Maximum Matchings via Algebraic Methods for Beginners

Pawel Parys
Warsaw University, Poland

November 5, 2009

Abstract

This is an introduction to algebraic methods for calculating maximum matchings.

1 Bipartite graphs

Definitions We are given a graph $G = (V, E)$. A *matching* is a set of edges from E such that each vertex is in at most one edge from the set. A matching is *maximum* if there is no matching of greater size. A matching is perfect, if it contains all vertices.

In the first section we consider only bipartite graphs. A graph is *bipartite*, it its set of vertices may be divided into two sets U, W such that there are no edges inside U and inside W. Let $n = |U| \geq m = |W|$ and $U = \{u_1, \ldots, u_n\}$, $W = \{w_1, \ldots, w_m\}$.

Classic algorithms The problem of finding a maximum matching in a bipartite graph was first considerd by Ford and Fulkerson, who have shown an $O(|E||V|) = O(n^3)$ algorithm [6]. Then Hopcroft and Karp presented an improved algorithm, which works in time $O(|E|\sqrt{|V|}) = O(n^{2.5})$ [8]. It was the best known algorithm until the (presented below) $O(n^{2.38})$ algorithm was discovered. Of course this algorithm is only theoretically better, since the fast matrix multiplication algorithm is very complicated, so in practice (for typical graph size) the Hopcroft-Karp algorithm is faster.

Adjacency matrix First fix some notation. For any matrix A its entry in i-th row and j-th column is denoted by $A_{i,j}$. When I and J are sets of numbers, by $A_{I,J}$ we denote a submatrix of A consisting of rows I and columns J. Finally, by $A_{del(i,j)}$ we denote a $(n-1) \times (n-1)$ submatrix obtained from A by deleting row i and column j.

A *symbolic bipartite adjacency matrix* of G is an $n \times m$ matrix $\widehat{A}(G)$ such that

$$\widehat{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } (u_i, w_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

where the $x_{i,j}$ are unique variables corresponding to edges of G.

At the beginning we present the problem of finding a perfect matching (containing all vertices) in a bipartite graph. A perfect matching may exist only if $|U| = |W|$ ($n = m$). In this case we get a squere matrix. It is easy to observe that

Theorem 1.1 (Tutte [15]) *For a bipartite graph G the symbolic determinant $\det \widehat{A}(G)$ is non-zero iff G has a perfect matching.*

Precisely, every component of $\det \widehat{A}(G)$ corresponds to a perfect matching. To see that recall the definition of determinant

$$\det \widehat{A}(G) = \sum_{p \in \Pi_n} \sigma(p) \prod_{i=1}^{n} x_{i,p_i} \ .$$

A non-zero component of this sum for every vertex u_i chooses another vertex w_{p_i}.

If we could calculate the determinant symbolically, the problem would be solved. However there is no fast method to do that (moreover the result can be very long). Lovász [10] proposed a randomized method: we substitute random numbers for the variables in the matrix. Precisely, we choose a prime number $p = n^{O(1)}$ and substitute each variable in $\widehat{A}(G)$ with a random number taken from the field $\mathbb{Z}_p$. Let us call the resulting matrix the *random adjacency matrix of G* and denote $A(G)$. We consider the matrix and its determinant in $\mathbb{Z}_p$. We have the following theorem:

Theorem 1.2 *If $\det \widehat{A}(G) \neq 0$ then with probability at least $1 - n/p$ it holds $\det A(G) \neq 0$ (and if $\det \widehat{A}(G) = 0$ then surely $\det A(G) = 0$).*

This gives a randomized algorithm for deciding whether a given bipartite graph has a perfect matching: Compute the determinant of $A(G)$ (using Gaussian elimination); with high probability this determinant is non-zero iff G has a perfect matching. The algorithm can be easily implemented to run in time $O(n^3)$. As we will see, it can be also implemented to run in time $O(n^\omega)$ using fast matrix multiplication (ω is the matrix multiplication exponent, currently $\omega \leq 2.38$, see Coppersmith and Winograd [3]). Observe that the sum and multiplication operations in $\mathbb{Z}_p$ may be done in constant time. Division operation requires $O(\log n)$ time, since one needs to perform the Euclid algorithm for greatest common divisor. However in all the algorithms mentioned in this paper division (namely taking inverse) is used only $O(n)$ times, so the additional logarithmic complexity of division does not impact the overall complexity.

