Algebraic Algorithms for Matching

Ioana Ivan, Madars Virza, Henry Yuen

December 14, 2011

1 Introduction

Given a set of nodes and edges between them, what's the maximum of number of disjoint edges? This problem is known as the *graph matching problem*, and its study has had an enormous impact on the develpoment of algorithms, combinatorics, optimization theory, and even complexity theory. Mathematicians have been interested in the matching problem since the 19th century, leading to celebrated theorems in graph theory due to Tutte, Menger, König, and Egerváry in the mid-twentieth century[18] However, the algorithmic aspects of the matching problem were not examined until the 60's, and the first efficient algorithm for the (unweighted) matching problem was described in Jack Edmond's seminal 1965 paper "Paths, Flowers and Trees" [4][1]. Since Edmond's paper, there has been a long and extensive literature on the algorithms for computing matchings. There are many different types of matching problems: *bipartite* versus *non-bipartite*, *perfect matching* versus *maximum matching*, *weighted* versus *unweighted*, and so on. Many of the algorithms developed to solve graph matching have been *combinatorial*, meaning the operation and analysis involve graphs and combinatorial structures. Examples include Edmond's algorithm [4], Micali and Vazirani's [10] algorithm, and the Ford-Fulkerson maximum flow algorithm applied to bipartite matching. However, many modern algorithms for matching are *algebraic*, involving techniques that superficially have little to do with graphs and matchings: computing determinants, updating matrix inverses, creating random matrices. The use of algebra has led to elegant and fast matching algorithms - and in certain settings, the fastest known. The development of algebraic matching algorithms has in turn spurred advances in linear algebraic techniques that have wide applicability to algorithms design for other combinatorial and graph-theoretic problems. In this survey, we explore an old combinatorial problem, but through a recent algebraic lens.

Nearly all the algebraic methods for solving matching have taken advantage of an important connection between determinants and matching discovered by Tutte in 1947 [18]: an undirected simple graph $G = (V, E)$ has a perfect matching if and only if its associated *Tutte matrix* is nonsingular. This characterization of the existence of perfect matchings immediately leads to a simple randomized algorithm for testing whether a graph has a perfect matching, and serves as the foundation for algorithms to find such a matching (if it exists), to find a maximum matching, and more. The Tutte matrix will also serve as a starting point for our survey.

The second key piece of technology used for the results covered here is the equivalence between computing the determinant, multiplying two matrices, inverting a matrix, and performing Gaussian Elimination. It is an extremely interesting and nontrivial fact that these problems all take $O(n^\omega)$ time, where ω is called the *matrix multiplication constant*, and is somewhere between 2 and 2.373 [20][2].

The paper will proceed as follows: (1) Using the Tutte matrix to detect and construct a perfect matching. (2) Computing maximum matchings via Gaussian Elimination in $O(n^\omega)$ time. (3) Computing perfect matchings via "lazy updates" of the inverse Tutte matrix in $O(n^\omega)$ time. (4) Solving the dynamic matching problem via a dynamic matrix inverse updating algorithm. (5) Counting perfect matchings in a graph via the determinant. (6) Open problems.

[1] This remarkable paper not only gave an efficient algorithm for matching, it was the first to articulate what "efficient algorithm" should even mean! In it, Edmonds essentially defined the class now known as $\mathcal{P}$, the set of polynomial-time solvable problems, and gave some philosophical arguments for why this class should be equated with efficient computation. As we know, Edmonds gave an extremely fruitful definition of efficient computation.

[2] We will assume that ω is strictly greater than 2; otherwise some of the time bounds here have additional logarithmic factors which we shall omit for simplicity.

The idea of using linear algebra to solve combinatorial problems is a very powerful one, and not restricted to the matching problem. For example, the best known algorithms for the sparse cut problem, counting the number of spanning trees, computing approximate max flow, and triangle finding in dense graphs, all use algebraic techniques extensively [9][3]. It appears that we are only beginning to understand the power of the "linear algebra method" for algorithms design.

2 Detecting and Constructing Perfect Matchings: A First Pass

2.1 The Tutte matrix

We begin by defining the matching problem formally.

Definition 2.1 (The matching problem)**.** *Let $G = (V, E)$ be an undirected simple graph. Then a* matching *of G is a subset M of E such that no two edges in M share a vertex. A* perfect matching *M is one where for every vertex $v \in V$, there is an edge $e \in M$ that is adjacent to v. A* maximum matching *M is one such that for all matchings M' of G, $|M| \geq |M'|$.*

Definition 2.2 (Tutte matrix)**.** *Let $G = (V, E)$ be an undirected simple graph, with $|V| = n$. Then the* Tutte matrix *of G is an $n \times n$ matrix T with entries:*

$$T_{ij} = \begin{cases} x_{ij} & \text{if } (i,j) \in E \text{ and } i < j \\ -x_{ij} & \text{if } (i,j) \in E \text{ and } i > j \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

where the x_{ij} are formal variables (i.e. not instantiated with a particular value).

In [18], Tutte demonstrated a striking and elegant connection between the existence of a perfect matching in a graph and the determinant of its associated Tutte matrix:

Theorem 2.1 (Tutte)**.** *Let $G = (V, E)$ be an undirected simple graph, and let T be its associated Tutte matrix. Then, $\det(T) \not\equiv 0$ if and only if G has a perfect matching.*

Proof. Writing out the definition of $\det(T)$:

$$\begin{aligned} \det(T) &= \sum_{\sigma \in S_n} \text{sgn}\,(\sigma) \prod_i T_{i\sigma(i)} \\ &= \sum_{\sigma \in S_n} \text{sgn}\,(\sigma) \prod_i (\pm \text{There exists an edge between vertex } i \text{ and } \sigma(i)) \end{aligned}$$

where S_n is the group of all permutations on n elements, and sgn (σ) denotes the parity of a permutation $\sigma \in S_n$. First, observe that any permutation σ where there is an i with $(i, \sigma(i)) \notin E$ contributes 0 to $\det(T)$. Call all other σ *feasible*. A feasible σ corresponds to a selection of edges $M_\sigma = \{(i, \sigma(i)) \mid 1 \leq i \leq n\}$. Suppose M_σ was not a perfect matching. Then consider σ^{-1}, the inverse permutation of σ. If M_σ is not a perfect matching, then there is a term in the determinant sum corresponding to σ^{-1} that's separate from the σ term: there must be an vertex i such that $\sigma(i) \neq \sigma^{-1}(i)$ (otherwise σ would be a perfect matching). Thus $\sigma \neq \sigma^{-1}$. Note that $M_{\sigma^{-1}} = M_\sigma$, and thus $\prod T_{i\sigma(i)} = \prod T_{i\sigma^{-1}(i)}$. However, sgn $(\sigma^{-1}) = -$sgn (σ). Thus, all non-matchings contribute exactly 0 to $\det(T)$.

