Perfect Matching in Parallel Computation

A thesis submitted in fulfillment
of the requirements for the degree of
Master of Science

by

Roded Sharan

supervised by
Prof. Avi Wigderson

Computer Science Department
The Hebrew University of Jerusalem
Jerusalem, Israel

June 1995

Acknowledgments

I am grateful to Avi Wigderson for his guidance, helpful ideas, and sincere efforts.
I want to thank Mike Luby for a fruitful conversation.
And lastly, I am eternally indebted to my parents, that stood by me all the way.

Contents

Chapter 1

Introduction

1.1 Introduction

1.1.1 Background

Given a graph $G = (V, E)$ with n vertices and m edges, a *Matching* is a set of edges $M \subset E$, such that no two edges of M are adjacent in G. A *Maximum Matching* is a matching of maximum cardinality in G. A *Perfect Matching* is a matching which is incident to all vertices of G. The *Perfect Matching Decision Problem* is to determine if G has a perfect matching. The *Search Problem* is to exhibit a perfect matching, if such exists. The *Maximum Matching Problem* is to find a matching of maximum cardinality in G.

The maximum matching problem is one of the most basic and well studied problems in combinatorics and computer science. Its importance is in it being from many aspects a universal problem. Aside of being a basic combinatoric problem, it has also many practical applications, such as routing in a network. Moreover, many fundamental problems have reductions to maximum matching, and perhaps the best example is the problem of maximum flow in a network.

The first sequential polynomial algorithm for the maximum matching problem was given by Edmonds [6], requiring $O(n^3)$ time. This algorithm as many others following it, solves the problem by finding a partial matching and augmenting it to a maximum matching using augmenting paths. The best algorithms currently known for this problem, by Blum [1] and by Micali and Vazirani [21] work in time $O(m\sqrt{n})$.

The methods used to solve the maximum matching problem for the sequential case, do not seem to parallelize. Even the problem of finding a single augmenting path is not known to be in $\mathcal{NC}$. New techniques, using tools from linear algebra, were developed to cope with the problem. All these methods are based on a theorem by Tutte [28] which reduces the perfect matching decision problem to deciding if a certain Tutte-matrix of the input graph is non-singular. This also lead to an $\mathcal{RNC}$ algorithm for solving the decision problem (See [17]).

The first parallel $\mathcal{RNC}$ algorithm for constructing a maximum matching was given by Karp, Upfal and Wigderson [15]. Another algorithm was later given by Mulmuley, Vazirani and Vazirani [22], and a Las-Vegas extension of both algorithms was given by Karloff [13], by bounding the size of a maximum matching from above. These two algorithms are described in the next section.

Deterministic $\mathcal{NC}$ algorithms, even for the decision problem, are not known. An outstanding open problem in parallel computation is to find a deterministic $\mathcal{NC}$ algorithm for the perfect matching problem. Sublinear time algorithms were given by Goldberg, Plotkin and Vaidya [10] - requiring $O(n^{2/3})$ time, and by Goldberg, Plotkin, Shmoys and Tardos [9] - requiring $O(\sqrt{m}\log^3 n)$ time.

Special cases of the perfect matching search problem turned out to be in $\mathcal{NC}$. Lev, Pippenger and Valiant [16] gave an $\mathcal{NC}$ algorithm to find a perfect matching in bipartite regular graphs. Vazirani [29] gave an $\mathcal{NC}$ algorithm to count the number of perfect matchings in $K_{3,3}$-free graphs, and Grigoriev and Karpinski [11] solved the the search problem for the case that the graph has only polynomially many matchings.

A problem closely related to the perfect matching problem is the *Bipartite Ear Decomposition Problem*:

Given a bipartite graph $G = (U, V, E)$, a *Bipartite Ear Decomposition* of G to r ears, is a decomposition $G = x + P_1 + ... + P_r$ such that: $x \in E$, and for each $i = 1 \ldots r$, the i'th ear P_i, is an open path of odd length with only its end vertices in $x + P_1 + ... + P_{i-1}$. The bipartite ear decomposition problem is, given a bipartite graph, to find a bipartite ear decomposition for it, if one exists.

The bipartite ear decomposition problem was first presented by Lovasz in [18], and it is still an open question whether there exists an $\mathcal{NC}$ algorithm to solve it. It is interesting to note, that the *Ear Decomposition Problem*, in which one is asked to find an ear decomposition for a general graph, with no requirements made on the length of the ears, is in $\mathcal{NC}$ (See [18],[20]).

1.1.2 New Results

Our work has two main parts.

In the first part, we present a new approach towards the perfect matching problem. This approach yields a new $\mathcal{NC}$ algorithm to solve the perfect matching search problem for bipartite 3-regular graphs.

Let $G = (V, E)$ be the input graph, $|V| = n$. Our Algorithm starts with a *Pseudo Perfect Matching* of G, which is a spanning subgraph of G with every vertex of odd degree in it. We then show how to efficiently augment this pseudo perfect matching until a perfect matching is produced.

The augmentation is done by finding certain cycle vectors in G, and xoring them to the pseudo perfect matching, reducing the number of edges in it. In other words the approach we take is augmenting by decreasing the number of edges, rather than augmenting by increasing as done in most sequential algorithms for the problem.

In total $O(\log n)$ phases are carried out. The input to each phase is a pseudo perfect matching, and the output of each phase is a new pseudo perfect matching, with number of 3-degree vertices in it, reduced by a constant factor. The algorithm works in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors.

As for the general case, we develop sufficient conditions for efficiently augmenting a pseudo perfect matching in a general graph, reducing its number of 3-degree vertices by a constant factor.

In the second part of our work, we show an $\mathcal{NC}$-equivalence between the perfect matching problem and the bipartite ear decomposition problem. The novel part of this equivalence relation is an $\mathcal{NC}$ reduction from the bipartite ear decomposition problem to the perfect matching problem. This reduction yields a first $\mathcal{RNC}$ algorithm for the bipartite ear decomposition problem.

1.2 Basic Definitions

1.2.1 Introduction to Graphs

An *Undirected Graph* G is composed of a finite non-empty set of elements $V(G)$ called *vertices*, and a multi-set of unordered pairs of vertices $E(G) \subset V(G) \times V(G)$ called *edges*. An edge from a vertex to itself is called a *Loop*. A *Directed* graph, is a graph in which the edges are ordered pairs of vertices. A *Simple* graph is a graph in which multiple edges and loops are not allowed. We will use the notation $G = (V, E)$ to denote a graph G with a set of vertices V and a set of edges E. Also $E(G)$ will denote the set of edges in G, and $V(G)$ the set of vertices in G.

Let G be a graph. The *Size* of G is its number of vertices. If G is undirected then the number of edges incident to a vertex is called the *Degree* of that vertex. If G is directed then the number of edges going out of a vertex is called the *Outdegree* of that vertex, and the number of edges going into a vertex is called the *Indegree* of that vertex. A *k-Regular* graph is an undirected graph, in which all vertices are of degree k.

A *Bipartite Graph* $G = (A, B, E)$ is a graph in which $V(G) = A \cup B$, and all edges in G join a vertex in A to a vertex in B. A, B are called the *Color Classes* of G.

A *Subgraph* of G is a graph $H = (U, F)$ such that $U \subset V$ and $F \subset E$. H is an *Induced Subgraph* of G if $\forall u, v \in U, (u, v) \in F \Leftrightarrow (u, v) \in E$. For any set of vertices $S \subset V$ we denote by $G[S]$ the subgraph of G induced by the vertices in S. H will be called a *Spanning Subgraph* if $U = V$.

A sequence of vertices $(v_1, \ldots, v_n)$ all distinct perhaps maybe the first and the last, defines a *Path*, which is composed of these vertices and the edges between them, (v_i, v_{i+1}) $i = 1 \ldots n-1$. A path is called *Simple*, if all its vertices are distinct, except maybe $v_1 = v_n$. An *Open Path* is a path in which the first and last vertices differ. If $v_1 = v_n$ the path is called a *Closed Path* or a *Cycle*.

Let $P = (v_1, \ldots, v_k)$ be an open path, $k > 1$. v_1, v_k are called the *End Vertices* of P. $(v_1, v_2), (v_{k-1}, v_k)$ are called *End Edges* of P. $v_2, \ldots, v_{k-1}$ are called *Internal Vertices* of P. A path with all internal vertices of degree 2 in G, is called a *2-Path*.

The *Length* of a path is defined as its number of edges. The *Distance* between two vertices in a graph is the length of the shortest path connecting them.

G is called a *Connected* graph, if there is a path between every pair of vertices in G. A maximal connected subgraph of G is called a *Connected Component*.

A *Tree* is a connected cycle-free graph. The *Leaves* of a tree are its vertices of degree 1. A *Spanning Tree* T of a graph G, is a tree for which $V(T) = V(G)$.

A *Rooted Tree* is a directed graph $T = (U, F)$ with a special vertex r, such that: Each $v \in U - \{r\}$ has outdegree 1; The outdegree of r is 0; and for every $v \in U - \{r\}$ there exists a directed path from v to r. For rooted trees, the *Height* of a tree is defined to be the length of the longest path from a leaf of that tree to its root.

Let $T = (U, F)$ be a tree, rooted at r. For a vertex $v \in U - \{r\}$, let (v, u) be the only edge going out of v, then u is called the *Father* of v, and v is called the *Son* of u. The *Lowest Common Ancestor* of any two vertices u, v in T, is a vertex w that is the ancestor of both u and v, and is farthest from r.

A *Breadth First Search* of a rooted tree, is a search beginning at the root of the tree, such that at step i all vertices of distance i from the root are searched, $0 \leq i \leq h$, where h denotes the height of the tree.

A *Forest* is a graph in which all connected components are trees. Given a tree $T = (U, F)$ in G, the set of *Tree-Edges* is F, and the set of *Non-Tree* edges is $E - F$. *Forest-edges* and *Non-Forest* edges are defined similarly. An *Induced Tree* in G, is a tree which is an induced subgraph of G. An *Induced Forest* in G is a forest in which all connected components are induced trees.

Let $V = \{v_1 \ldots v_n\}$. The *Incidence Matrix* of G is a matrix $D = (d_{ij})$ of size $n \times n$, defined as follows:

$$d_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Any subgraph of G can be described by its incidence vector with respect to the set of edges E. Let F be the set of all vectors representing cycles in G. Let $\overline{F}$ be the closure of F under the xor operation. Then $\overline{F}$ can be viewed as a space vector over $GF(2)$, with the xor operation. This space vector is called the *Cycle Space* of G, and its dimension is $m - n + 1$, provided G is connected. A vector in the cycle space, i.e. a union of cycles, is called a *Cycle Vector*.