The above theorem follows immediately from the following lemma proved independently by Zippel [16] and Schwartz [14], because $\det \widehat{A}(G)$ is a polynomial of degree n:

Lemma 1.3 *If $P(x_1, \ldots, x_m)$ is a non-zero polynomial of degree d with coefficients in a field $\mathbb{Z}_p$, then the probability that P evaluates to 0 on a random element $(s_1, \ldots, s_m) \in \mathbb{Z}_p^m$ is at most d/p.*

Proof
Induction on m (number of variables). For $m = 1$ we just have a one variable polynomial of degree d; it simply has at most d different roots.

For $m > 1$ we represent the polynomial in the following way:

$$P(x_1, \ldots, x_m) = x_1^0 P_0(x_2, \ldots, x_m) + \cdots + x_1^i P_i(x_2, \ldots, x_m)$$

where we require that P_i is a non-zero polynomial (i.e. that x_1^i is the highest power of x_1 appearing in P). The polynomial P_i has degree at most $d-i$. Firstly we randomly choose values $s_2, \ldots, s_m$ for $x_2, \ldots, x_m$. By induction assumption, it holds $P_i(s_2, \ldots, s_m) \neq 0$ with probability at least $1 - \frac{d-i}{p}$. Assume this is the case and $s_2, \ldots, s_m$ are fixed. We have a nonzero polynomial $P(x_1, s_2, \ldots, s_m)$ of one variable and degree i. We choose a value s_1 for x_1. With probability at least $1 - \frac{i}{p}$ we have $P(s_1, \ldots, s_m) \neq 0$ (it is also possible that $P(s_1, \ldots, s_m) \neq 0$ in the case when $P_i(s_2, \ldots, s_m) = 0$, which might cause that the probability is higher, but we only prove a lower bound). So both of these succeed with probability

$$\geq \left(1 - \frac{d-i}{p}\right)\left(1 - \frac{i}{p}\right) \geq 1 - \frac{d}{p}$$

□

Perfect matchings in $O(n^5)$ Now we will see how to find a perfect matching (till now we can only check if it exists). Let G be a bipartite graph having a perfect matching and let $A = A(G)$. Assume that $\det A \neq 0$. The first trivial approach is to take any edge and check whether the graph without its two ends still has a perfect matching (by calculating the determinant of the matrix for this smaller graph). If yes, we choose this edge and proceed in the smaller graph; if no, we try another edge. This gives a randomized algorithm running in time $O(n^5)$ (or $O(n^{\omega+2})$, when fast Gaussian elimination is used for computing $\det A$).

Perfect matchings in $O(n^4)$ An idea proposed by Ibarra and Moran [9] was to look at the inverse matrix A^{-1}. This is useful, because we have the following theorem:

Theorem 1.4 *The submatrix of A corresponding to the graph $G - \{u_i, w_j\}$ has a nonzero determinant iff $A^{-1}_{j,i} \neq 0$.*

Proof
The theorem follows immediately from the fact that $A^{-1}_{j,i} = (-1)^{i+j} \det A_{del(i,j)} \frac{1}{\det A}$. To see this formula it is enough to multiply the matrix defined in this way with

the matrix A and see that we get the identity matrix. Indeed, on the i-th element of the diagonal of the product we have

$$\sum_{k=1}^{n} a_{i,k}(-1)^{i+k} \det A_{del(i,k)} \frac{1}{\det A} ,$$

which is 1, because without the $\frac{1}{\det A}$ factor we have the formula for $\det A$. In any other place in row i, column j of the product we have

$$\sum_{k=1}^{n} a_{i,k}(-1)^{j+k} \det A_{del(j,k)} \frac{1}{\det A} ,$$

which is 0 because without the $\frac{1}{\det A}$ factor we have the formula for the determinant of a matrix A in which row i is copied to row j (a matrix with two identical rows). □

This theorem gives an algorithm working in time $O(n^4)$ (or $O(n^{\omega+1})$): We compute A^{-1}, we find an edge (u_j, w_i) for which $A^{-1}_{i,j} \neq 0$, we take the edge to the matching and we repeat the same for the matrix $A_{del(j,i)}$ (i.e. for the graph without the vertices u_j and w_i).

Perfect matchings in $O(n^3)$ The above algorithm calculates the inverse matrix every time a row and a column is removed from the original matrix. But it is not necessary to do that from scratch every time, as the new inverse matrix differs only slightly from the previous one. Precisely, the following fact holds.

