PARALLEL ALGORITHMS FOR MATCHING AND INDEPENDENCE PROBLEMS IN GRAPHS AND HYPERGRAPHS

by

AARON ANDREW WINDSOR

(Under the Direction of E. Rodney Canfield)

ABSTRACT

We consider the following problem: Given a greedy graph algorithm that seems to be inherently sequential, to what extent can we expect to speed up the computation by making use of additional processors? We obtain several positive results for particular problems, showing that it is theoretically possible to parallelize some greedy graph algorithms to the extent that their parallel running times are asymptotically much faster than their sequential running times. Highlights of our results include:

- A simple proof that a simple algorithm of Luby ("the permutation algorithm") is an $\mathcal{RNC}$ algorithm for finding a maximal independent set in a graph, and the first known derandomization of that algorithm.
- The first non-trivial upper bound on the deterministic time complexity of finding a maximal independent set in a hypergraph on a PRAM.

- A partial analysis of the permutation algorithm generalized to hypergraphs, showing that it outperforms the best known algorithm for finding a maximal independent set in a hypergraph in the following sense: For hypergraphs of dimension at least 6, any set of vertices is at least as likely to be added to the independent set by an iteration of the permutation algorithm as it is to be added by an iteration of the other algorithm.

- The first $\mathcal{NC}$ algorithm for finding a maximal forest in a hypergraph.

- The first $\mathcal{NC}$ algorithm for finding a maximal acyclic set in an undirected graph.

INDEX WORDS: Parallel algorithms, Maximal Independent Set, Maximal Acyclic Set, Maximal Forest, Graph, Hypergraph

# PARALLEL ALGORITHMS FOR MATCHING AND INDEPENDENCE PROBLEMS IN GRAPHS AND HYPERGRAPHS

by

AARON ANDREW WINDSOR

B.S., North Carolina State University, 1999

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2004

PARALLEL ALGORITHMS FOR MATCHING AND INDEPENDENCE PROBLEMS IN GRAPHS AND HYPERGRAPHS

by

AARON ANDREW WINDSOR

Major Professor: E. Rodney Canfield

Committee: Liming Cai
Ken Johnson
William McCormick
Robert W. Robinson

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2004

ACKNOWLEDGEMENTS

I sincerely thank Rod Canfield for his encouragement and support over the past few years. Rod has always given freely of his time and has consequently had to sit through countless explanations of ridiculous ideas that didn't end up in this thesis and only a few ridiculous ideas that, after many months of work, did. He always did so with the patience of a good teacher. His strikingly clear explanations have been the ideal that I've aspired to achieve in my teaching and in the communication of my work; hopefully this thesis is evidence of that. Bob Robinson has also been a great influence on me during my time at UGA. I learned more from the courses in Computational Complexity and Combinatorics I took from him my first year than from any other classes I took as a grad student. Finally, I thank Liming Cai, Ken Johnson, and William McCormick for giving their time to be on my committee.

Of course, I also owe a great deal of thanks to my parents. I would have never even started my Ph.D. were it not for their enthusiastic support of my education. But, more importantly, they have always supported me in all things not related to this thesis, and for that I'm extremely grateful.

Finally, I thank Phoebe. By far, my time with her over the last few years has been the best part of my graduate experience. For

believing in me much more than I think she had good reason to, I owe her the most.

TABLE OF CONTENTS

LIST OF FIGURES

## Chapter 1 - Introduction

Imagine the set of all of the employees of some company, some pairs of whom don't work well together on projects - we'll call two employees "incompatible" if this is the case. From all of this company's employees, a manager would like to choose a set of employees to assign to a project such that no two employees on the project team are incompatible. The manager also wants as many employees as possible on the project, so the only way an employee should get out of working on the project is if they are incompatible with at least one person already working on the project.

One way to select the project team is to line everyone up and, moving down the line, assign an employee to the project if and only if they are compatable with everyone already on the project. This process could take a while if there are a lot of employees, so we'd like to think about faster ways to select the project team. Clearly, our first attempt at solving the problem is as fast as possible if we only allow a single manager to choose the project team. So, could we speed things up by adding several more managers?

It won't necessarily be any faster to assign small teams of employees to each of several managers and try to run the selection process in parallel, since each manager might need to know about what decisions other managers have made in order to select a set of employees that are not only all compatible with each other, but also compatible with all employees selected by other managers. For example, say there are $n$ employees $e_1, e_2, \ldots, e_n$ and we use $n$ managers $m_1, m_2, \ldots, m_n$ to do the selection, with employee $e_i$ assigned to manager $m_i$ for all $i$. If, for all $i$ in the range $1 \ldots n-1$, employees $e_i$ and $e_{i+1}$ are incompatible, every manager needs to know what decision at least one other manager has made before he decides whether or not to include his employee on the project team. We'll need a more sophisticated way to parallelize this process to avoid simulating the original, slow process of selection in the worst case.

There is a way to select such a group in parallel that is asymptotically faster than the original sequential selection process, which takes time roughly logarithmic in the number of employees using a number of managers that is bounded by a polynomial in the number of employees. Chapter 2 of this thesis describes the algorithm and proves the necessary bounds on time and the number of managers. At its core, however, our project selection example is a specific instance of a more general problem easily described using the language of graph theory.

Let $G = (V, E)$ be an undirected graph, consisting of a finite set of vertices $V$ and a set of edges $E$, each of which is a set of two vertices. A set $S \subseteq V(G)$ induces a subgraph $G[S]$, which has vertex set $S$ and edge set $\{e \in E(G) \mid e \subseteq S\}$. Let $\mathcal{G}$ be the set of all graphs. We define a graph property $\pi$ to be a mapping $\pi : \mathcal{G} \rightarrow \{\text{true, false}\}$. A maximal subgraph of $G$ with property $\pi$ is a vertex-induced subgraph $G[S]$, for some $S \subseteq V(G)$, such that $G[S]$ has property $\pi$ but for any set $T$ that properly contains $S$, $G[T]$ does not have property $\pi$.

Typically, the graph properties we'll be interested in are simply stated, easily testable, non-trivial properties, such as "$G$ has no edges" or "$G$ contains no cycles". In either of these cases, there's a fairly efficient sequential algorithm to find a maximal subgraph of $G$ with property $\pi$: simply iterate through all of the vertices, maintaining a set of vertices $S$ as you go, adding any vertex $v$ to $S$ if and only if $G[S \cup \{v\}]$ has property $\pi$. After all vertices have been considered in this manner and if our $\pi$ is "well-behaved" in a certain sense that we'll define in section 1.2, $G[S]$ is a maximal subgraph of $G$ with property $\pi$.

We will define what it means to be a "fast parallel algorithm" in the next section. For now, the reader is encouraged to think of a fast parallel algorithm as one that, given access to multiple processors, can compute a function asymptotically faster than *any* algo-

rithm given access to only a single processor. The central question of this thesis is, informally, the following:

> Given a graph property $\pi$, is there a fast parallel algorithm that, given any graph $G$, computes a maximal subset $S \subseteq V(G)$ such that $G[S]$ has property $\pi$?

The example we started with fits into this framework by taking $V(G)$ to be the set of all employees, adding an edge to $E(G)$ for each pair of employees that are incompatible, and setting $\pi$ to be the property "$E(G) = \emptyset$". The general question above will be addressed and resolved in this thesis for various properties $\pi$.

## 1.1 Prerequisites

We expect this thesis to be understandable to any reader with an undergraduate-level background in theoretical computer science. We expect the reader to be familiar with basic asymptotic notation including big-$O$, big-$\Theta$, and big-$\Omega$ notation, basic computational complexity including the RAM model of computation and the complexity class $\mathcal{P}$, and some basic algebra at the level of understanding what fields and rings of polynomials are. A good background in probability theory is necessary, although some technical lemmas are included in the appendix that we'll use liberally to simplify our

probabilistic arguments. Finally, although we review some graph theoretic concepts in the following section, some prior experience with basic graph theory and combinatorics is necessary.

## 1.2 Some Graph Theory and Notation

For any set $S$, denote by $S^{(k)}$ the set of subsets of $S$ of size $k$, and by $\mathcal{P}(S)$ the powerset of $S$, which is the set of all subsets of $S$. A graph is an ordered pair $(V, E)$ where $V$ is a finite set and $E \subseteq V^{(2)}$. A hypergraph is a generalization of a graph, represented by an ordered pair $(V, E)$ where $V$ is again a finite set and $E \subseteq \mathcal{P}(V)$. In this thesis we deal with both graphs and hypergraphs, and we make the convention that we'll use the term "graph" only when we're specifically interested in excluding hypergraphs from consideration. When the hypergraph $H$ in question is clear, we'll often refer to $|V(H)|$ by $n$ and $|E(H)|$ by $m$.

An induced subgraph of a hypergraph can be specified by giving a subset of the vertex or edge set. For a hypergraph $H$ and some $S \subseteq V(H)$, we define $H[S]$, the subgraph induced by $S$, as

$$H[S] = (S, \{e \in E(H) \mid e \subseteq S\}).$$

For a hypergraph $H$ and some $F \subseteq E(H)$, we define $H[F]$, the subgraph induced by $F$, as

$$H[F] = \left( \bigcup_{e \in F} e \ , \ F \right).$$

A few special subclasses of hypergraphs will arise in this thesis. A *hypergraph of dimension* $d$ has no edge of cardinality more than $d$. A $k$-*uniform* hypergraph has only edges of cardinality $k$. Finally, a *linear hypergraph* has the property that no pair of edges is contained in more than one edge.

For any graph $G$, Let $\Gamma_G(v) = \{w \in V(G) \mid \{v, w\} \in E(G)\}$. We'll often drop the subscript and refer to $\Gamma(v)$ when the graph in question is clear. $d(v)$ will denote the degree of $v$, defined as $|\Gamma(v)|$.

A hypergraph property $\pi$ is *nontrivial* if infinitely many hypergraphs satisfy $\pi$ and at least one hypergraph does not satisfy $\pi$. $\pi$ is *hereditary* if, for all $S \subseteq V(H)$ and $T \subseteq S$, $\pi(H[S])$ implies $\pi(H[T])$. Finally, a property $\pi$ is *polynomial-time* testable if there exists a sequential algorithm that, for any hypergraph $H$, computes $\pi(H)$ in time polynomial in $n + m$. If a property $\pi$ is hereditary, then for any hypergraph $H$ and a subset $S \subseteq V(H)$ such that $H[S]$ has property $\pi$ and $H[S \cup \{v\}]$ does not have property $\pi$ for any $v \in H(G) \setminus S$, $H[S]$ is a maximal subgraph of $H$ with property $\pi$. If a property $\pi$ is hereditary, polynomial-time testable, and for any vertex set $V$, the

hypergraph $(V, \emptyset)$ satisfies $\pi$, then there's always a polynomial time algorithm to find a maximal subgraph with property $\pi$ of any graph that involves a simple iteration through the vertex set.

## 1.3 A Quick Introduction to Parallel Complexity Theory

Our model of a parallel computer is the PRAM, which assumes multiple processors working on a shared, random-access memory. The PRAM is a standard model in the field of parallel algorithms, although a little controversial because of it neglects issues like synchronization and the cost of communication between processors, which are major issues in any current parallel computational environment. The PRAM does not represent the best tradeoff between simplicity of analysis and applicability to predicting resource usage in a real setting; two other popular models that lean more toward the latter criterion are the BSP [41] and the LogP [14].

The questions addressed in this thesis, however, are of the form "Given a problem $\pi$, could we ever hope for a parallel algorithm to solve $\pi$ that is asymptotically faster than any sequential algorithm to solve $\pi$?", and we feel the PRAM is the best choice for answering questions of this nature. By ignoring aspects of the computation that are specific to the parallel environment being used, we are able to focus on whether or not a problem is parallelizable in what we

hope is some deeper sense than one can obtain by using a more (currently) realistic model. Furthermore, the PRAM can be simulated without much overhead by uniform circuit families that are "wide" and "shallow", which makes it the model of choice to approach complexity-theoretic questions.

The four main variants of the PRAM arise from deciding whether to allow two or more processors to read or write concurrently to a single memory location. These variants are the EREW, CREW, ERCW, and the CRCW PRAM, where EREW stands for "exclusive read, exclusive write", CRCW stands for "concurrent read, concurrent write", and the other two models are in between these two extremes. The distinction between these four models is more or less a formality in the context of the problems we wish to solve, since the CRCW PRAM model can be simulated by the EREW PRAM with only a logarithmic factor slow-down in time using a polynomial number of processors, and the general goal of this dissertation is to answer questions of the form "Is it possible to solve problem $X$ on a PRAM in poly-logarithmic time on a polynomial number of processors?".

For readers unfamiliar with the PRAM model and wanting a thorough introduction, we suggest the survey by Karp and Ramachandran [24]. For our purposes, it will be sufficient in most cases to notice that a few fundamental problems can be solved asymptotically more efficiently on a PRAM with multiple proces-

sors than on a single processor. We conclude this section with a short overview of two such fundamental problems.

Let $\circ$ be any associative binary operator. Then the value $x_0 \circ x_1 \circ \ldots \circ x_{n-1}$ can be computed by a PRAM using $\mathcal{O}(n)$ processors in time $\mathcal{O}(\log n)$, by way of a "binary tree" computation. We imagine all $x_i$'s as the leaves of a binary tree, each node in the tree as a PRAM processor, and the computation moving from the leaves to the root of the tree. During the first parallel computation step, for all $i$ from $0$ to $\lfloor \frac{n}{2} \rfloor - 1$, we combine $x_{2i}$ and $x_{2i+1}$ to form a parent containing $x_{2i} \circ x_{2i+1}$. Continuing in this manner, we arrive at the root in $\mathcal{O}(\log n)$ steps, and the root contains $x_0 \circ x_1 \circ \ldots \circ x_{n-1}$. This means that taking the sum, product, min, max, and many other simple functions of $n$ elements can be done efficiently on a PRAM. Using $\mathcal{O}(n^3)$ processors, this idea can also be used to obtain a $\mathcal{O}(\log^2 n)$ time algorithm for multiplication of two $n \times n$ matrices.