A set of *Fundamental Cycles* is a collection of cycles composing a basis of the cycle space of G. A set of fundamental cycles can be obtained the following way: Assume G is connected, and let T be any spanning tree of G. Any non-tree edge added to T, forms a unique cycle, composed of this edge and paths from its vertices to their lowest common ancestor. The collection of cycles defined by non-tree edges with respect to a spanning tree of G is a set of fundamental cycles. If G is not connected, we obtain a set of fundamental cycles in a similar way, by taking the cycles defined by non-forest edges with respect to a spanning forest.

A *Matching* M of G, is a set of vertex disjoint edges in G. If the matching M is incident to all the vertices in G, then M is a *Perfect Matching* of G. A *Maximum Matching* of G is a matching of maximum cardinality in G. Given a matching M, an *M-Alternating* path, is a path with edges alternately in and not in M. An *M-Augmenting* path is an M-alternating path with its end edges in $G - M$.

The *Perfect Matching Decision Problem* is, given a graph G, to determine if G has a perfect matching. The *Search Problem* is to exhibit a perfect matching, if one exists. The *Maximum Matching Problem* is to find a matching of maximum cardinality in G.

An *Ear Decomposition* of a graph G to r ears, is a decomposition $G = x + P_1 + ... + P_r$ such that $x \in E$, and for each $i = 1 \ldots r$, the i'th ear P_i, is a path with only its end vertices in $x + P_1 + ... + P_{i-1}$.

Let $G = (U, V, E)$ be a bipartite graph. A *Bipartite Ear Decomposition* of a bipartite graph G to r ears, is a decomposition $G = x + P_1 + ... + P_r$, such that $x \in E$, and for each $i = 1 \ldots r$, the i'th ear P_i, is an open path of odd length with only its end vertices in $x + P_1 + \ldots + P_{i-1}$.

The *Bipartite Ear Decomposition Problem* is to find a bipartite ear decomposition for G, if one exists.

1.2.2 Parallel Computation

A *Parallel Random Access Machine (PRAM)* is a theoretical model of parallel computation, in which a number of processors are working synchronously and communicating via a common random access memory. The model admits full parallelism: Any processor can access any cell of the shared memory in unit time.

The processors, that are random access machines, synchronously execute the same program, though each processor might work on different storage locations. In this model read-write conflicts may arise if two processors or more try to read from or write into the same memory location simultaneously. In the *Exclusive-Read Exclusive-Write (EREW)* PRAM, concurrent reading or writing are not allowed. In the *Concurrent-Read Exclusive-Write (CREW)* PRAM, any number of processors may simultaneously read from the same memory cell, but concurrent write is forbidden. The *Concurrent-Read Concurrent-Write (CRCW)* PRAM model permits both concurrent reading and concurrent writing.

The class $\mathcal{NC}$ is defined to be the class of problems that can be solved by deterministic algorithms in polylogarithmic time using a polynomial number of processors on a PRAM machine. This is the class of algorithms reckoned to be feasible in parallel computation.

The class $\mathcal{RNC}$ is defined to be the class of problems that can be solved by randomized algorithms with bounded error in polylogarithmic time using a polynomial number of processors on a PRAM machine.

For a survey on models of parallel computation and parallel algorithms see [8], [12], [14], and [25].

1.3 Previous Results

This section gives a survey of current randomized parallel algorithms for the perfect matching decision problem, and for the perfect matching search problem. Both problems are not known to be in $\mathcal{NC}$.

The first subsection presents an $\mathcal{RNC}$ algorithm, based on an idea by Lovasz [17], for testing if an input graph has a perfect matching. Lovasz showed an $\mathcal{RNC}$ reduction from the perfect matching decision problem to the problem of testing whether a specific integer Tutte-matrix is non-singular. This problem was shown by Borodin, von zur Gathen and Hopcroft [3] to be in $\mathcal{NC}$, yielding an $\mathcal{RNC}$ algorithm for the perfect matching decision problem.

The second subsection presents an $\mathcal{RNC}$ algorithm by Karp, Upfal and Wigderson [15] for constructing a perfect matching in a given graph, if such exists. Their algorithm starts with the set of all edges of the graph and each phase finds a large set of redundant edges and removes it, until a sparse graph is obtained. The problem is then easily solved.

The last subsection presents another $\mathcal{RNC}$ algorithm by Mulmuley, Vazirani and Vazirani [22] for this problem. This algorithm is based on a probabilistic lemma which ensures that when weights are assigned randomly to the edges of the graph, with high probability a unique minimum weight perfect matching exists. This unique perfect matching is then found using algebraic techniques.

1.3.1 An $\mathcal{RNC}$ Algorithm for the Perfect Matching Decision Problem

The algorithm for the perfect matching decision problem is based on a theorem by Tutte, stating that a graph has a perfect matching iff its Tutte matrix, which is a matrix of indeterminates, is non-singular. The problem of evaluating the determinant of the Tutte matrix is then reduced using randomization to the problem of computing the determinant of an integer matrix, which is in $\mathcal{NC}$ (See [3]).

Definition 1.3.1 *Let $G = (V, E)$ be a graph with no loops, $|V| = n$. Let $D = (d_{ij})$ denote the adjacency matrix of G. Let $x_{11} \ldots x_{nn}$ denote indeterminates.*

The **Tutte Matrix** *of G is an $n \times n$ skew-symmetric matrix $A = (a_{ij})$, defined as follows:*

$$a_{ij} = \begin{cases} x_{ij} & \text{if } d_{ij} = 1 \text{ and } i < j \\ -x_{ij} & \text{if } d_{ij} = 1 \text{ and } i > j \\ 0 & \text{if } d_{ij} = 0 \end{cases}$$

Definition 1.3.2 *Let A be an $n \times n$ matrix. For $\sigma \in S_n$ define:*

$$value_A(\sigma) = \prod_{i=1}^{n} a_{i\sigma_i}$$

Call σ **non-vanishing** *if $value_A(\sigma) \neq 0$.*

Definition 1.3.3 *Let $G = (V, E)$ be a graph, and let $\sigma \in S_n$. The* **trail** *of σ is defined as the set of ordered pairs $(v_i, v_{\sigma(i)})$, $1 \leq i \leq n$.*

Theorem 1.3.4 (Tutte [28]) *Let $G = (V, E)$ be a graph, and let A denote its Tutte-matrix. Let $|A|$ denote the determinant of A. Then $|A| \neq 0$ iff G has a perfect matching.*

Proof: ($\Leftarrow$): Assume M is a perfect matching of G. If we substitute the indeterminates in A, such that if $(v_i, v_j) \in M$ then $x_{ij} = 1$ and otherwise $x_{ij} = 0$, we get $|A| = (-1)^{n/2} \neq 0$.

($\Rightarrow$): We assume now that $|A| \neq 0$ and prove that there exists a perfect matching in G. By definition,

$$|A| = \sum_{\sigma \in S_n} sign(\sigma) \times value_A(\sigma)$$

and each permutation σ, in this sum, is non-vanishing iff $(v_i, v_{\sigma(i)}) \in E$ for all $1 \leq i \leq n$.

We now note that the trail of a permutation is a collection of disjoint cycles and single edges. If the trail of $\sigma \in S_n$ contains an odd cycle then there exists a permutation $\overline{\sigma} \in S_n$ such that the trail of $\overline{\sigma}$ is the same as the one of σ, except that the odd cycle is traversed in the opposite direction. Obviously, $value_A(\sigma) = -value_A(\overline{\sigma})$ and $sign(\sigma) = sign(\overline{\sigma})$. We conclude that permutations containing an odd cycle do not contribute to $|A|$.

Since $|A| \neq 0$, there exists a permutation $\sigma \in S_n$, such that the trail of σ is a perfect matching or contains only even cycles. But if the trail of

σ contains only even cycles we can pick from this trail all single edges and alternate edges from every even cycle, and obtain a perfect matching. ■

Following Tutte's theorem, the problem of deciding if a perfect matching exists, reduces to the problem of determining if the Tutte matrix of the graph is non-singular. To overcome this problem, we use a theorem by Schwarz [26].

Theorem 1.3.5 (Schwarz [26]) *Let $P(x_1, \ldots, x_n)$ be a polynomial in n variables over $\mathcal{Z}$, of degree $d \geq 1$. Let S be a non-empty set of integers. Let $a_1, \ldots, a_n$ be chosen randomly and independently from S. If P is not identically zero, then*

$$Pr[P(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

An $\mathcal{RNC}$ algorithm for the perfect matching decision problem is now straightforward. Since the determinant of a Tutte matrix is a polynomial of degree n, if we take $|S| = 2n$ then the substituted integer matrix will be non-singular with probability at least $1/2$, provided $|A| \neq 0$. Repeating this substitution or taking $|S|$ of larger (but still polynomial) size, we can reduce the probability of error. The determinant of the substituted matrix can be computed using the algorithm of Pan [23] in $O(\log^2 n)$ time using $M(n)$ processors, where $M(n)$ denotes the number of arithmetic operations required to multiply two $n \times n$ matrices over a ring (currently $M(n) = O(n^{2.376})$). The complexity of the algorithm is therefore $O(\log^2 n)$ time using $O(M(n))$ processors.

1.3.2 Constructing a Perfect Matching is in Random $\mathcal{NC}$

This subsection presents an $\mathcal{RNC}$ algorithm by Karp, Upfal and Wigderson [15] for the perfect matching search problem. The heart of their algorithm is a rank function which enables computing a large set of redundant edges, thereby reducing the input graph to a sparse graph, in which the problem is easily solved.

Let $G = (V, E)$ be the input graph, with n vertices and m edges. The algorithm differentiates between two cases. The first case, is when $m < \frac{3}{4}n$, i.e. the graph is sparse. In this case, assuming G has a perfect matching, at least half of its vertices are of degree 1, so we can identify at least $\frac{1}{4}n$ edges that must appear in a perfect matching of G, thus removing a constant fraction of the edges in G.

The second and more complicated case is when the graph is dense, i.e. $m \geq \frac{3}{4}n$. Here we use a procedure to find a large set R of redundant edges, such that with probability $1 - o(1/m^2)$, the graph $(V, E - R)$ has a perfect matching. This set is with constant probability $b > 0$ with size at least cm, for some constant $c > 0$.

It follows that with high probability, after $O(\log n)$ iterations, a perfect matching is obtained. We will describe below the algorithm for finding a large set of redundant edges.