Lemma 1.5 *Let*

$$A = \begin{bmatrix} a_{1,1} & v^T \\ u & B \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} \widehat{a}_{1,1} & \widehat{v}^T \\ \widehat{u} & \widehat{B} \end{bmatrix} ,$$

where $\widehat{a}_{1,1} \neq 0$. *Then* $B^{-1} = \widehat{B} - \widehat{u}\widehat{v}^T/\widehat{a}_{1,1}$.

Proof
Since $AA^{-1} = I$, we have

$$\begin{array}{cc} a_{1,1}\widehat{a}_{1,1} + v^T\widehat{u} = I_1 & a_{1,1}\widehat{v}^T + v^T\widehat{B} = 0 \\ u\widehat{a}_{1,1} + B\widehat{u} = 0 & u\widehat{v}^T + B\widehat{B} = I_{n-1} . \end{array}$$

Using these equalities we get

$$\begin{array}{c} B(\widehat{B} - \widehat{u}\widehat{v}^T/\widehat{a}_{1,1}) = I_{n-1} - u\widehat{v}^T - B\widehat{u}\widehat{v}^T/\widehat{a}_{1,1} = \\ = I_{n-1} - u\widehat{v}^T + u\widehat{a}_{1,1}\widehat{v}^T/\widehat{a}_{1,1} = I_{n-1} - u\widehat{v}^T + u\widehat{v}^T = I_{n-1} \end{array}$$

and so $B^{-1} = \widehat{B} - \widehat{u}\widehat{v}^T/\widehat{a}_{1,1}$ as claimed. □

The modification of $\widehat{B}$ described in this lemma is in fact a single step of the well known Gaussian elimination procedure. In this case, we are eliminating

the first variable (column) using the first equation (row). Similarly, we can eliminate from A^{-1} any other variable (column) j using any equation (row) i, such that $A^{-1}_{i,j} \neq 0$.

As an immediate consequence of Lemma 1.5 we get simple $O(n^3)$ algorithm for finding perfect matchings in bipartite graphs, by Mucha and Sankowski [12]:

SIMPLE-BIPARTITE-MATCHING(G):
$B = A^{-1}(G)$
$M = \emptyset$
for $c = 1$ to n do

1. find a row r, not yet eliminated, and such that $B_{r,c} \neq 0$ and $A(G)_{c,r} \neq 0$ (i.e. (u_c, w_r) is an allowed edge in $G - V(M)$);
2. eliminate the c-th column of B using the r-th row;
3. add (u_c, w_r) to M.

Perfect matchings in $O(n^\omega)$ The last step is to get an algorithm working in time $O(n^\omega)$. Bunch and Hopcroft [1] already in 1974 showed how to do Gaussian elimination by fast matrix multiplication in time $O(n^\omega)$. It allows to speed up previous algorithms, but it is not enough here: in the above algorithm we separately do n single steps of the Gaussian elimination. However the Bunch and Hopcroft algorithm may be adopted to solve our problem.

We consider first a particularly simple case of Gaussian elimination. Assume that we are performing a Gaussian elimination on a $n \times n$ matrix B and we always have $B_{i,i} \neq 0$ after eliminating the first $i-1$ rows and columns. In this case we can avoid any row or column pivoting, and the following algorithm performs Gaussian elimination of B in time $O(n^\omega)$:

SIMPLE-ELIMINATION(B):
for $i = 1$ to n do

1. lazily eliminate the i-th column of B using the i-th row;
2. let j be the largest integer such that $2^j | i$;
3. UPDATE($\{i+1, \ldots, i+2^j\}, \{i+1, \ldots, n\}$);
4. UPDATE($\{i+2^j+1, \ldots, n\}, \{i+1, \ldots, i+2^j\}$).

By „lazy elimination" we mean storing the expression of the form uv^T/c describing the changes required in the remaining submatrix without actually performing them. These changes are then executed in batches during calls to UPDATE(R, C) which updates the $B_{R,C}$ submatrix. Suppose that k changes were accumulated for the submatrix $B_{R,C}$ and then UPDATE(R, C) was called. Let these changes be $u_1 v_1^T/c_1, \ldots, u_k v_k^T/c_k$. Then the accumulated change of $B_{R,C}$ is

$$u_1 v_1^T/c_1 + \cdots + u_k v_k^T/c_k = UV$$

where U is a $|R| \times k$ matrix with columns $u_1, \ldots, u_k$ and V is a $k \times |C|$ matrix with rows $v_1^T/c_1, \ldots, v_k^T/c_k$. The matrix UV can be computed using fast matrix multiplication.