Sorting can also be done efficiently on a PRAM. Cole [9] has proven that sorting can be accomplished in time $\mathcal{O}(\log n)$ using $\mathcal{O}(n)$ processors on a PRAM. A much simpler classical algorithm by Batcher [3] is another alternative, running in time $\mathcal{O}(\log^2 n)$ with $\mathcal{O}(n)$ processors. Many of our algorithms deal with sets, and most set operations can be done efficiently in parallel using sorted lists. Testing for set membership can be done in $\mathcal{O}(\log n)$ time using a single processor and binary search, but also unions, in-

tersections, and many other binary operations on sets can be accomplished in $\mathcal{O}(\log n)$ time by concatenating the two lists, sorting them, then dealing with consecutive duplicates appropriately by examining consecutive pairs of elements in parallel.

### 1.3.1 $\mathcal{NC}$ and $\mathcal{RNC}$

Given a hypergraph function $f$, an $\mathcal{NC}$ algorithm for $f$ is an algorithm that, for any hypergraph $H$ with $n$ vertices and $m$ edges, computes $f(H)$ in time $\mathcal{O}(\log^k(n+m))$ on a PRAM with $\mathcal{O}((n+m)^c)$ processors for fixed $c, k > 0$.

Algorithms are sometimes simpler, faster, or both if they are supplied with a source of random bits and allowed to execute correctly with some probability less than 1. The random bits supplied turn the resources used by an algorithm (time, space, etc.) or even the output of the algorithm into a random variable. In this thesis, the randomness used by algorithms will cause the worst-case running time of the algorithm to become a random variable. In our analysis, we'll want to make a statement of the form "Algorithm $A$ computes function $f$ on any input of size $n$ in time $\mathcal{O}(t(n))$ using $\mathcal{O}(p(n))$ processors on an EREW PRAM with probability at least $1/2$", which hides a few details that we'll explain presently.

First, we imagine the algorithm $A$ taking, as one of its inputs, a string of $r$ random bits, where $r$ is bounded by some polynomial in $n$. For concreteness, we can think of $r$ as $c \cdot t(n)p(n)$, for some constant $c$, since we'll never be able to use more random bits than the total number of computation steps we use. Our probability space, then, is the set of all $2^r$ bitstrings with the uniform distribution. For some constant $c'$, after running $A$ for $c' \cdot t(n)$ steps, we stop $A$ and ask for its output, and if $A$ has finished its computation and can provide us with the correct answer it does so, otherwise $A$ will indicate that it has not finished the computation. The former case must occur with probability at least $1/2$. Even though the algorithm may not be finished when time is called, it will never output an incorrect answer.

In this setting, the constant $1/2$ is quite arbitrary and in fact we could substitute any constant $0 < \epsilon < 1$ and end up with the same class of algorithms. Given an algorithm $A$ that computes $f$ on an input of size $n$ in time $\mathcal{O}(t(n))$ with probability at least $\epsilon > 0$, we can run $A$ $1/\epsilon$ times, in which case $A$ still has execution time $\mathcal{O}(t(n))$, and $A$ succeeds iff at least one of those $1/\epsilon$ iterations succeed. The probability that $A$ has failed every iteration is now at most $(1-\epsilon)^{1/\epsilon} < e^{-1} < 1/2$.

Given a hypergraph function $f$, an $\mathcal{RNC}$ algorithm for $f$ is an algorithm that, for any hypergraph $H$ with $n$ vertices and $m$ edges,

computes $f(H)$ with probability at least $1/2$ in time $\mathcal{O}(\log^k(n+m))$ on a PRAM with $\mathcal{O}((n+m)^c)$ processors for fixed $c, k > 0$.

Although it's clear that $\mathcal{NC} \subseteq \mathcal{RNC}$ and $\mathcal{NC} \subseteq P$, it's not known whether any other inclusions among these complexity classes hold. $\mathcal{RNC}$ and $\mathcal{P}$ are incomparable, to the best of our current knowledge, since there are problems in $\mathcal{P}$ that are not known to be in $\mathcal{RNC}$ (the $\mathcal{P}$-complete problems, discussed below) and, perhaps more suprisingly, problems in $\mathcal{RNC}$ that are not known to be in $\mathcal{P}$ [32].

### 1.3.2 $\mathcal{P}$-Completeness

Any $\mathcal{NC}$ algorithm can be simulated by a single processor in polynomial time, since the product of processors used and time taken by an $\mathcal{NC}$ algorithm is always bounded from above by a polynomial. However, whether or not any algorithm that runs in polynomial time on a single processor can be simulated by an $\mathcal{NC}$ algorithm is a major open question in computational complexity theory. Analogous to the much more famous, and also unsolved, question of whether $\mathcal{NP} \subseteq \mathcal{P}$, in parallel complexity we ask whether $\mathcal{P} \subseteq \mathcal{NC}$.[1]

[1]The way we've defined $\mathcal{NC}$, $\mathcal{P} \subseteq \mathcal{NC}$ doesn't make much sense, since $\mathcal{P}$ is classically defined in terms of decision problems, i.e. functions mapping to the range $\{0, 1\}$, and we've defined $\mathcal{NC}$ in terms of function problems, with arbitrary ranges. Everything turns out okay, though, since $\mathcal{P}$ is the same class whether you define it in terms of function problems or decision problems, although the same

In parallel complexity theory, much like sequential complexity theory, we have a notion of complete problems. A $\mathcal{P}$-complete problem is one that is computable by a single processor in polynomial time and also has the property that if an $\mathcal{NC}$ algorithm exists for that problem, $\mathcal{P} \subseteq \mathcal{NC}$. Therefore, $\mathcal{P}$-complete problems are those problems in $\mathcal{P}$ that are least likely to have $\mathcal{NC}$ algorithms. The text by Greenlaw, Hoover, and Ruzzo [19] is a good introduction to the subject of $\mathcal{P}$-completeness and contains a laundry list of $\mathcal{P}$-complete problems.

Now is a good time to restate, more specifically and with better vocabulary, the fundamental question of this thesis:

> Given a non-trivial, hereditary, polynomial-time testable hypergraph property $\pi$, is there an $\mathcal{NC}$ algorithm for computing a maximal subgraph with property $\pi$ of any hypergraph?

As mentioned earlier, this problem is always in $\mathcal{P}$. The subgraph computed by the polynomial-time sequential algorithm is called the lexicographically first maximal subgraph with property $\pi$.

Miyano [30] has proven the following general result:

equivalence between decision problems and function problems is not known to be true of $\mathcal{NC}$.

**Theorem 1 (Miyano)** *For any non-trivial, hereditary, polynomial-time testable graph property $\pi$, the problem of finding the lexicographically first maximal subgraph with property $\pi$ is $\mathcal{P}$-complete.*

Clearly, Miyano's Theorem is still true if we replace the word "graph" by "hypergraph." The major properties considered in this thesis (independence and acyclicity) are non-trivial, hereditary, and polynomial-time testable, so by Miyano's result, there's little hope in simulating the corresponding sequential algorithms. However, this result does not mean that we can't obtain some sort of parallelization of these "inherently sequential" algorithms. Miyano's Theorem can be interpreted as follows: Given a non-trivial, hereditary, polynomial-time testable hypergraph property $\pi$, fix an ordering of the vertices of a graph. Then there probably is no $\mathcal{NC}$ algorithm that can always find the same maximal subgraph with property $\pi$ that the sequential algorithm would find based on that ordering. But, what about an $\mathcal{NC}$ algorithm that always finds a maximal subgraph that the sequential algorithm would have found based on *some* ordering of the vertices? This is exactly the sort of algorithm that we'll be able to find in many cases. These $\mathcal{NC}$ algorithms, although usually fairly simple to describe, look nothing like their sequential counterparts.

### 1.3.3 The Permutation Algorithm

As an introduction to PRAM algorithms and our algorithmic notation, we present a simple parallel algorithm for finding a maximal independent set in a hypergraph which we call "the permutation algorithm." Chapter 2 and much of Chapter 3 of this thesis focus on this algorithm. Luby [28] was the first to notice that this method of finding a maximal independent set was perfectly suited for parallel applications.

For any hypergraph $H$, the permutation algorithm computes a maximal set $S \subseteq V(H)$ such that $E(H[S]) = \emptyset$. Such a maximal set is called a maximal independent set (MIS). The property of independence ($E(H) = \emptyset$) is hereditary, non-trivial, and polynomial-time testable, so Miyano's Theorem applies to tell us that finding the lexicographically first MIS in a graph is $\mathcal{P}$-complete (and therefore, the problem of finding the lexicographically first MIS in a hypergraph is also $\mathcal{P}$-complete.) We start with an informal description of the permutation algorithm, then present our algorithmic notation using the permutation algorithm as an example.

Given a hypergraph with $n$ vertices, the algorithm first discards any edge $e$ that properly contains another edge $f$, since $f$ not being induced by the independent set chosen implies that $e$ will not be induced. Then, each vertex is independently assigned a rank chosen uniformly at random from the set $\{1, 2, \ldots, n^3\}$. With probability

$1 - \mathcal{O}(\frac{1}{n})$, no two vertices receive the same rank, so when analyzing the algorithm we'll assume that a permutation of the vertices has been chosen uniformly at random. All vertices initially consider themselves "marked."

Each edge now unmarks the maximum ranked vertex it contains. All vertices that remain marked are removed from the vertex set and put in the independent set. In the graph induced by the remaining vertices, any vertex that is left in an edge by itself is removed from the vertex set and discarded, since it can never join the independent set. These steps constitute one iteration of the algorithm, which is repeated on the remaining graph until there are no edges remaining in the hypergraph, at which point all remaining vertices are added to the independent set. Our algorithmic notation for this algorithm is shown in figures 1.1 and 1.2.

In our algorithmic notation, we always express scope by using indentation. The only clue that our algorithm is a parallel algorithm is the construct "**for each** $x \in S$ **pdo**", which means that for each $x \in S$, a unique processor is allocated to perform all steps in the scope of the **for** statement on $x$.

The correctness of the permutation algorithm can be deduced after a few simple observations:

- For a set $I \subseteq V(H)$, $I$ is an MIS in $H$ iff $I$ is an MIS in **CleanUp**(H). This is because (1) the only edges removed from

**Algorithm PermutationAlgorithm(H)**

(1) $I \leftarrow \emptyset$
(2) **while** $E(H) \neq \emptyset$ **do**
(3) $\quad H \leftarrow$ **CleanUp**$(H)$
(4) $\quad$ **for each** $v \in V(H)$ **pdo**
(5) $\quad\quad$ Set $\pi(v)$ to a random number in $[n^3]$
(6) $\quad\quad$ Mark $v$
(7) $\quad$ **for each** $e \in E(H)$ **pdo**
(8) $\quad\quad$ Unmark any $v \in e$ with $\pi(v) = \max\{\pi(v) \mid v \in e\}$
(9) $\quad$ **for each** marked $v$ **pdo**
(10) $\quad\quad I \leftarrow I \cup \{v\}$
(11) $\quad\quad V(H) \leftarrow V(H) \setminus \{v\}$
(12) **return** $I \cup V(H)$

Figure 1.1: The permutation algorithm

**Algorithm CleanUp(H)**

(1) **for each** $\{e, f\} \subseteq E(H)$ **pdo**
(2) $\quad$ **if** $e \subset f$ **then**
(3) $\quad\quad E(H) \leftarrow E(H) \setminus \{f\}$
(4) $\quad$ **else if** $f \subset e$ **then**
(5) $\quad\quad E(H) \leftarrow E(H) \setminus \{e\}$
(6) **for each** $e \in E(H)$ **pdo**
(7) $\quad$ **if** $|e| = 1$ **then**
(8) $\quad\quad V(H) \leftarrow V(H) \setminus e$
(9) **return** $H[V(H)]$

Figure 1.2: The clean-up subroutine for the permutation algorithm

$H$ during clean-up are those that properly contain other edges in $H$, so none of these edges can be induced by an MIS in $H$, and (2) the only vertices removed from $H$ during clean-up are those that are in an edge by themselves, so these vertices can't possibly be added to an MIS in $H$.

- The vertices marked after execution of line (8) of the permutation algorithm form an independent set in the current hypergraph. This is because lines (7) and (8) ensure that no edge is induced by marked vertices. Line (8) ensures that, even if the ranking function is not a permutation, at least one vertex from each edge is unmarked.

- The vertices marked after execution of line (8) of the permutation algorithm, together with the vertices in $I$, form an independent set in the original hypergraph. Assume otherwise, that there is an edge $e = \{v_1, v_2, \ldots, v_k\}$ in the original hypergraph that is induced by $I$ along with the marked vertices after line (8). Since both of these sets are assumed to be independent sets by themselves, part of $e$, say $v_1, v_2, \ldots, v_i$, must be in $I$, and the other part of $e$, $v_{i+1}, \ldots, v_k$, must be marked. But now, $v_{i+1}, \ldots, v_k$ should all be in an edge together in the current hypergraph and therefore would not be induced by the current vertex marking.

- The independent set returned by the permutation algorithm is maximal, since the only vertices not added to the independent set are those that appeared in an edge by themselves at some point.

Each of the for loops in the permutation algorithm and its clean-up subroutine can be implemented by $\mathcal{NC}$ algorithms. The only question remaining is how many times the main while loop in the permutation algorithm executes. We'll answer this question for graphs in Chapter 2 and partially answer the question for hypergraphs in Chapter 3.