Definition 1.3.6 *Let $G = (V, E)$ be a graph, and let $S \subset E$. Let P denote the set of all perfect matchings in G. We define*

$$Rank(S) = Max_{M \in P}|S \cap M|$$

Thus, $Rank(S)$ is the maximum number of edges from S that occur together in a perfect matching of G.

Lemma 1.3.7 *Let $S \subset E$. Let $R = \{e \in E - S \mid Rank(S \cup \{e\}) = Rank(S)\}$. If G has a perfect matching, then the graph $(V, E - R)$ has a perfect matching.*

Proof: Let A be a perfect matching in G such that $|A \cap S| = Rank(S)$. For each $e \in R$, $(S \cup \{e\}) \cap A = S \cap A$, so $e \notin A$. Therefore A is a perfect matching in $(V, E - R)$. ■

The algorithm for finding a redundant set is as follows:

Find Redundant Set:

- Choose randomly and uniformly $i \in \{1, \ldots, 5/6m\}$.
- Choose randomly a set $S \subset E$ from the uniform distribution over the i-element subsets of E.
- Let $R = \{e \in E - S \mid Rank(S \cup \{e\}) = Rank(S)\}$.

Lemma 1.3.8 *Let R denote the redundant set output of the algorithm, then*

$$Pr[|R| \geq m/60] \geq 1/60$$

.

Proof: Let p_i denote the probability that for an edge drawn at random from $E - S$, $Rank(S \cup \{e\}) > Rank(S)$, given that $|S| = i$. Linearity of expectation gives

$$E[Rank(S) \mid |S| = i] = \sum_{j=0}^{i-1} p_j$$

and since $Rank(E) = \frac{n}{2}$ we conclude that

$$\sum_{j=1}^{5/6m} p_j \leq \sum_{j=0}^{m-1} p_j = \frac{n}{2} \leq \frac{2}{3}m$$

We can now bound the expectation of $|R|$:

$$E[|R|] = \frac{1}{\frac{5}{6}m} \sum_{i=1}^{\frac{5}{6}m} (1 - p_i)(m - i) \geq$$

$$\frac{1}{\frac{5}{6}m} \sum_{i=1}^{\frac{5}{6}m} (1 - p_i)\frac{1}{6}m =$$

$$\frac{1}{5}(\frac{5}{6}m - \sum_{i=1}^{\frac{5}{6}m} p_i) \geq m/30$$

On the other hand,

$$E[|R|] = \sum_{i=1}^{m} i(Pr[|R| = i]) \leq \frac{m}{60} Pr[|R| < \frac{m}{60}] + mPr[|R| \geq \frac{m}{60}]$$

So, $Pr[|R| \geq \frac{m}{60}] \geq \frac{1}{60}$. ■

It remains to show how to compute the Rank function efficiently in parallel.

Theorem 1.3.9 *Let $G = (V, E)$, $S \subset E$. Define the $n \times n$ matrix $B^S = (b^S_{ij})$ as follows:*

$$b^S_{ij} = \begin{cases} yx_{ij} & \text{if } (i,j) \in S \text{ and } i < j \\ -yx_{ij} & \text{if } (i,j) \in S \text{ and } i > j \\ x_{ij} & \text{if } (i,j) \in E - S \text{ and } i < j \\ -x_{ij} & \text{if } (i,j) \in E - S \text{ and } i > j \\ 0 & \text{otherwise} \end{cases}$$

Denote the degree of y in $|B^S|$ by $d(B^S)$, then $d(B^S) = 2Rank(S)$.

Proof: Let M be a perfect matching in G, such that $|M \cap S| = Rank(S)$. There exists a permutation σ, whose trail is M. Denote the degree of y in $value_B(\sigma)$ by $d(\sigma)$, then $d(\sigma) = 2Rank(S)$, since each edge of M is traversed twice. Therefore $d(B^S) \geq 2Rank(S)$.

Permutations whose trail contains an odd cycle do not contribute to $|B(S)|$. From each permutation σ whose trail contains only edges and even cycles, we can obtain a permutation σ', whose trail is a perfect matching, and $d(\sigma') \geq d(\sigma)$. By definition, for any permutation σ, whose trail is a perfect matching, $d(\sigma) \leq 2Rank(S)$. Therefore $d(B^S) \leq 2Rank(S)$, which completes the proof. ■

Theorem 1.3.10 *There is an $\mathcal{RNC}$ algorithm for the perfect matching problem, with time $O(\log^3 n)$ using $O(n^{6.5})$ processors.*

Proof: By the previous theorem the Rank function can be computed by evaluating a determinant, whose entries are monomials of small degree. Using an algorithm by Borodin, Cook and Pippenger [2], this can be done in $O(\log^2 n)$ time using $O(n^{4.5})$ processors. Noting we have $O(\log n)$ phases and must compute the Rank for every edge in parallel, we get the stated bounds. ■

1.3.3 Matching is as Easy as Matrix Inversion

This subsection presents an algorithm is by Mulmuley, Vazirani and Vazirani [22] for the perfect matching search problem. This algorithm is based on a probabilistic isolating lemma which enables isolating one minimum weight perfect matching out of the possibly exponentially many. The processors are then coordinated to find this unique perfect matching.

Definition 1.3.11 *A* **set system** (S, F) *is a family* F *of subsets of a finite universe* S*.*

Lemma 1.3.12 (Isolating Lemma) *Let* (S, F) *be a set system,* $|S| = n$*. Assign integer weights chosen uniformly and independently from* $[1, 2n]$ *to the elements of* S*. Then the probability that there is a unique set of minimum weight in* F *is at least 1/2.*

Proof: Let $S = \{u_1, \ldots, u_n\}$. Assume weight w_i is assigned to element u_i. We now fix the weights of all elements except u_i and define the *threshold* for u_i as $\alpha_i \in \Re$ such that if $w_i \leq \alpha_i$ then u_i is contained in some minimum weight set, and otherwise u_i is not contained in any minimum weight set. An important observation is that if $w_i < \alpha_i$ then u_i is contained in every minimum weight set.

We now note that the threshold for u_i was defined independently of the weight assigned to u_i. Therefore w_i and α_i are two independent random variables and $Pr[w_i = \alpha_i] \leq \frac{1}{2n}$, since w_i is uniformly distributed. Summing for all elements of S we conclude that

$$Pr[\exists i,\ s.t.\ w_i = \alpha_i] \leq \frac{1}{2n} \times n = \frac{1}{2}$$

Therefore with probability at least $1/2$ each element of S is either in every minimum weight set or in no minimum weight set, or in other words the minimum weight set is unique. ∎

Let $G = (V, E)$ be the input graph, with n vertices and m edges. Assume G has a perfect matching. Let P denote the set of all perfect matchings of G. We can view (E, P) as a set system. Assign weights chosen uniformly and independently from $[1, 2m]$ to the edges of G. By the isolating lemma there is a unique perfect matching with minimum weight, with probability at least $1/2$. We will "coordinate" our processors to seek that solution.

Let $V = \{v_1, \ldots, v_n\}$. Denote the weight assigned to $(v_i, v_j) \in E$ by w_{ij}. Define a skew-symmetric matrix B of size $n \times n$, derived of the Tutte matrix of G, as follows:

$$b_{ij} = \begin{cases} 2^{w_{ij}} & (v_i, v_j) \in E \text{ and } i < j \\ -2^{w_{ij}} & (v_i, v_j) \in E \text{ and } i > j \\ 0 & \text{otherwise} \end{cases}$$

Lemma 1.3.13 *Assume there is a unique perfect matching M in G, and denote its weight by w. Then $|B| \neq 0$, and the highest power of 2 which divides $|B|$ is 2^{2w}.*

Proof: The value of the permutation corresponding to M is $(-1)^{n/2}2^{2w}$. If the trail of any other non-vanishing permutation is a perfect matching, then its value is a higher power of 2, by our hypothesis. Otherwise, if the trail of a non-vanishing permutation σ contains only even cycles and single edges, then we can build two perfect matchings M_1 and M_2, whose union is the trail of σ and $|value_B(\sigma)| = 2^{weight(M_1)+weight(M_2)} > 2^{2w}$. As in the proof of theorem 1.3.4 permutations that contain an odd cycle do not contribute to $|B|$. ∎

Lemma 1.3.14 *Let M be the unique minimum weight perfect matching in G, with weight w. Let $|B_{ij}|$ denote the (i,j) minor of B. Then $(v_i, v_j) \in M$ iff $\frac{|B_{ij}|2^{2w_{ij}}}{2^{2w}}$ is odd.*

Proof: By definition

$$|B_{ij}|2^{w_{ij}} = \sum_{\sigma:\sigma(i)=j} sign(\sigma) \times value_B(\sigma)$$

Non-vanishing permutations, that contain odd cycles in their trail, must contain at least two, since n is even. Therefore such permutations do not contribute to the sum, since each such permutation has a corresponding one with the same sign and opposite value.

Assume $(v_i, v_j) \in M$ then the value of the permutation, whose trail is M, is $\pm 2^{2w}$, and every other permutation which contributes to the sum has a value which is a higher power of 2.

Assume $(v_i, v_j) \notin M$, then all permutations that contribute to the sum have values which are higher power of 2, than 2^{2w}. ∎

From this lemma follows an algorithm for finding a minimum weight perfect matching:

Algorithm for finding a perfect matching:

- Compute $|B|$ and obtain w, the minimum weight of a perfect matching.

- For each $(v_i, v_j) \in E$ in parallel: Compute $\frac{|B_{ij}|2^{w_{ij}}}{2^{2w}}$, and if its odd add (v_i, v_j) to the perfect matching.

The complexity of the algorithm is essentially the complexity of computing $|B|$ and all the minors $|B_{ij}|$. If we use an algorithm by Pan [23] to compute these quantities, we get an algorithm with time $O(\log^2 n)$ using $O(n^{3.5}m)$ processors, for the perfect matching problem.

Theorem 1.3.15 *There is an $\mathcal{RNC}$ algorithm to construct a perfect matching, in time $O(\log^2 n)$ using $O(n^{3.5}m)$ processors.*

Chapter 2

Towards an $\mathcal{NC}$ Algorithm for the Perfect Matching Problem

2.1 Preliminaries

Let $G = (V, E)$ be an undirected simple graph.

Definition 2.1.1 *A* **Pseudo Perfect Matching** *of G is a spanning subgraph M, such that the degree of every vertex is odd in M.*

Definition 2.1.2 *The* **Complement** *in G of a pseudo perfect matching M, is the graph $C = G - M$, where isolated vertices are excluded.*

Definition 2.1.3 *An* **Augmenting Cycle** *of G with respect to a subgraph H, is a cycle vector C in G, such that $H \oplus C$ has less edges than H.*

Definition 2.1.4 *A* **Good Cycle** *of G, with respect to a pseudo perfect matching M, is a cycle vector C, such that $C - M$ is a matching.*

Rabin and Vazirani show in [24] that the maximum matching problem can be $\mathcal{NC}$ reduced to the perfect matching search problem on general graphs.

