# Investigation into parallel algorithms to find matchings in graphs.

Henry Ankers

April 2017

Supervisor: Artur Czumaj

# Contents

# Abstract

In this report we will examining different parallel algorithms for solving a range of problems related to finding matchings in graphs. We will look at some randomized algorithms for general graphs and some deterministic algorithms for regular bipartite graphs, and examine their correctness and runtime and analyse how useful they are for helping us to find a NC algorithm. We will also be looking at a number of key theorems which have proven extremely useful in developing these algorithms. Finally we will look at a sequential implementation and visualisation of one of these algorithms in Python.

Whilst the problem of finding and constructing a maximum or perfect matching are not yet known to lie in the parallel complexity class NC, we will see that lots of progress has been made in the past 30 years and look at possible ways algorithms will achieve this in the future.

# Chapter 1

# Introduction

## 1.1 Matchings In Graphs

Given a graph G = (V, E), a matching is a set of edges M $\subseteq$ E such that no two edges share any vertex. A maximum matching of G is a matching that contains the largest number of edges possible, a matching of maximum cardinality. A perfect matching is a matching which contains every vertex of a graph. The search problem for maximum and perfect matchings, and the decision problem of determining whether a given graph contains a perfect matching are all extremely well studied problems in graph theory. It is a very fundamental question to ask about a graph and many different graph problems reduce to it, such as the maximum flow problem. It is also a problem with many different practical applications such as pattern recognition [1], determining chemical bonds in certain types of compounds [2] and routing in permutation networks [3].

In 1957 Berge [4] proved that a matching was a maximum matching if and only if there was no augmenting path in the graph. This is a path P such that for a matching M, M$\oplus$P is a matching with a greater cardinality than M. The vast majority of sequential matching algorithms rely on this fact. The first polynomial time algorithm for finding a maximum matching in a general graph was given by Edmonds in 1965 [5]. Edmonds' approach was to find blossoms, cycles of odd length *2k+1* such that *k* edges belonged to M, and contracting this blossom to find augmenting paths. Edmonds algorithm ran in O($V^4$) time. Successive versions of the algorithm improved the runtime and currently the best runtime achieved by a sequential algorithm is O($\sqrt{V}$E) given by Vazirani and Micali [6].

Finding a maximum matching in a bipartite graph is easier than in the case of general graphs. Ford and Fulkerson [7] gave a O(VE) algorithm to find a maximum matching in bipartite graphs in 1956, finding a single augmenting path per iteration and using this to build a matching. Hopcroft and Karp [8] gave a O($\sqrt{V}$E) algorithm which found a maximal set of augmenting paths per iteration instead of just one. The principle behind this algorithm

was used in the previous algorithm developed by Vazirani and Micali for general graphs.

## 1.2 Parallel Computation

It is becoming harder and more expensive to increase the performance of processors. It may be the case that it becomes physically impossible for us to improve them in any meaningful way in the near future. As hardware itself becomes cheaper, parallel computation becomes a much more efficient way to improve the speed of algorithms.

We will be looking at parallel algorithms to solve a number of different problems related to matching which have been developed for the PRAM model of computation. This is a fairly simple parallel model which naturally follows from the standard RAM model of computation. We have a collection of RAM processors, each with their own local memory, and a unlimited amount of shared memory cells which can be accessed by all the processors. This model does not take into account issues such as the time taken to synchronize the processors, or the cost of reading/writing to the shared memory, however it provides us with a useful framework for measuring the performance of parallel algorithms.

An instruction for a PRAM works in the following way:

All processors read some data from the shared memory.

All processors perform some computation locally on the data.

All processors write some data back to the shared memory.

We need to specify what happens in the case that two processors try reading or writing to the same memory cell.

Concurrent Read/Write: Processors can simultaneously read/write to the same cell:

Exclusive Read/Write: Only one processor per instruction can read/write to the same cell.

As we clearly cannot have two different values stored in the same cell, we need some additional constraints on the CRCW (concurrent read and concurrent write) model:

Priority CRCW: Each processor is assigned a distinct priority, the processor with the highest priority is allowed to write

Arbitrary CRCW: A processor is picked randomly and allowed to write.

Common CRCW: If two processors try to write to the same cell then the values they try to write must be the same, otherwise the machines state becomes undefined.

When we are trying to design efficient parallel algorithms we have to take into consideration some additional notions. For example, we need to assign work to the processors equally, we are not effectively exploiting the parallelism of a problem if one processor is doing 90% of the total computation at every step. Another problem may be the amount of times we need to synchronise our processors, in real world settings this synchronisation probably comes

with a cost, meaning we would like to minimise this if possible.
We are particularly interested in finding algorithms that are in **NC**. This is the set of problems which can be solved in $O(\log^c(n))$ time using $O(n^k)$ processors where $c$ and $k$ are constants. Similarly to how the class **P** can be thought of as problems we can efficiently solve sequentially, the class **NC** can be considered to be the set of problems we can efficiently solve using parallel computers. It is important we make the distinction that this isnt the set of all problems which can be solved more efficiently in parallel; if a sequential algorithm runs in $O(\log n)$ time then it's in **NC** regardless of if a more efficient parallel solution exists or not.

We can see that $\mathbf{NC} \subseteq \mathbf{P}$, as we can simulate a **NC** algorithm in polynomial time on a sequential computer. It is unknown if $\mathbf{NC} = \mathbf{P}$, however it is very unlikely, because not every problem is easily parallelizable. We can think of the problems in **NC** as being the set of problems in **P** which are effectively parallelizable. We denote by **RNC** the algorithms which are solvable in **NC** time with access to randomness.

# Chapter 2

# RNC matching algorithms

## 2.1 Perfect Matching decision problem is in RNC

Finding a maximum matching in general graphs and the decision problem of determining whether a graph has a perfect matching is not yet known to be in **NC**. As we have seen, nearly all sequential algorithms to find a maximum matching rely on Berge's theorem, finding augmenting paths to increase the size of a matching until it is maximum. Unfortunately the problem of finding an augmenting path is not known to be in **NC**, meaning that we need to develop completely new techniques. We also must consider the fact that a graph can contain an exponential amount of maximum sized matchings; we know communication between different processors is limited so we need to find a method to get the processors to work towards the same matching.

A substantial amount of parallel algorithms rely on matrices. It's fairly obvious how multiple computers can be used to perform matrix computations faster than a single computer due to the way matrix multiplication works. This is one of the few areas where parallel algorithms aren't significantly more complicated than their sequential counterparts; making a matrix based approach to finding a maximum or perfect matching particularly attractive.

In 1979 Lovasz [9] noted that the decision problem of whether a graph had a perfect matching could be solved with an arbitrary degree of accuracy.

**Theorem (Tutte)**: *Let G = (V,E) be a simple graph. For each edge let $x_e$ be an indeterminate. Define the matrix B = $(B_{i,j})$ where:*

| | |
|---|---|
| $(B_{i,j}) = x_e$ | (i,j) ∈ E |
| $(B_{i,j}) = -x_e$ | (j,i) ∈ E |
| $(B_{i,j}) = 0$ | Otherwise |

*Then G has a perfect matching if and only if the determinant of B is not identically equal to 0 in the variables $x_e$*

The following proof is adapted from [10].

**Proof**:
The standard definition of a determinant is given by:
Det(T) = $\sum^{\pi}$Sign($\pi$)Value($\pi$) where:
$\pi$ is the set of all permutations over {1,2...n}.
Value($\pi$) = $\prod_{i=1}^{N}$ $B_{i,\pi(i)}$
Sign($\pi$) = 1 if $\pi$ is an even permutation, -1 if odd.
(An even permutation is a permutation which is the product of an even number of transpositions)
We can see that for Value($\pi$) to equal 0, we need $B_{i,\pi i} = 0$ for some $1 \leq i \leq |V|$. Therefore Value($\pi$) $\neq 0$ when (i,$\pi$(i)) $\in$ E for $1 \leq i \leq |V|$.

We define the trail of any $\pi$ with Value($\pi$) $\neq 0$ to be the subgraph of G consisting of edges (i,$\pi$(i)) for $1 \leq i \leq |V|$. Each trail passes through every vertex in G. We can also see that as $\pi$ is a permutation, there will always exist a unique x with $\pi$(x) = i, so every trail will contain the edge (x,i) as well as (i,$\pi$(i)) for $1 \leq i \leq |V|$, meaning the trail is a collection of disjoint cycles. If every cycle has a size of 2 then the trail corresponds to a perfect matching. This is because $\pi$ is the set of transpositions (i,j) where the edge (i,j) is in the perfect matching, so the trail will consist of edges (i,j) and (j,i) only.

Suppose $\pi$ has a trail with an odd cycle. We know there exists a permutation $\pi$' which is the same as $\pi$ but with the odd cycle transversed in the opposite direction. This gives us Value($\pi$) = -Value($\pi$'), and as Sign($\pi$) = Sign($\pi$') the net contribution to the sum is 0. Therefore Det(T) $\neq 0$ means there exists a permutation with it's trail consisting of only even cycles.

Suppose Det(T) $\neq 0$. Then there exists $\pi$ with only even cycles, going through every vertex. Therefore by taking alternating edges we find a perfect matching.