## 1.4 Derandomization

The existence of an $\mathcal{RNC}$ algorithm for a problem doesn't necessarily imply the existence of an $\mathcal{NC}$ algorithm for the same problem[2], but there are some nice settings where we can make exactly this conclusion based solely on the analysis of the $\mathcal{RNC}$ algorithm. We call this a derandomization of the randomized algorithm. The simplest way to derandomize an algorithm is to replace the sample space being used by one that's much smaller but has similar

[2]Although by using the trick in section 1.3.1 for reducing the error probability of a randomized algorithm, we can obtain a polynomial-processor, polylogarithmic-time algorithm that fails with probability at most $2^{-100}$, which is a lot smaller than the probability of a hardware failure or even getting struck by lightning while compiling your program.

properties, and then search the smaller sample space exhaustively. Constructions of small sample spaces with good probabilistic properties have been extensively developed and applied to derandomizations in the past decade.

### 1.4.1 $k$-wise Independence

Let $X_1, X_2, \ldots, X_n$ be random variables, all taking values from a common set $S$. $X_1, X_2, \ldots, X_n$ are *k-wise independent* if, for any set of $k$ distinct random variables $\{Y_1, Y_2, \ldots, Y_k\} \subseteq \{X_1, X_2, \ldots, X_n\}$ and any $a_1, a_2, \ldots a_k \in S$

$$\Pr\left[\bigwedge_{i=1}^{k} Y_i = a_i\right] = \prod_{i=1}^{k} \Pr[Y_i = a_i] \tag{1.1}$$

The notion of $k$-wise independence is a restricted form of the familiar "full" independence of random variables, where the probability of an intersection of any number of events is equal to the product of the probabilities of all of the events. The point of making the distinction between full independence and $k$-wise independence is that although we usually assume full independence when we analyze randomized algorithms, we often only *need* $k$-wise independence, for some constant $k$, and sample spaces that are only $k$-wise independent can be much smaller than fully independent

sample spaces. In fact, if $k$ is a constant, $k$-wise independent sample spaces can be created that have size polynomial in $n$, which is small enough so that it's not prohibitively expensive to exhaustively search the sample space for a good point.

The construction of a $k$-wise independent sample space that will be important to us in this thesis is the following:

**Proposition 2** *Let $q$ be a prime power, $F_q$ be the field with $q$ elements, and $F_q[x]^{<k}$ be the set of all polynomials in $F_q[x]$ of degree less than $k$. Take $F_q[x]^{<k}$ to be a probability space with the uniform distribution. Let $X_0, X_1, \ldots, X_{q-1}$ be random variables, where for any point $f \in F_q[x]^{<k}$, $X_i(f) = f(i)$. Then the random variables $X_0, X_1, \ldots, X_{q-1}$ are $k$-wise independent.*

**Proof:** To see that equation 1.1 holds for this sample space, notice that the left-hand side is the probability that any $f \in F_q[x]^{<k}$ evaluates to $k$ particular values on $k$ distinct points. Consider a "generic" polynomial in $F_q[x]^{<k}$ of the form $\sum_{i=0}^{k-1} c_i x^i$, where the $c_i$'s are variables. There are $q^k$ polynomials in $F_q[x]^{<k}$, since we can specify any polynomial of degree less than $k$ by fixing each of the $c_i$'s to be values in $F_q$. Since any polynomial of degree less than $k$ is uniquely determined by its values on $k$ distinct points, the left-hand side is then just $1/q^k$. We now need to show that, for any $X_i$ and fixed $a \in F_q$, $\Pr[X_i = a] = 1/q$, since then the right-hand side of

1.1 will match the left. So, how many of the polynomials in $F_q[x]^{<k}$ evaluate to $a$ on $i$? We can force a polynomial to evaluate to $a$ on $i$ by setting the variables $c_{k-1}, c_{k-2}, \ldots, c_1$ arbitrarily to any values in $F_q$, then solving for $c_0$. Since $F_q$ is a field, there's a unique solution for $c_0$. So, there are $q^{k-1}$ polynomials in $F_q[x]^{<k}$ that evaluate to $a$ on $i$, and hence, for any fixed $a$, the probability $X_i = a$ is $q^{k-1}/q^k = 1/q$.

■

### 1.4.2 Indyk's Permutation Family

Let $P_n$ be the set of all permutations of $[n]$. We create a probability space out of $P_n$ by giving $P_n$ the uniform distribution. Given a set $X \subseteq [n]$ and a distinguished element $a \in X$, a simple computation shows that

$$\Pr_{\pi \in P_n}[\pi(a) = \min(\pi(X))] = \frac{1}{|X|}.$$

$|P_n| = n!$, which is too large to search exhaustively, so for the purposes of derandomization we'd like to be able to construct a family of functions $\mathcal{F}$ that has similar properties to $P_n$ but has size polynomial in $n$.

A set $\mathcal{F}$ of functions all mapping the set $[n]$ to $[n]$ is called $\epsilon-$*minwise independent* if, when $\mathcal{F}$ is given the uniform measure, for any

set $X \subseteq [n]$ and $a \in X$,

$$\frac{1-\epsilon}{|X|} \leq \Pr_{f \in \mathcal{F}}[f(a) = \min\{f(X)\}] \leq \frac{1+\epsilon}{|X|} \tag{1.2}$$

Indyk [21] has proven the following theorem, which we will rely upon to derandomize an algorithm in Chapter 2:

**Theorem 3 (Indyk)** *There exist constants $c, c' > 1$ such that for any $0 < \epsilon < 1$, any $c' \log(1/\epsilon)$-wise independent set of functions satisfies (1.2) for any $|X| \leq \epsilon n/c$.*

If we're not concerned with exactly what the range of these functions is, we can easily adapt Indyk's construction to satisfy inequality (1.2) for any $X \subseteq [n]$ by taking $\mathcal{F}$ to be an $\epsilon$-min-wise independent set of functions on the set $[\lceil cn/\epsilon \rceil]$. To extract a set of $\epsilon$-min-wise independent functions on $[n]$ from this construction, we just ignore the largest $\lceil cn/\epsilon \rceil - n$ elements from the domain.

So, by Proposition 2 and Theorem 3, there exist constants $c, c' > 1$ such that given any $0 < \epsilon < 1$, for any $n > 0$ we can create a set of $\epsilon$-min-wise independent functions $\mathcal{F}$ with domain $[n]$ by taking $\mathcal{F}$ to be the set of all polynomials of degree less than $c' \log(1/\epsilon)$ over the field $F_p$, where $p$ is the first prime greater than $\lceil cn/\epsilon \rceil$. By Bertrand's Postulate (cf. [20], Theorem 418, p. 343), for any $n \geq 2$ there exists a prime $p$ with $n < p \leq 2n$, so the set $\mathcal{F}$ has size polynomial in $n$.

## 1.5 Overview of Results

We conclude the introduction with a statement of the results covered in this thesis, now that we have the language to describe them.

Luby [28] proved in 1985 that a maximal independent set in a graph could be found by a remarkably simple $\mathcal{RNC}$ algorithm, but used a different algorithm and pairwise independence to obtain an $\mathcal{NC}$ algorithm. In Chapter 2, we give a much simpler proof that the simple algorithm is an $\mathcal{RNC}$ algorithm. Our simpler proof has the added advantage that it can easily be derandomized into an $\mathcal{NC}$ algorithm, whereas no derandomization of the permutation algorithm was previously known to exist.

In Chapter 3, we consider the MIS problem on hypergraphs. We first present a sub-linear time deterministic parallel algorithm for finding an MIS in a hypergraph. Then, we consider the permutation algorithm on hypergraphs of dimension at least 6and compare it to a similar algorithm for finding an MIS in a hypergraph of bounded dimension that is known to be an $\mathcal{RNC}$ algorithm, showing that during any particular iteration, the probability a set $X$ is chosen by the permutation algorithm for addition to the independent set is strictly greater than the probability that the known $\mathcal{RNC}$ algorithm adds that same set to its independent set.

In Chapter 4, we give the first known $\mathcal{NC}$ algorithm for finding a maximal acyclic set in a graph, or, equivalently, a maximal forest in a hypergraph. This problem had previously been studied by multiple researchers, with sub-linear time parallel algorithms for special cases presented by Chen and He [10], Chen and Uehara [11], Chen and Zhang [12], and Uehara [40]. Our main subroutine for solving this problem turns out to be, conveniently enough, a variant of an algorithm for finding an MIS in a graph. We also present two applications of this result to approximating $\mathcal{NP}$-complete optimization problems using $\mathcal{NC}$ algorithms.

Finally, we present some open problems in Chapter 5. In theory, Chapters 2, 3, and 4 are independent of each other given the material presented in this introduction, but we feel they are best digested in the order given.

## Chapter 2 - Maximal Independent Sets in Graphs

An independent set in a graph $G$ is a set $I \subseteq V(G)$ such that $G[I]$ has an empty edge set. A maximal independent set (MIS) is an independent set not properly contained in any other independent set. Karp and Widgerson [26] gave the first $\mathcal{RNC}$ algorithm for finding an MIS in a graph and provided a derandomization of their algorithm. Following their proof, both Luby [28] and Alon, Babai, and Itai [1] discovered simpler proofs and found more general settings for the derandomization of randomized parallel algorithms. Goldberg and Spencer later presented more efficient $\mathcal{NC}$ algorithms for the MIS problem in [17] and [18].

In this chapter, we present a quick analysis of Luby's permutation algorithm (figure 1.1) on graphs, which gives a short proof that the MIS problem is in $\mathcal{NC}$. Our proof that the permutation algorithm on graphs is an $\mathcal{RNC}$ algorithm is much simpler than Luby's original proof and has the advantage that it admits a straightforward derandomization, whereas the original $\mathcal{RNC}$ algorithm pre-

sented by Luby was not previously known to have a derandomization.

## 2.1 Analysis of the Permutation Algorithm on Graphs

Since each iteration of the permutation algorithm can be implemented by an $\mathcal{NC}$ algorithm, we wish to bound the number of iterations needed to produce a graph with no edges. Once a graph contains no edges, all remaining vertices can be added to the independent set and the algorithm terminates. We begin by considering the expected number of edges removed by a single iteration.

**Lemma 4** *Let $X$ be the number of edges removed from a graph $G$ by one iteration of the permutation algorithm. Then $\boldsymbol{E}[X] \geq |E(G)|/2$.*

**Proof:** Let $I$ be the set of vertices that remain marked during a single iteration of the permutation algorithm after the execution of step (8). Let $\mathbf{M}(v, S)$ be the event that $v$'s rank is less than the rank of all other vertices in the set $S$. With this notation, $v$ is added to the independent set during an iteration of the permutation algorithm

iff the event $\mathbf{M}(v, \Gamma(v))$ occurs. Now,

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{\{u,v\}\in E(G)} \Pr\{u \in I \cup \Gamma(I) \text{ or } v \in I \cup \Gamma(I)\} \\
&\geq \frac{1}{2} \sum_{\{u,v\}\in E(G)} \Pr\{u \in I \cup \Gamma(I)\} + \Pr\{v \in I \cup \Gamma(I)\} \\
&= \frac{1}{2} \sum_{v\in V(G)} d(v) \Pr\{\ v \in I \cup \Gamma(I)\ \} \\
&\geq \frac{1}{2} \sum_{v\in V(G)} d(v) \Pr\{\ v \in \Gamma(I)\ \} \\
&= \frac{1}{2} \sum_{v\in V(G)} d(v) \Pr\left\{ \bigcup_{w\in\Gamma(v)} \mathbf{M}(w, \Gamma(w)) \right\} && (2.3) \\
&\geq \frac{1}{2} \sum_{v\in V(G)} d(v) \Pr\left\{ \bigcup_{w\in\Gamma(v)} \mathbf{M}\left(w, \Gamma(w) \cup (\Gamma(v) \setminus \{w\})\right) \right\} && (2.4) \\
&= \frac{1}{2} \sum_{v\in V(G)} d(v) \sum_{w\in\Gamma(v)} \Pr\{\ \mathbf{M}(w, \Gamma(w) \cup (\Gamma(v) \setminus \{w\}))\} && (2.5) \\
&= \frac{1}{2} \sum_{v\in V(G)} d(v) \sum_{w\in\Gamma(v)} \frac{1}{d(v) + d(w)} && (2.6) \\
&= \frac{1}{2} \sum_{v\in V(G)} \sum_{w\in\Gamma(v)} \frac{d(v)}{d(v) + d(w)} \quad = \quad |E(G)|/2
\end{aligned}$$

Where (2.4) follows from the fact that $\mathbf{M}(w, \Gamma(w) \cup (\Gamma(v) \setminus \{w\})) \subseteq \mathbf{M}(w, \Gamma(w))$, and (2.5) follows from noticing that all events in the union in (2.4) are pairwise disjoint. ∎

Standard arguments from probability theory can now be used to show that the permutation algorithm on graphs is an $\mathcal{RNC}$ algorithm. Using Lemma 22, we see that if the expected number of

edges removed in any iteration is at least $|E(G)|/2$, the probability that less than $|E(G)|/3$ edges are removed in an iteration is at most $5/6$. Call an iteration "successful" if it removes at least $|E(G)|/3$ edges, and notice that we need $\mathcal{O}(\log n)$ successful iterations until we're left with the empty graph. Corollary 24 then tells us that, for any $c$, we need only $c \log n$ iterations to guarantee $(c/2) \log n$ successful iterations with probability $1 - o(1)$.