Now suppose that M_σ were a perfect matching. This implies that $\sigma(i) = \sigma^{-1}(i)$ for all i. Thus $\sigma = \sigma^{-1}$. Observe that sgn $(\sigma) \prod T_{i\sigma(i)} = \pm x_{e_1}^2 \cdots x_{e_{n/2}}^2$, where $M_\sigma = \{e_1, \ldots, e_{n/2}\}$ is the perfect matching. Note that no other permutation $\tau \in S_n$ will induce a term with that particular monomial in the determinant sum, ensuring that $\det(T)$ is a nonzero polynomial. □

It is important to note that the Tutte matrix T is not a matrix of numbers, but rather is a *formal* matrix in the sense that its entries are variables. Thus, the determinant of a Tutte matrix is not a number either, but a polynomial in the variables in the matrix. When we write $\det(T) \equiv 0$, we mean that the polynomial is the *identically* 0 polynomial (its coefficients are all 0).

2.2 Detecting the presence of perfect matchings

This theorem almost immediately gives us an algorithm to detect whether a graph G has a perfect matching: compute the Tutte matrix T, compute $\det(T)$, and test whether $\det(T)$ is the 0 polynomial or not. However, there is a significant barrier to this proposed algorithm becoming efficient: T is a formal matrix, so computing $\det(T)$ could take exponential time. Indeed, even *writing down* $\det(T)$ could take superpolynomial time. Suppose that G were $K_{n/2,n/2}$, the complete bipartite graph. There are $(n/2)!$ matchings, and hence more than $(n/2)!$ terms in $\det(T)$.

However, we don't necessarily need to compute polynomial corresponding to $\det(T)$ explicitly; we simply need to test whether that polynomial is identically 0. Fortunately, we have an efficient way to accomplish this - provided we use randomization. Suppose G had no perfect matching. Then $\det(T)$ is the 0 polynomial, so for every numerical assignment to the variables x_{ij} in T, evaluating $\det(T)$ on that assignment will yield 0. If G had a perfect matching, then $\det(T)$ is not identically 0, so there must be *some* assignment to the variables x_{ij} such that the evaluation of $\det(T)$ on that assignment will be nonzero. The following theorem strengthens this by saying that for nonzero $\det(T)$, *most* assignments to the variables x_{ij} cause $\det(T)$ to evaluate to a nonzero value.

Theorem 2.2 (Schwartz-Zippel)**.** *Suppose $\det(T) \not\equiv 0$; then suppose each variable in T was set to an element in $\{1, \ldots, n^2\}$ uniformly at random. Then, $\Pr[\det(T)(x) = 0] \leq 1/n$, where the randomness is taken over the setting of each variable.*

This is a restatement of the celebrated Schwartz-Zippel lemma [17], which has diverse application to many areas of theoretical computer science. For a succinct proof that uses field theory, see [11].

The following randomized algorithm was presented first by Lovász [7]: Given a graph G, compute its Tutte matrix T. Substitute randomly chosen values from $\{1, \ldots, n^2\}$ for the variables in T. Compute the determinant of the "instantiated" Tutte matrix T. If the determinant is 0, then reject. If the determinant is nonzero, then accept. With high probability, this algorithm will accept graphs with perfect matchings. The algorithm will never accept graphs without a perfect matching. The probability of failure can be made exponentially small by repetition. The running time of this algorithm is bounded by $O(n^\omega)$, the time it takes to compute the determinant of an $n \times n$ matrix (that is instantiated with numbers for entries, not formal variables).

2.3 Constructing a perfect matching

The previous algorithm only detects the presence of a perfect matching in G, with high probability. What if you want to construct a perfect matching, given that there is one? The previous algorithm can be extended in a natural way to accomplish this, and the ideas used here are central to the more advanced algorithms covered later in this survey.

Rabin and Vazirani developed the following algorithm, which uses the perfect matching detection algorithm by Lovász recursively. Suppose we have established that a graph G has a perfect matching, and suppose that we have even identified an edge e that belongs in a perfect matching of G (e is then called *admissible*). The key observation is that the subgraph G' of G obtained by removing e and all edges adjacent to e also has a perfect matching. Any perfect matching of G', combined with e will form a perfect matching for G. Provided that we have a method for identifying whether an edge is in a perfect matching, we have a recursive algorithm for constructing a perfect matching. Rabin and Vazirani provides this method; the *inverse* of the Tutte matrix contains information about admissible edges. They make use of the following linear algebra fact to compute the inverse:

Fact 1 (Adjoint formula)**.** *For any non-singular $n \times n$ matrix A we have $(A^{-1})_{i,j} = \frac{(adj\ A)_{i,j}}{\det A}$, where the value of $(adj\ A)_{i,j}$ (called the adjoint) is the determinant of A after i^{th} row and j^{th} column have both been deleted.*

The following theorem immediately follows from the adjoint formula and Theorem 2.1:

Theorem 2.3 (Rabin, Vazirani)**.** *Let $G = (V, E)$ be an undirected simple graph having a perfect matching and T be its associated Tutte matrix. Then $(T^{-1})_{i,j} \neq 0$ if and only if $G - \{i, j\}$ has a perfect matching.*

Proof. Suppose $G - \{i, j\}$ has a perfect matching M. Then if vertices i and j are added back to $G - \{i, j\}$, (i, j) will not be incident on any edge in M. $M' = M \cup \{(i, j)\}$ will be a perfect matching for G. Since G has a perfect matching, T_G^{-1} exists and by the adjoint formula, $(T^{-1})_{i,j} = \frac{(adjT)_{i,j}}{\det T}$. Since G has a perfect matching M', $\det T \neq 0$. Since $G - \{i, j\}$ has a perfect matching M, $(\text{adj } T)_{i,j} \neq 0$. Thus $(T^{-1})_{i,j} \neq 0$.

Now suppose G has a perfect matching and $(T^{-1})_{i,j} \neq 0$. Then the adjoint formula says that $(\text{adj } T)_{i,j} \neq 0$, which implies that $G - \{i, j\}$ has a perfect matching.

□

From this theorem naturally follows an iterative algorithm that recursively removes edges in a perfect matching, as described above. We present the Rabin-Vazirani algorithm below formally:

Algorithm 1 Rabin-Vazirani algorithm

```
M ← ∅
while G is non-empty do
    Compute T_G, and instantiate each of the variables with a random value from {1, ..., n^2}
    Compute T_G^{-1}
    Find i, j such that (v_i, v_j) ∈ G and (T_G^{-1})_{i,j} ≠ 0
    M ← M ∪ {(v_i, v_j)}
    G ← G − {v_i, v_j}
end while
return M
```

Note that the Rabin-Vazirani algorithm is randomized, just as the Lovász algorithm is. The algorithm will find a perfect matching with constant probability, but this can be amplified to an arbitrarily high (constant) amount via (constant) repeated trials. One trial of the Rabin-Vazirani algorithm takes $O(n^{\omega+1})$ time to compute the perfect matching, because matrix inversion can be performed in $O(n^{\omega})$ time and all other operations in each of $O(n)$ loop iterations are dominated by matrix inversion.

3 Maximum Matchings via Gaussian Elimination

3.1 O(n^3) algorithm for general undirected graphs

The seminal paper of Mucha and Sankowski not only introduced the fastest algorithms for perfect and maximum matchings, but also gave a very simple extension of the Rabin-Vazirani algorithm, which achieves $O(n^3)$ running time and which we present below.