Suppose G has a perfect matching. Let $\pi$ be the permutation which is the product of the transpositions corresponding to the perfect matching.
Then Sign($\pi$) = $(-1)^{N/2}$ as $\pi$ consists of $\frac{N}{2}$ transpositions.
Value($\pi$) = $(-1)^{N/2}$ $\prod x_e^2$.
We can set $x_e = 1$ if e $\in$ M, 0 otherwise.
Therefore Sign($\pi$)Value($\pi$) = $\prod 1^2 = 1$
For any other permutation $\pi$', as the trail will consist of an edge not in M, Value($\pi$') = 0.
Therefore we can see that Det(T) = 1. □

The determinant of B forms a polynomial with degree |V| with $|V|^2$ variables, therefore we can immediately see that checking the determinant is identically equal to 0 would be very computationally expensive.

**Lemma (Schwartz-Zippel)**[11]: *Let P be a non-zero polynomial over the variables $\{x_1 \ldots x_n\}$ with a degree of D over a field F. If we pick values $\{v_1 \ldots v_n\}$ randomly from a set $S \subseteq F$ and let $P(\{v_1 \ldots v_n\})$ be the value obtained by setting $x_1 = v_1$, $x_2 = v_2 \ldots$ then $P(\{v_1 \ldots v_n\}) \neq 0$ with probability $(1 - \frac{D}{|S|})$.*

We can see that the above lemma allows us to determine with an arbitrary accuracy if a graphs Tutte matrix has a non-zero determinant or not and therefore whether the graph has a perfect matching. This leads to a sequential Monte Carlo algorithm (a randomized algorithm which has a chance of producing an incorrect answer) to solve the decision problem of whether a graph has a perfect matching. As our polynomial is over the field of real numbers, we can achieve a very high probability. In fact if it were possible to generate a completely random real number in the set [0,1] we would have a deterministic algorithm. As the problem of calculating the determinant of a matrix is in **NC**, this leads to an **RNC** algorithm to effectively solve the decision problem.

## 2.2 Constructing a Perfect Matching is in RNC

An algorithm presented by Karp, Upfal and Wigderson [12] uses this to give a **RNC** algorithm to find a perfect matching in a graph G which has one. The algorithm relies on finding sets of redundant edges, a set of edges which when removed still preserves the property of G having a perfect matching.

The algorithm's method of finding set of redundant edges depends on the amount of edges in the graph. We will look at the more general case of identifying a large set of redundant edges. This is where $|\mathrm{E}| \geq \frac{3}{4}|\mathrm{V}|$.

**Definition**: *Let **PM** denote the set of perfect matchings in G. For any $S \subseteq E$ define: Rank(S) = Max(/$S \cap A$/) for all $A \in$ **PM**.*

**Lemma**: *For $S \subseteq E$ define RE = {e ∈ (E-S); Rank(S ∪ e)) = Rank(S)}. If G has a perfect matching then so does G' = (V, (E-RE)).*

**Proof**: We can clearly see this is the case, if A is a perfect matching and $|(\mathrm{S} \cup \mathrm{e}) \cap \mathrm{A}| = |\mathrm{S} \cap \mathrm{A}|$ then clearly $\mathrm{e} \notin \mathrm{A}$, therefore A is a perfect matching of G'. □

**Procedure (Find Redundant-Set):**

*1) Randomly choose a number i from the uniform distribution over the range $\{1,2,\ldots,\frac{5}{6}|E|\}$*
*2) Choose $S \subseteq E$ from the uniform distribution over the i-element subsets of E*
*3)RE = {e ∈ E-S ; Rank(S ∪ e)} = Rank(S)}*

By looking at the probabilities that for an edge chosen at random from E-S, Rank(S∪{e}) > Rank(S), it can be shown that:

**Prob**(|RE|$\geq \frac{1}{60}$|E|) $\geq \frac{1}{60}$

If we can find a way to compute Rank effectively then by calculating the Rank for every edge in parallel we will be able to find a perfect matching.

**Theorem:** *Given S ⊆ E, Define the matrix* $B^S = (B^S_{i,j})$ *where:*

| | |
|---|---|
| $(B^S_{i,j}) = yx_e$ | (i,j) ∈ E |
| $(B^S_{i,j}) = -yx_e$ | (j,i) ∈ E |
| $(B^S_{i,j}) = x_e$ | (i,j) ∈ E - S |
| $(B^S_{i,j}) = -x_e$ | (j,i) ∈ E - S |
| $(B^S_{i,j}) = 0$ | Otherwise |

*Then Rank(S) = Half of the degree of y in Det(*$B^S$*)*

We shall omit the proof due to length, however it is fairly simple and similar to the proof of Tutte's theorem. The full proof can be found in [12].

## 2.3 Matching is as easy as matrix inversion

This algorithm given by Mulmuley, Vazirani and Vazirani [13] gives another **RNC** method for finding a perfect matching in a graph G = (V, E) which contains a perfect matching. Again relying on the Tutte matrix, instead of removing edges it instead attempts to isolate a certain perfect matching and work towards it.

**Definition:** *Let* $S = \{x_1, \ldots, x_n\}$. *Let* $F = \{s_1, \ldots s_m\}$ *where* $s_i \subseteq S$ *for* $0 \leq i \leq m$. *We call (S, F) a set system.*

**Isolating Lemma:** *Let (S, F) be a set system where S is a collection of elements. For each* $x_i \in S$ *define a unique weight* $w_i$ *chosen uniformly from the set [1, 2n]. Define the weight of* $s_i \in F$ *to be the sum of the weight of all it's elements. Then the probability that the minimum weight set is unique is* $\geq \frac{1}{2}$

**Proof:** For an element $x_i$ let W be the smallest weight of a set not containing the element. Let W' be the smallest weight of a set containing the element.
Let us define the threshold for an element $x_i$ to be the real number $r_i$ where $r_i$ = W - W' when we ignore the weight $w_i$ of $x_i$.
Clearly if $w_i < r_i$ then $x_i$ must be in every minimum weight set, as W' + $w_i$ < W.
We can also see that $w_i$ $r_i$ then $x_i$ isn't in any minimum weight set as W' + $w_i$ > W.
Suppose that $w_i = r_i$. Then W' + $w_i$ = W, so we can see there is a minimum weight set containing $x_i$ and a minimum weight set not containing $x_i$ .

As we defined $r_i$ without using the weight $w_i$ they are independent variables. As $w_i$ is a uniformly distributed integer in [1, 2n] we have:
**Prob**($r_i = w_i$ ) = $\frac{1}{2n}$
As each of the weights are assigned independently we can see that the probability is the same for every $x_i \in$ S. We can therefore sum the probabilities up to get:
**Prob**(Every element is either in every minimum weight set or in none) = $\frac{1}{2}$ □

If we let S = E, we give every edge a unique weight from [0, 2|E|], and we let F = Set of all perfect matchings in G, then we can see that the probability that the minimum weight perfect matching is unique is $\geq \frac{1}{2}$.
Denote the weight given to an edge (i, j) $\in$ E by $w_{i,j}$. Define an adjacency matrix $A^S = (a^S_{i,j})$ where:

| | |
|---|---|
| $(a^S_{i,j}) = 2^{w_{i,j}}$ | (i,j) $\in$ E and i < j |
| $(a^S_{i,j}) = -2^{w_{i,j}}$ | (i,j) $\in$ E and i > j |
| $(a^S_{i,j}) = 0$ | i = j or (i,j) $\notin$ E |

**Lemma:** *Let M be a unique minimum weight perfect matching of G with weight W. Then Det(A) $\neq$ 0 and $2^{2W}$ is the greatest power of 2 which divides Det(A)*

The following two proofs are taken from [14]

**Proof:**
We will define Sign and Value the same way as seen in the proof of Tutte's theorem. Recall also that Det(A) $\neq$ 0 if and only if there exists a permutation with its trail consisting only of even cycles.
Let $\pi$ be a permutation which is a product of transpositions corresponding to a perfect matching. We saw in the proof of Tuttes theorem this consists of $\frac{N}{2}$ cycles.
Sign($\pi$) = $(-1)^{N/2}$ as $\pi$ consists of $\frac{N}{2}$ cycles.
Value($\pi$) = $(-1)^{N/2}2^{2W'}$ where W' is the weight of the matching.
Therefore we can see the matching contributes $2^{2W'}$ to Det(A), meaning our unique minimum weight matching contributes $2^{2W}$ whilst all others must contribute $2^{2W+c}$ where c $\geq$ 1, meaning $2^{2W}$ is the largest power of two which can divide any contribution by a perfect matching.
We need to check there aren't any other trails which don't correspond to perfect matchings but contribute a smaller power of 2 to Det(A), as this would mean that $2^{2W}$ does not divide Det(A).
If such a trail existed then it would consist only of even cycles. Let $\pi$ be the permutation with such a trail. If they all had a size of 2 we would have a perfect matching so one cycle must have a size of 4. In this case there exist two separate perfect matchings with weights $W_1$ and $W_2$ where $W_1 \cup W_2$ is equal to the trail. We can see therefore that Value($\pi$) =