This algorithm has time complexity $O(n^\omega)$ because of the following lemma.

Lemma 1.6 *The number of changes performed by the SIMPLE-ELIMINATION algorithm in steps 3. and 4. is at most* 2^j.

Proof
In the i-th iteration rows $i+1,\ldots,i+2^j$ and columns $i+1,\ldots,i+2^j$ are updated. Since $2^j|i$, we have $2^{j+1}|i-2^j$, so these rows and columns were also updated in step $i-2^j$. Thus, the number of changes is at most 2^j. □

It follows from this lemma, that the cost of the update in i-th iteration is proportional to the cost of multiplying the $2^j \times 2^j$ matrix by a $2^j \times n$ matrix. By splitting the second matrix into $2^j \times 2^j$ square submatrices, this can be done in time $(2^j)^\omega n/2^j = n(2^j)^{\omega-1}$. Now, every j appears $n/2^j$ times, so we get the total time complexity of

$$\sum_{j=0}^{\lceil \log n \rceil} n(2^j)^{\omega-1} n/2^j = n^2 \sum_{j=0}^{\lceil \log n \rceil} (2^{\omega-2})^j = O(n^2(2^{\omega-2})^{\lceil \log n \rceil}) = O(n^\omega) \ .$$

This finishes the proof that a naive iterative Gaussian elimination without row or column pivoting can be implemented in time $O(n^\omega)$ using a lazy updating scheme.

Now move to the real problem of finding a perfect matching in a bipartite graph. Let $A = A(G)$ be a bipartite adjacency matrix and let $B = A^{-1}$. We may still eliminate columns from left to right. However now, when eliminating column c, we can not always use row c (like in the previous simple case), but we need to choose a row r such that $B_{r,c} \neq 0$ as well as $A_{c,r} \neq 0$. The algorithm should be appropriately adopted.

The basic idea is that since columns are eliminated from left to right, we do not need to update the whole rows, only the parts that will be needed soon:

BIPARTITE-MATCHING(G):
$B = A^{-1}(G)$
$M = \emptyset$
for $c = 1$ to n do

1. find a row r, not yet eliminated, and such that $B_{r,c} \neq 0$ and $A(G)_{c,r} \neq 0$ (i.e. (u_c, w_r) is an allowed edge in $G - V(M)$);
2. lazily eliminate the c-th column of B using the r-th row;
3. add (u_c, w_r) to M;
4. let j be the largest integer such that $2^j|c$;
5. update columns $c+1,\ldots,c+2^j$.

Notice that in this case the update operation is a bit more complicated than in the SIMPLE-ELIMINATION algorithm. Suppose we update columns $c+1,\ldots,c+2^j$. These columns were updated in the $(c-2^j)$-th iteration, so we have to perform updates resulting from elimination of columns $c-2^j+1,\ldots,c$.

Let us assume without loss of generality that the corresponding rows have the same numbers (otherwise we may renumber them). The first update comes from elimination of the $(c-2^j+1)$-th row using the $(c-2^j+1)$-th column. The second update comes from elimination of the $(c-2^j+2)$-th row using the $(c-2^j+2)$-th column, but we do not know the values $B(c-2^j+2, c+1), \ldots, B(c-2^j+2, c+2^j)$ in this row without performing the first update. Fortunately, we can use the lazy computation trick again! We lazily perform the postponed updates one after another and after performing the i-th update, we only compute the actual values for the rows used in the next 2^l updates, where l is the largest number, such that $2^l|i$. In other words, for each i from 1 to 2^j we update rows $c-2^j+i+1, \ldots, c-2^j+i+2^l$ (in all columns $c+1, \ldots, c+2^j$) applying changes coming from columns/rows $c-2^j+i-2^l+1, \ldots, c-2^j+i$ (note that values in these rows have already been fully updated). In this manner rows $c-2^j+1, \ldots, c$ will be correctly updated; we also need to update all rows below c, but this may be done using straight matrix multiplication.

What is the time complexity of updating columns $c+1, \ldots, c+2^j$? We have to perform 2^j updates and performing i-th update requires multiplication of a $2^l \times 2^l$ matrix by a $2^l \times 2^j$ matrix, where l as before. This is the same situation as in the analysis of the SIMPLE-ELIMINATION algorithm, but now we have only 2^j rows instead of n. The complexity is thus $O(2^{j\omega})$. We also have to count the time required to update the rows below c and this requires multiplication of a $(n-c) \times 2^j$ matrix by a $2^j \times 2^j$ matrix. This can be done in time $O(2^{j\omega} n/2^j) = O(n(2^j)^{w-1})$. We now have to sum it up over all j, but again, this is the same sum as before and we get $O(n^\omega)$.