## 2.2 The Derandomization

Luby's proofs that both the permutation algorithm on graphs and another algorithm for finding an MIS in a graph are $\mathcal{RNC}$ algorithms both follow the same computation as our proof of Lemma 4 up until (2.3), when the probability of a union of events is encountered. The Bonferroni inequalities, one of which states that

$$\Pr\left[\bigcup_i E_i\right] \geq \sum_i \Pr[E_i] - \sum_{i \neq j} \Pr[E_i \cap E_j],$$

are then used to lower bound this probability. To derandomize the result, one then needs a probability space that has size bounded by a polynomial in $n$ and gives some sort of guarantee about what $\Pr[E_i \cap E_j]$ is, for arbitrary events $E_i$ and $E_j$, $i \neq j$. For example, in Luby's other algorithm for finding an MIS in a graph, each event $E_i$

simply indicates that a particular weighted coin flip came up heads. These events are simple enough that a small pairwise independent probability space can be used, where $\Pr[E_i \cap E_j] = \Pr[E_i] \cdot \Pr[E_j]$ for each pair of events. However, in the permutation algorithm, the events $E_i$ indicate that a particular vertex is a local minimum under the permutation, and no construction of a small probability space with good bounds on $\Pr[E_i \cap E_j]$ is known. Our simpler proof of Lemma 4 that avoids using the Bonferonni inequalities is therefore critical to obtaining a derandomization since we avoid terms that involve pairwise intersections of events altogether.

To derandomize the permutation algorithm, we use Indyk's construction of a family of $\epsilon$-min-wise independent functions described in Section 1.4.2. The only change needed in our proof of Lemma 4 is that equation (2.6) is replaced by a lower bound of

$$\frac{1}{2}\sum_{v \in V} d(v) \sum_{w \in \Gamma(v)} \frac{1-\epsilon}{d(v)+d(w)}$$

which tells us that the expected number of edges removed by a single iteration of the permuatation algorithm using $\epsilon$-min-wise independent functions to rank vertices is at least $(1-\epsilon)|E(G)|/2$. This set of $\epsilon$-min-wise independent functions can now be searched exhaustively in parallel for a function that removes at least $(1-\epsilon)|E(G)|/2$

edges, the only downside being that there is a polynomial blow-up in the number of processors needed to execute the algorithm.

## 2.3 Using the MIS Algorithm to Find a Maximal Set Packing

Karp and Widgerson [26] noticed that an $\mathcal{NC}$ algorithm for finding an MIS in a graph could be used to implement an $\mathcal{NC}$ algorithm for what they called the Maximal Set Packing problem. Their version of the Maximal Set Packing problem takes as input a set $S$ of $m$ subsets of a ground set of size $n$, and outputs a maximal set $P \subseteq S$ such that for any two distinct $x, y \in P, x \cap y = \emptyset$. We define a generalized version of this problem and show how it can be reduced to finding an MIS in a graph. Our version will be essential to our solution of the Maximal Acyclic Set problem in Chapter 4.

For our purposes, we'll need to add an additional parameter to the Maximal Set Packing function. Our function takes a hypergraph $H$ and a set $X \subseteq V(H)$ as input parameters, and returns a maximal set $S \subseteq E(H)$ such that for any two distinct edges $m, n \in S$, $m \cap n \subseteq X$. If $X = \emptyset$, this reduces to the same definition used by Karp and Widgerson.

Figure 2.3 gives a high-level description of how to use an $\mathcal{NC}$ algorithm for finding an MIS in a graph to find a Maximal Set Packing in a hypergraph. It follows directly from the definition of an MIS that the algorithm in figure 2.3 computes a maximal set packing in

**Algorithm MaximalSetPacking(H,X)**

(1) $V(G) \leftarrow E(H)$
(2) $E(G) \leftarrow \emptyset$
(3) **for** each pair $\{m, n\} \subseteq V(G)$ **pdo**
(4) **if** $m \cap n \subseteq X$ **then**
(5) $E(G) \leftarrow E(G) \cup \{m, n\}$
(6) **return MIS**$(G)$

Figure 2.3: An algorithm that finds a maximal set packing in a hypergraph

a hypergraph, and the graph $G$ created sent to the MIS subroutine has $|E(H)|$ vertices and $\sum_{v \in V(H)} \binom{d(v)}{2}$ edges. The reduction used takes constant time using $\sum_{v \in V(H)} \binom{d(v)}{2}$ processors. We'll return to this algorithm when we use it as a subroutine in Chapter 4.

## Chapter 3 - Maximal Independent Sets in Hypergraphs

With the question of the parallel complexity of finding an MIS in a graph resolved, we now turn to the generalization of the MIS problem to hypergraphs. Recall that an MIS $M$ in a hypergraph $H$ is a maximal set such that no edge $e \in E(H)$ is contained in $M$. Finding an MIS in a hypergraph is a much more difficult problem than finding an MIS in a graph, and despite a great deal of effort by many researchers, not much is known about the parallel complexity of computing an MIS in a hypergraph.

Dahlhaus, Karpinski, and Kelsen [15] extended an $\mathcal{NC}$ algorithm of Goldberg and Spencer [17] for finding an MIS in graphs to an $\mathcal{NC}$ algorithm for finding an MIS in hypergraphs of dimension 3. Beame and Luby [6] presented two algorithms that they conjectured were $\mathcal{RNC}$ algorithms for finding an MIS in a hypergraph, and Kelsen [27] completed their analysis for one of the algorithms under the assumption that the dimension of the hypergraph was $O(1)$. Łuczak and Szymańska [29] proved that the problem of finding an MIS in a linear hypergraph is in $\mathcal{RNC}$, and Szymańska [37]

later derandomized their result to obtain an $\mathcal{NC}$ algorithm. Recently, Shachnai and Srinivasan[36] showed that the permutation algorithm on hypergraphs always finds a "large" MIS in $k$-uniform hypergraphs (hypergraphs where all edges are of size $k$) and derandomized the algorithm for two special cases of hypergraphs with degree constraints.

In section 3.1, we give the first non-trivial upper bound on the deterministic parallel complexity of finding an MIS in a hypergraph. In section 3.2, we revisit the permutation algorithm used for finding an MIS in a graph and compare it to the algorithm proven to be an $\mathcal{RNC}$ algorithm for finding an MIS in hypergraphs of bounded dimension by Beame, Kelsen, and Luby. Both algorithms are identical except for the procedure used to mark vertices to be added to the independent set, and we show that the probability that any set of vertices is marked by the permutation algorithm is strictly greater than the probability that the $\mathcal{RNC}$ algorithm marks that same set of vertices.

## 3.1 A Sub-linear Time Parallel Algorithm

There are two known non-trivial upper bounds for the problem of finding an MIS in a hypergraph on a PRAM. Karp, Widgerson, and Upfal [25] gave a randomized parallel algorithm that finds a

maximal independent set in a hypergraph in time $\mathcal{O}(\sqrt{n} \cdot (\log n + \log m))$ using $mn$ processors, and Kelsen [27] was able to derandomize his $\mathcal{RNC}$ algorithm for finding an MIS in a hypergraph of dimension $d$ into a complicated deterministic algorithm that runs in time $(\log n)^{\mathcal{O}(1)} \cdot o(n^{(2d)^{2d}\epsilon})$ using $n^{\mathcal{O}(1/\epsilon)}$ processors, for any $\epsilon \geq 2^{d+1} \cdot (\log\log n / \log n)$. Neither one of these algorithms gives a deterministic sublinear time bound for finding an MIS in a hypergraph: the upper bound of Karp, Widgerson, and Upfal only applies to randomized algorithms and the upper bound of Kelsen only applies to hypergraphs of bounded dimension.

In this section, we give the first non-trivial upper bound on the deterministic time complexity of finding an MIS in a hypergraph on a PRAM. We present a simple deterministic algorithm for finding an MIS in any hypergraph that runs in time $\mathcal{O}(\log^k(m+n) \cdot \max\{m^{1-\epsilon+\delta}, n^{2(\epsilon+\delta)}, n^{1-\delta}\})$ for some fixed $k > 0$ and any $\epsilon, \delta > 0$. As long as $0 < \delta < \epsilon \leq 1/4$, this algorithm runs in sub-linear time in the size of the input.

The central idea in our upper bound is the *elimination* of an edge in a hypergraph. Elimination of an edge $e = \{v_1, v_2, \ldots, v_k\}$ in the hypergraph $H$ involves adding all but one of $e$'s vertices (say, $v_1, v_2, \ldots, v_{k-1}$) to the independent set we're currently maintaining and replacing $H$ by **Cleanup**$(H')$, where **CleanUp** is the subroutine

defined in figure 1.2 and $H'$ is defined by

$$H' = (V(H) \setminus e, E(H) \setminus \{e \in E(H) \mid v_k \in e\}).$$

The justification for this step is that $v_k$ can never be added to the independent set at any later step in the algorithm, since it would then cause $e$ to be induced by the independent set, but since $v_k$ will never be added, any edge that contains $v_k$ cannot possibly be induced by any set of vertices chosen by the algorithm. We will refer to the vertex $v_k$ that is not added to the independent set as the *discarded vertex* of an eliminated edge.

We can also eliminate several edges at once, but here we must be a little more careful - for example, if we chose to eliminate $\{a, b\}$ and $\{c, d\}$ by adding $a$ and $c$ to the independent set, we'd need to know that $\{a, c\} \notin E(H)$. In our algorithm below, we'll avoid this difficulty by eliminating a set of edges $S$ such that no two edges in $S$ meet each other or both intersect a common third edge. Such a set of edges can clearly be safely eliminated.

Our algorithm is described in figure 3.4. The subroutine **MIS** used by the algorithm is any one of the $\mathcal{NC}$ algorithms for finding an MIS in a graph. The correctness of our algorithm can be verified by noticing that edges are only added to the independent set by being eliminated, and the only time a set of edges is eliminated

**Algorithm MISH(H)**

(1) **while** $\exists e \in E(H)$ such that $|e| \geq n^{\delta}$
(2) eliminate $e$ from $H$
(3) **while** $\exists v \in V(H)$ such that $d(v) \geq n^{\epsilon}$
(4) eliminate exactly one $e \ni v$ from $H$
(5) **while** $E(H) \neq \emptyset$
(6) $V(G') \leftarrow E(H)$
(7) **for each** pair $\{u, v\} \subseteq V(G')$ **pdo**
(8) **for each** $w \in V(G')$ **pdo**
(9) **if** $u \cap w \neq \emptyset$ and $v \cap w \neq \emptyset$ **then**
(10) $E(G') \leftarrow E(G') \cup \{u, v\}$
(11) $M \leftarrow MIS(G')$
(12) **for each** $e \in M$ **pdo**
(13) eliminate $e$ from $H$

Figure 3.4: An algorithm that finds an MIS in a hypergraph

is when the set $M$ is eliminated, but $M$ is, by construction, a set such that no two edges in $M$ intersect each other or a common third edge.

As for the time complexity, we'll analyze each while loop separately. The first while loop removes $n^\delta$ vertices from $V(H)$ each time it's executed, so it can execute at most $n^{1-\delta}$ times before $V(H) = \emptyset$. Each iteration can be accomplished by an $\mathcal{NC}$ algorithm.

The second while loop removes a vertex from at least $n^\epsilon$ edges each time it executes, so it removes at least $n^\epsilon$ from the sum $\sum_{e \in E(H)} |e|$, which is at most $m \cdot n^\delta$. This can be done at most $m \cdot n^{\delta-\epsilon} \leq m^{1+\delta-\epsilon}$ times (Here we assume that $m \geq n$, but if this isn't true, we get an even better bound of at most $n^{1+\delta-\epsilon}$ iterations.) Each iteration can again be accomplished by an $\mathcal{NC}$ algorithm.

The final while loop constructs a graph with a vertex for each edge in the original hypergraph and graph edges between any two edges in the original hypergraph that either meet each other or both meet a common third edge. An **MIS** subroutine is then called on this graph. Since each edge in the hypergraph at this point meets at most $n^{\delta+\epsilon}$ other edges, the maximum degree in this graph is $n^{2(\delta+\epsilon)}$. Any MIS in a graph with $n$ vertices and maximum degree $\Delta$ must have size at least $n/(\Delta + 1)$, since each vertex in the MIS excludes at most $\Delta$ other vertices. So, the call to the **MIS** subroutine here returns a set of at least $\Omega(m/n^{2(\delta+\epsilon)})$ edges that

can safely be eliminated concurrently. We can eliminate this many edges $\mathcal{O}(\log m \cdot n^{2(\delta+\epsilon)})$ times before there are no edges left.

Each iteration of the while loop starting at line (5) can be implemented by an $\mathcal{NC}$ algorithm, since both the double for loop starting on line (7) and the for loop starting on line (12) can be implemented in constant time if $\mathcal{O}(m^3)$ processors are available, and we know already that **MIS** is an $\mathcal{NC}$ algorithm.

Putting everything together, we see that the bodies of each of the while loops are $\mathcal{NC}$ algorithms and the while loops execute $\mathcal{O}(n^{1-\delta})$, $\mathcal{O}(m^{1-\epsilon+\delta})$, and $\mathcal{O}(\log m \cdot n^{2(\epsilon+\delta)})$ times, respectively. This gives us the following result:

**Theorem 5** *An MIS in a hypergraph can be computed on a PRAM in time* $\mathcal{O}(\log^k(m+n)\max\{m^{1-\epsilon+\delta}, n^{2(\epsilon+\delta)}, n^{1-\delta}\})$ *using* $\mathcal{O}(m^3)$ *processors, for some fixed* $k > 0$ *and any* $\epsilon, \delta > 0$.

### 3.2 The Permutation Algorithm Revisited

Using the structure of the permutation algorithm from section 1.3.3 as a general outline, one can devise other parallel algorithms for finding an MIS in a hypergraph by simply replacing lines (4) through (8) in the permutation algorithm (figure 1.1) with any other

procedure that marks an independent set in a hypergraph[3]. Although they may not perform as well as the permutation algorithm, other selection procedures can be designed so that they are easier to analyze - particularly if all vertices are marked using independent, identically distributed random variables..