$2^{W1+W2}$ which is a multiple of $2^{2W+c}$. □

**Lemma:** *The edge (i,j) is in the unique minimum-weight matching M if and only if Det($A_{i,j}$) * $2^{w_{i,j}}/2^{2W}$ is an odd integer*

**Proof:**
$\mathrm{Det}(\mathrm{A}_{i,j})2^{w_{i,j}} = \sum_{\pi:\pi(i)=j}\mathrm{Sign}(\pi)\mathrm{Value}(\pi)$
For the same reasoning before if the trail of $\pi$ contains an odd cycle then it's net contribution to the determinant is equal to zero.
If (i,j) ∈ M then the trail corresponding to M contributes $2^{2W}$ to the determinant. We know from the previous lemma that every other trail contributes a multiple of $2^{2W+1}$. Therefore the determinant must be an odd integer.
On the other hand,if (i,j) ∉ M then all trails given by $\pi$ in the determinant contribute a multiple of $2^{2W+1}$ and therefore the determinant must be an even integer. □

To find a perfect matching we need to:
Calculate Det(A) and find the minimum weight cost of a matching
Calculate $\mathrm{Det}(\mathrm{A}_{i,j})2^{w_{i,j}}/2^{2W}$ in parallel for every edge, and if it's odd add it to the matching.
This gives us a runtime of $\mathrm{O}(\log^2 \mathrm{n})$ time using $\mathrm{O}(\mathrm{n}^{3.5}\ \mathrm{m})$ processors using Pan's randomized algorithm for matrix inversion [15].

The isolating lemma gives us a powerful tool for finding perfect matchings in a graph. If we are given a graph G with preset edge weights (let $(\mathrm{i,j})_w$ define weight of an edge in E), we are able to use a variation on this algorithm to find a perfect matching of maximum weight. Using this we can find a maximum matching of a graph G in the following way:
Let G = (V,E') where:
E' = {(i,j); $0 \leq \mathrm{i}, \mathrm{j} \leq |\mathrm{V}|$}
$(\mathrm{i,j})_w = 0$ if (i,j) ∉ E
$(\mathrm{i,j})_w = 1$ if (i,j) ∈ E
Then a maximum weight perfect matching of G' can easily be converted to a maximum matching of G by selecting all the edges in the perfect matching with a weight of 1.

The isolating lemma has proven to be a powerful tool for finding efficient parallel algorithms to solve matching problems in various different types of graphs. Lots of work has been done in trying to construct the weighting function in such a way that the minimum weight perfect matching is always unique, meaning we could find a perfect matching deterministically. Whilst it is currently unknown how to do this for general graphs, success has been had for specific types of graphs such as graphs with a polynomial number of perfect matchings [16] and chordal graphs (graphs which have the property that for every cycle containing more than 4 vertices, there exists an edge not in the cycle which connects two vertices in the

cycle) [17].

In [18] a interesting condition is given for a weighting function to be isolating in bipartite graphs. For a cycle C in the graph, we say that C has nonzero circulation if the sum:
$|w(v_1,v_2) - w(v_2,v_3) + w(v_3,v_4) \ldots - w(v_k,v_1)| \neq 0$
Where $v_1..v_k$ are the vertices in the cycle and w is a weighting function.

**Lemma:** *Let G be a bipartite graph with a perfect matching, and let w be a weight function such that all cycles in G have nonzero circulation. Then the minimum weight perfect matching is unique*
**Proof:** Suppose G has two minimum weight perfect matchings. The graph $M_1 \oplus M_2$ is a graph in which every vertex has a degree of 0 or 2, so ignoring any isolated vertices we can see it consists of only cycles. As the graph is bipartite, all these cycles have even length. As all these cycles have non-zero circulation, either the edges in $M_1$ or $M_2$ contribute a smaller amount to the circulation. We can take the edges with minimum weight from this cycle and form a perfect matching with a smaller total weight, which gives us a contradiction. □

In [19] this theorem proves crucial in constructing a isolating weight function, by constructing a polynomial sized set of weighting functions such that one gives all cycles nonzero circulation and then attempting to find this weighting function. By doing this they managed to show that finding a perfect matching in bipartite graphs can be done deterministically in $O(\log^2 n)$ time using $O(n^{O(\log n)})$ processors (referred to as **Quasi-NC**), getting us close to a **NC** algorithm for bipartite graphs.

# Chapter 3

# NC Algorithms for 3-regular bipartite graphs

Much work has been done to solve the perfect matching problem in special types of graphs. As we have seen, finding a perfect matching in a bipartite graph is in **Quasi-NC**, we also have **NC** algorithms for the case of bipartite planar graphs [18] [20]. It is possible that these algorithms may lead to the discovery of methods which work for general graphs. We will look at 2 different **NC** algorithms for finding a perfect matching in 3-regular bipartite graphs.

## 3.1 Sharan and Wigderson

The first algorithm we will look at was given by R Sharan and A Wigderson in 1996 [21]. It takes a 3-regular bipartite graph and attempts to increase the number of different connected components in a pseudo perfect matching of a graph by a constant fraction per iteration, eventually leaving us with a perfect matching.

### 3.1.1 Definitions

**3-regular bipartite graph:** A graph which is 3-regular, meaning that each vertex has a degree of 3, and bipartite.
**Pseudo perfect matching:** A subgraph of a graph G where every edge has an odd degree.
**Augmenting Cycle:** An augmenting cycle for G is defined to be a cycle C such that G$\oplus$C has less edges than G. Despite the similar name, these are not related to augmenting paths.
**Good Cycle:** A good cycle for G with respect to a pseudo perfect matching M is a cycle such that C - M is a matching.
**Fundamental cycle:** For every edge (u,v) in G and not in a spanning tree T of G, the

fundamental cycle is a path including the edge (u,v) (called the fundamental edge) and the two paths in T to their lowest common ancestor.

### 3.1.2 Converting a pseudo perfect matching into a forest

Our goal here is to transform our pseudo perfect matching into as simple a structure as possible. As G is 3-regular, we will take a copy of G to be our initial pseudo perfect matching.

**Theorem:** *Let G be a undirected graph with a maximum degree of 3 which contains a perfect matching. Let M be a pseudo perfect matching of G. If M is not already a perfect matching, then we can always find a good augmenting cycle of M.*

**Proof:** If we take a pseudo perfect matching M of G, and a perfect matching N, we can see that M⊕N gives us a set of disjoint cycles and some isolated vertices. This is because any vertex of degree 1 becomes either a vertex of degree 2 or 0, and a vertex of degree 3 must become a vertex of degree 2. As all vertices have a degree of 2 we can see it consists only of cycles.

Clearly we have $|N| < |M|$ unless M is also a matching. So we know there must exist a cycle in M⊕N with more edges in M than edges in N. Therefore if we were to XOR this cycle to M, we would remove more edges than we added, giving us an augmenting cycle. Furthermore as N is a matching, removing the edges of M from M⊕N would leave us with a matching, so this cycle is also a good cycle. □

We now know that we can always find a good augmenting cycle of a pseudo perfect matching. Therefore we want to simplify our pseudo perfect matching so this becomes easier to do. We will do this by first converting it into an odd forest.
Let M be our pseudo perfect matching, a copy of G. Let P(e,T,G) define the parity of the amount of times the edge e appears in the set of fundamental cycles for G with respect to T. Then if P(e,T,G) = 1, e would be removed from M by xoring the set of fundamental cycles to M.

**PPM to Forest:**

Find a spanning forest T of G
Calculate P(e,T,G) for every edge in M
For every edge in M:
    If P(e,T,G) = 1, set M = M⊕e

**Correctness**

Clearly as every non-forest edge appears in only a single fundamental cycle, we remove it from M. We are therefore left with some forest edges, meaning M is now a forest. An obvious question to ask is 'how do we know we won't isolate any vertices, and how do we

know the degree of vertices will be odd'?
If a vertex V has degree 3 and we xor a fundamental cycle, we will remove 2 edges. If another cycle contains the same two edges, these edges will be added, giving V a degree of 3. If the cycle contains a different edge, the different edge will be removed and the edge the cycles shared in common will be re-added to the vertex, giving V a degree of 1. We can therefore see that the degree of any vertex in M is always odd, and none will be isolated.

**Complexity**
We need to show we can compute a spanning forest of G, and we can compute P(e,T,G) for every edge in G in NC time.

**Computing P(e,T,G):**
Suppose e is a non-forest edge, then clearly P(e,T,G) = 1. Otherwise we use the following lemma.

**Lemma:** *Let e = (u,v) be a forest edge, with v being the child of u. Let T(e) denote the subtree of T rooted at v. Then P(e,T,G) is equal to the parity of the number of non-forest edges incident to vertices in T(e)*
**Proof:** Let (x,y) be a non-forest edge. If neither x or y is in T(e) then clearly the fundamental cycle containing (x,y) does not pass through e.

If both x and y are in T(e) then the lowest common ancestor of x and y is contained in T(e). Therefore the fundamental cycle containing (x,y) only passes through (x,y) and edges in T(e), and as e isn't in T(e) then we can see the fundamental cycle doesn't contain e.