Maximum matchings The above algorithms were looking for a perfect matching (a matching containing all vertices). If it does not exist, one may be interested in finding a maximum matching (a matching with maximum number of edges). As previously, the basic object is the adjacency matrix of G, which now is not necesarly a square matrix.

A *rank* of a matrix A (denoted $rank(A)$) is a maximum size of a square submatrix of A with a nonzero determinant. Recall that each submatrix of A is an adjacency matrix for a graph restricted to a subset of vertices corresponding to the rows and columns of that submatrix. Hence from Theorem 1.1 immediatelly follows

Theorem 1.7 (Lovász [10]) *The rank of the symbolic adjacency matrix $\widehat{A}(G)$ is equal to the size (the number of edges) of a maximum matching of G. Moreover a maximum matching contains vertices corresponding to rows and columns of a squere submatrix of $\widehat{A}(G)$ of size $rank(\widehat{A}(G))$ with a nonzero determinant.*

Like previously, we may look at the random adjacency matrix $A(G)$:

Theorem 1.8 *With probability at least $1-n/p$ it the ranks of $\widehat{A}(G)$ and $A(G)$ are equal.*

Of course it follows from Lemma 1.3. Indeed, take any squere submatrix of $\widehat{A}(G)$ of size $rank(\widehat{A}(G))$ with a nonzero determinant. Its determinant is a

polynomial of degree $\leq n$, hence with probability $\geq 1 - n/p$ the determinant of the same submatrix in $A(G)$ is also nonzero.

From the above follows that it is enough to find (in time $O(n^\omega)$) a maximum submatrix of the random adjacency matrix $A(G)$ with a nonzero determinant. Then we may apply the BIPARTITE-MATCHING algorithm for that submatrix and we have a maximum matching in G. A maximum submatrix with nonzero determinant may be found using Gaussian elimination (which may be implemented in time $O(n^\omega)$, very similarly to the BIPARTITE-MATCHING algorithm): the columns and rows used for elimination form the demanded submatrix. Indeed, after the elimination we get a form, which is similar to the upper triangular form: everywhere under the pivots there are only zeroes, so this submatrix has a nonzero determinant. Moreover all rows not used in elimination contain only zeroes, so no bigger submatrix has nonzero determinant. The key point is that a step of the elimination procedure does not change the rank, since a step is invertible and the rank can not increase. Althrough a determinant of a submatrix may change during elimination, this is not the case for our submatrix, since all rows and columns used for elimination come from inside this submatrix.

Monte Carlo vs Las Vegas In general there are two kinds of probabilistic algorithms. *Monte Carlo algorithms* have fixed running time, but their result is correct only with some high probability. On the other hand *Las Vegas algorithms* return always correct results, but pesimistically they may work very long (their expected running time is used as a complexity measure). See that a Las Vegas algorithm can be always made a Monte Carlo algorithm with the same complexity: just run the algorithm, and when it runs too long strop it and return any result. The inverse is not true in general, one needs a (fast) procedure which checks whether the result is correct. Then one may repeat the Monte Carlo algorithm until it will return a correct answer.

All the algorithms presented above are Monte Carlo algorithms. However they can be made Las Vegas. One method is described at the end of Section 2 and works also for general graphs. The algorithm is purely algebraic, however the correctness proof is quite complicated. For bipartite graphs a much simpler method works, but it is not algebraic. Assume the algorithm has found some, in its opinion, maximum matching. We want to check if it is correct, which means that no bigger matching exists or, equivalently, no augmenting path exists. Existance of an augmenting path may be tested using a BFS algorithm in time $O(n^2)$ (see [4]), so the expected running time of the Las Vegas algorithm remains $O(n^\omega)$.

2 General graphs

In this section we show how the above ideas generalise to an arbitrary, not necessarly bipartite, graph. Let now $n = |V|$ and $V = \{v_1, \ldots, v_n\}$.

Classic algorithms Solving the maximum matching problem in time polynomial in n remained an elusive goal for a long time until Edmonds [5] gave the first algorithm, with running time $O(|V|^2|E|) = O(n^4)$. Several additional improvements culminated in the $O(\sqrt{|V|}|E|) = O(n^{2.5})$ algorithm of Micali and Vazirani [11]. Unfortunately, both these algorithms mentioned above, as well as the others, are quite complicated. In my oponion the randomized $O(n^3)$ algebraic algorithm presented below is the simplest one.