In [6], Beame and Luby describe an alternative to the permutation algorithm called the *maximum degree algorithm* that uses a modified marking procedure and analyze the maximum degree algorithm on hypergraphs of bounded dimension. Their incomplete analysis of the maximum degree algorithm was completed by Kelsen [27], who showed that the algorithm is indeed an $\mathcal{RNC}$ algorithm for finding an MIS in hypergraphs of bounded dimension. To describe their algorithm, we will need to introduce some notation.

For a set $S \subseteq E(H)$ and integer $i$, define $d_i(S)$, the *$i$th degree* of $S$, as

$$d_i(S) = |\{e \in E(H) \mid S \cap e \neq \emptyset, |e \setminus S| = i\}|.$$

Under the same conditions, we define $\Delta_i(S)$, the *normalized $i$th degree* of $S$, to be

$$\Delta_i(S) = d_i(S)^{\frac{1}{i}}.$$

[3]Provided, of course, that the selection process either always chooses at least one vertex, or if it is randomized, always chooses at least one vertex with positive probability.

Furthermore, define $\Delta(S)$ as

$$\Delta(S) = \max_i \{\Delta_i(S)\}.$$

Finally, define $\Delta_H$ on a hypergraph of dimension $d$ as

$$\Delta_H = \max_{\substack{S \subseteq V(H) \\ 1 \leq |S| \leq d}} \{\Delta(S)\}.$$

Let $H$ be a hypergraph of dimension $d$. The selection step in the maximum degree algorithm can be described simply as follows:

> Each vertex independently marks itself with probability $1/(2^{d+1}\Delta_H)$. If any edge has all of its vertices marked, it unmarks all of those vertices.

Since the vertices that remain marked after this step form an independent set, this is a valid marking procedure. Notice, however, that $\Delta_H$ can be close to $n$ in dense hypergraphs, so this marking procedure may execute several times before a single vertex is marked, whereas the permutation algorithm always marks at least one vertex. Beame and Luby go on to prove that the probability that a set of vertices gets added to the independent set, given that they were marked, is at most $1/2$. This gives us the following lemma:

**Lemma 6 (Beame and Luby)** *Let $X$ be any set of vertices not containing any edge of $H$. Then the probability that the maximum degree algorithm chooses $X$ during its selection step is at least $1/(2 \cdot (2^{d+1}\Delta_H)^{|X|})$.*

In this section, for any set $X \subseteq V(H)$, we wish to compare the probability that the permutation algorithm selects $X$ during a selection step with the probability that the maximum degree algorithm selects $X$ during a selection step. Let $S_v$ be the event that $v$ stays marked after the selection step in the permutation algorithm, that is, $v$ is not the minimum of any of the edges it's contained in. We will prove the following result:

**Lemma 7** *Let $X$ be any set of vertices not containing any edge of $H$. Then*

$$\Pr\left[\bigcap_{v \in X} S_v\right] > \frac{1}{d(2de\Delta(X))^{|X|}}$$

For $d \geq 6$ and *any* set of vertices $X$ that doesn't induce an edge in the current hypergraph, the probability the permutation algorithm will select the set $X$ is strictly greater than the probability that the maximum degree algorithm would have selected $X$. For $d < 6$, the lower bound from lemma 7 only falls below the lower bound in lemma 6 by a small constant factor. It would seem that this, along with the fact that the maximum degree algorithm is an

$\mathcal{RNC}$ algorithm would be enough to imply that the permutation algorithm is an $\mathcal{RNC}$ algorithm for finding an MIS in a hypergraph. However, Kelsen's analysis of the maximum degree algorithm consists of the application of a highly technical upper tail bound on the sum of small products of independent, identically distributed random variables, and this bound does not necessarily carry over to the probability space of all permutations. Thus, we consider lemma 7 very good evidence that the permutation algorithm is an $\mathcal{RNC}$ algorithm for finding an MIS in a hypergraph of bounded dimension although the true complexity of the permutation algorithm on such hypergraphs remains open.

Call a vertex $v$ $\epsilon$-large under a permutation $\pi$ if

$$|\{w \in V(H) \mid \pi(w) < \pi(v)\}| \geq (1-\epsilon)n.$$

Let $L_{v,\epsilon}$ be the event that $v$ is $\epsilon$-large. We will need the following technical lemma to prove lemma 7:

**Lemma 8** *Let $e \in E(H)$, $Y \subseteq e$, $X \subseteq V(H) \setminus e$, and $v \in e$. Then*

$$\Pr\left[v = \min(e) \,\middle|\, \bigcap_{y \in Y} L_{y,\epsilon} \cap \bigcap_{x \in X} L_{x,\epsilon}\right] \leq \Pr\left[v = \min(e) \,\middle|\, \bigcap_{y \in Y} L_{y,\epsilon}\right]$$

**Proof:** First,

$$\begin{aligned}
\Pr\left[v = \min(e) \,\middle|\, \bigcap_{y\in Y} L_{y,\epsilon}\right] &= \frac{\binom{\epsilon n}{|e|}(|e|-1)!(n-|e|)!}{n!} \cdot \frac{\binom{n}{|Y|}}{\binom{\epsilon n}{|Y|}} \\
&= \frac{1}{|e|} \cdot \frac{(\epsilon n)_{(|e|)}}{n_{(|e|)}} \cdot \frac{n_{(|Y|)}}{(\epsilon n)_{(|Y|)}} \\
&= \frac{1}{|e|} \cdot \frac{(\epsilon n - |Y|)_{(|e|-|Y|)}}{(n-|Y|)_{(|e|-|Y|)}}. \qquad (3.7)
\end{aligned}$$

Also,

$$\begin{aligned}
&\Pr\left[v = \min(e) \,\middle|\, \bigcap_{y\in Y} L_{y,\epsilon} \cap \bigcap_{x\in X} L_{x,\epsilon}\right] \\
&= \frac{\binom{\epsilon n}{|e|}(|e|-1)!\binom{\epsilon n-|e|}{|X|}|X|!(n-|X|-|e|)!}{n!} \cdot \frac{\binom{n}{|X|+|Y|}}{\binom{\epsilon n}{|X|+|Y|}} \\
&= \frac{1}{|e|} \cdot \frac{(\epsilon n)_{(|e|)}(\epsilon n - |e|)_{(|X|)}}{n_{(|X|+|e|)}} \cdot \frac{n_{(|X|+|Y|)}}{(\epsilon n)_{(|X|+|Y|)}} \\
&= \frac{1}{|e|} \cdot \frac{(\epsilon n)_{(|X|+|e|)}}{n_{(|X|+|e|)}} \cdot \frac{n_{(|X|+|Y|)}}{(\epsilon n)_{(|X|+|Y|)}} \\
&= \frac{1}{|e|} \cdot \frac{(\epsilon n - |X| - |Y|)_{(|e|-|Y|)}}{(n-|X|-|Y|)_{(|e|-|Y|)}} \qquad (3.8)
\end{aligned}$$

So, to prove the lemma, we need $(3.8) \leq (3.7)$, which is true iff for any $i \geq j$, $\epsilon \in (0,1]$, the inequality $(\epsilon n - i)/(n-i) \leq (\epsilon n - j)/(n-j)$ holds. A little manipulation shows that this is indeed the case.

■

We're now in position to prove lemma 7:

**Proof:**[of lemma 7] To get a good lower bound on $\Pr[\bigcap_{v\in X} S_v]$, we first get an upper bound on the complement of this event, conditioned on all of the vertices in $X$ being $\epsilon$-large. The idea here is that, for $\epsilon$ sufficiently small, $\epsilon$-large vertices are very unlikely to be the minimum of any of the edges they are in, and therefore unlikely to get unmarked.

$$\begin{aligned}\Pr\left[\bigcup_{v\in X}\overline{S_v}\;\middle|\;\bigcap_{v\in X}L_{v,\epsilon}\right] &= \Pr\left[\bigcup_{v\in X}\bigcup_{e\ni v} v=\min(e)\;\middle|\;\bigcap_{v\in X}L_{v,\epsilon}\right]\\ &\leq \sum_{v\in X}\sum_{e\ni v}\Pr\left[v=\min(e)\;\middle|\;\bigcap_{v\in X}L_{v,\epsilon}\right] \quad (3.9)\end{aligned}$$

Since there may be vertices in $X$ that are not in $e$, we use lemma 8 to upper bound (3.9) with

$$\begin{aligned}
\sum_{v\in X}\sum_{e\ni v}\Pr\left[v=\min(e)\;\middle|\;\bigcap_{v\in X\cap e}L_{v,\epsilon}\right] &= \sum_{v\in X}\sum_{e\ni v}\frac{\binom{\epsilon n}{|e|}(|e|-1)!(n-|e|)!}{n!}\cdot\frac{\binom{n}{|X\cap e|}}{\binom{\epsilon n}{|X\cap e|}}\\
&= \sum_{v\in X}\sum_{e\ni v}\frac{1}{|e|}\cdot\frac{(\epsilon n)_{(|e|)}}{n_{(|e|)}}\cdot\frac{n_{(|X\cap e|)}}{(\epsilon n)_{(|X\cap e|)}}\\
&\leq \sum_{v\in X}\sum_{e\ni v}\frac{1}{|e|}(\epsilon)^{|e\setminus X|}\\
&= \sum_{\substack{e\in E(H)\\ e\cap X\neq\emptyset}}\frac{|e\cap X|}{|e|}(\epsilon)^{|e\setminus X|}\\
&= \sum_{j=2}^{d}\sum_{i=1}^{j-1}\sum_{\substack{e\in E(H)\\ |e|=j,|e\cap X|=i}}\frac{i}{j}\,(\epsilon)^{j-i}\\
&= \sum_{j=2}^{d}\sum_{i=1}^{j-1}\frac{i}{j}\,d_{j-i}(X)\,(\epsilon)^{j-i}
\end{aligned}$$

Where the index in the second sum of the penultimate term above only runs up to $j-1$ because our set $X$ doesn't contain any edge $e\in E(H)$. Setting $\epsilon = 1/(2d\Delta(X))$ at this point will yield a decent bound:

$$\begin{aligned}
\sum_{j=2}^{d}\sum_{i=1}^{j-1}\frac{i}{j}\,d_{j-i}(X)\,\left(\frac{1}{d\Delta(X)}\right)^{j-i} &= \sum_{j=2}^{d}\sum_{i=1}^{j-1}\frac{i}{j}\left(\frac{1}{2d}\right)^{j-i} \\
&\leq \frac{1}{d}\sum_{j=2}^{d}\frac{1}{j\cdot 2^j}\sum_{i=1}^{j-1} i\cdot 2^i \\
&= \frac{1}{d}\sum_{j=2}^{d}\left(1-\frac{2}{d}+\frac{1}{d2^{d-1}}\right) \\
&\leq \left(\frac{d-1}{d}\right)^2
\end{aligned}$$

For $d \geq 2$, using the identity $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1}+2$.

So, $\Pr\left[\bigcup_{v\in X}\overline{S_v} \;\middle|\; \bigcap_{v\in X} L_{v,1/(2\Delta(X))}\right] \leq (d-1)^2/d^2$ and therefore the complement of this event occurs with probability strictly greater than $1/d$. Computation of the probability of the event we're conditioning on shows

$$\Pr\left[\bigcap_{v\in X} L_{v,1/(2\Delta(X))}\right] = \frac{\binom{n/(2\Delta(X))}{|X|}}{\binom{n}{|X|}} \geq \left(\frac{n}{2\Delta(X)|X|}\cdot\frac{|X|}{ne}\right)^{|X|} = \left(\frac{1}{2e\Delta(X)}\right)^{|X|}$$

using the well-known inequalities $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$. We now complete the proof by noticing that

$$\begin{aligned}
\Pr\left[\bigcap_{v\in X} S_v\right] &\geq \Pr\left[\bigcap_{v\in X} S_v \;\cap\; \bigcap_{v\in X} L_{v,1/(2\Delta(X))}\right] \\
&= \Pr\left[\bigcap_{v\in X} S_v \;\middle|\; \bigcap_{v\in X} L_{v,1/(2\Delta(X))}\right] \cdot \Pr\left[\bigcap_{v\in X} L_{v,1/(2\Delta(X)}\right] \\
&> \frac{1}{d(2de\Delta(X))^{|X|}}
\end{aligned}$$

■

## Chapter 4 - Maximal Acyclic Sets in Graphs

Let $G = (V, E)$ be an undirected graph with $m$ edges and $n$ vertices. A cycle in $G$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ where $k > 2$, $\{v_i, v_{i+1}\} \in E(G)$ for $i$ in the range $1 \leq i \leq k-1$, and $\{v_k, v_1\} \in E(G)$. A graph is acyclic if it contains no cycles. An acyclic set in the graph $G$ is a set $A \subseteq V(G)$ such that the induced graph on $A$ is acyclic, and, of course, a maximal acyclic set (MAS) is an acyclic set that isn't properly contained in another acyclic set.

Acyclicity is hereditary, non-trivial, and polynomial-time testable, so by Miyano's Theorem finding the lexicographically first MAS is $\mathcal{P}$-complete. The sequential algorithm for finding the lexicographically first MAS can be implemented to run in time $\mathcal{O}(m \cdot \alpha(m, n))$, where $\alpha$ is a slow-growing inverse of the Ackerman function, using the classic union-find data structure with path compression and union-by-rank (See [38], Ch. 2, for a good presentation of this data structure.)