If only x is in T(e) then as the lowest common ancestor of x and y is not contained in T(e), the fundamental cycle must pass through e

Therefore we can see that P(e,T,G) is equal to the parity of the number of non-forest edges incident to only a single vertex in T(e). This is clearly equivalent to the parity of the number of non-forest edges in T(e), note that if we calculate P(e,T,G) by summing the number of non-forest edges incident on each vertex, an edge with both endpoints in T(e) will be counted twice and therefore not change the parity. □

Let N(v) denote the number of non-forest edges incident to v. We can use a tree contraction algorithm to efficiently calculate P(e,T,G). This is an algorithm which merges leaf nodes with their parents, along with any value stored in the leaf node. We can use this to merge the values N(v), and when v becomes a leaf we will have the parity N(v) = P(e,T,G).

An effective tree contraction algorithm is given by Abrahamson, Dadoun, Kirkpatrick and Przytycka requiring O(log n) time using O(n / log n) processors [22]. Unfortunately a tree contraction algorithm requires us to not only know the parent vertex of the leaf nodes, but the parent of the parent of the leaf nodes as well. For this reason we are required to arbitrarily root the tree first.

**Rooting a tree:**
To do this we will use something called the Euler tour technique. This was introduced by Tarjan and Vishkin [23]. We will show their method of constructing this tour.

Given a tree T, create a new tree $T_D$ such that for every undirected edge (u,v) in T, we have directed edges {u,v} and {v,u} in $T_D$. Then $T_D$ is a Eulerian graph, meaning we can find a directed circuit which visits every edge once. We will describe how to find this circuit:

**Finding a Euler tour:**
Construct a list of directed edges {u,v},{v,u} as above.
Select a vertex r to be the root of the tree.
Sort the edge list lexicographically, then place all the edges {r,u} first in order.
For each edge {u,v} in the list:
If the previous edge {x,y} has $x \neq u$, let First(u) = {u,v}
Otherwise, let Next(y,u) = {u,v}
For each edge {u,v} set the pointer Successor(u,v) where:
Successor(u,v) = Next(u,v) if Next(u,v) $\neq \emptyset$
Successor(u,v) = First(v) otherwise

Successor(u,v) is always in the form {v,x} and defined uniquely for every edge, and produces a Eulerian circuit of $T_D$. Once we have the sorted list, it takes O(1) time to construct the circuit in parallel. There are many different parallel sorting algorithms we can use, Cole [24] gives a O(log n) algorithm using O(n / log n) processors on the EREW PRAM.

We can use this to root the tree. Starting at the root vertex, for each edge (u,v) in T, the first occurrence of {u,v} or {v,u} is called the *advanced edge* and the second occurrence the *retreat edge*. The distance from the root increases at every advanced edge. Therefore if the advanced edge is {u,v}, u must be the parent of v in the tree. Given the Successor function we can do this in O(1) time.

We can therefore root our tree in O(log t) time using O(t / log t) processors, where t is the number of vertices contained in the tree.

**Finding a spanning forest:**
Finding a spanning forest efficiently in parallel is itself a problem which has been the focus of lots of study, Cole and Vishkin [25] and Iwana and Kambayashi [26] give two different algorithms for the Arbitrary CRCW, requiring O(log n) time using O(n*A(n) / log n) processors. A(n) denotes the inverse Ackermann function (this is an extremely slowly growing function, $A(n) \leq 5$ for $n \leq 10^{10^{20}}$.

Therefore the final complexity is:

| | |
|---|---|
| **Finding a spanning forest:** | O(log n) time using O(n*A(n) / log n) processors. |
| **Rooting a tree:** | O(log n) time using O(n / log n) processors. |
| **Calculating P(e,T,G):** | O(log n) time using O(n / log n) processors. |
| **Total:** | O(log n) time using O(n*A(n) / log n) processors. |

### 3.1.3 Converting a forest into an induced forest

We want to convert our forest into an induced forest. If we take a tree T in the forest and let X be the set of vertices in T, we say that T is induced if every edge connecting two vertices in X is in T.

When we look at the complement of this induced forest, we can see that each edge connects two different connected components, and furthermore we can see that this must form a set of edge disjoint cycles (as every non-isolated vertex has a degree of 2). We will use this fact to find a perfect matching of the graph in the next stage.

Let a cut vertex denote any vertex v in T such that v has between $\frac{t-1}{3}$ and $\frac{2t}{3}$ descendants including itself. If |T| = 4, then let the vertex of degree 3 be the cut vertex.
Let G[T] denote the graph acquired from inducing the vertices in T over G
Let S[v] denote the set of vertices including v and all the neighbours of v
Let M be our forest from the previous step, let G be our input graph
**Induce(M):**
While M is not induced, in parallel for every uninduced tree T:
Root T arbitrarily at a 3-vertex.
Calculate the number of descendants of every vertex in the tree.
Find a cut vertex, v.
Try to find a cycle C in G[T] which contains v.
If we find such a cycle, let M = M⊕C.
Else, let I = Induce(T - v) and M = (M - T) ∪ I ∪ T[S[v]].

**Correctness**
**Lemma:** *Let $T_i$ be an uninduced tree in T at the start of an iteration. Then $|T_i|$ has decreased by $\frac{1}{3}$ at the end of the iteration.*

**Proof:** Let v be our cut-vertex
Case 1) We have a cycle C in G[T] that passes through v.

In this case, T⊕C clearly separates the graph at v into two different connected components. As we know v has between $\frac{t-1}{3}$ and $\frac{2t}{3}$ descendants, we know that each connected component is at most of size $\frac{2|T|}{3}$ . Therefore the size of any uninduced component has decreased by a third.
Case 2) We don't have a cycle, in which case we remove v and all its edges, getting three

different connected components. We know that each connected component is at most of size $\frac{2|T|}{3}$. □

**Lemma:** *In $log_{\frac{3}{2}}(n)$ iterations the forest will be induced*

**Proof:** Any tree with two vertices is induced, as we do not allow multiple edges between the same vertices.

We know the size of any uninduced component of M decreases by at least a third every iteration, we can see that after $\log_{\frac{3}{2}}(n)$ iterations, we will be left with an induced forest. □

**Complexity**

There are 3 procedures required here. We need to find a way to find the different trees in the forest, which is the same as finding the connected components of M. We also need to be able to arbitrarily root a tree and calculate the number of descendants given this rooting. Finally, we need to be able to check if our cut vertex lies on a cycle in G[T].

**Arbitrarily rooting a tree and calculating the number of descendants:**

We have seen how we can root a tree, so we just need to effectively calculate the number of descendants. Fortunately we can do this using the same Euler tour technique we used to root the tree.

Give every advanced edge a value of 1, and every retreat edge a value of 0. Let {u,v} be the advanced edge containing v. Then every advanced edge {x,y} between {u,v} and {v,u} gives us a descendant y of v. We can compute the prefix sum of the list of edges in parallel to find this value for every vertex. Computing the prefix sum of a list can be done in O(log n) time using O(n / log n) processors [27].

**Determining if a cut vertex lies on a cycle:**

Let the lowest common ancestor for an edge (x,y) in G[T] and not in T be denoted z. The paths from x to z and from y to z in T combined with the edge (x,y) form a cycle. If v is also a descendant of z (or v = z) and v is an ancestor of either x or y then clearly v lies on this cycle.

So given a non tree edge (x,y), we first need to find the lowest common ancestor z. This is fairly simple and quite similar to calculating the number of descendants. We give every advanced edge in the Euler tour a value of 1, and every retreat edge a value of -1. By computing the prefix sums again, the value at the advanced edge {a,b} is the level of b in the tree. The lowest common ancestor is the vertex with the smallest level which appears between x and y in the Euler tour. We can also see that v is only a descendant of z and an ancestor of x or y if it appears between x and y in the Euler tour with a smaller level than one of the two. The technique for finding the lowest common ancestors using a Euler tour was first given by Vishkin and Berkman [28].

**Finding the connected components of M:** Either of the algorithms given by Cole and Vishkin or Iwana and Kambayashi for finding a spanning tree can also be used for finding the connected components of a graph.

Therefore the final complexity is:
**Number of iterations:** $O(\log_{\frac{3}{2}} n)$

| | |
|---|---|
| **Rooting tree, calculating number of descendants:** | O(log t) time using O(t / log t) processors. |
| **Finding the connected components of M:** | O(log n) time using O(n*A(n) / log n) processors. |
| **Determining if a cut vertex lies on a cycle:** | O(log t) time using O(t / log t) processors. |
| **Total:** | $O(\log^2 n)$ time using O(n*A(n) / log n) processors. |

### 3.1.4 Converting an induced forest into a perfect matching

**Lemma:** *Let G be a 3-regular bipartite 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:** Every vertex in M has a degree of 3 or 1. The vertices in M with degree 1 have a degree of 2 in the complement, and the 3-vertices become isolated. Excluding these non-isolated vertices, we see every vertex has a degree of 2 and as such the complement is a collection of cycles. As these cycles all exist in G, and every cycle in a bipartite graph is even, we can see these are all even cycles. □

Using the lemma above, finding a perfect matching of the vertices in the complement becomes extremely easy, we can just take alternating edges of every cycle. We can use this to decrease the number of 3-vertices by a constant fraction.