3.2 Speeding up Rabin-Vazirani

Mucha and Sankowski noticed that the Rabin-Vazirani algorithm is inefficient in that it recomputes the inverse of the Tutte matrix at each iteration, when two rows and two columns are removed (corresponding to removing vertices i and j from the graph). Each recomputation is expensive and fails to take advantage of the relation between the Tutte matrices in each iteration; the $(r+1)$th Tutte matrix T_{r+1} is simply T_r with two rows and two columns removed (corresponding to the vertices i, j removed). Intuitively, there should be some relationship between T_r^{-1} and T_{r+1}^{-1}. Indeed, there is: Mucha and Sankowski showed that T_{r+1}^{-1} is obtained by essentially performing Gaussian Elimination on T_r^{-1}. This key insight allows us to speed up the update procedure. They use the following theorem from linear algebra.

Theorem 3.1 (Elimination theorem)**.** *Let A be a nonsingular $n \times n$ matrix. Then we can write $A = \begin{pmatrix} a_{1,1} & v^\top \\ u & B \end{pmatrix}$ and $A^{-1} = \begin{pmatrix} \hat{a}_{1,1} & \hat{v}^\top \\ \hat{u} & \hat{B} \end{pmatrix}$, $a_{1,1}, \hat{a}_{1,1}$ are numbers, and $u, \hat{u}, v$, and $\hat{v}$ are vectors of length $n-1$. Furthermore, $\hat{a}_{1,1} \neq 0$. Then $B^{-1} = \hat{B} - \hat{u}\hat{v}^\top / \hat{a}_{1,1}$.*

This theorem gives us a way to update the inverse of a matrix, after deleting a row/column pair, without recomputing the inverse from scratch. How is this related to Gaussian elimination? If we think of the columns as being variables, and the rows as being equations, the procedure above corresponds to eliminating the first variable using the

first equation. Equipped with this theorem we can present an $O(n^3)$ algorithm for perfect matchings due to Mucha and Sankowski:

Algorithm 2 $O(n^3)$ algorithm for finding a perfect matching

1: $M \leftarrow \emptyset$
2: $A \leftarrow T_G^{-1}$
3: **for** $i \leftarrow 1 \dots n$ **do**
4: find j such that $(v_i, v_j) \in G$ and $A_{i,j} \neq 0$
5: $M \leftarrow M \cup \{(v_i, v_j)\}$
6: $G \leftarrow G - \{v_i, v_j\}$
7: eliminate i-th row and j-th column of A
8: eliminate j-th row and i-th column of A
9: **end for**
10: **return** M

Remember than in our algorithm for finding a perfect matching, every time we remove an edge, we have to remove all edges adjacent to both its endpoints, so we need to perform the elimination procedure twice. One concern might be that, after the first application of the procedure, $T_{j,i}^{-1}$ is now 0, so we can't perform the elimination again. But $T_{i,i}^{-1} = T_{j,j}^{-1} = 0$ (because otherwise we would have a perfect matching both in the original graph and the graph with vertex i removed by Tutte's theorem, which is not possible), so the value of $T_{j,i}^{-1}$ changes by $0 \cdot 0 / T_{i,j}^{-1} = 0$ during the first elimination. This algorithm is the Rabin-Vazirani algorithm implemented using dynamic rank-1 updates.

3.3 O(n^ω)algorithm for bipartite graphs

The idea of updating the inverse via Gaussian elimination is central to Mucha's and Sankowski's $O(n^\omega)$ algorithm for finding a maximum matching in a bipartite graph.

To do this we need to introduce an analogue of the Tutte matrix for bipartite graphs: the Edmond's matrix, for which analogues of Theorem 2.1 and Theorem 2.3 hold:

Definition 3.1 (Edmond's matrix)**.** *Let $G = (U \cup V, E)$ be undirected bipartite graph, with $|U| = |V| = n$. Then the* Edmonds matrix *of G is an $n \times n$ matrix $\mathfrak{D}$ with entries:*

$$\mathfrak{D}_{ij} = \begin{cases} x_{ij} & \text{if } (u_i, v_j) \in E \\ 0 & \text{if } (u_i, v_j) \notin E \end{cases}$$

where the x_{ij} are formal variables (i.e. not instantiated with a particular value).

The corresponding $O(n^3)$ algorithm for matching in bipartite graphs is the same as Algorithm 2, except that we do one elimination (i-th row and j-th column for edge (u_i, v_j)) instead of two in order to update the inverse.

Mucha and Sankowski noted that each update to $\mathfrak{D}_G^{-1}$ amounts to subtracting a matrix of form uv^T/c and that many such updates $u_1 v_1^T/c_1, \dots, u_k v_k^T/c_k$ can be performed in one sweep by fast matrix multiplication. Mucha's and Sankowski's paper [13] gives an iterative version of the algorithm, which is equivalent to the recursive version presented in Mucha's PhD thesis [12]. As we found the recursive version easier to understand, we present it as Algorithm 3 here.

Algorithm 3 recurses on the range of columns and recursively performs elimination on the first half of columns, while queueing updates that need to be performed to the second half. When the recursive call returns, it actually performs updates from all recursive levels to the second half of columns and finishes by recursively doing elimination on the second half. The main insight behind this is that while eliminating a row and column is expensive ($O(n^2)$ entries need to be updated), the results of such updates are not needed until much later. One can show that if we defer updates to a later point, we can do batch update much faster than performing k individual updates consecutively.

By "lazy elimination" on line 10 in Algorithm 3 we mean queuing the unevaluated expression of form uv^T/c, whose actual computation is deferred to later (i.e. storing u, v and c separately to be multiplied out later). Update

Algorithm 3 $O(n^\omega)$ algorithm for finding a perfect matching in bipartite graph

```
1:  M ← ∅
2:  A ← 𝔇_G^{-1}
3:  MATCH(1, n)
4:  return M
5:
6:  procedure MATCH(p, q)
7:      if p = q then
8:          find v_r such that (u_p, v_r) ∈ E and A_{r,p} ≠ 0
9:          M ← M ∪ {(u_p, v_r)}
10:         lazily eliminate r-th row and p-th column of A
11:     else
12:         m ← ⌊(p+q)/2⌋
13:         MATCH(p, m)
14:         update uneliminated rows in columns m + 1 ... q of A
15:         MATCH(m + 1, q)
16:     end if
17: end procedure
```

(line 14) to set of rows R and columns C means adding the terms of the form uv^T/c to $A_{R,C}$, the matrix A restricted to rows from R and columns from C (and u and v similarly restricted). Suppose k such delayed updates are to be performed. Then the cumulative update to $A_{R,C}$ is $u_1v_1^T/c_1 + \ldots + u_kv_k^T/c_k$. This is a multiplication of two matrices, one of which has columns $u_1/c_1, \ldots, u_k/c_k$ and the other has rows $v_1^T, \ldots, v_k^T$. While computing the matrices takes time $k|C|$ and $k|R|$, we can speed up the computation by using the fast matrix multiplication algorithm, instead of naive multiplication.