Dahlhaus and Karpinski show in [5] that the perfect matching problem for general graphs is $\mathcal{NC}$-equivalent to the perfect matching problem restricted to graphs of maximum degree 3, and this reduction reserves bipartiteness. They also prove that the general perfect matching problem can be $\mathcal{NC}$-reduced to the perfect matching problem on 3-regular graphs, though this reduction does not reserve bipartiteness.

We will adopt the second reduction and from now on try to solve the perfect matching search problem on graphs of maximum degree 3 and an even number of vertices. Note that for such graphs if n denotes the number of vertices and m the number of edges, then $m = O(n)$.

2.2 High-level Description of the Algorithm

A pseudo perfect matching is basically a spanning subgraph of G, which we will use as a starting point to constructing a perfect matching of G. Our algorithm for finding a perfect matching is based on finding a pseudo perfect matching and constantly "improving" it until a perfect matching is obtained.

The way we "improve" our pseudo perfect matching is using augmenting cycles. Since the degree of every vertex in a pseudo perfect matching is odd, when we xor a cycle to it we reserve its structure, i.e. we get another pseudo perfect matching. Each time we xor an augmenting cycle to the pseudo perfect matching we get closer to a perfect matching, since the number of edges of the pseudo perfect matching decreases by at least 1. The following theorem shows that a good augmenting cycle exists whenever the pseudo perfect matching is not a perfect matching.

Theorem 2.2.1 *Let G be an undirected graph with no loops, having a perfect matching. Let M be a pseudo perfect matching of G. If M is not a perfect matching, then there exists a good augmenting cycle in G.*

Proof: Let N be a perfect matching of G. Examine $M \oplus N$. This graph, not including the isolated vertices, is a collection of vertex-disjoint cycles. Since $|M| > |N|$ there exists a cycle with more edges from M than from N. This cycle is an augmenting one. It is moreover a good augmenting cycle, since N is a matching. ■

We present below a highlevel description of our $\mathcal{NC}$ algorithm for the perfect matching search problem on bipartite 3-regular graphs. Such graphs have a perfect perfect matching by the following theorem.

Theorem 2.2.2 *Every bipartite regular graph has a perfect matching.*

Proof: See [19]. ■

Most stages of the algorithm, except the last one (Section 2.6), can be applied to general graphs as well. Sections 2.3, 2.4, 2.5, 2.6 describe the different stages of the algorithm. Section 2.6 proves correctness and complexity.

Notation 2.2.3 *Let G be a graph with $2n$ vertices, having a perfect matching. Let H be a subgraph of G. We denote by $\Delta(H)$ the excess of edges in H over the number of edges in a perfect matching of G, i.e. $\Delta(H) = |E(H)| - n$.*

Perfect Matching Algorithm:

Input: A bipartite 3-regular graph $G = (U, V, E)$ with n vertices and m edges.
Output: A perfect matching M of G.

```
begin
    /* Section 2.3 */
    Construct a pseudo perfect matching of G, and denote it M.
    /* Section 2.4 */
    Convert M to a forest in G.
    /* Section 2.5 */
    Convert M to an induced forest in G.
    /* Section 2.6 */
    while M is not a matching do:
    begin
        Find a perfect matching N in the complement of M.
        Find an augmenting cycle L in M ∪ N, such that
        Δ(M ⊕ L) ≤ (1 − c)Δ(M), for a constant c > 0.
        M = M ⊕ L.
    end
end
```

2.3 Constructing A Pseudo Perfect Matching

This section describes the first stage of our algorithm, namely constructing a pseudo perfect matching of a given graph. Note that if the input graph is

3-regular, we can take the whole graph as our pseudo perfect matching.

Let $G = (V, E)$ be a graph with n vertices and m edges. We define below a system of equations over $GF(2)$, whose solution is a pseudo perfect matching of G.

Let us assign for each edge $e \in E$ a variable $X_e \in \{0, 1\}$. The set of equations is as follows:

$$\forall v \in V : \oplus_{\{e:v\in e\}} X_e = 1 \tag{2.3.1}$$

Lemma 2.3.1 *Let $\vec{X}$ be a solution to this system of n equations. Define $M(\vec{X}) = \{e \in E \mid \vec{X}_e = 1\}$, then $M(\vec{X})$ is a pseudo perfect matching of G.*

Proof: We just need to show that the degree of every vertex is odd in $M(\vec{X})$, and this follows directly from equations 2.3.1. ∎

Theorem 2.3.2 *There is an $\mathcal{NC}$ algorithm for constructing a pseudo perfect matching of a given graph in time $O(\log^2 n)$ using $O(nM(n)/\log n)$ processors.*

Proof: **Correctness:** By the previous lemma a solution to the system of equations is a pseudo perfect matching. If G has a perfect matching then it is in particular a solution. Therefore if no solution is found then G has no perfect matching.

Complexity: The complexity of this algorithm is essentially the complexity of solving a linear system of n equations with m variables. This can be done (See [12]) in $O(\log^2 n)$ time using $O(nM(n)/\log n)$ processors, $M(n)$ denoting the number of processors required for a PRAM machine to multiply two $n \times n$ matrices over a general ring in $O(\log n)$ time. Currently $M(n) = O(n^{2.376})$ (See [4]). ∎

2.4 Converting a Pseudo Perfect Matching into a Forest

This section and the next one deal with the problem of converting a pseudo perfect matching of a graph G into one which is also an induced forest in

G. The motivation for this transformation is to simplify the structure of the pseudo perfect matching as much as possible.

Definition 2.4.1 *An* **Odd Forest** *of a graph G, is a pseudo perfect matching which is a forest in G.*

Definition 2.4.2 *An* **Odd Induced Forest** *of a graph G, is a pseudo perfect matching which is an induced forest in G (See figure 2.4.1).*

Definition 2.4.3 *A* **3-vertex** *of a pseudo perfect matching M, is a vertex of degree 3 in M.*

Definition 2.4.4 *An* **edge-tree** *of a pseudo perfect matching M, is a connected component of M which is a single edge.*

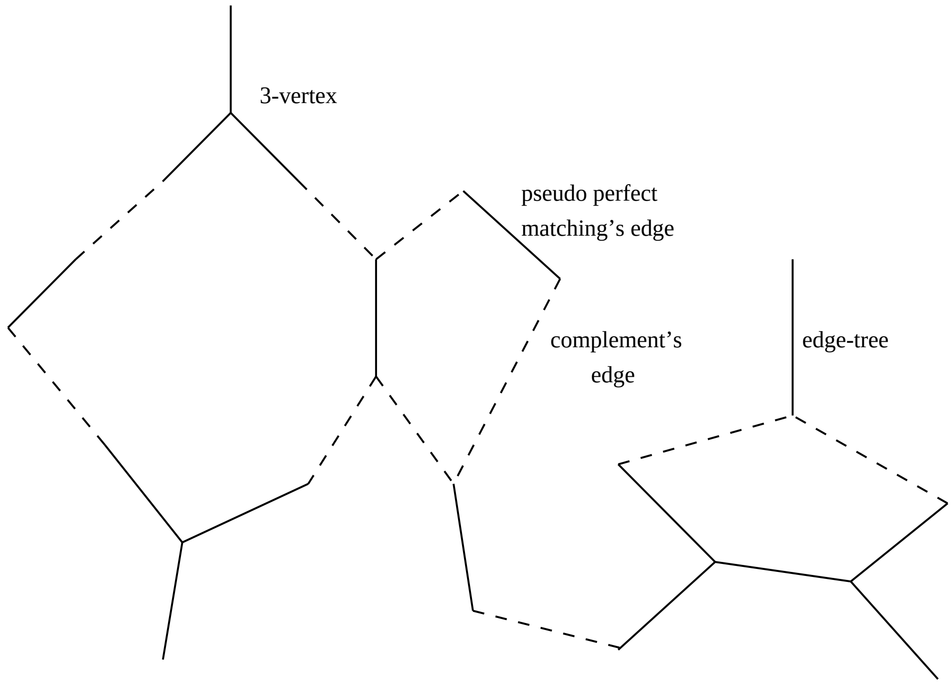

Figure 2.4.1: An Odd Induced Forest

Let M be a pseudo perfect matching of G. We now describe an algorithm which converts M into an odd forest of G. A key observation here, is that xoring a set of fundamental cycles to M results in an odd forest.

Algorithm for Constructing an Odd Forest:

Input: A graph $G = (V, E)$ with n vertices and m edges;
A pseudo perfect matching M of G.
Output: An odd forest M of G.

begin
    `In parallel for every connected component` K `in` M **do**:
    begin
        `Find a spanning tree` T `of` K
        `Find a set` S `of fundamental cycles in` K,
        `with respect to` T.
        `In parallel for every` $c \in S$ **do**:
            $K = K \oplus c$.
    end
end

Lemma 2.4.5 *Let K be a connected graph. Let S denote a set of fundamental cycles with respect to a spanning tree of K. Define $K' = K \oplus_{c \in S} c$, then K' is cycle free.*

Proof: Every fundamental cycle in S, is composed of a unique non-tree edge and some tree edges. When we xor the set of fundamental cycles to K all non-tree edges vanish and only some tree edges remain. Therefore K' is cycle-free. ■

Theorem 2.4.6 *There is an $\mathcal{NC}$ algorithm to convert a pseudo perfect matching into an odd forest in $O(\log n)$ time using $O(n^2/\log n)$ processors.*

Proof: **Correctness:** Let K be a connected component of M. By the previous lemma, the graph obtained from K at the end of the algorithm is cycle-free. Since xoring cycles to a pseudo perfect matching reserves its structure, the result is an odd forest.

Complexity: The algorithm consists of the following steps: Finding the connected components of M, finding a spanning tree in each component, computing a set of fundamental cycles with respect to that tree, and xoring those cycles to the pseudo perfect matching.

Finding the connected components of a graph, for which $m = O(n)$, is done in $O(\log n)$ time using $O(n)$ processors (See [12]). An arbitrary spanning forest of G can be computed in $O(\log n)$ time using $O(n)$ processors (See [12]).