Let M be our odd induced forest from the previous step, G our input graph
**Induced forest to perfect matching**
While M is not a perfect matching:
    Compute a perfect matching N in the complement of M
    Let H = M ∪ N
    Compute a spanning forest T of H
    For every edge in T:
        If P(e,T,H) = 1; M = M⊕e

**Correctness**
A 2-path is defined as a path in which all internal vertices have a degree of 2, and the two end vertices have a degree other than 2.

N and M are edge disjoint, therefore in N, every vertex of degree 1 is matched with an edge not in M. Therefore every vertex of degree 1 becomes a vertex of degree 2 in H. We

can therefore see that the end vertices of a 2-path must have a degree of 3 as every vertex in H has a degree of 2 or 3.

**Lemma:** *Let Q be a 2-path in H with u, v being the two end vertices. If e ∈ Q and P(e,T,H) = 1 then u and v are removed as 3-vertices from M.*

**Proof:** Let f ∈ Q, f = (u,x) for some vertex x in Q.

If a fundamental cycle includes an edge e of a 2-path, it clearly must include all the edges of the 2-path, meaning we have P(e,T,H) = P(f,T,H).

Then as P(e,T,H) = P(f,T,H) = 1, f is removed and so the degree of u is no longer equal to 3. Note that we cannot have another edge incident to u added, as all the edges incident to u were already present in M. We can apply the exact same argument to the other end vertex. □

Let n(M) = Number of 3-vertices in M

**Lemma:** *Let S be the exclusive-or of a set of fundamental cycles of H. Let M' = M⊕S. Then n(M') ≤ $\frac{1}{2}$n(M)*

**Proof:** Take any connected component I of H. Suppose I has 2p 3-vertices and 2q 2-vertices. Then we have 2p + 2q - 1 edges in any spanning tree T of I.

To find the minimum number of non-tree edges we need to find the minimum number of edges possible in I. This is when we have 6p = 4q edges. 6p = 3p + 2q, so the number of non-forest edges is

(3p + 2q) - (2p + 2q -1) = p + 1

Every one of these edges appears in only one fundamental cycle, so every edge is contained in S. Let e be one of these edges. We have already seen that every vertex has degree 2 or 3, so e is either incident to two 3-vertices or it is in a 2-path.

Case 1) e is incident to two 3-vertices

In this case both the vertices that e is incident to are removed as 3-vertices. If h is another non-forest edge incident to two 3-vertices, then h can only be incident to one of the same two vertices, meaning that e removes at least one 3-vertex from I

Case 2) e is in a 2-path

In this case from our previous lemma both end vertices are removed as 3-vertices. There can only be one non-forest edge per 2-path, otherwise we would isolate at least 1 vertex, so this removes at least one 3-vertex from I.

Therefore as the number of 3-vertices has decreased by at least half in every connected component of H, we can see that M' has at most p-1 3-vertices. □

**Theorem:** *After log(n) iterations, M is a perfect matching*

**Proof:** We at least half the number of 3-vertices every iteration, we have seen that we don't isolate any vertices, therefore after log(n) iterations we must be left with every vertex having a degree of 1, giving us a perfect matching. □

**Complexity**
We need to be able to compute a perfect matching in the complement, find a spanning tree of H and calculate P(e,T,H) for every edge in H.
We have already seen how to find a spanning tree and calculate P(e,T,H), all that is left is finding a perfect matching in the complement. This is extremely simple, as we know the complement is a series of edge disjoint even cycles, we can just take alternating edges from each connected component. We therefore just need to find the connected components, which again we already know how to do.

Therefore the final complexity is:
**Number of iterations:** O(log n)

| | |
|---|---|
| **Finding connected components:** | O(log n) time using O(n*A(n) / log n) processors. |
| **Finding a spanning tree of H:** | O(log n) time using O(n*A(n) / log n) processors. |
| **Calculating P(e,T,H):** | O(log n) time using O(n / log n) processors. |
| **Total:** | $O(\log^2 n)$ time using O(n*A(n) / log n) processors. |

We can therefore see that the algorithm runs in $O(\log^2 n)$ time using O(n*A(n) / log n) processors, giving us an NC algorithm for finding a perfect matching in 3-regular bipartite graphs. We will examine how this may help us find an algorithm for general graphs in the next section.

### 3.1.5 A sequential Python implementation

Part of this project was to fully implement this algorithm sequentially in Python, and visualise the result. The code is fairly long and relies on lots of subroutines, so a full description of how it works is left in comments in the code. I will give a brief description here of how the program works, and how it visualises the algorithm.

The program relies extensively on the graph-tool Python library created by Tiago Peixoto [29], and requires this installed to run.

**Subroutines**
These are the subroutines used to carry out the 3 main sections of the algorithm.

*newBiRegGraph*(n): Creates a 3-regular bipartite graph of size n.

*findSpanningForest*(G): Given a graph G, runs a breadth first search from some random vertex. Repeats until every vertex is in the forest.

*findFCycle*(T,u,v): Given a tree T, this finds the fundamental cycle connecting vertices u and v in the tree. It returns a graph with the edges of the fundamental cycle.

*edgesInTree*(G,T): This finds and returns a list of fundamental edges, edges which are in G but not in the spanning forest T.

*FindCycle*(G,v): Given a vertex v, it checks to see if v lies on a cycle in G using depth first search.

*VInduce*(G,T): Returns the graph induced by G over the non-isolated vertices in T.

*FindCC*(G): Uses breadth first search to find the different connected components.

*isInduced*(T,G): Determines if T is induced over G.

*numDescendants*(G,v,visited,descendants): Recursively calculates the number of descendants every vertex has in G.

*FindCutVertex*(T,root): Finds a cut vertex given a tree T and a root vertex.

*Complement*(G,M): Returns the complement of M with respect to edges in G.

*matchingInComplement*(N): Computes a perfect matching in N, a complement of a pseudo perfect matching, by taking alternating edges from each connected component.

*XOREdge*(G,e): Performs G = G$\oplus$e.

*Union*(G,H): Returns the union of G and H.

**Main Algorithm**

These are sequential implementations of the three main sections of the algorithm. Each one uses the subroutines above to carry out the tasks described. Also included is a function *checkPM* which automatically runs at the end of the algorithm. This will check that every edge in the perfect matching found is in the graph, and checks that every vertex has a degree of 1. This is used to verify that the algorithm has found a correct perfect matching.

*ppmToForest*(G): Creates a copy of a 3-regular bipartite graph G and turns this into a odd forest. It does this by finding a spanning tree of G, finding a set of fundamental edges and then finding a fundamental cycle for all these edges. It then calculates the number of times that each edge appears in a fundamental cycle, stored in a 2D array, and removes them from G if the parity is odd.

*Induce*(G,M): Given G and a odd forest M, it converts the odd forest into an odd induced forest. We find the different connected components of M, we use *isInduced* to check if the components are induced. For every non-induced tree, we find a cut vertex and determine if the cut vertex lies on a cycle C. If it does, we let M = M$\oplus$C. If not we set I = Induce(G, T - {v}) and M = (M - T) $\cup$ I $\cup$ T[S[v]].

*IForestToPM*(G,M): Given G and an odd induced forest M, it will convert M into a perfect matching of G. It will find the complement of M with respects to G, find a perfect matching

N in the complement. It creates a new graph H = M∪N, finds a spanning forest and a set of fundamental cycles for these edges, and then calculates the number of time that an edge in H appears in a fundamental cycle, and removes the edge from H if the parity is odd.

*checkPM*(): After finding a perfect matching by running the 3 algorithms above, this checks that:

1) Every vertex has a degree of 1
2) Every edge in the perfect matching is in the original graph G.

If these conditions are not met, it will produce an error message in the console.

**Visualisation**

As part of my project I wanted to visualise this algorithm working. Initially this was going to run alongside the algorithm itself, however this was essentially not possible when converting an odd forest into an odd induced forest, as the visualisation can only work on a single graph and multiple different graphs are required at this stage. Another problem with this approach is it's very unclear what is actually happening, we can only show what appears to be a random sequence of edges being removed.

The program offers animated visualisations of the three main parts of the algorithm. In the case of ppmToForest, it shows an animation of the original graph G having a set of fundamental cycles XORed to it. The vertices and the edges in the fundamental cycle are coloured, so it is slightly easier to understand what is happening.
IForestToPM works in the same way, although the fundamental cycles are now of the graph H = M∪N, where M is our odd forest and N is a perfect matching in the complement of this forest.
For Induce it shows the uninduced edges of any tree in the forest in red, and it displays the new edges which will be added as a result of inducing the forest in yellow.

The program also offers a second option for visualisation, which is to display the graph at different stages of the algorithm. This offers a better demonstration of the algorithm in my opinion. It allows the user to clearly see important structures and features of the graph which are crucial to the algorithm's correctness (for example, the complement of the odd forest being a series of disjoint even cycles, or the structure of the graph H as defined above).

When running the algorithm, a list of options is clearly displayed in the terminal so the user can pick whichever visualisation they would like to view.