One can check that eliminated rows and columns are always up-to-date, therefore the algorithm is correct and equivalent to Algorithm 2.

Theorem 3.2. *Algorithm 3 runs in time $O(n^\omega)$.*

Proof. Assume without loss of generality that n is a power of two, $n = 2^l$. Then in each call to MATCH the update is done on 2^k columns and at most n rows and we need to multiply a $2^k \times 2^k$ matrix by $2^k \times O(n)$ matrix. We can partition this $2^k \times n$ matrix in $n/2^k$ submatrices of size $2^k \times 2^k$ and use the fast matrix multiplication $n/2^k$ times to multiply two $2^k \times 2^k$ matrices. This takes time $\frac{n}{2^k}2^{k\omega} = n(2^{\omega-1})^k$ for such update. Each update to 2^k columns happens in $n/2^{k+1}$ calls to MATCH, so the total time taken is

$$\sum_{k=0}^{l-1} n/2^{k+1} \cdot (2^{\omega-1})^k = \frac{n^2}{2}\sum_{k=0}^{l-1}(2^{\omega-2})^k = O(n^2(2^{\omega-2})^l) = O(n^\omega)$$

□

Readers extremely familiar with computational linear algebra might notice that this is essentially the same as Bunch-Hopcroft algorithm for performing Gaussian elimination in $O(n^\omega)$ time: the only difference is that to perform Gaussian elimination without pivoting one chooses to eliminate r-th row and r-th column, instead of looking for an entry that corresponds to an edge in the matching. Note that this algorithm is the familiar $O(n^3)$ Gaussian elimination algorithm with lazy updates.

3.4 O(n^ω) algorithm for bipartite graphs

For the rest of this paper we will assume that we can use Gaussian elimination as a black-box; interested readers are encouraged to refer to Bunch's and Hopcroft's original paper [1]. Gaussian elimination is central to proving the following theorem [13]:

Theorem 3.3. *Let G be a graph having a perfect matching. For any matching M of G, a maximal submatching M' of M, such that M_0 can be extended to a perfect matching, can be found in time $O(n^\omega)$.*

The procedure for finding maximal extendable submatching, together with the $O(n^\omega)$ algorithm for finding perfect matching in a bipartite graph, is used to implement an algorithm for finding perfect matchings in general graphs. (Algorithm 4). The rough plan for the algorithm is as follows: greedily find a matching. If this matching has a large extendable matching, then we have reduced the problem size by a constant fraction. Otherwise we have identified a large set of forbidden edges, which will allow us to use the special structure of the graph to reduce the problem size by a constant fraction.

A very involved structural theorem by Lovász and Plummer [8] allows the PARTITION procedure to partition a graph with a large number of forbidden edges into a bipartite graph and a number of other subgraphs, so that each of these components has a perfect matching. So one can use the $O(n^\omega)$ algorithm for the bipartite graph and recurse with general algorithm for the other parts.

The partitioning procedure is combinatorial and doesn't provide new algebraic insights, so we refer the interested reader to the original paper [13]. However it uses algorithms for dynamic vertex connectivity problem as a black box, which are algebraic. We discuss algebraic methods for dynamic updates in Section 5.

Algorithm 4 $O(n^\omega)$ algorithm for finding a perfect matching in general graph

```
1: procedure GENERAL-MATCHING(G)
2:     Find matching M of size at least n/4 using the greedy algorithm
3:     Find maximum allowed submatching M' of M.
4:     Match vertices of M' and remove them from G, obtaining G'
5:     if |M'| ≥ n/8 then
6:         Call GENERAL-MATCHING(G')
7:     else
8:         Call PARTITION(G')
9:     end if
10: end procedure
```

One might wonder if finding a maximum matching is harder than finding a perfect matching. As Rabin and Vazirani proved in their paper [14], the answer is negative, because there is an easy reduction from the problem of finding a perfect matching:

Theorem 3.4. *Let G be a graph with a maximum matching of size k. Let G' be a graph obtained from G by adding $n - 2k$ new vertices, and edges connecting each new vertex to each original vertex. Then G' has a perfect matching, furthermore any perfect matching for G' has a maximum matching of G as a submatching.*

Proof. A maximum matching in G will leave $n - 2k$ vertices unmatched, but those can be matched against new vertices of G', therefore G' has a perfect matching.

No perfect matching of G' can match new vertices among themselves, therefore it matches $n - 2k$ new vertices to $n - 2k$ vertices of G. We have $2k$ vertices left in G and the perfect matching in G' restricted to them corresponds to a maximum matching in G. □

4 Perfect Matchings via Lazy Updating

Mucha and Sankowski's $O(n^\omega)$-time maximum matching algorithm was a breakthrough in 2004, but it used rather technical methods from both algebra and graph theory. In the case of bipartite matching, however, they gave a simpler algorithm that implemented Gaussian Elimination in a lazy manner. Their technique seemed to break down in the general, non-bipartite case, which led to the question of whether it was possible to solve the maximum matching problem in $O(n^\omega)$ time, using a lazy updating strategy [13]. In 2008, Nicholas Harvey answered this in the affirmative by giving a purely algebraic algorithm for perfect matching in general graphs that uses a lazy update method [5]. This lazy update method is derived from the Mucha-Sankowski $O(n^2)$ matrix inverse update procedure to speed up the Rabin-Vazirani perfect matching algorithm from $O(n^{\omega+1})$ time to $O(n^3)$ time, described in a previous section.

4.1 Harvey's divide-and-conquer algorithm

Harvey generalizes the elimination theorem Harvey extends this rank-1 update method to achieve an $O(n^\omega)$ time algorithm to find a perfect matching. The key idea is to adopt a divide-and-conquer approach, but defer the matrix inverse updates whenever possible - just as in Mucha-Sankowski's bipartite matching algorithm. Harvey's algorithm `FindPerfectMatching` is presented below:

Algorithm 5 FindPerfectMatching(G)

procedure FINDPERFECTMATCHING(G)
  $M \leftarrow \emptyset$.
  Construct Tutte matrix T from G.
  Assign values to the variables in T, selected from $\{1, \ldots, n^2\}$ uniformly at random.
  Compute $N = T^{-1}$.
  Let S be the vertex set of G.
  FindAllowedEdges(G)
end procedure

procedure FINDALLOWEDEDGES(S)
  if $|S| > 2$ **then**
    Partition S into equal-sized sets $S_1, \ldots, S_\alpha$, where α is a fixed constant in the algorithm.
    for Each unordered pair $\{S_a, S_b\}$ **do**
      FindAllowedEdges($S_a \cup S_b$)
      Update N.
    end for
  else
    (Base case: $S = \{i, j\}$ for some i, j)
    if $T_{ij} \neq 0$ and $N_{ij} \neq 0$ **then**
      Add (i, j) to the matching M.
      Update N.
    end if
  end if
end procedure

Proof of Correctness (assuming the correctness of the update procedure). Note that it is unimportant how the set S is partitioned in `FindAllowedEdges`, as long as each partition is of equal size. Assume that whenever the algorithm enters the base case in the procedure `FindAllowedEdges`, the Tutte inverse matrix N is accurate with respect to the pair of edges being considered, and with respect to the current matching M; i.e. in the base case where $S = \{i, j\}$, $N_{ij} \neq 0$ iff edge (i, j) can be added to the current matching M without conflict (however there is no guarantee that for any other pair of edges $\{k, \ell\}$, $N_{k\ell}$ will be accurate). Then, this algorithm is simply a variant of the Rabin-Vazirani perfect matching algorithm.

