Next, we will bound the running time and prove that our algorithm runs in NC, modulo the calls to $\mathcal{O}$; this constitutes the most challenging part of the analysis.

7.1 Correctness

We will need the following lemma.

▶ **Lemma 40.** *Suppose that $\mathcal{L}$ is a laminar family of sets in a node-weighted graph $G = (V, E)$, and we replace every $S \in \mathcal{L}$ whose node weight is larger than half of the total node weight by the complement, i.e., $V - S$. Then the resulting family of sets $\mathcal{L}'$ is also laminar.*

Proof. Let S, S' be two sets in $\mathcal{L}$. They are either disjoint or one is contained in the other.

If $S \cap S' = \emptyset$: They cannot both have node weight more than $1/2$. So at most one of them gets replaced by its complement. Then it is easy to see that the resulting sets do not cross.

If $S \subseteq S'$: There are three possibilities. If none of them gets replaced by their complements, or both of them get replaced by their complements, they remain nested and therefore do not cross. If one of them gets replaced by its complement, it has to be the larger set S'. In that case the resulting sets become disjoint, and still do not cross. ◀

Using Lemma 40 and Lemma 20, we deduce that Reduce always returns a matching minor of its input graph. By definition, PartialMatching also returns a matching minor of its graph when it finishes (for the analysis of running time see Section 7.2).

This proves the correctness of the algorithm, since we always find a matching minor that has a unique perfect matching (itself), and by Lemma 20, we can extend it to a perfect matching, independently in the preimage of each node.

7.2 Running time

First we analyze PerfectMatching (Algorithm 2) assuming the calls to PartialMatching (Algorithm 3) are in NC.

▶ **Lemma 41.** *Assuming the calls to* PartialMatching *are in* NC*,* PerfectMatching *is in* NC*.*

Proof. We simply need to bound the number of levels in the recursion. We will prove that when PartialMatching returns a matching minor H, the node weight of every node is at most $5/6$ the total node weight. This proves that in each recursive call to PerfectMatching, the number of vertices gets reduced by a factor of $5/6$.

Note that the first time in Algorithm 3 that a node's weight goes above $1/6$ the total weight, the algorithm stops and returns a two-node minor. So we just need to prove that the weight of the node that just went above $1/6$ is not more than $5/6$. The current minor was obtained from the previous minor by either Reduce, or by shrinking triads. But Reduce never creates nodes with weight more than half the total weight. The weight of each node in a triad is also at most $1/6$ the total weight, so after shrinking the triad, the new weight can be at most $1/6 + 1/6 + 1/6 = 1/2$ the total weight. This finishes the proof. ◀

Finally, we need to prove that PartialMatching finishes in a polylogarithmic number of steps. Using the structural facts, Lemma 42 and Lemmas 32 and 34, we establish the following lemma.

▶ **Lemma 42.** *In each iteration of Algorithm 3, the number of non-isolated edges gets reduced by a factor of* $1 - \Omega(1/\log^2|V|)$*.*

Proof. First assume G is a connected graph. Then we can directly apply Lemmas 32 and 34 for some fixed $\epsilon < 1/9$ to show that we either find $c_1|E|$ triads or there exist $c_2|E|/\log^2|V|$ edge-disjoint even walks. In the former case, after contracting the triads, the number of edges gets reduced by a factor of $1 - c_1$. In the latter case, let $C_1, C_2, \ldots, C_k$ be the edge-disjoint even walks, and let $w \in \mathcal{W}$ be the weight vector such that $\langle w, \text{sign}(C_i)\rangle \neq 0$. Note that w is guaranteed to exist by Lemma 26. In the call to Reduce(G, w), every C_i loses at least edge by Lemmas 24 and 25, either because one of its edges becomes disallowed or it gets shrunk as a result of shrinking top-level tight odd sets. Therefore, one of the candidate graphs in U in Algorithm 3 will have a factor of $1 - c_3/\log^2|V|$ fewer edges, for some constant $c_3 > 0$.

Next assume G is not connected. If so, we apply the above-stated argument to each connected component that is not an isolated edge. We can further assume the same weight vector w works for all connected components. Now if H_1 is the graph obtained from shrinking triads, and H_2 is the result of Reduce(G, w), then we know that the average number of edges in H_1 and H_2 for each connected component is at most $1 - c_3/2\log^2|V|$ times the number of edges in the connected component. So one of H_1, H_2 must have at most $(1 - c_3/2\log^2|V|)$ times as many non-isolated edges as G. ◀

Note that Lemma 42 gives a polylogarithmic upper bound on the number of iterations in Algorithm 3, since if we track the number of non-isolated edges, after every $\Theta(\log^2|V|)$ steps we get a constant factor reduction, and therefore it takes at most $O(\log|E| \cdot \log^2|V|)$ iterations for it to reach 0.