## 3.2 Kulkarni's algorithm

The following algorithm for finding a perfect matching in 3-regular bipartite graphs is given by Kulkarni [30]. The approach is very different from the last algorithm we looked at, and primarily relies on something called a matching polytope.

### 3.2.1 Polytopes related to matchings

Given a graph G = (V, E), we can define a geometric object called a matching polytope. For every matching M $\subseteq$ E, if we let $X^M$ represent the characteristic vector of M then the matching polytope is given by finding the convex hull of the set {$X^M$, M is a matching}.

The perfect matching polytope is defined in the same way, with M instead being a perfect matching. From this point we will denote the matching polytope of G as M(G) and the perfect matching polytope as PM(G)
Edmonds gave a description of both these polytopes for bipartite graphs. Let w(e) refer to the weight of an edge e with respect to some weighting function w, and w(v) refer to the weight of all edges incident to v.

PM(G) is given by:
w(v) = 1 for all v $\in$ V.
w(e) $\geq$ 0 for all e $\in$ E.
M(G) is given by:
w(v) $\leq$ 1 for all v $\in$ V.
w(e) $\geq$ 0 for all e $\in$ E.

Whilst these are sufficient for bipartite graphs, in the case of general graph we require a tighter constraint. This is, for every subset U $\subseteq$ V, w(E[U]) $<$ $\frac{1}{2}$ (|U| - 1)).
The fractional matching polytope (FPM(G)) is defined the same way for all graphs as the perfect matching polytope is for bipartite graphs. We shall see this fact is crucial for the correctness of the algorithm.

### 3.2.2 2-3 graphs and backtrackers

A 2-3 graph is a graph such that:
Every vertex has degree 2 or 3.
If a vertex has degree 2, its two neighbouring vertices have a degree of 3.
We will use E(G) to refer to the edges of the graph G

**Definition:** *Given graphs G and H, and respective weighting functions $w_g$ and $w_h$, a backtracker function is a function F: E(G) → {0,1} X E(H) ∪ {0,1} such that:*

| | |
|---|---|
| F(e) = 0 if | $w_g$(e) = 0 |
| F(e) = 1 if | $w_g$(e) = 1 |
| F(e) = (0,$e_h$) if | $w_g$(e) = $w_h$($e_h$) |
| F(e) = (1,$e_h$) if | $w_g$(e) = 1 - $w_h$($e_h$) |

Kulkarni and Mahajan prove the following theorem in [31], which is the key to this algorithm.

**Theorem:** *Given graphs G, H and interior points A = (G, $w_g$) of FPM(G) and B = (H, $w_h$) of FPM(H) and a backtracker function F: E(G) → {0,1} X E(H) ∪ {0,1}, there exists an NC algorithm which outputs:*

*i. a 2-3 graph G' and an interior point C = (G', $w_{g'}$) of FPM(G') such that $w_{g'}$(e') > 0 for all e' ∈ E(G')*

*ii. Another backtracker function F': E(G) → {0,1} X E(G') ∪ {0,1}*

We will define the procedure *make2-3* which takes two interior points A and B and a backtracker F as inputs and outputs a interior point C of a 2-3 graph G' and a backtracker F' as given above. We will denote this: (C',F') = *make2-3*(A,F,B). The NC algorithm described above constructs the 2-3 graph in such a way that any edges of weight 0 from G are deleted.

It isn't immediately obvious why we need these backtracker functions. Given a perfect matching in H, we can use the backtracker to find a perfect matching of G. Suppose we have a perfect matching $M_h$ of H, we will say $M_h$(e) = 1 if e is in the perfect matching, and 0 otherwise. Let $M_g$ be a perfect matching of G. Suppose we have a backtracker F: E(G) → {0,1} X E(H) ∪ {0,1} such that:

| | |
|---|---|
| F(e) = 0 if | $M_g$(e) = 0 |
| F(e) = 1 if | $M_g$(e) = 1 |
| F(e) = (0,$e_h$) if | $M_g$(e) = $M_h$($e_h$) |
| F(e) = (1,$e_h$) if | $M_g$(e) = 1 - $H_h$($e_h$) |

To use this to find a perfect matching in G we would need to find a way to construct this backtracker function so it maps every edge of G to 0 or 1 whilst remaining in the FPM(G) (which again is equal to PM(G) in the bipartite case)

### 3.2.3 Manipulating edge weights

We have already explored the concept of fundamental cycles whilst looking at the previous algorithm. We'll be using them again here to find a big-even cycle vector. This is a cycle in a 2-3 graph G such that the cycle contains at least half of the 3-vertices in G.

*Find big-even(G)*:
Find a spanning tree T of G
Find S, the set of fundamental cycles in G with respect to T
Let C = $\oplus$ c $\in$ S

We've performed a very similar operation in our previous algorithm, by calculating p(e,T,G) for every edge we find the parity of the number of fundamental cycles e is contained in, therefore C = {e; p(e,T,G) = 1}. We have already seen we can do this in **NC**. Our proof that it contained half the 3 vertices of our odd forest also holds here, our only difference here is every 2-path has a length of 1. We will look at the simpiler proof for 2-3 graphs however.

**Lemma:** *find big-even(G) finds a big-even cycle in G*

**Proof:** Suppose we have K 3-vertices and L 2-vertices in G The minimum amount of edges is given when 2L = 3K
If we have 2L = 3K then L = 3K/2
We therefore have 2L = L + 3K/2 edges in the graph
We know the amount of edges in a spanning tree is precisely K + L - 1
So we can see that we have at least K/2 edges not in the spanning tree
All of these K/2 edges appear in our big-even cycleas they only appear in a single fundamental cycle
We know that every edge in G has at least one endpoint being a 3-vertex
So we can see that we have at least K/2 3-vertices in our cycle, so our cycle contains at least half of the 3-vertices. □

We need to define one more procedure, *simple-manip*, which will allow us to move to a different interior point of the perfect matching polytope, given an interior point and a big-even cycle C.

*simple-manip(C,G)*:
For every cycle c in C:
Pick a minimum weight edge e in the cycle, let w(e) = k
Add k to the weights of edges in the cycle which are an odd distance from e
Remove k from the weights of edges in the cycle which are an even distance from e

We can see that we still have w(e) $\geq 0$ for every edge, and the weights of each vertex remains unchanged, so this gives us a different interior point of our perfect matching polytope.

### 3.2.4 Finding a perfect matching

Given a bipartite cubic graph G:

Assign w(e) = $\frac{1}{3}$ for all e $\in$ E. We then have an interior point A = (G, $w_g$) of PM(G).
Let B = (H,$w_h$) = (G,$w_g$) be an interior point of H, a copy of G.
Let F: E(G) $\rightarrow$ {0,1} X E(H) $\cup$ {0,1} such that F(e) = (0,e) for all e in E(G).
While H is non-empty:
C = *find-big-even*(H)
B = *simple-manip*(C,H)
(B,F) = *make2-3*(A,F,B)
Backtrack a perfect matching in G using F.

#### Correctness

**Lemma:** *During the course of the algorithm the weights of the edges of H will be either 0,* $\frac{1}{3}$ *or* $\frac{2}{3}$.

**Proof:** We start with every edge having a weight of $\frac{1}{3}$ and therefore both *simple-* and *make2-3* are only able to change them to 0 or $\frac{2}{3}$. □

**Lemma:** *At the start of every while loop, H is a 3-regular bipartite graph with every weight equal to* $\frac{1}{3}$

**Proof:** H is a 2-3 graph at the start of every while loop, so every edge has at least 1 endpoint being a 3-vertex. All edges with a weight of 0 are deleted by make2-3. Therefore the edges have a weight of $\frac{1}{3}$ or $\frac{2}{3}$.
If an edge has a weight of $\frac{2}{3}$ then as the other two edges incident to the three vertex can have a minimum weight of $\frac{1}{3}$ each, we see that the weight function does not give us an interior point of PM(H) as $w_h(v) \geq \frac{4}{3}$. Therefore all edges must have weight of $\frac{1}{3}$. As $w_h(v)$ = 1 for every vertex, we can see that every vertex must have a degree of 3. □

**Lemma:** *At the end of each while iteration we have deleted a constant fraction of edges.*

**Proof:** As all the edges at the start of a while loop have a weight of $\frac{1}{3}$, the minimum weight edge found by *simple-manip* will have a weight of $\frac{1}{3}$, and this weight will be removed from half the edges in the big-even cycle, which contains at least half of the 3-vertices in H.
As every vertex in H is a 3-vertex, we can see that there are $3|V_H|$ edges in total. Our big-even cycle contains $\frac{1}{2}*2*|V_H|$ edges, and we remove half of these. Therefore we remove a minimum of $\frac{1}{6}$ of the edges per iteration. □

**Theorem:** *The algorithm finds a perfect matching in 3-regular bipartite graphs in NC time.*

**Proof:** As H is empty, our backtracker now has to map every edge of G to either 0 or 1 whilst remaining in PM(G). We can see this gives us a perfect matching.

As our while loop deletes a constant fraction of edges, it terminates after O(log n) iterations. All our procedures run in **NC** so therefore our algorithm is **NC**.□

We will analyse this algorithm and examine how the techniques may be adapted for general graph however we will first look at a reduction which makes this easier.