At every level of the recursion, for every pair of vertices $\{i, j\}$, there are two subsets $S_a, S_b \subseteq V$ such that $\{i, j\} \subseteq S_a \cup S_b$. Thus, every pair of vertices $\{i, j\}$ (and in particular every edge) gets considered in a base case. If in a base case, an edge is deemed admissible with respect to the current matching, it is added to the matching. This is precisely the Rabin-Vazirani perfect matching algorithm, but where the edges are processed in a particular order. Hence after executing `FindPerfectMatching`, M will contain a perfect matching of G.

Time complexity. Assume that updating N takes $O(s^\omega)$ time for each subproblem with s vertices. Each subproblem of size s will recurse to solve $\binom{\alpha}{2}$ subproblems of size $2s/\alpha$. Thus, the total time complexity of this algorithm satisfies the recurrence relation $T(s) = \binom{\alpha}{2} T(2s/\alpha) + O\left(\binom{\alpha}{2} \cdot s^\omega\right)$, where the latter term is due to the update procedure. The solution to this recurrence relation is $T(n) = O(n^\omega)$, provided that α satisfies $\log_{\alpha/2} \binom{\alpha}{2} < \omega$.

4.2 The lazy update procedure

We will give a high-level description and intuition for the update procedure used in the $O(n^\omega)$ algorithm. For full details, we refer the reader to [5].

Consider the base case of the `FindAllowedEdges` subroutine: every time an edge is added to the matching M, an update to N is called. If a full update is performed, each update must take $O(n^2)$ time, and there are $n/2$ edges added to form a perfect matching. Thus, if we are to accomplish an overall running time of $O(n^\omega)$, we cannot update the entire N matrix in the base case. Instead, we will perform the minimum amount of necessary work to update parts of N and defer the rest of the updates to when recursive subproblems are completed (i.e. the Update call in line 15 of `FindAllowedEdges`).

What is the minimum amount of update work necessary? We will maintain the invariant that when a recursive subproblem begins or completes, the submatrix of N corresponding to the parent's subproblem is updated "correctly". To elaborate, suppose a parent subproblem is $S_{\text{parent}} = \{g_1, g_2, \ldots, g_t\}$. S_{parent} will be partitioned into α sets, and there are $\binom{\alpha}{2}$ child subproblems. After each child subproblem is processed, the submatrix of N corresponding to rows and columns $g_1, \ldots, g_t$ will be exactly as if the lazy update approach were not used, but full updates were performed every time. The entries of N outside of this submatrix are not guaranteed to be "fresh" and "correct", but the key observation is that *the entries outside the parent subproblem submatrix are not used in the child subproblems!* This is why we can afford to be lazy. Hence, each update only takes time that's a function of the subproblem size, not of n.

Harvey's algorithm carefully maintains this invariant. Intuitively, every time a child subproblem defers update work (that is, the update work of the entries outside its parent submatrix), it will make a record of it. This is used later when ancestor subproblems (after the recursion has "unwound" itself) have to perform their own update work; in order to perform the right updates, the ancestors have to know of the parameters used in their descendants' updates. The records will store those parameters. The algorithm is able to achieve a $O(s^\omega)$ update time for each recursive subproblem, and thus we have a $O(n^\omega)$-time algorithm to find a perfect matching (if it exists).

4.3 Extending Harvey's algorithm to maximum matching

As described in the previous section, given an algorithm to compute perfect matching, one can compute maximum matching in $O(n^\omega)$ time.

5 Dynamic Matching

We've seen that the crucial element behind the fast $O(n^\omega)$-time algorithms for matching were methods to quickly update matrix inverses. We pursue this technique further in this section, showing several quick methods for updating matrix inverses. The algorithms for updating the matrix inverse presented in this section don't give us the $O(n^\omega)$ algorithm for finding matchings, but they alow more general updates, and can be used to solve an interesting form of the matching problem - the *dynamic update* matching problem. Furthermore, they are conceptually simpler, and can immediately be applied to other dynamic graph problems, while the update method used in $O(n^\omega)$ algorithm for finding matchings for general graphs is specifically tailored to that problem.

Suppose we've already computed the inverse of the Tutte matrix for a graph, and found its determinant to be non-zero. As we've previously mentioned, it's now easy to determine whether some edge is part of any perfect matching, by just quering the corresponding entry in the inverse, and checking if it's non-zero. But if we then make small changes to the graph, like adding or removing a single edge, the inverse we've previously computed can't be directly used to answer this question anymore. In this section, we'll present fast algorithms for updating the inverse, determinant, and adjoint of a matrix in the presence of small changes, and show how these algorithms can be applied to solve dynamic graph problems.

In [15] Sankowski gives the following algorithms for dynamic matrix inverse (for *non-singular* matrices):

- An algorithm that supports changes to a single row or column, and takes $O(n^2)$ time for updates and O(1) time for queries

- An algorithm that supports changes to n^ϵ rows or columns, in $O(n^{\omega(1,\epsilon,1)})$[3] update time, and O(1) query time
- Two algorithms that support changes to a single matrix entry and have the following tradeoffs between update and query time: one requires $O(n^{1.575})$ time for updates, and $O(n^{0.575})$ time for queries, and the other one supports both updates and queries in $O(n^{1.495})$ time.

These results can easily be extended to give the same time bounds for matrix determinant and adjoint computations, as well as solving linear systems.

The main idea. The main technique behind the algorithms for dynamically updating matrix inverses is the following: suppose we want to modify a small part of a matrix A, and we already have the precomputed inverse A^{-1}. Let A' be the matrix after the change. Instead of computing A'^{-1} directly, we use the "Easy Representation" $A'^{-1} = B^{-1} \cdot A^{-1}$, where B turns out to be a sparse matrix whose inverse is easy to compute. For the single-entry updates we combine this with another technique: we update lazily, which leads to a better update time bound, at the cost of a worse query time.

5.1 Algorithm A that supports row and column updates

Update to A: We want to update the ith column of A to a vector v (row updates are handled similarly). Denote the updated matrix A'. To relate this to graph problems, not that in order to modify the neighborhood of a vertex, we need to change an entire row and an entire column, which is exactly the type of update this algorithm supports.

Easy representation of A^{-1}: We write $A' = A \cdot B$, where B is a sparse matrix. The Easy Representation is $A'^{-1} = B^{-1} \cdot A^{-1}$, where B turns out to be a matrix whose inverse is easy to compute and has many zeros.