Given a spanning forest, a set of fundamental cycles with respect to that forest can be found in $O(\log n)$ time using $O(n^2/\log n)$ processors. This is done by computing for each non-forest edge the lowest common ancestor of its vertices (See [12], [27]), and outputting the non-forest edge along with the paths from its end vertices to their lowest common ancestor. The complexity of xoring those cycles is $O(\log n)$ time using $O(n^2/\log n)$ processors.

The overall time of the algorithm is therefore $O(\log n)$ using $O(n^2/\log n)$ processors. ■

2.5 Converting an Odd Forest into an Induced one

Let M be an odd forest of a graph G. We describe below an algorithm to convert M into an odd induced forest.

Definition 2.5.1 *Let T be a rooted tree, with $n > 1$ vertices. A $\frac{1}{3}-\frac{2}{3}$* **cut-vertex** *of T, is a vertex v, such that v has at least $(\lceil\frac{1}{3}n\rceil - 1)$ descendants and at most $\lceil\frac{2}{3}n\rceil$ descendants, including itself.*

Lemma 2.5.2 *Let T be a tree in M with $t > 2$ vertices. Root T arbitrarily at a 3-vertex, then T contains a $\frac{1}{3}-\frac{2}{3}$ cut-vertex.*

Proof: The root of T has three sons, so one of them has at least $\lceil\frac{1}{3}t\rceil - 1$ descendants, including itself. Denote this vertex v. If v is a $\frac{1}{3}-\frac{2}{3}$ cut-vertex we are done. Otherwise, v has at least $\lceil\frac{2}{3}t\rceil + 1$ descendants. One of its two sons has therefore at least $\lceil\frac{1}{3}t\rceil$ descendants including itself. We continue descending along the tree using these arguments, until a $\frac{1}{3}-\frac{2}{3}$ cut-vertex is encountered. Such a vertex will be found since each time our height in the tree shortens by one, and the height of the tree is finite. ■

Following is an algorithm to convert an odd forest M in G to an induced one. The algorithm uses two subroutines:

The first subroutine, Split(M, v), inputs a graph $M = (V, E)$, and a vertex $v \in V$ of degree 3. Let u_1, u_2, u_3 be the vertices adjacent to v. The output of this subroutine is a new graph $M' = (V', E')$, such that $V' = (V - \{v\}) \cup \{v_1, v_2, v_3\}$ and $E' = (E - \{(v, u_1), (v, u_2), (v, u_3)\}) \cup \{(v_1, u_1), (v_2, u_2), (v_3, u_3)\}$. v is practically split into three new vertices. Each split vertex is kept in a database together with the vertices it was split to.

The second subroutine Merge(M) inputs a graph M and merges back all split vertices. This subroutine scans the database of split vertices and merges them back in M.

Algorithm for Constructing an Odd Induced Forest:

Input: A graph $G = (V, E)$ with n vertices and m edges;
An odd forest M in G.
Output: An odd induced forest M of G.

begin
  While M `is not induced,`
  `In parallel for every uninduced tree` T `in` M **do**:
  begin
    `Root` T `arbitrarily at a 3-vertex.`
    `Find a` $\frac{1}{3}-\frac{2}{3}$ `cut-vertex of` T`, denote it` v`.`
    `Search for a cycle` C `in` $G[T]$`,`
    `which includes` v`.`
    If `such a cycle` C `exists` **then**
      $T = T \oplus C$
    else
      `Split(`M, v`)`
  end
  `Merge(`M`)`
end

Proposition 2.5.3 *The algorithm terminates after at most* $\lceil \log_{3/2} n \rceil$ *iterations, with* M *being an odd induced forest of* G.

Proof: We will show, that each iteration the size of any uninduced tree component, decreases by at least a factor of $\frac{2}{3}$.

First observe, that during the execution of the algorithm the only uninduced trees dealt by the algorithm, are originally uninduced trees, or subgraphs of those trees.

Let T be an uninduced tree at the beginning of an iteration. Let v denote a $\frac{1}{3}-\frac{2}{3}$ cut-vertex of T. Following the algorithm two cases are possible:

Case a) There exists a cycle C in $G[T]$ which passes through v. In that case $T \oplus C$ comprises of two or more connected components and the size of each is at most $\frac{2}{3}|T|$, since v is a $\frac{1}{3}-\frac{2}{3}$ cut-vertex.

Case b) There is no such cycle. In this case, splitting v we get three connected components of size at most $\frac{2}{3}|T|$.

At the k'th iteration therefore, all uninduced trees are of size at most $(\frac{2}{3})^k n$.

We now note that once a vertex is split, three new edges are formed, and these edges remain throughout the execution of the algorithm, since no cycle passes through any of them. Thus, the merging step in the end is possible.

It follows that the algorithm terminates after at most $\lceil \log_{3/2} n \rceil$ iterations, with M being an odd induced forest. ■

Theorem 2.5.4 *There is an $\mathcal{NC}$ algorithm to convert an odd forest into induced one in $O(\log^2 n)$ time using $O(n)$ processors.*

Proof: **Correctness:** By the previous proposition the algorithm terminates with an odd induced forest.

Complexity: The algorithm runs at most $\lceil \log_{3/2} n \rceil$ iterations. Each iteration includes finding the connected components of M, and for each tree component the following: Rooting the tree, computing a $\frac{1}{3}-\frac{2}{3}$ cut-vertex, finding a cycle passing through that vertex, if such exists, and xoring it to M.

The connected components of M are found in $O(\log n)$ time using $O(n)$ processors (See [12]).

Given a tree T with k vertices, we can use the euler tour technique developed by Tarjan and Vishkin [27] to perform our computations on T. Rooting T can be done in $O(\log k)$ time using $O(k/\log k)$ processors (See [12]). Finding a $\frac{1}{3}-\frac{2}{3}$ cut vertex of T can be done by computing for each vertex in T the number of descendants and then determining a cut vertex. This computation can be performed in $O(\log k)$ time using $O(k/\log k)$ processors (See [12]).

In order to check if a cycle passes through a given vertex v, we do the following: We check in parallel for each non-tree edge (u, w) in $G[T]$, the lowest common ancestor of u and w being z: Is v is an ancestor of u or w? and is v a descendant of z, or z itself? If the answer to both questions is 'yes', we construct the cycle by taking the edge (u, w) and the paths from u and w to z. This requires $O(\log k)$ time using $O(k)$ processors. Xoring the cycle can be done in $O(1)$ time using $O(k)$ processors.

The merging step at the end can be done in $O(1)$ time using $O(n)$ processors. The overall time of the algorithm is therefore $O(\log^2 n)$, using $O(n)$ processors. ■

2.6 Finishing it up

This section describes the last stage of our algorithm. We assume that we are given an odd induced forest M of a bipartite 3-regular graph G. An algorithm is given for finding a good augmenting cycle, such that xoring it to M will result in an odd induced forest in which the number of 3-vertices is only a constant fraction of the original one. Of course if there are no 3-vertices in M then it is a perfect matching. We will use the notation of the previous section.

The following lemma establishes a measure of augmentation and characterizes good augmenting cycles.

Lemma 2.6.1 *Let G be a graph, and let M be an odd forest of G. Let C be a good cycle in G with respect to M. Define $\Delta = |E(M)| - |E(M \oplus C)|$. Denote by $n(C)$ the number of 3-vertices of M in C. Then $\Delta = n(C)$.*

Proof: Denote by k the number of edge-trees of M in C. Denote by t the number of other trees of M with edges in C.

$$\Delta = |E(M)| - |E(M \oplus C)| = |E(M)| - (|E(M - C)| + |E(C - M)|) =$$

$$(|E(M)| - |E(M - C)|) - |E(C - M)| = (k + (n(C) + t)) - (k + t) = n(C)$$

■

Lemma 2.6.2 *Let G be a bipartite 3-regular graph. Let M be an odd induced forest of G, then the complement of M in G is a collection of vertex-disjoint even cycles.*

Proof: G is bipartite, therefore every cycle in G is of even length. G is 3-regular, and every vertex is of odd degree in M, therefore every vertex in the complement of M is of degree 2. ■

Following is an algorithm for finding a perfect matching in a bipartite 3-regular graph G, given an odd induced forest of that graph. The algorithm consists of $O(\log n)$ iterations. The input of each iteration is an odd induced forest of G, and the output is a new odd induced forest of G with number of 3-vertices only a constant fraction of the original one. This new forest is obtained by augmenting with a good cycle, passing through a constant fraction of the 3-vertices of the given odd induced forest.

Algorithm:

Input: A bipartite 3-regular graph $G = (U, V, E)$ with
n vertices and m edges; An odd induced forest M of G.
Output: A perfect matching M of G.

begin
While M `is not a perfect matching` **do**:
begin
`Compute a perfect matching` N `in the complement of` M.

`Let` W `denote the set of 3-vertices in` M.
`Build a multi-graph` K `with a set of vertices` W,
`which is obtained from` $M \cup N$ `by contracting`
`every 2-path, connecting two 3-vertices,`
`to a single edge.`

`Compute a set of fundamental cycles in` K.
`Compute the xor` L `of all these cycles,`
`expanded to` $M \cup N$.
$M = M \oplus L$.

end

end

The first step in the algorithm is computing a perfect matching in the complement of M. All edges of the complement are on even cycles, by lemma 2.6.2, therefore we can in parallel on each cycle, number its edges successively and take the even-numbered edges to be our perfect matching. Denote this perfect matching N. Let H denote the graph $M \cup N$.

Lemma 2.6.3 *Any linear combination (over $GF[2]$) of cycle vectors in H, containing a 3-vertex, is a good augmenting cycle.*

Proof: N is a matching, therefore every linear combination of cycle vectors in H is a good one. If a good cycle in H contains at least one 3-vertex, then by lemma 2.6.1 it is an augmenting cycle. ∎

We now continue by transforming H into a multi-graph K. Let W denote the set of 3-vertices in M. Define $K = (W, F)$ as the graph obtained by contracting every 2-path in H connecting two 3-vertices, to a single edge.

Lemma 2.6.4 *K is 3-regular.*

Proof: N is a perfect matching, and therefore it is incident to all the leaves of M. Thus in H all vertices, that are not 3-vertices in M, are of degree 2. It follows that K is 3-regular. ∎

The motivation for constructing K is given in the next proposition.