# Chapter 4

# Reduction from matching to matching in 3-regular graphs

We have seen two **NC** algorithms for perfect matchings in 3-regular bipartite graphs. Before looking at how useful these algorithms may be for general graphs, we will first look at a reduction from maximum matchings in general graphs to maximum matchings in 3-regular graphs given by Bidel in [32].

## 4.1 Transformations of the graph

**Transformation $T_D$: Doubling the graph**
This is a single transformation in which we double the edges in the graph. This clearly preserves the size of a maximum matching, and therefore $T_D(G)$ will have a perfect matching if and only if G has a perfect matching. Our motivation for this transformation is it will always leave us with a even number of vertices with a degree of 2, something which will be required for later transformations.

**Transformation $T_1$: Removing vertices of degree 1**
Let a be a vertex in G with deg(a) = 1
Add 4 additional vertices to G: b, c, d and e
Add the following edges to G: {(a,b), (b,c), (c,d), (d,a), (b,e), (c,e), (d,e)}

Let $T_1(G)$ denote G after applying this transformation to a single vertex of degree 1. Let $T'_1(G)$ denote G after applying this transformation to all vertices of degree 1.
If we have a matching in $T_1(G)$, due to the fact that the 7 additional edges are incident to 5 different vertices, we can have a maximum of 2 of these edges contained in the matching.
Therefore the size of a maximum matching M* of $T_1(G)$ is equal to the size of maximum

matching M of G plus 2. Notice also that we can always find a maximum matching of this size by letting M* = M ∪ {(b,c), (d,e)}.

We can also see this transformation preserves the property of G having a perfect matchings, if M is perfect then M* as defined above is also perfect, so $T_1(G)$ has a perfect matching if and only if G does. We can recover a maximum matching of G from $T'_1(G)$ in O(1) time by by removing any edges we have added in the transformation from the matching.

**Transformation $T_2$: Removing vertices of degree 2**

Let a, b be two vertices with a degree of 2
Add two vertices c and d
Add edges {(a,c), (b,c), (c,d)}

Let $T_2(G)$ denote G after applying this transformation to two vertices of degree 2. Let $T'_2(G)$ denote G after applying this transformation to all vertices of degree 2. Note that we need to apply $T_D$ to G before we can apply $T_2$, because if not we cannot guarantee we have an even number of vertices with a degree of 2.

If we have a matching in $T_2(G)$, as all three additional edges are incident to vertex c we can see that we can only add one of these three edges to the matching.

Therefore the size of a maximum matching M* of $T_2(G)$ is equal to the size of maximum matching M of G plus 1. We can always find a maximum matching of this size, let M* = M ∪ (c,d). Furthermore if M is a perfect matching of G then clearly M* is a perfect matching of $T_2(G)$, so we can see this preserves the property of whether G has a perfect matching. Again, it is simple for us to recover a maximum matching of G from a maximum matching of $T_2(G)$ by removing any edges we have added in the transformation from the matching.

**Transformation $T_4$: Removing vertices of degree $\geq 4$**

Let a be a vertex of degree 4
Let $v_1$, $v_2$ be two neighbouring vertices of a
Remove the edges (a,$v_1$) and (a,$v_2$)
Add vertices b and c
Add edges {(a,b), (b,c), ($v_1$,c), ($v_2$,c)}

We will obviously have to apply this transformation multiple times for each vertex with a degree $\geq 4$, as each iteration will only decrease the degree of a vertex by 1.
If we have a matching in $T_4(G)$, then as b is only incident to a and c only two vertices in the matching can be incident to a, b or c.
Suppose M is a matching of G. If (a,$v_1$) ∈ M then M' = M - (a,$v_1$) ∪ {(a,b),(c,$v_1$)} is a matching of cardinality |M| + 1 in $T_4(G)$.

If (a,$v_2$) ∈ M then M' = M - (a,$v_2$) ∪ {(a,b),(c,$v_2$)} is a matching of cardinality |M| + 1 in $T_4(G)$.

If neither of these edges are in M, then M' = M ∪ {(b,c)} is a matching of cardinality |M| + 1 in $T_4$(G).
If M' is a matching of $T_4$(G) then at most only two edges are incident to a, b, and c as b is only incident to a and c. By removing one of these edges, we get a matching of size M of size |M'-1|. Contracting the edges (a,b) and (b,c). as only one edge in M is incident to a, b or c, we can see that M is a matching in the resulting graph which is G.
Therefore we can see that $T_4$(G) has a matching of size K+1 if and only if G has a matching of size K, and we can recover a matching of G from a matching of $T_4$(G) in O(1) time.

Applying the transformations $T'_1(T'_2(T'_D(T'_4(G))))$ we have turned G into a 3-regular graph. We can recover a matching from the transformed graph in O(1) time simply by keeping a list of which edges we have added to the graph and removing these from any matching.

## 4.2 Examining this reduction

We may ask if this preserves bipartiteness, however we can actually see the transformation $T_1$ alone doesn't. If a is a vertex of degree 1 in G which has been transformed, as we have added edges {(b,c), (c,e), (b,e)}, we can see that b and c would have to be on opposite sides, and then one of the edges edges (c,e) and (b,e) would destroy bipartiteness.

It's actually fairly obvious this is the case. If we don't have the same number of vertices on both sides of the bipartite graph, there clearly cannot exist a perfect matching. Yet a 3-regular bipartite graph must always have a perfect matching. In fact, it has 3 completely edge disjoint perfect matchings, meaning the total number of perfect matchings is exponential.

Regardless, this gives us a good starting point to consider how the algorithm given by Sharan and Wigderson will work on general graphs. Let T(G) denote the general graph G after having the transformations above applied to it. We can now see that the theorem given at the beginning of Section 3.1.2 holds for a pseudo perfect matching of every graph T(G). Converting a pseudo perfect matching of T(G) (which we can now take as just a copy of the graph) to an odd induced forest can be done with no issues. The main problems come when we need to find a perfect matching in the complement.

Applying the transformation to the graph greatly simplifies the structure of the complement of an odd forest. We cannot say anything about the complement for G, if the maximum degree of a vertex in G is $n$, then the degrees of the vertices in the complements range from 0 to ($n$-3). Finding a perfect matching, or any kind of matching, is just as hard here as it is in G.

In T(G), every vertex has a degree of 3. Therefore as every vertex has a degree of 1 or 3 in the odd forest, ignoring any isolated vertices we can see that given an odd forest M, the

complement of M is a collection of edge disjoint cycles again. Unfortunately these cycles are not even cycles, so by taking alternating edges we may leave 1 vertex unmatched. The smallest odd cycle contains 3 vertices, so our worst case is $\frac{1}{3}$ vertices in the complement aren't matched in the matching N. It could easily be the case that every 3-vertex is connected to one of these unmatched vertices. Then to convert $H = M \cup N$ into a graph with every vertex having degree 2 or 3, we would have to remove these unmatched vertices, and H would contain no vertices of degree 3, so we couldn't remove any from M, and the algorithm fails.

It would be interesting to look at this in greater detail. If T(G) does have a perfect matching, can we always find a perfect matching of the vertices in the complement? Is there any sort of relationship between the number of unmatched vertices in the complement and whether or not T(G) has a perfect matching? All these cycles exist in G, is there any way they could prove useful in constructing an isolating weighting function of the graph?

Unfortunately this reduction doesn't make the algorithm given by Kulkarni any more useful for general graphs. The reason is simple, the perfect matching polytope PM(G) has much harder constraints to satisfy than the fractional matching polytope FPM(G) for general graphs. The **NC** algorithm given by Kulkarni and Mahajan in [31] is unlikely to prove useful for finding an interior point of PM(G) for general graphs because of these additional constraints.

# Chapter 5

# Conclusions and future work

## 5.1 Future Work

If I were to extend more on this project I think it would be an interesting but realistic challenge to implement the algorithm given in "Matching is as easy as matrix inversion" in parallel, as parallel matrix algorithms are probably the most simple parallel algorithms to implement. It would be useful to try implementing this algorithm using different methods of parallel computation, and comparing the efficiencies to the best sequential algorithms. If I were to continue working on this project, this would be my next objective.

I think it would be interesting to look at the isolating lemma in closer detail, and examine the different techniques used to make the minimum weight perfect matching unique in certain structures of graphs. It seems to be the most useful tool we have for creating efficient parallel matching algorithms, and the success in derandomizing the weight assignment for different types of graphs means this may be the most currently promising method of finding a **NC** matching algorithm.

Whilst Tutte's theorem is extremely useful for solving the decision problem of whether a graph contains a perfect matching, it would also be beneficial to look at other possible ways of doing this. Determining if the determinant of the Tutte matrix is identicially equal to zero is always going to be very hard to compute, purely because the determinant can contain an exponential amount of terms. Whilst in practice the probability that the algorithm is correct can be made arbitrarily close to 1, we are still probably never going to get a **NC** decision algorithm using the Tutte matrix, meaning other methods will be needed.

It would be interesting to consider the algorithm given by Sharan and Wigderson further. It's probably unlikely we would make any real progress adapting the methods to general graphs, however it may be beneficial examining further how the algorithms techniques could be adapted to 3-regular graphs.