Suppose that after the update, the i^{th} column of A is the vector v, and call the resulting matrix A'. Then $A' = A + (v - A_{*,i}) \cdot e_i^T$ (here $A_{*,i}$ is the i^{th} column of A, and e_i is the unit vector with a 1 in the i^{th} position). If the matrix A is non-singular, we can write $A' = A \cdot B$ for some matrix B. So $A' = A \cdot (I + A^{-1}(v - A_{*,i}) \cdot e_i^T) = A \cdot B$, and therefore $B = I + A^{-1}(v - A_{*,i}) \cdot e_i^T$. Let $b = A^{-1}(v - A_{*,i})$. Then $B = I + b \cdot e_i^T$, and therefore $B^{-1} = (I + b \cdot e_i^T)^{-1}$.

Algorithm 6 Algorithm A for updating the inverse

1: Compute $b = A^{-1}(v - A_{*,i})$
2: Compute $B^{-1} = (I + b \cdot e_i^T)^{-1}$
3: Return $A'^{-1} = B^{-1}A^{-1}$

Claim 1: We can compute $A^{-1}(v - A_{*,i})$ in $O(n^2)$ time. This is obvious, since we're multiplying an $n \times n$ matrix by an $n \times 1$ vector.

Claim 2: $B^{-1} = (I + b \cdot e_i^T)^{-1}$ can be computed in $O(n)$ time. We can derive the formula below for the inverse of $(I + b \cdot e_i^T)^{-1}$ by noting that the determinant of B is $b_i + 1$, and computing adjoint matrix for B (this is easy to do, because B is sparse, and most entries of the adjoint will turn out to be 0 as well).

$$\begin{pmatrix} 1 & 0 & \dots & 0 & -\frac{b_1}{b_i+1} & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & -\frac{b_2}{b_i+1} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & \dots & 1 & -\frac{b_i}{b_i+1} & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & -\frac{1}{b_i+1} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & \dots & 0 & -\frac{b_n}{b_i+1} & 0 & \dots & 1 \end{pmatrix}$$

[3] $\omega(\alpha, \beta, \gamma)$ is the exponent of multiplying an n^α x n^β matrix by a n^β x n^γ matrix. One bound we have is $\omega(1, \alpha, \alpha) \leq \alpha \cdot (\omega - 1) + 1$

Looking at the formula for the inverse, it's easy to see that in can be computed in $O(n)$ time once we've computed b (which we do in the previous step).

Claim 3: $A'^{-1} = B^{-1} \cdot A^{-1}$ can be computed in $O(n^2)$ time. This is because B^{-1} only contains $O(n)$ non-zero entries.

So overall, the algorithm takes $O(n^2)$ time for updates. Queries can be done in O(1) time, since we explicitly store the inverse.

5.2 Algorithm B that supports n^ϵ row and column updates

Update: We want to update the first $k = n^\epsilon$ columns of A to the vectors $v_1, \ldots, v_k$ (row updates are handled similarly). Denote the updated matrix A'. Note the we can easily generalize this algorithm to handle updates to arbitrary k columns by reordering them. To relate this to graph algorithms, this algorithm is useful for updating the inverse of the adjacency/Tutte matrix after we've made changes to the neighborhoods of n^ϵ vertices in the graph. It is also used as subroutine for the fast algorithms that support a single entry update, which we'll present in the next subsection.

Easy Representation of A^{-1}: We use the same Easy Representation as before, namely, $A'^{-1} = B^{-1} \cdot A^{-1}$, where B is a matrix whose inverse is easy to compute and has many zeros.

Suppose the first k columns of A have been changed to $v_1, \ldots, v_k$. Then

$$\begin{aligned} A' &= A + \begin{pmatrix} | & | & & | & \\ v_1 & v_2 & \cdots & v_k & 0 \cdots 0 \\ | & | & & | & \end{pmatrix} - \begin{pmatrix} | & | & & | & \\ A_{*,1} & A_{*,2} & \cdots & A_{*,k} & 0 \cdots 0 \\ | & | & & | & \end{pmatrix} \\ &= A \left(I + A^{-1} \left[\begin{pmatrix} | & | & & | & \\ v_1 & v_2 & \cdots & v_k & 0 \cdots 0 \\ | & | & & | & \end{pmatrix} - \begin{pmatrix} | & | & & | & \\ A_{*,1} & A_{*,2} & \cdots & A_{*,k} & 0 \cdots 0 \\ | & | & & | & \end{pmatrix} \right] \right) \\ &\triangleq A \cdot B \end{aligned}$$

Algorithm 7 Algorithm B for updating the inverse

1: Compute $B = I + A^{-1} \cdot [(v_1 \cdots v_k \quad 0 \cdots 0) - (A_{*,1} \cdots A_{*,k} \quad 0 \cdots 0)]$
2: Compute B^{-1}
3: Return $A'^{-1} = B^{-1}A^{-1}$

We get the desired time bound of $O(n^{\omega(1,\epsilon,1)})$ for updates because B has non-zero entries only on the diagonal and at most n^ϵ columns, so it's easy to perform matrix multiplications involving it, and it turns out that computing its inverse is also easy.

5.3 Algorithms that supports single entry updates

Update: We want to update entry (i, j) in A to some new value.

Easy representation of A^{-1}: Here, instead of updating the entire matrix, we keep the inverse in the following lazy form: $A^{-1} = (N + I) \cdot M$. Note that, in contrast to the previous algorithms, where we represented A' as $A \cdot B$, but actually explicitly computed A' after each update, here we only store M and N, and then perform the necessary multiplications to answer queries, when needed. We only update N for the first n^ϵ updates, and afterwards, we reset N to 0, and make a batch update including all the n^ϵ changes to M, using algorithm 7 above.

We have two algorithms: one updates all of N in each step, and performs the multiplication $e_i \cdot (N + I) \cdot M \cdot e_j^\top$, when we want query the (i, j) position of A^{-1}, giving us an update cost of $O(n^{1.575})$, and query time $O(n^{0.575})$. We refer to this as algorithm C.

For the other algorithm, we exploit the idea of delaying the updates further, to obtain a better time for updates ($O(n^{1.495})$), at the cost of a higher query time (also $O(n^{1.495})$). The key difference between this new algorithm and algorithm C, is that we only explicitly compute some rows of the matrix N at each step, and the others are computed only when needed for answering queries. We refer to this as algorithm D.

Note that which algorithm is better to use depends on the ratio of the number of updates and queries needed for our application. If the number of queries is a lot larger than the number of updates, algorithm C is better. On the other hand, if we make the same number of queries and updates, we'll want to use algorithm D.

5.4 Applications to Dynamic Matchings

Testing whether edge (i, j) is part of a perfect matching. Algorithm C can be immediately applied to support fast edge updates to the inverse of the Tutte matrix corresponding to a graph. An edge update in the graph only changes 2 entries in the matrix. We can then maintain the inverse in time $O(n^{1.575})$ using algorithm C above. To check whether some edge (i, j) is part of a perfect matching, we need to check whether entry (i, j) in the inverse of the Tutte matrix is non-zero, which can be done in time $O(n^{0.575})$

We can also use algorithm D, which gives us $O(n^{1.495})$ time for updating the Tutte matrix, and also $O(n^{1.495})$ time for checking whether the edge is part of a perfect matching.