An MAS in a graph is similar to a spanning forest in a graph, the former being a maximal vertex-induced forest and the latter a

maximal edge-induced forest. Finding a spanning forest in a graph has been studied extensively in the context of parallel algorithms and many efficient $\mathcal{NC}$ algorithms for finding spanning forests are known (see, e.g. [13], [22], [34], [35]), but finding an MAS even in a graph of bounded degree was not previously known to be in $\mathcal{NC}$. Following Pearson and Vazirani's [33] work on finding maximal bipartite subgraphs in parallel, Chen and He [10] considered the MAS problem and gave several EREW PRAM algorithms for finding an MAS in special cases, including an $\mathcal{NC}$ algorithm for planar graphs and a $\mathcal{O}(\sqrt{n}\log^3 n)$ time, $\mathcal{O}(n^2)$ processor algorithm for graphs with bounded degree. Chen and Zhang [12] later extended the results from [10] to obtain an $\mathcal{NC}$ algorithm for finding an MAS in $K_{3,3}$-free graphs. Chen and Uehara [11] gave $\mathcal{NC}$ algorithms for finding a maximal set of vertices that induces an acyclic graph with maximum degree 2. Finally, Uehara [40] has shown that finding an MAS in a $k$-chordal graph (every cycle of length greater than $k$ has a chord) is in $\mathcal{NC}$ for $k = 3$ and $\mathcal{RNC}$ for fixed $k > 3$.

In this chapter, we present the first $\mathcal{NC}$ algorithm for finding an MAS in any graph, running in time $\mathcal{O}(\log^4 n)$ with $\mathcal{O}((m^2 + n)/\log n)$ processors on an EREW PRAM. Our algorithm uses a reduction of the problem of finding an MAS in a graph to the problem of finding a maximal forest in a hypergraph. Our $\mathcal{NC}$ algorithm for finding a maximal forest in a hypergraph may be of independent interest,

since the problem has not been previously addressed in the literature on parallel algorithms. We conclude the chapter with some applications of the MAS subroutine to approximating $\mathcal{NP}$-complete problems with $\mathcal{NC}$ algorithms.

## 4.1 Preliminaries

Although the problem of finding a spanning forest in a graph has been widely studied in the context of $\mathcal{NC}$ algorithms, the generalization of this problem to hypergraphs has not. It turns out that finding a maximal set of edges in a hypergraph that induces an acyclic hypergraph is equivalent to finding an MAS in a graph; we prove this equivalence in Section 4.3. For now, we focus on solving the problem of finding an edge-induced, acyclic sub-hypergraph of any hypergraph. We first need to extend the definition of a cycle so that it applies to hypergraphs.

A cycle in a hypergraph is a sequence of distinct vertices $v_0, v_1, \ldots, v_{k-1}$ and a sequence of distinct edges $e_0, e_1, \ldots, e_{k-1}$ such that $k > 1$ and $v_i \in e_i \cap e_{(i+1) \bmod k}$ for all $i$, $0 \leq i < k$. In the sequel, addition on cycle indices is implicitly done mod $k$, where $k$ is the number of vertices in the cycle.

A forest is a subset of the edge set of a hypergraph that induces no cycles. A maximal forest of a hypergraph $H$ is a forest that is

not contained in any other forest of $H$. In a graph with no isolated vertices, the fact that a forest is maximal is enough to imply that the forest covers the vertex set. The same is not true for hypergraphs, however. Take, for example, $V(H) = \{v_1, v_2, \ldots, v_n\}$ and $E(H) = \{\{v_1, v_2, v_k\} \ : \ 3 \leq k \leq n\}$, so that any maximal forest of $H$ consists of a single edge, and therefore covers at most 3 of the $n$ vertices. The problem of deciding whether or not a hypergraph containing only edges of size $k \geq 3$ even has a spanning forest is NP-complete [39].

## 4.2 An $\mathcal{NC}$ Algorithm for Finding a Maximal Forest in a Hypergraph

Our algorithm for finding a maximal forest in a hypergraph is similar at a high level to a well known parallelization of Borůvka's algorithm for finding a minimum spanning tree in a graph [5]. During each step of the algorithm, a forest is found in the hypergraph. All of the edges in the forest are then contracted and the resulting hypergraph has a new vertex in place of each connected component induced by the forest in the original hypergraph. Any edge from the original hypergraph that met the vertex set of a single connected component of the forest in more than two vertices is removed before contraction, so that the only edges remaining in the

hypergraph after contraction are those that met each component in at most one vertex. Throughout the description of the algorithm and its analysis, we assume that all edges in the hypergraph contain at least two vertices, since any edge containing a single vertex cannot be part of a cycle and can therefore be added to the forest in a preprocessing step.

### 4.2.1 Contraction of Forests

The contraction procedure can be described formally as follows: Let $H$ be a hypergraph, and let $F$ be a forest in $H$. Partition $F$ into $F_1, F_2, \ldots, F_k$ using an equivalence relation between edges defined as $e \sim f$ iff $e$ and $f$ are in the same connected component in $H[F]$. The hypergraph $H' = \textbf{Contract}(H, F)$ has vertex set

$$V(H') = \left( V(H) \setminus \bigcup_{f \in F} f \right) \cup \{v_1, v_2, \ldots, v_k\}$$

Where $v_1, v_2, \ldots, v_k$ are all new labels not appearing in $V(H)$. Define a function $\rho : V(H) \to V(H')$ as

$$\rho(v) = \begin{cases} v_i, & \text{if } v \text{ is covered by } F_i, \text{ for some } i \\ v, & \text{otherwise} \end{cases}$$

Each edge $e \in E(H)$ naturally corresponds to a subset of $V(H')$, which is $\{\rho(v) \ : \ v \in e\}$. Set $\hat{\rho}(e) = \{\rho(v) \ : \ v \in e\}$ for all $e \in E(H)$, and now the edge set $E(H')$ can be defined as

$$E(H') = \{\hat{\rho}(e) \ : \ e \in E(H), \ |e| = |\hat{\rho}(e)| \ \}$$

Contraction of forests gives a convenient way of computing a maximal forest iteratively, with only a little awkwardness in the specification of the algorithm. We would like to repeatedly find a forest in the hypergraph, contract it, and add it to the set of edges that will eventually be a maximal forest. But, since the vertex set is changing during this process, the edges we wish to add during an iteration may not be edges in the hypergraph we started with. What's more, an edge in a contracted hypergraph may correspond to more than one edge in the original hypergraph, in which case, at most one of the edges in the original hypergraph corresponding to a single edge in the current hypergraph may be added to the forest. To avoid this complication when stating the algorithm, we assume the use of a function **Resolve** that maps edges in the current hypergraph to edges in the original hypergraph, choosing a unique edge in the original hypergraph arbitrarily when there are multiple candidates.

To formalize the fact that repeated contraction using **Resolve** to obtain edges from the original hypergraph yields forest in the original hypergraph is formalized by the next two lemmas. We first prove a technical lemma, using the function $\hat{\rho}$ associated with any hypergraph contraction.

**Lemma 9** *Let $H$ be a hypergraph. Let $F$ be a forest in $H$, and let $H' =$ **Contract**$(H, F)$. For any $F_1 \subseteq E(H)$ and $F_2 \subseteq E(H')$ such that $\hat{\rho}$ restricted to $F_1$ is a bijection between $F_1$ and $F_2$, then $F_2$ is a forest in $H'$ iff $F \cup F_1$ is a forest in $H$.*

**Proof:** Assume that $F \cup F_1$ induces a cycle in $H$. Of all of the cycles induced by $F \cup F_1$, choose a cycle $C$ with the minimum number of edges from $F_1$. Let $e_1, e_2, \ldots, e_k$ be the sequence of edges in the cycle, and choose distinct $v_i \in e_i \cap e_{i+1}$ for all $i$. By definition of $\hat{\rho}$, for each $e \in F$, $\hat{\rho}(e) = \{v\}$ for some $v \in V(H') \setminus V(H)$, and it also follows from this definition that, for every set of edges $X$ all belonging to the same connected component in $H[F]$, $\bigcup_{e \in X} \hat{\rho}(e) = \{v\}$ for some $v \in V(H') \setminus V(H)$. Each edge $e \in F_1$, however, gets mapped to a unique edge in $F_2$ of size $|e|$ by $\hat{\rho}$, since $\hat{\rho}$ is a bijection between the two sets. Since $F$ is a forest in $H$, at least one of the edges in $C$ must be a member of $F_1$. If only one edge $e_i$ from $C$ is in $F_1$, then since two other edges in the cycle meet $e_i$ in two distinct vertices and all other edges in the cycle map to the same $v \in V(H') \setminus V(H)$,

$|\hat{\rho}(e)| < |e|$, contradicting the fact that $\hat{\rho}(e) \in E(H')$. So, there are at least two edges from $F_1$ in the cycle.

Call these edges from $F_1$ $f_1, f_2, \ldots, f_l$, labeled in order of their appearance in the original cycle $C$. We claim that the sequence of distinct edges $\hat{\rho}(f_1), \hat{\rho}(f_2), \ldots, \hat{\rho}(f_l)$ forms a cycle in $H'$, which we'll show by finding a sequence of distinct vertices $w_1, w_2, \ldots, w_l$ such that $w_i \in \hat{\rho}(f_i) \cap \hat{\rho}(f_{i+1})$ for all $i$. If $f_i$ and $f_{i+1}$ were adjacent in $C$, then there is some $v_i \in f_i \cap f_{i+1}$ and so $\rho(v_i) \in \hat{\rho}(f_i) \cap \hat{\rho}(f_{i+1})$. Otherwise, $f_i$ and $f_{i+1}$ were separated in $C$ by some sequence of edges from $F$ that all belong to the same connected component in $H[F]$, so, under $\hat{\rho}$, all of these edges from $F$ map to $\{v\}$ for some $v \in V(H') \setminus V(H)$. This vertex $v$ is clearly in both $\hat{\rho}(f_i)$ and $\hat{\rho}(f_{i+1})$. Proceeding in this manner, we get a sequence of vertices $w_1, w_2, \ldots w_l$ as required, and the only way that we are forced to choose the same vertex twice is if $w_i = w_j = v$ for some $i \neq j$ and $v \in V(H') \setminus V(H)$. In this case, we can replace the sequence of edges $f_{i+1}, f_{i+2}, \ldots, f_j$ with a path through the connected component in $H[F]$ induced by edges that map to $\{v\}$ under $\hat{\rho}$, creating a cycle with fewer edges from $F_1$ than $C$ and contradicting the fact that $C$ was a cycle using a minimum number of edges from $F_1$.

In the other direction, assume $F_2$ induces a cycle $C$ in $H'$. Let $f_1, f_2, \ldots, f_l$ be the sequence of distinct edges and $w_1, w_2, \ldots, w_l$ be the sequence of distinct vertices. Using $\hat{\rho}$, we can pull the edges

$f_1, f_2, \ldots, f_l$ back to distinct edges $\hat{\rho}^{-1}(f_1), \hat{\rho}^{-1}(f_2), \ldots, \hat{\rho}^{-1}(f_l)$ in $H$, and the only way that there does not exist a distinct $v_i \in V(H)$ such that $v_i \in \hat{\rho}^{-1}(f_i) \cap \hat{\rho}^{-1}(f_{i+1})$ for any $i$ is if $w_i$ is the image of a connected component in $H[F]$. But, since all $w_i$'s are pairwise distinct, all of the connected components in $H[F]$ they correspond to are pairwise distinct as well, so we can form a cycle in $H$ by connecting any such pair $\hat{\rho}^{-1}(f_i)$, $\hat{\rho}^{-1}(f_{i+1})$ that don't share a common vertex with a path through the connected component that maps to $\{w_i\}$ under $\hat{\rho}$.

■

We now need to formally define the function **Resolve** used to map edges from a hypergraph resulting from some sequence of contractions to edges in the original hypergraph. Given any sequence of hypergraphs $H_0, H_1, \ldots, H_{k+1}$ and a sequence of forests $F_0, F_1, \ldots F_k$ such that $F_i$ is a forest in $H_i$ and $H_{i+1} = \textbf{Contract}(H_i, F_i)$ for all $i$, let $\hat{\rho}_i$ be the function associated with the contraction **Contract**$(H_i, F_i)$. Define a function $\phi_i : E(H_{i+1}) \mapsto E(H_i)$ by associating each edge $e$ in $E(H_{i+1})$ with an edge $\phi_i(e)$ in $E(H_i)$ that maps to $e$ under $\hat{\rho}_i$. Notice that $\hat{\rho}_i$ restricted to the image of $\phi_i$ is a bijection between the image of $\phi_i$ and $E(H_{i+1})$. Now, for any $i \geq j$, the function $\textbf{Resolve}_{(i,j)}$ used to map an edge in the hypergraph $H_i$ to an edge in the hypergraph $H_j$ is defined as $\phi_j \circ \phi_{j+1} \circ \ldots \circ \phi_{i-2} \circ \phi_{i-1}$,

or the identity function if $i = j$, in either case a bijection between edges in $H_j$ and some set of edges in $H_0$.