Proposition 2.6.5 *Let G be an undirected 3-regular graph of $2n$ vertices, with no loops. Let S be a set of fundamental cycles in G, with respect to some spanning tree of G. Denote by C the xor of all cycles in S. Then C contains at least $n+1$ vertices.*

Proof: Let I be any connected component of G, with $2p$ vertices and q edges. The dimension of the cycle space of I is $q - 2p + 1 = p + 1$. We can therefore find a set of $p+1$ fundamental cycles in I, each such cycle containing a distinct non-tree edge. The vertices of these non-tree edges are on C. Since I is 3-regular, such a vertex is common to at most two non-tree edges. It follows that C contains at least $p+1$ vertices of I. Since this is true for any connected component of G we conclude that C contains at least $n+1$ vertices. ■

Let S denote a set of fundamental cycles with respect to some spanning tree of K. Denote by C the xor of all these cycles. From the previous proposition we deduce that at least half of K's vertices are on C. By lemma 2.6.3 if we expand C to H (edges in K become 2-paths in H) we obtain a good augmenting cycle, containing at least half of the 3-vertices in M. We state this result in the following theorem.

Theorem 2.6.6 *Let L denote the expansion of C to H. Let $M' = M \oplus L$. Denote by $n(M)$ the number of 3-vertices in M, then $n(M') \leq \frac{1}{2}n(M)$*

Proof: Following the proof of proposition 2.6.5, the xor of all fundamental cycles in K, is a cycle vector containing at least $\frac{n(M)}{2}+1$ vertices (originally 3-vertices in M). It follows that L contains at least $\frac{n(M)}{2}+1$ 3-vertices, since we only exchange edges in K by 2-paths in H.

As proved above L is a good augmenting cycle, therefore augmenting with L will reduce the number of 3-vertices in M by at least a factor of 2. ■

Theorem 2.6.7 *Let G be a 3-regular bipartite graph. Given an odd induced forest M in G, there is an $\mathcal{NC}$ algorithm to obtain a perfect matching of G, working in time $O(\log^2 n)$ using $O(n^2/\log n)$ processors.*

Proof: **Correctness:** Each iteration we augment M, using a good augmenting cycle, at least halving the number of 3-vertices in M by theorem 2.6.6. The algorithm therefore terminates after at most $\lceil \log n \rceil$ iterations with M being a perfect matching of G.

Complexity: By theorem 2.6.6, $\lceil \log n \rceil$ iterations suffice to produce a perfect matching. Each iteration consists of four stages: Computing a perfect matching in the complement; Constructing K; Computing a set of fundamental cycles in K; and xoring those cycles, expanded, to M.

Computing a perfect matching in the complement can be done in time $O(\log n)$ using $O(n/\log n)$ processors.

Building K can be done by reducing 2-paths to edges using pointer jumping along those paths. This takes $O(\log n)$ time using $O(n)$ processors.

Computing a set of fundamental cycles is done in $O(\log n)$ time using $O(n^2/\log n)$ processors. Expanding the cycles back to H can be done in $O(1)$ time using $O(n)$ processors, if we keep for each edge of K its 2-path source in H. Xoring the cycles to M can be done in $O(1)$ time using $O(n^2/\log n)$ processors.

The overall time of the algorithm is $O(\log^2 n)$ using $O(n^2/\log n)$ processors. ∎

Theorem 2.6.8 *There is an $\mathcal{NC}$ algorithm for the perfect matching problem, restricted to bipartite 3-regular graphs, working in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors.*

Proof: **Correctness:** We start with the input graph (which is 3-regular) as our initial pseudo perfect matching. Correctness now follows from theorems 2.4.6, 2.5.4, and 2.6.7.

Complexity: By theorems 2.4.6, 2.5.4, and 2.6.7, the overall time of the algorithm is $O(\log^2 n)$, using $O(n^2/\log n)$ processors ∎

2.7 Succeeding when the Number of Edge-Trees is Small

This section describes how to reduce the number of 3-vertices of an odd induced forest by a constant factor, when the number of its edge trees is at most a constant times the number of its 3-vertices.

Proposition 2.7.1 *Let $G = (V, E)$ be a graph of maximum degree 3, with no loops. Let $|V| = n$, $|E| = m$, and $\varepsilon > 0$. If $\frac{m}{n} \geq 1 + \varepsilon$(*) then there is a cycle vector in G containing at least $\lceil \varepsilon n \rceil$ of its 3-degree vertices.*

Proof: Let l denote the number of vertices of degree 3 in G, and k denote the number of vertices of degree 1 in G. Using this notation,

$$m = \frac{3l + k + 2(n - k - l)}{2}$$

Substituting this term for m in (*) gives

$$\frac{3l + k + 2(n - k - l)}{2n} \geq 1 + \varepsilon$$

It follows that $l - k \geq 2\varepsilon n$.

Contract now all open 2-paths in G to a single edge, and remove all 2-cycles. The new graph H obtained, contains only vertices of degree 1 or 3. Since $l - k \geq 2\varepsilon n$, the dimension of the cycle space of H is at least $\lceil \varepsilon n \rceil$. Therefore, if we compute a set of fundamental cycles in H, with respect to some spanning forest of H, then at least $\lceil \varepsilon n \rceil$ of the 3-degree vertices are on the xor of all these cycles (The proof is similar to that of proposition 2.6.5). The proof is now completed since 3-degree vertices of H correspond to 3-degree vertices of G. ■

Let G be a 3-regular graph with $2n$ vertices. Let M be an odd induced forest in G. Let $2g + 1$ be the minimum length of an odd cycle in G, $g > 0$. Denote by l the number of 3-vertices in M.

The algorithm to augment M, such that only a constant fraction of its 3-vertices remain, consists of the following steps:

- Compute a maximum matching N in the complement of M.

- Contract open 2-paths in $M \cup N$ to a single edge, and remove 2-cycles, obtaining the graph H.
- Compute a set of fundamental cycles S in H, with respect to some spanning forest of H.
- Let S' denote the set of fundamental cycles expanded to $M \cup N$.
- In parallel for all $c \in S'$ do: $M = M \oplus c$.

Theorem 2.7.2 *If $l(g+1) \geq n(1 + c(4g+2))$, $c > 0$, then there is an $\mathcal{NC}$ algorithm for computing an odd induced forest with only $(1-c)l$ 3-vertices. The algorithm works in $O(\log n)$ time using $O(n^2/\log n)$ processors.*

Proof: **Correctness:** Assume that there are n_i trees in M with i 3-vertices each, $0 \leq i \leq n$. It follows that:

$$|E(M)| = \sum_{i=0}^{n} n_i(2i+1)$$

$$|V(M)| = 2n = \sum_{i=0}^{n} 2n_i(i+1)$$

Let C denote the complement of M in G. Then, $|E(C)| = 3n - |E(M)|$. C is a collection of vertex-disjoint cycles, therefore a maximum matching in C contains at least $\frac{g}{2g+1}(3n - |E(M)|)$ edges, since half the edges of an even cycle are in a maximum matching, and at least $g/(2g+1)$ of the edges of an odd cycle are in a maximum matching.

Let N be a maximum matching in C. Examine the graph $M \cup N$. Since $|E(N)| \geq \frac{g}{2g+1}(3n - |E(M)|)$, and $E(M) \cap E(N) = \emptyset$, we get that,

$$|E(M \cup N)| \geq |E(M)| + \frac{g}{2g+1}(3n - |E(M)|)$$

Define $p = |V(M \cup N)|$ and $q = |E(M \cup N)|$ then,

$$\frac{q}{p} \geq \frac{(g+1)|E(M)| + 3ng}{2n(2g+1)}$$

We now notice that $l = \sum_{i=0}^{n} n_i \cdot i$. It follows that,

$$|E(M)| = \sum_{i=0}^{n} n_i \cdot i + \sum_{i=0}^{n} n_i(i+1) = \sum_{i=0}^{n} n_i \cdot i + n = l + n$$

and,

$$\frac{q}{p} \geq \frac{(4g+1)n + (g+1)l}{(4g+2)n}$$

We conclude that $\frac{q}{p} \geq 1 + c \Leftrightarrow l(g+1) \geq n(1 + c(4g+2))$.

By proposition 2.7.1, if $l(g+1) \geq n(1 + c(4g+2))$, $c > 0$, at least $\lceil cn \rceil$ of the 3-vertices in M are on the xor of all cycles in S. Thus, by xoring those cycles, expanded, to M, we will reduce the number of 3-vertices in M by at least a factor of $1/c$ (The proof is similar to that of theorem 2.6.6).

Complexity: The complexity of the algorithm is $O(\log n)$ time using $O(n^2/\log n)$ processors (See theorem 2.6.7). ■

Chapter 3

Perfect Matching and Bipartite Ear Decomposition are $\mathcal{NC}$-Equivalent

3.1 Preliminaries

Let $G = (V, U, E)$ be an undirected bipartite simple graph.

Definition 3.1.1 *An edge in G is called* **Allowed** *if it lies in some perfect matching of G. A graph G is said to be* **Elementary** *if its allowed edges form a connected subgraph of G.*

Theorem 3.1.2 *If G is elementary bipartite then G is connected and every edge in G is allowed.*

Proof: See [19]. ■

Definition 3.1.3 *A* **Bipartite Ear Decomposition** *of a bipartite graph G to r ears, is a decomposition $G = x + P_1 + ... + P_r$, such that $x \in E$, and for each $i = 1 \ldots r$, the i'th ear P_i, is an open path of odd length with its end vertices in $x + P_1 + \ldots + P_{i-1}$, and no other vertex in common with $x + P_1 + \ldots + P_{i-1}$.*

Theorem 3.1.4 *A graph G is elementary bipartite iff G has a bipartite ear decomposition. If G has n vertices and m edges then the number of ears in such a decomposition is $m - n + 1$.*

Proof: See [19]. ■

3.2 Main Theorem

Theorem 3.2.1 *The perfect matching problem and the bipartite ear decomposition problem are $\mathcal{NC}$-equivalent for bipartite elementary graphs.*

The proof of the easier part of this theorem, an $\mathcal{NC}$ reduction from the perfect matching problem to the bipartite ear decomposition problem, is given below. An $\mathcal{NC}$-reduction from the bipartite ear decomposition problem to the perfect matching problem, is given in the following sections: Sections 3.3 and 3.4 describe an algorithm which uses a perfect matching oracle to construct a bipartite ear decomposition in $\mathcal{NC}$. Correctness and Complexity of this algorithm are proved in Section 3.5.

Theorem 3.2.2 *The perfect matching problem can be $\mathcal{NC}$-reduced to the bipartite ear decomposition problem, on bipartite elementary graphs.*

Proof: Let $G = (U, V, E)$ be an elementary bipartite graph with n vertices and m edges. Let $x + P_1 + ... + P_r$ be a bipartite ear decomposition of G to odd ears, given as r linked lists where the i'th list contains the edges of the i'th ear. Let k_i denote the length of P_i, $i = 1 \ldots r$. The algorithm for obtaining a perfect matching consists of two steps:

- In Parallel for all $i = 1 \ldots r$: Number the edges of P_i from 1 to k_i, starting with an end edge.
- Let M denote the set of even numbered edges in all P_i's, and the edge x. Output M.