Finding a perfect matching. In order to find a perfect matching, we generate a random Tutte matrix, compute its inverse, and find an edge whose corresponding entry in the inverse matrix is non-zero, remove it, and all its adjacent edges, and then recompute the inverse of the Tutte matrix, and continue until we've added $\frac{n}{2}$ edges to the matching.

Removing an edge and all its adjacent edges requires making changes to one column and one row of the Tutte matrix, and then recomputing its inverse(to check which edges are part of the submatching), which can be done in $O(n^2)$ time, using algorithm A. We need to do this $O(n)$ times to find a matching, so this immediately gives us an $O(n^3)$ algorithm for finding perfect matchings.

While not immediately obvious, algorithm C can also be extended to support clear operations in $O(n^{1.575})$ time (a clear operation allows us to set one row or one column to 0), which gives us an algorithm for finding a perfect matching with runtime $O(n^{2.575})$. This time bound is improved to $O(n^\omega)$ by the perfect matching algorithm that uses Gaussian elimination, but we include these algorithms as well since they're conceptually simpler.

Maximum matchings. In [16], Sankowski gives an algorithm for dynamically computing the size of the maximum matching in an unweighted undirected graph with $O(n^{1.495})$ update cost, and $O(1)$ query cost, that supports single edge insertions and deletions. Prior to this work, the best upper bound for both this problem $O(m)$, so the new algorithm performs better for dense graphs.

From the original graph G, we construct a graph G^d such that the following lemma holds:

Lemma 5.1. *G^d has a perfect matching iff G has a matching of size at least $\frac{|V|}{2} - d$.*

For the full construction of G^d, we refer the reader to [16], but one property that will be important for our algorithm is that for $d_1 \neq d_2$, G^{d_1} and G^{d_2} are identical, except that there is a subset of nodes S, such that G^{d_1} has d_1 in the subgraph induced on S, while G^{d_2} has d_2 edges (G^d has the same set of vertices for any d).

Note: From our algorithm that supports both updates and queries in $O(n^{1.495})$ when updating a constant number of entries (this can model inserting or deleting a single edge) for matrix inverse and matrix determinant, we can get an algorithm that performs edge updates to a graph in $O(n^{1.495})$ time such that the updates succeeds iff the graph after the update has a perfect matching, and otherwise it just returns failure. We do this by just checking the determinant of the Tutte matrix after the update, and returning failure if it's non-zero.

Now, using this fact, and the lemma about G^d, we can construct a dynamic algorithm for maximum matching with $O(n^{1.495})$ update time, and $O(1)$ query time.

The algorithm works as follows: Suppose we want to add an edge e to G. We add e to G^d, to obtain the graph G'^d, and then try to delete one of the edges from the subgraph induced by S. If it's possible, then G'^{d-1} also has a perfect matching, so we decrease d by 1. If it's not possible, we don't make any changes, we keep the graph G'^d.

Edge deletions are handled similarly. We try to delete an edge corresponding to e from G^d to obtain G'^d if it's not possible, we first add an edge to a pair of vertices in S, and then remove the edges, obtaining a graph of type G'^{d+1}.

Whenever we want to find the size of the perfect matching in G, we just return $\frac{|V|}{2} - d$, for the current value of d.

Other applications. The algorithms that support dynamic updates to matrix inverse, determinant, and adjoint have applications for numerous dynamic graph algorithms, for problems such as computing the transitive closure, computing the maximum number of vertex disjoint s-t paths, and testing directed vertex k-connectivity of a graph. For more details, we refer the reader to [16].

6 Counting Matchings

We now turn our attention from algorithms that find matchings to algorithms that *count* them. That is, given a graph G, how many perfect matchings does it have? It would not do to explicitly enumerate them all, as there could be an exponential number of perfect matchings. For example, the complete bipartite graph $K_{n,n}$ has $n!$ perfect matchings.

It turns out that it is unlikely there is a polynomial time algorithm to count the number of perfect matchings in a graph; Valiant proved that this problem is $\#\mathcal{P}$-complete, a class of *extremely* difficult problems[4] [19]. This was shown by reducing 0-1 PERMANENT (the problem of evaluation the permanent - as opposed to the determinant - of a matrix with 0-1 entries) to the problem of counting matchings, and 0-1 PERMANENT is itself known to be $\#\mathcal{P}$-complete.

However, despite the hardness result, it turns out that the number of perfect matchings in a graph G can be efficiently approximated to any desired degree of accuracy, provided that you are willing to trade time for accuracy. Jerrum, Sinclair and Vigoda gave a FPRAS (fully polynomial randomized approximation scheme) that approximates the number of perfect matchings in a *bipartite* graph to within $1 \pm \epsilon$. Their result involves a series of powerful Monte Carlo techniques for sampling an exponentially large graph, and introduces many novel ideas that have been used in other algorithms [6]. Later, Chien introduced a simple algorithm for approximating the number of perfect matchings in a *general* graph by estimating the determinant of a random Tutte matrix [2]. Chien's algorithm is not comparable to that of Jerrum, Sinclair and Vigoda's, however, because his algorithm is not guaranteed to terminate in polynomial time. Despite this, we will present Chien's algorithm, because it makes novel use of the ideas that we have explored above.

6.1 Estimators and critical ratios

Before describing Chien's algorithm we will make a brief digression into the topic of *estimators* and *critical ratios*. Suppose you want to approximate a quantity Q by sampling values from a distribution $\mathcal{D}$ and computing some function on the sampled values. Let X be the random variable corresponding to the output of the function on a value drawn from $\mathcal{D}$. Essentially, we are hoping that the value of X is closely related to Q. We call X an *estimator* for Q. If it is the case that $\mathbb{E}[X] = Q$, then we call X an *unbiased* estimator. If we have an unbiased estimator X for Q, then this suggests a simple algorithm to approximate Q: sample X repeatedly, and take the average of the samples. The question then becomes: how many samples do we have to take? Intuitively, if the estimator has low variance, then we need only to take a few samples. On the other hand, if the variance is extremely large, then we have to take many samples. This intuition is quantified by $\frac{\mathbb{E}[X^2]}{\mathbb{E}[X]^2}$, called the *critical ratio* of an estimator. With high probability, given an unbiased estimator X of Q, the number of samples needed to approximate Q within $1 \pm \epsilon$ is $O(1/\epsilon^2)\frac{\mathbb{E}[X^2]}{\mathbb{E}[X]^2}$. This follows from a straightforward application of Chebyshev's inequality.

6.2 An unbiased estimator for the number of perfect matchings

Chien gives a simple estimator for the number of perfect matchings in a graph. Let $G = (V, E)$ be an undirected simple graph on $2n$ vertices. Let $M(G)$ denote the number of perfect matchings in G. Let T be the associated $2n \times 2n$ Tutte matrix for G. Obtain matrix B_G by instantiating each variable in T with a value chosen from $\{-1, 1\}$ selected uniformly and independently at random.