8 Discussion

This paper has identified what appears to be the "core" of the difficult open problem of obtaining an NC matching algorithm, namely the decision problem. We must immediately mention that both decision problems stated in Section 1.2 have been the subject of numerous attacks over the past decades and hence resolution is not likely to be an easy matter. At the same time, we hope that since the "target" has been more precisely identified, the resolution of the open problem will gain added impetus.

An obvious open question is to build on the quasi-NC algorithm of [11] and related results of [12] to obtain the appropriate oracle-based NC algorithms and pseudo-deterministic RNC algorithms for linear matroid intersection and for finding a vertex of a polytope with faces given by totally unimodular constraints. An interesting problem defined by Papadimitriou and Yannakakis [29] , called Exact Matching, is the following: Given a graph G with a subset of the edges marked red and an integer k, find a perfect matching with exactly k red edges. This problem is known to be in RNC [27], even though it is not yet known to be in P. Is there a pseudo-deterministic RNC algorithm for it?

The phenomenon identified in Section 1.3 clearly deserves to be studied in depth. To the best of our knowledge, there are only two algorithmic results for bipartite matching that have not been extended to general graphs. The first is obtaining a fully polynomial randomized approximation scheme for counting the number of perfect matchings [14]; this is also among the outstanding open problems of theoretical computer science today. The second is obtaining an $O(m^{10/7})$ algorithm for maximum matching [23], which beats the earlier algorithms for sparse graphs.

References

1. Nima Anari and Vijay V. Vazirani. Planar Graph Perfect Matching is in NC. In *Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 2018.
2. Alberto Caprara, Alessandro Panconesi, and Romeo Rizzi. Packing cycles in undirected graphs. *Journal of Algorithms*, 48(1):239–256, 2003.
3. Laszlo Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, 1976.
4. Marek Cygan, Harold N Gabow, and Piotr Sankowski. Algorithmic Applications of Baur-Strassen's Theorem: Shortest Cycles, Diameter and Matchings. *arXiv preprint*, 2012. `arXiv:1204.1616`.
5. Samir Datta, Raghav Kulkarni, and Sambuddha Roy. Deterministically isolating a perfect matching in bipartite planar graphs. *Theory of Computing Systems*, 47(3):737–757, 2010.
6. Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.
7. David Eppstein and Vijay V. Vazirani. NC Algorithms for Computing a Perfect Matching, Number of Perfect Matchings, and a Maximum Flow in One-Crossing-Minor-Free Graphs. In *Proceedings of the Thirty-First ACM Symposium on Parallelism in Algorithms and Architectures*, 2019.

8 Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-NC. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 754–763. ACM, 2016.

9 Eran Gat and Shafi Goldwasser. Probabilistic Search Algorithms with Unique Answers and Their Cryptographic Applications. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 136, 2011.

10 Shafi Goldwasser and Ofer Grossman. Perfect Bipartite Matching in Pseudo-Deterministic RNC. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 22, page 208, 2015.

11 Rohit Gurjar and Thomas Thierauf. Linear matroid intersection is in quasi-NC. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 821–830. ACM, 2017.

12 Rohit Gurjar, Thomas Thierauf, and Nisheeth K Vishnoi. Isolating a vertex via lattices: Polytopes with totally unimodular faces. *arXiv preprint*, 2017. `arXiv:1708.02222`.

13 John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

14 Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM (JACM)*, 51(4):671–697, 2004.

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

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

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

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

19 Alexander V Karzanov. O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh; title translation: On finding maximum flows in networks with special structure and some applications. *Mathematicheskie Voprosy Upravleniya Proizvodstvom*, 1973.

20 László Lovász. On determinants, matchings, and random algorithms. In *FCT*, volume 79, pages 565–574, 1979.

21 László Lovász and Michael D Plummer. *Matching theory*, volume 367. American Mathematical Soc., 2009.

22 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

23 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 253–262. IEEE, 2013.

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

25 Silvio Micali and Vijay V Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 17–27, 1980.

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

27 Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

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

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

30 Michael O Rabin and Vijay V Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, 1989.

31 Piotr Sankowski. NC Algorithms for Weighted Planar Perfect Matching and Related Problems. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

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

33 Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

34 Vijay V Vazirani. NC algorithms for computing the number of perfect matchings in K3, 3-free graphs and related problems. *Information and computation*, 80(2):152–164, 1989.

35 Vijay V Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{|V|}|E|)$ general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, 1994.