Correctness: M is a matching, since by the algorithm, we don't take adjacent edges from P_i's, nor their end edges.

To prove that M is a perfect matching, we show that it matches all vertices of G, inducting on the number of ears. If $G = x$, the claim is obvious. Assume M matches all vertices of the first $k-1$ ears. In particular M matches the end vertices of the k'th ear. Noting that the even-numbered edges of the k'th ear are incident to all its internal vertices the claim follows.

Complexity: Numbering the edges can be done using list ranking and a perfect matching can be obtained in $O(\log m)$ time using $O(m/\log m)$ processors. ■

3.3 Constructing a Bipartite Ear Decomposition

Theorem 3.3.1 *The bipartite ear decomposition problem can be $\mathcal{NC}$-reduced to the perfect matching problem, on bipartite elementary graphs.*

Proof: In subsection 3.3.1 we describe a sequential polynomial algorithm to construct a bipartite ear decomposition, given a perfect matching oracle. In subsection 3.3.2 we show how to parallelize this algorithm, and obtain the required result. ■

Let $PM(G)$ be an oracle which inputs a graph G and outputs a perfect matching of G. We can use this oracle to build a new oracle $PM(G, e)$, which inputs an elementary bipartite graph G, and an edge $e = (u, v) \in E(G)$, and outputs a perfect matching of G containing e. The algorithm implementing $PM(G, e)$ is as follows:

- Let G' be the graph obtained from G by deleting u and v.
- Call $PM(G')$ and denote its output by M.
- Output $M \cup \{e\}$.

We will denote throughout this section by F_e the output of $PM(G, e)$.

3.3.1 Sequential Algorithm:

Following is a polynomial algorithm for constructing a bipartite ear decomposition of a bipartite elementary graph, given a perfect matching oracle. The algorithm is inspired by the proof to theorem 3.1.4 in [19].

Algorithm:

Input: An elementary bipartite graph $G = (U, V, E)$ with n vertices and m edges, and an oracle $PM(G, e)$ for the perfect matching problem.

Output: A bipartite ear decomposition of G:
The edges of the k'th ear are labeled by k.

begin

 If $m = 1$ label the single edge by 1 and halt.
 Let $x, y \in E$ be any two adjacent edges.
 Let $k = 1$.
 Compute $PM(G, x)$ and $PM(G, y)$, and denote the results F_x and F_y respectively.
 Label the edges of the connected component in $F_x \oplus F_y$, which includes x and y, by 1.
 Let L denote the set of labeled (by 1) edges.
 while There exists an unlabeled edge $e \in E - L$ adjacent to a labeled edge (in L) **do**:
 begin
  Compute $PM(G, e)$ and denote the result F_e.
  Let P denote the connected component in $F_x \oplus F_e - L$, which includes e.
  $k = k + 1$.
  Label the edges of P by k.
  $L = L \cup P$.
 end

end

Lemma 3.3.2 *Let M and N be two perfect matchings of an undirected graph G. Then $M \oplus N$ is a union of vertex-disjoint even cycles and isolated vertices.*

Proof: M and N are both perfect matchings, therefore in $M \oplus N$ every vertex is of degree 2 or 0. Thus $M \oplus N$ is a union of vertex disjoint cycles, and isolated vertices. All the cycles are alternating between M and N, and therefore of even length. ■

Theorem 3.3.3 *The above algorithm computes in polynomial time a bipartite ear decomposition of G.*

Proof: **Correctness:** We prove correctness by inducting on the number of ears built by the algorithm.

The first ear built by the algorithm, $x + P_1$, is an even cycle by the previous lemma. Assume we already constructed k ears $x + P_1 + ... + P_k$ and let $e \in E$ be an edge not already included in $x + P_1 + ... + P_k$, adjacent to some edge in $x + P_1 + ... + P_k$. If no such edge exists we are done, since G is connected. Otherwise, following the notation of the algorithm, denote by L the set of edges $x + P_1 + ... + P_k$. Denote by P, the connected component of $F_x \oplus F_e - L$, which includes e.

By the induction hypothesis $\exists f \in L \cap F_x$, such that f is adjacent to e. Obviously f belongs to the connected component in $F_x \oplus F_e$, which includes e, but $f \notin P$. Therefore P is an open path. By the construction all internal vertices of P are not incident to any labeled edge and the end vertices are.

It remains to show that P is of odd length. Following the algorithm P is F_x-alternating. Since both end edges of P are not in F_x, P must be of odd length.

Complexity: The algorithm consists of $(m - n + 1)$ iterations, one for each ear. Each iteration consists of calling the perfect matching oracle, finding a connected component and labeling edges, all of which can be done in polynomial time (in $(m + n)$). The overall time is therefore polynomial.

■

3.3.2 Parallel Algorithm:

We continue now by parallelizing the algorithm presented in the previous subsection.

Definition 3.3.4 *A* **Cycle Component** *of an edge e with respect to an edge x, is the connected component in $F_x \oplus F_e$, which includes e. It is denoted $C(x, e)$.*

Our parallel algorithm starts by building the first ear, containing a special edge x. Certain priorities are then assigned to some edges of the graph, and lastly, these edges try to force their pre-assigned priorities on their cycle components with respect to x.

The initial priorities of the edges are assigned by the subroutine *Assign-Priorities*, implemented in the next section. The priority of each edge may change, as hinted, during the execution of the algorithm. Each edge tries to "write" its priority on a set of edges composing its cycle component with respect to x. The rule is, that an edge e succeeds in writing its priority on edge f (overriding f's priority) iff e has higher priority than f.

Definition 3.3.5 *A* **Priority-Path** P *with* **Value** $v_P = (p, e)$, *is a path in which the priority assigned to all edges is p, and the edge with the lowest serial number is e. We compare two values using the left lexicographical order, i.e.* $(p_1, e_1) < (p_2, e_2)$ *iff* $p_1 < p_2$, *or* $p_1 = p_2$ *and* $e_1 < e_2$.

Algorithm:

Input: An elementary bipartite graph $G = (U, V, E)$ with n vertices and m edges, and an oracle $PM(G, e)$ for the perfect matching problem.

Output: A bipartite ear decomposition of G:
The edges of the k'th ear are labeled by k.

begin

If $m = 1$ label the single edge by 1 and halt.
Let $x, y \in E$ be any two adjacent edges.
Compute $C(x, y)$ and label all its edges by 1.
Let $P_1 = C(x, y)$. Let $E' = E - (P_1 \cup F_x)$.
Compute in parallel $C(x, e)$ for all $e \in E'$.
AssignPriorities(G, x, P_1).
In parallel for every edge $e \in E'$ **do**:
begin
e tries to write its original
priority on the edges of $C(x, e) - P_1$.
end
Compute the set of all priority paths obtained
In parallel for every priority path P **do**:

```
      Compute the value $v_P$ of $P$.
   Sort the computed values in an increasing order.
   In parallel for every priority path $P$ do:
   begin
      Let $k$ denote the place of $P$'s value
      at the sorted list.
      Label all edges of $P$ by $k+1$.
   end
end
```

Note that $C(x, e)$ is computed once for every $e \in E'$, and remains fixed throughout the execution of the algorithm. The same is true for F_x and P_1. Correctness and Complexity of this algorithm are proved in section 3.5.

3.4 Implementing AssignPriorities Subroutine

This section describes the implementation of the subroutine AssignPriorities. The purpose of this subroutine is to enable an imitation of the sequential algorithm presented in subsection 3.3.1, by giving certain priorities to the edges in E', such that after the priority writing stage, a bipartite ear decomposition will be obtained by the algorithm.

In order to assign priorities to the edges of G we first build an auxiliary directed graph $G^* = (E' \cup \{x\}, F)$. Let $u, v \in V(G^*)$, $u \neq v$, and denote their corresponding edges in G by e_u and e_v. If $e_v = x$ then $u \to v \in F \Leftrightarrow e_u$ is adjacent to P_1. Otherwise, $u \to v \in F \Leftrightarrow e_u$ is adjacent to $C(x, e_v)$.

Notation 3.4.1 *For an edge $e \in E'$ we denote by v_e its corresponding vertex in G^*. v_x will denote the vertex corresponding to x.*

AssignPriorities Subroutine:

Input: An elementary bipartite graph $G = (U, V, E)$ with n vertices and m edges;
An edge x; The first ear P_1.

Output: Priorities assigned to the edges of G.

begin
```
    Build an auxiliary graph G*, as above.
    In parallel for all e ∈ E' do:
    begin
       If v_e and v_x are connected (in G*) then
         Let d be he distance from v_e to v_x.
       else
         Let d = ∞
       Assign e the priority (d, e).
    end
```
end

Note that each priority is an ordered pair of numbers. The first number denotes a certain distance in G^*, and the second, is a serial number of an edge in G.

Definition 3.4.2 *Let $e, f \in E(G)$. Let (d_e, e) and (d_f, f) denote the priorities assigned to e and f respectively. We define that e has* **higher priority** *than f iff $d_e < d_f$, or $d_e = d_f$ and $e < f$. A priority (d, e) is called* **Finite***, if $d < \infty$.*

Complexity: Building G^* can be done, using our computations of cycle components in the main algorithm, in $O(1)$ time using $O(m^2)$ processors. Computing the distance of every vertex from v_x can be done by computing a BFS-tree rooted at v_x in time $O(\log^2 m)$ using $O(M(m))$ processors (See [7]). The assigning of priorities is done in $O(1)$ time using $O(m)$ processors.

The overall time of AssignPriorities subroutine is therefore $O(\log^2 m)$ using $O(M(m))$ processors.

3.5 Proof of Main Theorem:

Notation 3.5.1 *Throughout this section whenever an ear is mentioned, we aim at an ear produced by the algorithm, i.e. a set of edges, with no previous assumptions about its structure. Let r denote the number of ears produced by the algorithm. For $i = 1 \ldots r$ denote the set of edges labeled by i, at the end of the algorithm, by P_i.*

For $i = 1 \ldots r$: Define $E_i = \cup_{j=1}^{i} P_j$; Define $V_i = \cup_{j=1}^{i} V(P_j)$; Let G_i denote the graph (E_i, V_i).

Let P be any ear produced by the algorithm. Denote by $C(P)$ the cycle component with respect to x, of the edge which wrote its priority on P.