Adjacency matrix We need an improved version of a adjacency matrix, since now edges may lead between any two vertices. A *skew symmetric adjacency matrix* of G is an $n \times n$ matrix $\widehat{A}(G)$ such that

$$\widehat{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } (v_i, v_j) \in E \text{ and } i < j \\ -x_{j,i} & \text{if } (v_i, v_j) \in E \text{ and } i > j \\ 0 & \text{otherwise} \end{cases}$$

where the $x_{i,j}$ are unique variables corresponding to edges of G (each variable apears twice in the matrix).

Theorem 1.1 is true also for an adjacency matrix defined in this way:

Theorem 2.1 (Tutte [15]) *For any graph G the symbolic determinant* $\det \widehat{A}(G)$ *is non-zero iff G has a perfect matching.*

Proof
Assume first that there is a perfect matching M. Look at the component of $\det \widehat{A}(G)$ containing $\det \widehat{A}(G)_{i,j}$ and $\det \widehat{A}(G)_{j,i}$ for each $(v_i, v_j) \in M$. This is a multiplication of $x_{i,j}^2$ for each $(v_i, v_j) \in M, i < j$, so it is nonzero. Moreover such component appears only once (so it does not reduce with some other its copy), no other component may have all these variables in squeare. So $\det \widehat{A}(G) \neq 0$.

Assume now that $\det \widehat{A}(G) \neq 0$. Recall that

$$\det \widehat{A}(G) = \sum_{p \in \Pi_n} \sigma(p) \prod_{i=1}^{n} \widehat{A}(G)_{i,p_i} \ .$$

Consider a permutation p with an odd-length cycle. Let p' be the permutation p with one odd-length cycle reversed (say this which contains the smallest number through all numbers in all odd-length cycles). Compare the components for p and for p': odd number of factors $\widehat{A}(G)_{i,p_i}$ is replaced by $\widehat{A}(G)_{p_i,i} = -\widehat{A}(G)_{i,p_i}$. Moreover $\sigma(p) = \sigma(p')$, so one component is opposite to the other and they reduce. Hence (if $\det \widehat{A}(G) \neq 0$) there is some permutation p with all cycles of even length such that each $\widehat{A}(G)_{i,p_i}$ is nonzero. Such permutation gives a set of cycles of even length in G such that each vertex is on exactly one cycle. It is enough to take every second edge from each cycle to get a perfect matching. □

Like previously we may substitute random numbers for the variables in $\widehat{A}(G)$, getting a random adjacency matrix A; an analogue of Theorem 1.2 holds. It allows us to check in time $O(n^3)$ if a perfect matching exists and to find one in time $O(n^5)$.

Perfect matchings in $O(n^4)$ and $O(n^3)$ We consider the same $O(n^4)$ algorithm as for bipartite graphs: compute A^{-1}, find an edge (v_i, v_j) for which $A^{-1}_{i,j} \neq 0$, take the edge to the matching and repeat the same for the matrix A without the i-th and the j-th row and column. This algorithm in the general case was invented by Rabin and Vazirani [13] For correctness we need some analogue of Theorem 1.4:

Theorem 2.2 *Assume* $\det A \neq 0$. *The submatrix of A corresponding to the graph $G - \{v_i, v_j\}$ has a nonzero determinant iff $A^{-1}_{j,i} \neq 0$.*

The proof is a little bit more complicated now: to get the submatrix of A corresponding to $G-\{v_i, v_j\}$ we need to remove from A both the i-th and the j-th row and both the i-th and the j-th column. On the other hand $A^{-1}_{i,j}$ guaranties only that after removing the j-th row and the i-th column we get a nonzero determinant (after removing the second row and column the determinant might potentially become zero, but this in not the case). In the proof we use three easy facts from the linear algebra. First, when after removing one row (or column) from a matrix, its rank may decrease by at most one. Second, when some rows are removed from a matrix, the rank stays the same iff the removed rows are linear combinations of the remaining rows. Third, a determinant of any skew symmetric matrix of odd size is zero. Indeed, $\det B = \det B^T = \det(-B) = \det B$ (the last equality is true only for B of odd size; for B of even size we have $\det(-B) = \det B$). In the proof we also need the following lemma (which is used also further):

Lemma 2.3 *For any skew symmetric matrix A, if $A_{R,C}$ is a square submatrix of A of size $rank(A)$ with a nonzero determinant, then $A_{R,R}$ also has a nonzero determinant.*

Proof
Look at the subatrix $A_{R,*}$ consisting of rows R and all columns. Its rank is equal to $rank(A)$. Hence all rows outside R are linear combinations of the rows in R. Since A is skew symmetric, also all columns outside R are linear combinations of columns in R. This means that $rank(A_{R,*}) = rank(A_{R,R})$, thus $A_{R,R}$ has a nonzero determinant. □

Proof
From theorem 1.1 the right side is equivalent to $\det A_{del(i,j)} \neq 0$.