[4]Even if $\mathcal{P} = \mathcal{NP}$, there could still be no polynomial time algorithm for counting matchings! On the other hand, a polynomial time algorithm for counting matchings would imply $\mathcal{P} = \mathcal{NP}$.

Theorem 6.1 (Chien). *$X_G = \det(B_G)$ is an unbiased estimator for $M(G)$.*

Proof. We give a simpler, slightly different proof to this statement than in [2]. We follow the proof of Theorem 2.1 instead. Permutations σ corresponding to perfect matchings contribute 1 to the determinant sum, and permutations not corresponding to matchings contribute 0. To see this, observe that a perfect matching σ gives rise to a monomial of the form $x_{e_1}^2 x_{e_2}^2 \cdots x_{e_n}^2$ in the determinant of the Tutte matrix, and when each of the variables are instantiated with a random value from $\{-1, 1\}$, the expectation of this monomial is 1, because each variable is squared and is uncorrelated with every other variable. On the other hand, when σ does not correspond to a perfect matching, the monomial has a variable that is not squared, and is uncorrelated with all other variables. $\mathbb{E}[x_{ij}] = 0$, so non-matching permutations contribute 0 to the determinant sum. Since there is exactly one σ for every perfect matching, and each of these σ contributes 1 to the determinant, this concludes the proof of the statement. □

The following theorem shows that this estimator will not in general lead to a polynomial time randomized algorithm for estimating the number of perfect matchings in a graph.

Theorem 6.2 (Chien). *The critical ratio of the estimator X_G is bounded by $3^{n/2}$, and there exist a graph G such that the critical ratio of X_G is at least $2^{n/2}$.*

One such graph that has an exponential lower bound on the critical ratio consists of $n/2$ copies of the complete bipartite graph $K_{2,2}$. On the other hand, counting perfect matchings in a random graph is quite easy; Chien shows that almost every graph $G \in \mathcal{G}_{2n,p=1/2}$ (i.e. the set of graphs on $2n$ vertices where each edge is included with probability 1/2) has critical ratio $O(n\omega(1))$, where $\omega(1)$ is any superconstant function. Thus, given a random graph, one needs only to sample X at most a slightly superlinear number of times in order to get a good approximation of $M(G)$.

7 Conclusion

We have seen various algorithms that solve the matching problem by using algebraic techniques instead of combinatorial methods. These algebraic techniques largely take advantage of the close relationship between perfect matchings in a graph and the graph's associated Tutte matrix. The perfect matching algorithms that utilize this relationship are due to Lovász and Rabin, Vazirani. Later, Mucha and Sankowski, and Harvey extended the Rabin-Vazirani algorithm to obtain the fastest known algebraic algorithms for matching, running in $O(n^\omega)$ time. The core of their algorithms is speeding up the computation for updating the inverse Tutte matrix. Sankowski also developed general techniques for quickly updating the inverse of a matrix, with applications to *dynamic* matching - where we want to find the cardinality of the maximum matching after adding or removing an edge, and we want to do so in time that's faster than computing the maximum matching from scratch. Finally, we saw how the Tutte matrix could be used to estimate the *number* of perfect matchings in a graph, a problem for which the exact solution is $\mathcal{NP}$-hard.

However, there are many interesting open questions left, some of which we raise here:

1. Can the n^ϵ update procedure used in the dynamic matrix inverse algorithm be used to speed up the lazy updates procedure in Harvey's algorithm?

2. Can Sankowski's algorithms for dynamic matching be extended to solve dynamic *weighted* matching?

3. The best algorithm for finding matchings in general graphs is due to Micali and Vazirani [10], which runs in time $O(\sqrt{n}m)$, where n is the number of vertices and m is the number of edges. This is faster than the $O(n^\omega)$ time algorithm we saw, when $m = o(n^{\omega-0.5})$ - so the graph has to be extremely dense in order for the algebraic algorithm to run faster. However, their algorithm uses many complicated combinatorial techniques. Is it possible that there is an algebraic algorithm that achieves this running time or better?

4. Chien's algorithm to estimate the number of perfect matchings will not run in polynomial time for all graphs. Is there a FPRAS to count perfect matchings in general graphs?

References

[1] BUNCH, B. J. R., AND HOPCROFT, J. E. Factorization and inversion by fast matrix multiplication. *Mathematics of Computation 28*, 125 (1974), 231–236.

[2] CHIEN, S. A determinant-based algorithm for counting perfect matchings in a general graph. In *In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2004), pp. 728–735.

[3] CHRISTIANO, P., KELNER, J. A., MADRY, A., SPIELMAN, D. A., AND TENG, S.-H. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd annual ACM symposium on Theory of computing* (2011), STOC '11, pp. 273–282.

[4] EDMONDS, J. Paths, trees, and flowers. *Canadian Journal of Mathematics 17*, 3 (1965), 449–467.

[5] HARVEY, N. J. A. Algebraic structures and algorithms for matching and matroid problems. In *In Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2006), pp. 531–542.

[6] JERRUM, M., SINCLAIR, A., AND VIGODA, E. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM 51* (July 2004), 671–697.

[7] LOVÁSZ, L. On determinants, matchings, and random algorithms. In *FCT* (1979), pp. 565–574.

[8] LOVÁSZ, L., AND PLUMMER, M. *Matching theory*. AMS Chelsea Publishing Series. AMS Chelsea Pub., 2009.

[9] MATOUŠEK, J. *Thirty-three miniatures: mathematical and algorithmic applications of linear algebra*. Student mathematical library. American Mathematical Society, 2010.

[10] MICALI, S., AND VAZIRANI, V. V. An $O(|V|^{1/2}|E|)$ algorithm for finding maximum matching in general graphs. In *In Proceedings of the 21st Annual Symposium on Foundations of Computer Science, Syracuse (FOCS)* (1980), pp. 17–27.

[11] MOSHKOVITZ, D. An alternative proof of the Schwartz-Zippel lemma. *Electronic Colloquium on Computational Complexity (ECCC) 17* (2010), 96.

[12] MUCHA, M. *Finding maximum matchings via Gaussian elimination*. PhD thesis, Warsaw University, 2005.

[13] MUCHA, M., AND SANKOWSKI, P. Maximum matchings via Gaussian elimination. In *In Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2004), pp. 248–255.

[14] RABIN, M. O., AND VAZIRANI, V. V. Maximum matchings in general graphs through randomization. *Journal of Algorithms 10*, 4 (1989), 557 – 567.

[15] SANKOWSKI, P. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *In Proceedings of the 45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings* (2004), pp. 509–517.

[16] SANKOWSKI, P. Faster dynamic matchings and vertex connectivity. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* (2007), SODA '07, pp. 118–126.

[17] SCHWARTZ, J. T. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM 27*, 4 (1980), 701–717.

[18] TUTTE, W. T. The Factorization of Linear Graphs. *Journal of the London Mathematical Society s1-22*, 2 (1947), 107–111.

[19] VALIANT, L. G. The complexity of computing the permanent. *Theoretical Computer Science 8* (1979), 189–201.

[20] WILLIAMS, V. V. Breaking the Coppersmith-Winograd barrier.