Lemma 3.5.2 *For every $1 \leq i \leq r$, P_i is a path.*

Proof: P_1 is a cycle. For $2 \leq i \leq r$, by definition P_i is connected, with all vertices of degree at most 2, since at the beginning each edge (in E') has a distinct priority, and throughout the algorithm, an edge tries to write its priority only on its cycle component. ■

Corollary 3.5.3 *The set of all priority paths in G is well defined.*

Lemma 3.5.4 *At the end of the algorithm every edge $e \in E$ has a finite priority attached to it.*

Proof: Following the proof of theorem 3.3.3, G has a bipartite ear decomposition, in which the first ear is P_1, and every other ear is part of a cycle component $C(x, e)$, for some $e \in E'$.

In such a decomposition each ear contains an edge whose distance from v_x in $G^*$is finite, and therefore of finite priority. In particular this edge will attempt to write on its containing ear during the execution of the algorithm, and the lemma follows. ■

Lemma 3.5.5 *For every $1 \leq i \leq r$ $E_i \cap F_x$ is a perfect matching of G_i.*

Proof: We prove by induction. For $i = 1$ by the construction, P_1 is an F_x-alternating even cycle, and the claim follows.

We assume that the claim is true for $i > 0$ and prove it for $i+1$. By the algorithm P_{i+1} is F_x-alternating. Therefore all vertices of degree 2 in P_{i+1} are matched by $E_{i+1} \cap F_x$. Assume on the contrary, that v is a vertex of degree 1 in P_{i+1}, not matched by $E_{i+1} \cap F_x$. Let $e \in C(P_{i+1}) - P_{i+1}$ be incident to v.

The only reason that $e \notin P_{i+1}$ is that the priority attached to e is higher than the one written on the edges of P_{i+1}. It follows that $\exists j$, s.t. $1 \leq j \leq i$ and $v \in V_j$. By the induction hypothesis v is matched by $E_i \cap F_x$, a contradiction. ■

Lemma 3.5.6 *For every $2 \leq i \leq r$, P_i is an open path.*

Proof: Let $2 \leq i \leq r$. By lemma 3.5.2 P_i is a path. On the contrary assume P_i is a cycle. Moreover assume there exists an edge $e \in P_i$ adjacent to some ear P_j, $j < i$, at vertex v.

By the previous lemma the vertices of P_j are matched by $E_j \cap F_x$, so $e \notin F_x$. But P_i is an F_x-alternating cycle, so $\exists f \in P_i \cap F_x$ incident to v, a contradiction since F_x is a matching. We conclude that for each $e \in P_i$, e is not adjacent to any previous ear.

Let $w \in P_i$ be the edge which wrote its priority on P_i. By lemma 3.5.4 v_w is of finite distance, say d, from v_x in G^*. Therefore w is adjacent to some cycle component $C(x, e)$, such that $e = y$, i.e. $C(x, e) = P_1$), or the distance from v_e to v_x in $G^*$is $d - 1$. In any case w is adjacent to a previous ear, contradicting our conclusion. It follows that P_i must be an open path. ■

Lemma 3.5.7 *Let $2 \leq i \leq r$. Denote the end edges of P_i by a, b. Then a and b are both adjacent to E_{i-1}.*

Proof: W.l.o.g. we prove that a is adjacent to E_{i-1}. The same arguments are valid for b. P_i is open, therefore there exists an edge $e \in C(P_i) - P_i$, adjacent to a. e must have higher priority than the one written on P_i, otherwise it would have been a part of P_i. As a result $e \in E_{i-1}$, which means that a is adjacent to E_{i-1}. ■

Lemma 3.5.8 *Let $2 \leq i \leq r$. Denote the end edges of P_i by a, b. Then $a, b \notin F_x$.*

Proof: We prove w.l.o.g. that $a \notin F_x$. The arguments for b are similar. Using the previous lemma, a is adjacent to E_{i-1}. It now follows from lemma 3.5.5 that a is adjacent to some edge of F_x. But F_x is a matching, so $a \notin F_x$. ■

Corollary 3.5.9 *For every $i = 2 \ldots r$, P_i is of odd length.*

Lemma 3.5.10 *Let $2 \leq i \leq r$. Denote the end edges of P_i by a, b. Then no edge of P_i, other then a, b, is adjacent to E_{i-1}.*

Proof: On the contrary, assume there exists an edge $e \in P_i$, $e \neq a$ and $e \neq b$, which is adjacent to E_{i-1} at a vertex $v \in V_{i-1}$. By lemma 3.5.5 $e \notin F_x$, but since e is not an end edge $\exists f \in F_x \cap P_i$, which is adjacent to e at v. Thus a contradiction results, since again by lemma 3.5.5 $f \notin F_x$. ■

Proof of Main Theorem: Correctness: By the algorithm the first ear produced is an even cycle. By lemmas 3.5.2 - 3.5.10 all other ears produced by the algorithm are paths of odd length satisfying the requirements of a bipartite ear decomposition.

Complexity: To compute $C(x, e)$ for any two edges x and e, we must call the perfect matching oracle twice, and then compute the connected component which includes e in $F_x \oplus F_e$. Computing this component can be done in $O(\log n)$ time using $O(n)$ processors (See [12]). This computation is done $O(m)$ times in parallel and therefore requires $O(\log n)$ time and $O(mn)$ processors.

AssignPriorities subroutine takes $O(\log^2 m)$ time using $O(M(m))$ processors (See section 3.4). The priority writing stage, can be done in $O(\log \log m)$ time using $O(m^2)$ processors, by saving for each edge the priorities attempted to be written on it, and choosing the highest.

Computing the set of priority paths obtained, can be done in $O(\log m)$ time using $O(m)$ processors. For each priority path, computing its value can be done in $O(\log \log n)$ time using $O(n)$ processors. This computation is done $O(m)$ times in parallel.

Sorting the set of values computed can be done in $O(\log m)$ time using $O(m)$ processors (See [12]), and the ears can be labeled in $O(1)$ time using $O(m)$ processors.

The overall complexity of the algorithm is therefore $O(\log^2 m)$ time using $O(M(m)$ processors.

■

3.6 Extensions:

Suppose we have an $\mathcal{NC}$ algorithm for a class of graphs to test whether a graph in this class has a perfect matching. For example, such an algorithm exists for $K_{3,3}$-free graphs (See [29]). This class should have an additional property, that subgraphs of graphs in this class are also in the class. Let G be a graph in such a class. We can take G and remove from it in parallel all edges that do not lie in any perfect matching. Now by definition each connected component of the resulted graph is elementary.

Call a class that meets these two requirements a *Perfect Class*, then we deduce that the perfect matching problem and the bipartite ear decomposition problem are $\mathcal{NC}$ equivalent for bipartite perfect graphs.

Another aspect of the $\mathcal{NC}$-equivalence shown here is a first $\mathcal{RNC}$ algorithm for the bipartite ear decomposition problem, simply using an $\mathcal{RNC}$ algorithm for finding a perfect matching such as in [15] or [22], and our $\mathcal{NC}$ reduction to the perfect matching problem. Of course when the input graph is regular we can use the $\mathcal{NC}$ algorithm in [16] and obtain an $\mathcal{NC}$ algorithm to solve the bipartite ear decomposition problem for bipartite regular graphs.

The problem of finding a deterministic $\mathcal{NC}$ algorithm for the bipartite ear decomposition problem on a general bipartite graph, first presented by Lovasz in [18], remains open.

Bibliography

[1] N. Blum. A new approach to maximum matching in general graphs. In M. Paterson, editor, *ICALP 90: Automata, Languages, and Programming (LNCS 443)*, pages 586–597, 1990.

[2] A. Borodin, S. Cook, and N. Pippenger. Parallel computation for well-endowed rings and space bounded probabilistic machines. *Information and Control*, 31:113–136, 1983.

[3] A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and gcd computations. *Information and control*, 53:241–256, 1982.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *The 19th Annual ACM Symposium on Theory of Computing*, pages 1–6, 1987.

[5] E. Dahlhaus and M. Karpinski. Perfect matching for regular graphs is AC^0-hard for the general matching problem. *J. Comput. and System Sci.*, 44(1):94–102, 1992.

[6] J. Edmonds. Paths, trees and flowers. *Canada J. Math*, 17:449–467, 1965.

[7] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Inf. Proc. Letters*, 28:61–65, 1988.

[8] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[9] A. Goldberg, S. Plotkin, D. Shymoys, and E. Tardos. Using interior-point methods for fast parallel algorithms for bipartite matching and related problems. *Siam J. Computing*, 21(1):140–150, 1992.

[10] A. V. Goldberg, P. M. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *The 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[11] D. Grigoriev and M. Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in $\mathcal{NC}$. In *The 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 166–172, 1987.

[12] J. Ja'Ja'. *An introduction to parallel algorithms.* Addison-Wesely Publishing Company, 1992.

[13] H. J. Karloff. A las vegas $\mathcal{NC}$ algorithm for maximum matching. *Combinatorica*, 6(4):387–391, 1986.

[14] R. M. Karp and R. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 17. MIT Press, Cambridge, Mass., 1990.

[15] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a maximum matching is in random $\mathcal{NC}$. *Combinatorica*, 6:35–48, 1986.

[16] A. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. on Computers*, 30(2):93–100, 1981.

[17] L. Lovasz. On determinants, matchings and random algorithms. In L. Budach, editor, *Fundamentals of Computing Theory.* Akademia-Verlag, Berlin, 1979.

[18] L. Lovasz. Computing ears and branchings in parallel. In *The 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 464–467, 1985.

[19] L. Lovasz and M. D. Plummer. *Matching Theory*, pages 122–126. Academic Press, Budapest, Hungary, 1986.

[20] Y. Maon, B. Schreiber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. In *VLSI Algorithms and Architectures (LNCS 227)*, pages 34–45, 1986.

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

[22] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[23] V. Pan. Fast and efficient algorithms for the exact inversion of integer matrices. In *The fifth Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 504–521, 1985.

[24] M. O. Rabin and V. V. Vazirani. Maximum matching in general graphs through randomization. *J. Algorithms*, 10:557–567, 1989.

[25] J. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufman, San Mateo, CA, 1993.

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

[27] R. E. Tarjan and U. Vishkin. An efficient biconnectivity algorithm. *Siam J. Computing*, 14(4):862–874, 1985.

[28] W. T. Tutte. The factorization of linear graphs. *J. London Math Society*, 22:107–111, 1947.

[29] V. Vazirani. $\mathcal{NC}$ algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. In R. Karlsson and A. Lingas, editors, *SWAT 88: Scandinavian Workshop on Algorithms Theory (LNCS 262)*, pages 233–242, 1988.