Finally, we can prove the correctness of iterative contraction in producing a forest using the following lemma:

**Lemma 10** *Given any sequence of hypergraphs $H_0, H_1, \ldots, H_{k+1}$ and a sequence of forests $F_0, F_1, \ldots F_k$ such that $F_i$ is a forest in $H_i$ for all $i$, $H_{i+1} =$ **Contract**$(H_i, F_i)$ for all $i, 0 \leq i \leq k$, and $H_{k+1}$ has an empty edge set, $\bigcup_{i=0}^{k} \bigcup_{f \in F_i}$ **Resolve**$_{(i,0)}(f)$ is a maximal forest in $H$.*

**Proof:** We will show that $\bigcup_{i=j}^{k} \bigcup_{f \in F_i}$ **Resolve**$_{(i,j)}(f)$ is a maximal forest in $H_j$ by induction on $k - j$. When $k - j = 0$, **Resolve**$_{(k,j)}$ is just the identity function on $E(H_k)$ and $F_k$ is clearly maximal since $E(H_{k+1}) = \emptyset$. For general $k - j > 0$, assuming that $\bigcup_{i=j}^{k} \bigcup_{f \in F_i}$ **Resolve**$_{(i,j)}(f)$ is a maximal forest in $H_j$, we wish to show that $\bigcup_{i=j-1}^{k} \bigcup_{f \in F_i}$ **Resolve**$_{(i,j-1)}(f)$ is a maximal forest in $H_{j-1}$. Set $F' = \bigcup_{i=j}^{k} \bigcup_{f \in F_i}$ **Resolve**$_{(i,j)}(f)$, and now $\hat{\rho}_{j-1}$ restricted to $\phi_{j-1}(F')$ is a bijection between $\phi_{j-1}(F')$ and $F'$, so Lemma 9 applies to show that

$$\begin{aligned}
\phi_{j-1}(F') \cup F_{j-1} &= \phi_{j-1}\left(\bigcup_{i=j}^{k} \bigcup_{f \in F_i} \textbf{Resolve}_{(i,j)}(f)\right) \cup F_{j-1} \\
&= \bigcup_{i=j}^{k} \bigcup_{f \in F_i} \textbf{Resolve}_{(i,j-1)}(f) \;\cup\; F_{j-1} \\
&= \bigcup_{i=j-1}^{k} \bigcup_{f \in F_i} \textbf{Resolve}_{(i,j-1)}(f)
\end{aligned}$$

is a forest in $H_{j-1}$, which is maximal in $H_{j-1}$ since $F'$ was maximal in $H_j$. ■

### 4.2.2 The Algorithm

For a set of edges $S \subseteq E(H)$ and an edge $e \in E(H)$, we define the following set:

$$\textbf{meet}(e, S) \;=\; \{v \in e \;:\; \exists f \in S,\; v \in f\} = \bigcup_{f \in S} (e \cap f)$$

One component in our algorithm for finding a maximal forest in a hypergraph is an algorithm for finding a generalization of a maximal matching in a hypergraph. A matching $M \subseteq E(H)$ is a set of pairwise disjoint edges; the matching is maximal iff for any $e \in E(H)$, $|\textbf{meet}(e, M)| \geq 1$. Repeatedly finding and contracting just a maximal matching in a hypergraph, however, is not enough to guarantee rapid progress towards finding a maximal forest in a hypergraph. For example, if the original hypergraph consisted of $m$ edges, all of which intersect in a single common vertex, $m$ stages of matching and contraction would be needed to create a maximal forest. It turns out that not too much more than a maximal matching is needed to guarantee rapid progress towards a maximal forest - each iteration of our algorithm augments a maximal matching with a few additional edges to create a forest $F$ such that

**Algorithm MaximalForest(H)**

(1) $F \leftarrow \emptyset$
(2) **while** $E(H) \neq \emptyset$ **do**
(3) $\quad M_1 \leftarrow$ **MaximalSetPacking**$(E(H), \emptyset)$
(4) $\quad E' \leftarrow \{e \in E(H) : |\textbf{meet}(e, M_1)| = 1\}$
(5) $\quad M_2 \leftarrow$ **MaximalSetPacking**$(E', \bigcup_{m \in M_1} m)$
(6) $\quad F \leftarrow F \cup \bigcup_{m \in M_1 \cup M_2}$ **Resolve**$(m)$
(7) $\quad H \leftarrow$ **Contract**$(H, F)$
(8) **return** $F$

Figure 4.5: An algorithm that finds a maximal forest in a hypergraph

for each $e \in E(H)$, $|\textbf{meet}(e, F)| \geq 2$. The difference between such a forest $F$ and a maximal matching is highlighted by the earlier example: given $m$ edges all intersecting in a single vertex, the only such forest $F$ is the entire edge set.

To describe the procedure that finds the additional edges that augment our original maximal matching, we use the function **MaximalSetPacking**$(E, X)$ defined in Section 2.3. Recall that **MaximalSetPacking**$(E, X)$ returns a maximal set $M \subseteq E$ such that for any $m, n \in M$, $m \cap n \subseteq X$.

Our algorithm for finding a maximal forest in a hypergraph is given in figure 4.5.

**Lemma 11** ***MaximalForest(H)*** *returns a maximal forest in* $H$*.*

**Proof:** Assume that during some iteration of the main while loop, $M_1 \cup M_2$ induces a cycle in $H$. Edges in $M_1$ form a matching in $H$, so any cycle in $H$ that is made up of edges from $M_1 \cup M_2$ must contain at least one edge from $M_2$. But, no edge $m \in M_2$ can be part of any cycle, since $|\textbf{meet}(m, M_1 \cup (M_2 \setminus \{m\}))| = 1$ by construction. So, during any iteration, $M_1 \cup M_2$ is a forest. At least one edge is chose per iteration by the first call to **MaximalSetPacking**, so the algorithm eventually terminates when $E(H) = \emptyset$. The correctness of the algorithm now follows from Lemma 10. ■

Given a hypergraph $H$, let $\mu(H)$ be the matching number of $H$, defined as the maximum cardinality of any matching on $H$. The proof of the following proposition is straightforward, and thus omitted:

**Proposition 12** *For any hypergraph $H$ and any set $S \subseteq V(H)$ such that every edge $e \in E(H)$ contains at least two vertices from $S$, $\mu(H) \leq |S|/2$.*

**Lemma 13** *The main while loop in **MaximalForest(H)** executes $O(\log n)$ times.*

**Proof:** Each iteration of the main while loop in **MaximalForest** finds a forest $F$ in the current hypergraph $H$, so the entire algorithm produces a sequence of hypergraphs $H_0, H_1, \ldots, H_k$ and a sequence of forests $F_0, F_1, \ldots, F_{k-1}$ such that $F_i$ is a forest in $H_i$ and

$H_{i+1} = \textbf{Contract}(H_i, F_i)$ for all $i$, $0 \leq i \leq k-1$. $H_0$ is the original hypergraph, and $H_k$ is a hypergraph containing no edges.

We now claim that $\mu(H_{i+1}) \leq \mu(H_i)/2$ for all $i$, $0 \leq i \leq k-1$, which will prove the lemma since $\mu(H_0) \leq n/2$ and any hypergraph $H$ having $\mu(H) < 1$ must have an empty edge set. Let $c(H_i)$ be the number of connected components in $H_i[F_i]$, for all $i$. $c(H_i) \leq \mu(H_i)$, since the set of connected components in $H_i[F_i]$ can be converted into a matching of size $c(H_i)$ on $H_i$ by choosing one edge per connected component.

By definition of a maximal matching, $|\textbf{meet}(e, M_1)| \geq 1$ for any $e \in E(H)$. Set $V_1 = \bigcup_{m \in M_1} m$ so that $\{e \setminus V_1 \ : \ e \in M_2\}$ is a maximal matching in the hypergraph $\{e \setminus V_1 \ : \ |\textbf{meet}(e, M_1)| = 1\}$ and therefore any edge $e$ with $|\textbf{meet}(e, M_1)| = 1$ has $|\textbf{meet}(e \setminus V_1, M_2)| \geq 1$. So, for any edge $e \in E(H)$,

$$|\textbf{meet}(e, F)| \ = \ |\textbf{meet}(e, M_1 \ \cup \ M_2)| \ \geq \ 2 \tag{4.10}$$

For any hypergraph $H$ and a forest $F$ in $H$, the only edges in **Contract**$(H, F)$ are those edges $e \in E(H) \setminus F$ that intersect each connected component in $H[F]$ in at most one vertex, so from (4.10) we see that each edge $e \in E(H)$ corresponding to an edge $e'$ in **Contract**$(H, F)$ intersects at least two separate connected components in $H[F]$. There are $c(H)$ connected components in $H[F]$, all

of which get mapped to distinct vertices in **Contract**$(H, F)$. These vertices satisfy the assumptions of Proposition 12, so for any $i$, $\mu(H_{i+1}) \leq c(H_i)/2 \leq \mu(H_i)/2$. ∎

**Theorem 14** ***MaximalForest***$(H)$ *runs in time* $\mathcal{O}(\log n \log^3 m)$ *using* $\mathcal{O}((m + \sum_{v \in V(H)} \binom{d(v)}{2})/\log m)$ *processors on an EREW PRAM.*

**Proof:** In Section 2.3, we noticed that the dependency graph created in the algorithm **MaximalSetPacking** from a hypergraph $H$ had $|E(H)|$ vertices and at most $\sum_{v \in V(H)} \binom{d(v)}{2}$ edges. The algorithm then finds an MIS in this dependency graph. To find a maximal indpendent set in a graph, we use an algorithm of Goldberg and Spencer [17] which runs in time $\mathcal{O}(\log^3 n)$ using $\mathcal{O}((n+m)/\log n)$ processors. This gives us a $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}((m + \sum_{v \in V(H)} \binom{d(v)}{2})/\log m)$ processor EREW PRAM algorithm for finding a maximal set packing in a hypergraph with $m$ edges. Computing **Contract**$(H, F)$ involves computing the edge-induced connected components of a hypergraph, which can be accomplished by a reduction to the problem of finding the connected components of a graph: simply represent each hyperedge with a path of edges in the corresponding graph. Identifying connected components in a graph can be done in time $\mathcal{O}(\log^{3/2} n)$ using $\mathcal{O}(m + n)$ processors using Karger, Nisan, and Parnas's algorithm from [23]. The time and processor bounds for identifying connected components, as well as performing the

other set operations needed during an iteration of **MaximalForest** are clearly dominated by those for finding a matching in a hypergraph, so the analysis above combined with Lemma 13 gives the theorem. ■

### 4.3 An $\mathcal{NC}$ Algorithm for Finding a Maximal Acyclic Set in a Graph

For a graph $G$ and any $v \in V(G)$, let $\gamma(v) = \{e \in E(G) : v \in e\}$. The following lemma formalizes the reduction between finding a maximal forest in a hypergraph and finding an MAS in a graph:

**Lemma 15** *Let $G$ be a graph, and let $H$ be a hypergraph with vertex set $E(G)$ and edge set $\{\gamma(v) : v \in V(G)\}$. Then $A \subseteq V(G)$ is an acyclic set in $G$ iff $\{\gamma(a) : a \in A\}$ is a forest in $H$.*

**Proof:** Assume there is a cycle induced by vertices in $A$, so that there exists a sequence of distinct vertices $a_0, a_1, \ldots, a_{k-1}$, each $a_i \in A$, and a sequence of distinct edges $e_0, e_1, \ldots, e_{k-1}$, each $e_i \in E(G)$, such that $k > 1$ and $a_i \in e_i \cap e_{i+1}$ for all $i$. Notice that for any two vertices $u, v \in V(G)$, $\gamma(u) \cap \gamma(v) \subseteq \{\{u, v\}\}$ with equality iff $u$ and $v$ are adjacent in $G$. So, all edges in the sequence $\gamma(a_0), \gamma(a_1), \ldots, \gamma(a_{k-1})$ are distinct, since the only way $\gamma(u) = \gamma(v)$ for $u, v \in V(G)$ is if both $u$ and $v$ have degree at most 1, in which case they couldn't have appeared in the original cycle. Since $G$ is a graph, it must

be the case that $e_i = \{a_{i-1}, a_i\}$ for all $i$, so that $e_i \in \gamma(a_{i-1}) \cap \gamma(a_i)$ for all $i$, and the sequence $e_0, e_1, \ldots, e_k$ of vertices from $V(H)$ and $\gamma(a_{k-1}), \gamma(a_0), \ldots, \gamma(a_{k-2})$ of edges in $E(H)$ forms a cycle in $H$.

In the other direction, assume there is a cycle in $H$, so that there exists a sequence of distinct vertices $e_0, e_1, \ldots, e_{k-1}$ from $V(H)$ and distinct edges $\gamma(a_0), \gamma(a_1), \ldots, \gamma(a_{k-1})$ from $E(H)$ such that $k > 1$ and $e_i \in \gamma(a_i) \cap \gamma(a_{i+1})$ for all $i$. Since $a_i = a_j \implies \gamma(a_i) = \gamma(a_j)$, the vertices from $V(G)$ in the sequence $a_0, a_1, \ldots, a_{k-1}$ are all distinct. Furthermore, since $e_i \in \gamma(a_i) \cap \gamma(a_{i+1})$ for all $i$, $e_i = \{a_i, a_{i+1}\}$ for all $i$, and therefore the sequence of vertices $a_1, a_2, \ldots, a_{k-1}, a_0$ from $V(G)$ together with the sequence of edges $e_0, e_1, \ldots, e_{k-1}$ from $E(G)$ forms a cycle in $G$. ■

The construction in Lemma 15 is essentially the construction of the dual of a hypergraph (cf. [4], p. 390), except that the dual of a hypergraph is technically a multi-hypergraph and our construction doesn't make use of multiple edges.

Our algorithm for finding an MAS in a graph is now straightforward - create the hypergraph described in Lemma 15, find a maximal forest in it, and return the set of vertices corresponding to the edges in that forest. Since the edge set of the hypergraph contains no more than $|V(G)|$ edges, this is an efficient reduction, and we have the following theorem:

**Theorem 16** *There exists an algorithm to find an MAS in any graph $G$ in $\mathcal{O}(\log^4 n)$ steps using $\mathcal{O}((m^2+n)/\log n)$ processors on a EREW PRAM.*

**Proof:** Set $n = |V(G)|$ and $m = |E(G)|$. The construction in Lemma 15 produces a hypergraph $H$ with at most $m$ vertices and $n$ edges. $H$ is linear (no two edges in the hypergraph intersect in more than one vertex) and has maximum degree 2. The sum $\sum_{v \in V(H)} d(v)$ is therefore initially bounded from above by $2m$. Let $H'$ be any hypergraph that results from the contraction of any set of edges in $H$. $H'$ must also have $\sum_{v \in V(H')} d(v) \leq 2m$, so

$$\sum_{v \in V(H')} \binom{d(v)}{2} \leq \binom{\sum_{v \in V(H')} d(v)}{2} = \mathcal{O}(m^2)$$

Plugging this in to Theorem 14 gives the required bounds. ∎

## 4.4 An $\mathcal{NC}$ Algorithm for Approximating Maximum Planar Subgraph

Given a graph property $\pi$, we have been interested in this thesis with finding a maximal set with property $\pi$, and we could have just as easily asked for the minimal set with the complement of property $\pi$. The minimal (resp. maximal) solutions can be thought of as local minimums (resp. maximums), and we now turn to the problem of computing the global minimum or maximum size set with property

$\pi$. Unfortunately, for most interesting properties $\pi$ (independence and acyclicty included), the problem of finding the global minimum or maximum size set with property $\pi$ is $\mathcal{NP}$-complete.