## 5.2 Author's Assessment of the Project

My initial goal for this project, set out in the specification, was to analyse in detail the two algorithms for 3-regular bipartite graphs and discuss their usefulness in finding a general **NC** maximum matching algorithm. I feel that my project accomplishes this, and moreover provides a fairly good summary of existing **RNC** matching algorithms. Whilst there are obviously many more algorithms not covered upon here, many rely on variations on the isolating lemma, or employ the use of Tutte's theorem, both of which have been covered.

Another goal was to visualise both the algorithms for 3-regular bipartite graphs we looked at. Implementation was difficult, the initial graph library I wanted to use (JGraphT) was unsuitable for visualisations in my opinion, and the library I did use often worked in slightly unintuitive and complicated ways. This is the reason why the algorithm code is so long despite the algorithm itself being relatively simple. In total it took around 5 weeks to implement the algorithm and get the visualisations working, by this time I simply didn't have enough time to implement the algorithm given by Kulkarni. Constructing the *make2-3* function would have been the main issue; the **NC** algorithm described would have been fairly complicated to implement, implementing this algorithm alone would have been a fairly substantial challenge for this project.

Instead of implementing this algorithm I spent more time researching different **NC** algorithms and techniques to solve the matching problem. In hindsight I feel this was more useful. Originally I only planned to look at these two algorithms, but this allowed me to present the **RNC** algorithms we looked at in the beginning of the paper, which I think was a very important part of this project.

I wanted to explain to the reader exactly how the algorithms worked on a PRAM, explaining for example how the different processors were used as I feel it can sometimes be hard to understand exactly how an algorithm works in parallel. I spent a lot of the first term looking at PRAM algorithms in the book by JaJa [33] for tree contraction, Euler tours and other algorithms used. The reason it mostly wasn't included in this report was simply length. To describe any of these would have required explaining the work-time presentation framework, multiple basic parallel algorithm techniques and lots of preliminary definitions and theorems. The correctness of these algorithms is often quite complicated as well. Ultimately this was content that is covered in any basic parallel algorithms book and I didn't believe adding it all to my report would have been worthwhile.

If I were to do this project again, I would have examined the **Quasi-NC** algorithm given for bipartite perfect matching instead of Kulkarni's algorithm as I feel this is more relevant and a more important result. I would also have spent much less time looking at PRAM algorithms for various problems, as these mostly didn't end up in my final report. When

visualising the algorithm, I would have stored the graph in a traditional data structure (adjacency matrix or adjacency lists) until I needed to visualise it, this would have probably made the code a lot simpiler.

Overall I am pleased with what I have accomplished in this project. I have had to independently search for relevant research papers and gain a good understanding of them, something I have never done before. I've also had to learn about the PRAM model of parallel computation and parallel complexity theory, things I had not encountered before in my studies. I've had to directly implement an algorithm from one of these papers, which again was something I had not had to do before. Lastly I've had to write this report, which is the first piece of formal writing I have done at university. Whilst I found the project very challenging in places, I have enjoyed it immensely and think it has been an important part of my degree.

# Chapter 6

# Acknowledgements

The algorithm I implemented relied heavily on the graph-tool Python library [29] created by T. Peixoto, which provided an extremely intuitive method for animating the visualisations. The majority of my initial research on PRAM algorithms came from books by Ja'Ja [33] and Gibbons & Rytter [34]. The book *Introduction to Parallel Algorithms and Architectures* by Leighton [14] gave a very good insight into the proofs of the isolating lemma and Tuttes theorem by introducing the concept of the trail of a permutation.
Finally I'd like to thank my supervisor Artur Czumaj for all the help and advice he has given me over the duration of this project.

# References


[1] M. Wu and J. Leou. *A bipartite matching approach to feature correspondence in stereo vision*. Pattern Recognition Letters, 1995 (cit. on p. 4).

[2] N. Trinajstić ; D. J. Klein and M. Randić. *On some solved and unsolved problems of chemical graph theory*. International Journal of Quantum Chemistry, 1986 (cit. on p. 4).

[3] G. F. Lev ; N. Pippenger and L. G. Valiant. *A fast parallel algorithm for routing in permutation networks*. IEEE Transactions on Computers, 1981 (cit. on p. 4).

[4] C. Berge. *Two theorems in graph theory*. Proceedings of the National Academy of Sciences of the United States of America, 1957 (cit. on p. 4).

[5] J. Edmonds. *Paths, trees, and flowers*. Canadian Journal of Mathematics, 1965 (cit. on p. 4).

[6] V. V. Vazirani and S. Micali. *An O ($V^{1/2}E$) algorithm for finding maximum matching in general graphs*. Proc. 21st IEEE Symp. Foundations of Computer Science, 1980 (cit. on p. 4).

[7] L. R. Ford and D. R. Fulkerson. *Maximal flow through a network*. Canadian Journal of Mathematics, 1956 (cit. on p. 4).

[8] J. Hopcroft and R. M. Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*. SIAM Journal on Computing, 1973 (cit. on p. 4).

[9] L. Lovasz. *On Determinants, Matchings And Random Algorithms*. Fundamentals of Computation Theory, 1979 (cit. on p. 7).

[10] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Morgan Kaufmann, 2000 (cit. on p. 8).

[11] R. Zipple. *Probabilistic algorithms for sparse polynomials*. EUROSAM, 1979 (cit. on p. 9).

[12] R. M. Karp ; E. Upfal and A. Wigderson. *Constructing a perfect matching is in random NC*. 1985 (cit. on pp. 9, 10).

[13] K. Mulmuley ; U. V. Vazirani and V. V. Vazirani. *Matching is as easy as matrix inversion*. 1987 (cit. on p. 10).

[14] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*. Morgan Kaufmann, 1991 (cit. on pp. 11, 37).

[15] V. Pan. *Fast and efficient algorithms for the exact inversion of interger matrices*. Fifth Annual Foundations of Software Technology and Theoretical Computer Science Conference, 1985 (cit. on p. 12).

[16] M. Agrawal; T. M. Hoang and T. Thierauf. *The polynomially bounded perfect matching problem is in $NC^2$*. 24th International Symposium on Theoretical Aspect of Computer Science, 2007 (cit. on p. 12).

[17] E. Dahlhaus and M. Karpinski. *Matching and multidimensional matching in chordal and strongly chordal graphs*. Discrete Applied Mathematics, 1998 (cit. on p. 13).

[18] S. Datta; R. Kulkarni and S. Roy. *Deterministically isolating a perfect matching in bipartite planar graphs*. Theory of Computing Systems., 2010 (cit. on pp. 13, 14).

[19] S. Fenner ; R. Gurjar and T. Theirauf. *Bipartite Perfect Matching is in quasi-NC*. 2016 (cit. on p. 13).

[20] Meena Mahajan and Kasturi Varadarajan. *A new NC algorithm to find a perfect matching in planar and bounded genus graphs*. Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, 2000 (cit. on p. 14).

[21] R. Sharan and A. Wigderson. *A new NC Algorithm for Perfect Matching in Bipartite Cubic Graphs*. 1996 (cit. on p. 14).

[22] K. R. Abrahamson ; N. Dadoun ; D. G. Kirkpatrick and T. M. Przytycka. *A simple parallel tree contraction algorithm*. Journal of Algorithms, 1989 (cit. on p. 16).

[23] U. Vishkin and R. E. Tarjan. *Finding biconnected components and computing tree functions in logarithmic parallel time*. Proceedings of FOCS, 1984 (cit. on p. 17).

[24] R. Cole. *Parallel merge sort*. 27th Annual Symposium on Foundations of Computer Science, 1986 (cit. on p. 17).

[25] R. Cole and U. Vishkin. *Approximate parallel scheduling. Part II: Applications to logarithmic-time optimal graph algorithms*. 27th Annual Symposium on Foundations of Computer Science, 1991 (cit. on p. 17).

[26] K. Iwama and Y. Kambayashi. *A simpler parallel algorithm for graph connectivity*. Journal of Algorithms, 1994 (cit. on p. 17).

[27] R. E. Ladner and M. J. Fischer. *Parallel Prefix Computation*. Journal of the ACM, 1980 (cit. on p. 19).

[28] U. Vishkin and O. Berkman. *Recursive Star-Tree Parallel Data Structure*. SIAM Journal on Computing, 1993 (cit. on p. 19).

[29] T. Peixoto. *The graph-tool python library*. URL: https://graph-tool.skewed.de (cit. on pp. 22, 37).

[30] R. Kulkarni. *A New NC-Algorithm for Finding a Perfect Matching in d-Regular Bipartite Graphs*. Lecture Notes in Computer Science, vol 3998, 2006 (cit. on p. 25).

[31] R. Kulkarni and M. Mahajan. *Seeking a vertex of the planar matching polytope in NC*. 12th European Symposium on Algorithms ESA, 2004 (cit. on pp. 26, 33).

[32] T. Biedl. *Linear reductions of maximum matching*. Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, 2000 (cit. on p. 30).

[33] J. Ja'Ja. *An introduction to parallel algorithms*. Addison wesley, 1992 (cit. on pp. 35, 37).

[34] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1990 (cit. on p. 37).