Assume first that the left side holds: $\det A_{del(\{i,j\},\{i,j\})} \neq 0$. As we have already said, $\det A_{del(i,j)} = 0$ (since the matrix is skew symmetric of odd size), so the ranks of both these matrices are $n-2$. Removing the j-th column from $A_{del(i,j)}$ do not change the rank, so this column is a linear combination of the other columns (we mean here columns consisting of all rows except the i-th row). Consider the matrix A withot row i. Its rank is $n-1$. As its j-th column is a linear combination of the other columns (as previously here we also have columns without row i), removing the j-th column does not change the rank, hence the rank of $A_{del(i,j)}$ is $n-1$, which is what we should prove.

Now assume that that the left side does not hold. Then the rank of $A_{del(\{i,j\},\{i,j\})}$ is at most $n-3$. When it is $n-3$, by Lemma 2.3 there is a skew symmetric matrix of size $n-3$ with nonzero determinant, which is impossible, sinse $n-$ is odd. Hence the rank of $A_{del(\{i,j\},\{i,j\})}$ is at most $n-4$. It is get from A by removing two rows and two columns, hence after removing each of them the rank should decrease by one, so the rank of $A_{del(i,j)}$ is $n-2$. □

Like in the bipartite case, thanks to Lemma 1.5 we get a simple $O(n^3)$ algorithm:

SIMPLE-BIPARTITE-MATCHING(G):
$B = A^{-1}(G)$
$M = \emptyset$
for $c = 1$ to n do
  if column c is not yet eliminated then

1. find a row r, not yet eliminated, and such that $B_{r,c} \neq 0$ and $A(G)_{c,r} \neq 0$ (i.e. (v_c, v_r) is an allowed edge in $G - V(M)$);
2. eliminate the c-th column of B using the r-th row;
3. eliminate the r-th column of B using the c-th row;
4. add (v_c, v_r) to M.

Perfect matchings in $O(n^\omega)$ For bipartite graphs we could use the lazy computation mechanism because columns could be processed from left to right. In the general case we need to process both rows and columns in an arbitrary order; the lazy computation mechanism does not work any more.

A first randomized $O(n^\omega)$ algorithm for perfect matchings was given by Mucha and Sankowski [12]. It relies on a nontrivial structural decomposition of graphs called the ,,canonical partition", and uses sophisticated dynamic connectivity data structures to maintain this decomposition online.

A little simpler and strictly algebraic algorithm was given by Harvey [7]. It uses a divide-and-conquer approach. Assume that the number of vertices of G is a power of two. It can be easily achieved by adding new pairs of vertices connected by an edge. The algorithm is outlined below:

FindAllowedEdges(S, B)
  If this S was already processed then return
  If $|S| > 2$ then
    Partition S into 4 equal-sized parts $S_1, \ldots, S_4$
    For each unordered pair $\{S_a, S_b\}$ of parts
      FindAllowedEdges($S_a \cup S_b, B$)
      Update B
  Else
    This is a base case: S consists of two vertices i and j
    If $A_{i,j} \neq 0$ and $B_{i,j} \neq 0$ (i.e. edge $\{i, j\}$ is allowed) then
      Add $\{i, j\}$ to the matching

We start the function with $S = V$ and $B = A^{-1}(G)$. The arguments to recursive calls (in particular the matrix B) is passed by value, i.e. it is coppied and changes from recursive calls are not visible outside. The ,,Update B" command means that we eliminate rows and columns corresponding to vertices added to the matching by the last recursive call to FindAllowedEdges. In particular values in these rows and columns of B are made zero.

First see correctness of the algorithm. A crutial observation is that the algorithm considers each pair of vertices in at least one base case. The proof is an easy inductive argument: fix a pair of vertices $\{i, j\}$, and note that at each level of the recursion, at least one unordered pair of parts $\{S_a, S_b\}$ has $\{i, j\} \subseteq S_a \cup S_b$. We may always look at the first such pair, then for sure $S_a \cup S_b$ was not processed before, so the first condition would not return immediatelly from the procedure. Hence, the algorithm is simply a variant of the Rabin-Vazirani $O(n^4)$ algorithm that considers edges in an unusual order.

Now analyse the complexity. Let us suppose for now that the updating scheme requires only $O(|S|^\omega)$ time for a subproblem with set S; this will be demonstrated later. To get the desired complexity we need additionally assume that the partition of S into 4 parts is done in a friendly way: there are only 2^{i+1} parts on level i (all parts are disjoint). This can be easily guaranted (fix some order of vertices, choose the parts such that they consist of $n/2^{i+1}$ consequtive vertices, then S consists always of two fragments of consequtive vertices). The issue of checking the first condition may be solved by a bit vector (of size $O(n^2)$) indicating which subproblems have been solvedl queries and updates to the vector are done in constant time. Let us analyse the recursion. At level i the size of a subproblem is $n2^{-i}$ and the number of subproblems is $\binom{2^{i+1}}{2} \leq 2^{2i+1}$. The total time to apply updates at level i is $O((n2^{-i})^\omega 2^{2i}) = O(n^\omega 2^{-(\omega-2)i})$. Suming over all levels yields a bound of $O(n^\omega)$ (assuming $\omega > 2$; if $\omega = 2$ we get $O(n^2 \log n)$).

The only thing left is to do the update of B. Note that we does not need to update the whole B. Only the part corresponing to S is used, so it is enough to update this part. We need to eliminate columns and rows corresponing to the vertices added to the matching in the call to FindAllowedEdges which have just returned. So we already know which rows/columns should be eliminated and in which order (in fact any order gives the same result). Hence we may eliminate them using the lazy elimination scheme. Namely, we may renumber the columns and rows so that in the i-th step we should eliminate column i using row i. Then we may simply apply the $SIMPLE - ELIMINATION$ algorithm (with the only difference that no all rows/columns are eliminated, only some fixed number of them). Thus the update may be done in $O(|S|^\omega)$. Beside the updates, we also need to copy the matrix B while doing a recursive call. But, againg, only the fragent corresponding to B is needed, so it takes time $O(|S|^2)$.

Maximum matchings Like for bipartite graphs, with probability $1 - n/p$ the ranks of $\widehat{A}(G)$ and $A(G)$ are equal. Of course rank can not be greater than

the maximum size of matching: a matching gives a submatrix of the same size with a nonzero determinant. The inverse is not immediate: it is possible that a submatrix $A_{R,C}$ which witnessess the rank does not correspond to a subgraph, because R is different than C. But according to Lemma 2.3 also $A_{R,R}$ has a nonzero determinant, so in the subgraph corresponding to rows R there is a perfect matching. We may find a submatrix witnessess the rank as previously, using Gaussian elimination.

Monte Carlo vs Las Vegas There is an probabilistic algebraic algorithm, which allows to find the size of a maximum matching, but makes mistakes in the opposite direction: if the random adjacency matrix is incorrectly choosen it may return too much. The algorithm is based on the Gallai-Edmonds decomposition of a graph. So we have two algorithm calculating the size of a maximum matching, one which may return too less and other which may return too much. Hence both the algorithms returned correct values iff the values are the same. The algoithm comes from [2].

To be extended...

A The fast matrix multiplication algorithm

To be written...

References

[1] J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[2] J. Cheriyan. Randomized $\widetilde{O}(M(|V|))$ algorithms for problems in matching theory. *SIAM J. Comput.*, 26(6):1635–1669, 1997.

[3] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, pages 1–6, New York, NY, USA, 1987. ACM.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[5] J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[6] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] N. J. A. Harvey. Algebraic structures and algorithms for matching and matroid problems. In *FOCS*, pages 531–542, 2006.

[8] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[9] O. H. Ibarra and S. Moran. Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication. *Inf. Process. Lett.*, 13(1):12–15, 1981.

[10] L. Lovász. On determinants, matchings, and random algorithms. In *FCT*, pages 565–574, 1979.

[11] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27, 1980.

[12] M. Mucha and P. Sankowski. Maximum matchings via gaussian elimination. In *FOCS*, pages 248–255, 2004.

[13] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *J. Algorithms*, 10(4):557–567, 1989.

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

[15] W. T. Tutte. The factorization on linear graphs. *J. London Math. Soc.*, 22:107–111, 1947.

[16] R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, pages 216–226, London, UK, 1979. Springer-Verlag.