When faced with an $\mathcal{NP}$-complete graph optimization problem $\pi$, one line of attack is the design of an approximation algorithm for $\pi$. A *factor-$\alpha$ approximation algorithm* for a graph minimization problem is an algorithm that is guaranteed to return a solution whose size is at most $\alpha$ times the optimal solution. A *factor-$\alpha$ approximation algorithm* for a graph maximization problem is an algorithm that is guaranteed to return a solution whose size is at least $\alpha$ times the optimal solution. In the case of minimization problems $\alpha$ will be at least one, and in the case of maximization problems $\alpha$ will be at most one. In either case, the closer $\alpha$ is to one, the better the approximation. For a complete introduction to the subject of approximation algorithms, we suggest the text by Vazirani [42].

Even better than a polynomial time approximation algorithm is an $\mathcal{NC}$ approximation algorithm; recently the importance of parallel approximation algorithms has begun to be noticed and a textbook on the young subject has even appeared [16].

The maximum planar subgraph problem is: given a graph $G$, find the maximum cardinality subset $S \subseteq E(G)$ such that $G' = (V(G), S)$ is planar. In [11], Chen and Uehara describe how to use a parallel algorithm for finding an MAS in a graph of degree 3 to

parallelize the factor 7/18 approximation algorithm of Câlinescu, Fernandes, Finkler, and Karloff [8] for finding a maximum planar subgraph in a graph. We outline their approach here and refer the reader to the original papers for more details.

Chen and Uehara reduce the problem of parallelizing this approximation algorithm to the following problem: Given a bipartite graph $G$ with bipartition $X, Y$, where each vertex in $Y$ has degree 3, find a maximal acyclic set in $G$ that contains $X$. Our algorithm for finding an MAS from section 4.3 can be easily adapted to solve this problem: simply initialize the acyclic set $A$ to $X$ instead of the empty set at the beginning of the algorithm. The full approximation algorithm requires two more major computations: finding a spanning tree in a graph and identifying the connected components in a graph, both of which can be done in less time and with fewer processors on an EREW PRAM than our algorithm for finding an MAS, so we have the following theorem:

**Theorem 17** *There exists a factor-*$7/18$ $\mathcal{NC}$ *approximation algorithm for the maximum planar subgraph problem that runs in time* $\mathcal{O}(\log^4 n)$ *using* $\mathcal{O}((m^2 + n)/\log n)$ *processors on an EREW PRAM.*

## Chapter 5 - Open Questions

We close with a few open questions that have been raised by the work in this thesis.

**Borůvka's Algorithm on Hypergraphs**

The following randomized variant of Borůvka's algorithm for graphs generalized to hypergraphs is a good candidate for a simpler algorithm for finding a maximal forest in a hypergraph:

1. Choose a random permutation of the set $\{1, \ldots, m\}$ and use this to assign a distinct non-negative integer weight to each edge. Each vertex then votes for the edge containing it of least weight.

2. Any edge $e$ that receives at least $|e| - 1$ votes gets added to the forest.

3. Contract all edges added to the forest from step (2), then repeat from step (1) until the edge set is empty.

It can be shown that the edges added in step (2) form a forest, which along with Lemma 10 shows the correctness of the algorithm. We assign weights randomly since one can easily

construct hypergraphs with edge rankings that cause this algorithm to run for $O(m)$ iterations. For example, take

$$E(H) = \{\{v_1, v_2, v_3\}, \{v_2, v_3, v_4\}, \{v_3, v_4, v_5\}, \ldots, \{v_{n-2}, v_{n-1}, v_n\}\}$$

with the edges ranked by the index of the smallest vertex they contain. We conjecture the following:

**Conjecture 18** *The algorithm above is an $\mathcal{RNC}$ algorithm for finding a maximal forest in a hypergraph.*

**Maximal Acyclic Sets in Directed Graphs**

The ultimate goal of the line of research from Chapter 4 is to find an $\mathcal{NC}$ algorithm for finding a maximal vertex-induced acyclic subgraph in a *directed* graph; this problem has applications to resolving deadlock situations efficiently. Unfortunately, even the problem of finding a maximal edge-induced acyclic subgraph in a directed graph is not known to be in $\mathcal{NC}$, although Berger and Shor [7] have developed an interesting $\mathcal{RNC}$ algorithm that is guaranteed to always find a "large" edge-induced acyclic subgraph of a directed graph.

The problem of finding a maximal edge-induced acyclic subgraph of a directed hypergraph (each edge in a directed hypergraph is an ordered pair of disjoint subsets of the vertex set) is equivalent to the problem of finding a maximal vertex-induced acyclic subgraph

in a directed graph under the same dual-hypergraph transformation we used in section 4.3 for the maximal acyclic set problem.

**Approximating Weighted Minimum Feedback Vertex Set in Graphs**

Given an acyclic set $S \subseteq V(G)$, the set $V(G) \setminus S$ is called a *feedback vertex set*. The weighted minimum feedback vertex set problem is: given a graph $G$ and a function $f : V(G) \mapsto Q$, find the feedback vertex set $S \subseteq V(G)$ such that $\sum_{v \in S} f(v)$ is minimized. Recently, Bafna, Berman, and Fujito [2] devised a factor-2 approximation algorithm for this problem. Their approach involves the repeated extraction of a minimal feedback vertex set from a series of vertex-induced subgraphs of the original graph. In light of our results from Chapter 4, this approximation algorithm may be amenable to parallelization, since our $\mathcal{NC}$ algorithm for finding a maximal acyclic set in a graph can easily be converted into an $\mathcal{NC}$ algorithm for finding a minimal feedback vertex set in a graph. The worst case number of stages in the graph decomposition should be able to be cut from $n$ to $\log n$ by sacrificing a little in the approximation guarantee, which would mean that we would no longer expect a factor-2 approximation algorithm. We therefore conjecture the following:

**Conjecture 19** *For any $\epsilon > 0$, there exists a factor-$(2+\epsilon)$ $\mathcal{NC}$ approximation algorithm for the problem of finding a weighted maximum feedback vertex set in a graph.*

**Finding an MIS in a Hypergraph**

There are still many open questions concerning parallel algorithms for finding an MIS in a hypergraph. Our analysis of the permutation algorithm in section 3.2 suggests that it performs much better than an algorithm already known to be an $\mathcal{RNC}$ algorithm, so we conjecture the following:

**Conjecture 20** *The permutation algorithm is an $\mathcal{RNC}$ algorithm for finding an MIS in a hypergraph of bounded dimension.*

Furthermore, we conjecture

**Conjecture 21** *The permutation algorithm is an $\mathcal{RNC}$ algorithm for finding an MIS in a hypergraph.*

**Bibliography**

[1] N. Alon, L. Babai and A. Itai, A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem, *Journal of Algorithms* **7** (1986) 567-583.

[2] V. Bafna, P. Berman and T. Fujito, A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem, *SIAM Journal of Discrete Math* **12** (1999) 289-297.

[3] K. Batcher, Sorting Networks and Their Applications, In *Proc. AFIPS Spring Joint Computer Conference 32*, AFIPS Press, Reston, VA, (1968) 307-314.

[4] C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Company, Amsterdam, 1973.

[5] O. Borůvka, O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti* **3** (1926) 37-58.

[6] P. Beame and M. Luby, Parallel Search for Maximal Independence Given Minimal Dependence, In *Proc. First*

*ACM-SIAM Symposium on Discrete Algorithms*, ACM Press, New York, NY, (1990) 212-218.

[7] B. Berger and P. Shor, Tight Bounds for the Maximum Acyclic Subgraph Problem, *Journal of Algorithms* **25** (1997) 1-18.

[8] G. Câlinescu, C.G. Fernandes, U. Finkler and H. Karloff, A Better Approximation Algorithm for Finding Planar Subgraphs, In *Proc. Seventh ACM-SIAM Symposium on Discrete Algorithms*, ACM Press, New York, NY, (1996) 16-25.

[9] R. Cole, Parallel Merge Sort, *SIAM Journal on Computing* **17** (1988) 770-785.

[10] Z.-Z. Chen and X. He, Parallel Algorithms for Maximal Acyclic Sets, *Algorithmica* **19** (1997) 354-368.

[11] Z.-Z. Chen and R. Uehara, Parallel Algorithms for Maximal Linear Forests. *IEICE Transactions on Information and Systems* **E80-A** (1997) 627-634.

[12] Z.-Z. Chen and S. Zhang, A Tight Upper Bound on the Number of Edges in a Bipartite, $K_{3,3}$-free or $K_5$-free graph with an Application. *Information Processing Letters* **84** (2002) 141-145.

[13] K. Chong, Y. Han and T. Lam, Concurrent Threads and Optimal Parallel Minimum Spanning Trees Algorithm, *Journal of the ACM* **48** (2001) 297-323.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian and T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, In *Proc. Fourth ACM Symposium on Principles and Practice of Parallel Programming*, ACM Press, New York, NY, (1993) 1-12.

[15] E. Dahlhaus, M. Karpinski and P. Kelsen, An Efficient Parallel Algorithm for Computing a Maximal Independent Set in a Hypergraph of Dimension 3, *Information Processing Letters 42* (1992) 309-313.

[16] J. Díaz, M. Serna, P. Spirakis and J. Torán, *Paradigms for Fast Approximability*, Cambridge University Press, New York, NY, 1997.

[17] M. Goldberg and T. Spencer, A New Parallel Algorithm for the Maximal Independent Set Problem, *SIAM Journal of Computing* **18** (1989) 419-427.

[18] M. Goldberg and T. Spencer, An Efficient Parallel Algorithm that Finds Independent Sets of Guaranteed Size, *SIAM Journal of Discrete Mathematics* **6** (1993) 443-459.

[19] R. Greenlaw, H. Hoover and W. Ruzzo, *Limits to Parallel Computation : P-completeness Theory*, Oxford University Press, New York, NY, 1994.

[20] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, Clarendon Press, Oxford, 1965.

[21] P. Indyk, A Small Approximately Min-Wise Independent Family of Hash Functions, *Journal of Algorithms* **38** (2001) 84-90.

[22] D. Karger, P. Klein and R. Tarjan, A Randomized Linar-Time Algorithm to Find Minimum Spanning Trees, *Journal of the ACM* **42** (1995) 321-328.

[23] D. Karger, N. Nisan and M. Parnas, Fast Connected Components Algorithms for the EREW PRAM, *SIAM Journal of Computing* **28** (1999) 1021-1034.

[24] R. Karp and V. Ramachandran, Parallel Algorithms for Shared Memory Machines, In *J. van Leeuwen, ed., Handbook of Theoretical Computer Science Vol. A*, Elsevier, New York, NY, 1990.

[25] R. Karp, E. Upfal, and A. Widgerson, The Complexity of Parallel Search, *Journal of Computer and Systems Sciences* **36** (1988) 225-253.

[26] R. Karp and A. Widgerson, A Fast Parallel Algorithm for the Maximal Independent Set Problem, *Journal of the ACM* **32** (1985) 762-773.

[27] P. Kelsen, On the Parallel Complexity of Computing a Maximal Independent Set in a Hypergraph, In *Proc. Twenty-fourth ACM Symposium on Theory of Computing*, ACM Press, New York, NY, (1992) 339-350.

[28] M. Luby, A Simple Parallel Algorithm for the Maximal Indpendent Set Problem, *SIAM Journal of Computing* **15** (1986) 1036-1053.

[29] T. Łuczak and E Szymańska, A Parallel Randomized Algorithm for Finding a Maximal Independent Set in a Linear Hypergraph, *Journal of Algorithms* **25** (1997) 311-320.

[30] S. Miyano, The Lexicographically First Maximal Subgraph Problems: P-Completeness and NC Algorithms, *Math. Systems Theory* **22** (1989) 47-73.

[31] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, NY, 1995.

[32] K. Mulmuley, U. Vazirani and V. Vazirani, Matching is as Easy as Matrix Inversion, *Combinatorica* **7** (1987) 105-113.

[33] D. Pearson and V. Vazirani, Efficient Sequential and Parallel Algorithms for Maximal Bipartite Sets, *Journal of Algorithms* **14** (1993) 171-179.

[34] S. Pettie and V. Ramachandran, A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest, *SIAM Journal on Computing* **31** (2002) 1879-1895.

[35] C. Poon and V. Ramachandran, A Randomized Linear Work EREW PRAM Algorithm to Find a Minimum Spanning Forest, In *Proc. Seventh International Symposium on Algorithms and Computation*, LNCS 1350, Springer-Verlag, Berlin, (1997) 212-222.

[36] H. Shachnai and A. Srinivasan, Finding Large Independent Sets in Hypergraphs in Parallel, In *Proc. Thirteenth Symposium on Parallelism in Algorithms and Architectures*, ACM Press, New York, NY, (2001) 163-168.

[37] E. Szymańska, Derandomization of a Parallel MIS Algorithm in a Linear Hypergraph, *Proc. ICALP Satellite Workshops*, Carleton Scientific, Waterloo, Ontario, Canada, (2001) 39-52.

[38] R. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics Vol. 44, SIAM, Philadelphia, PA, 1983.

[39] I. Tomescu, M. Zimand, Minimum Spanning Hypertrees, *Discrete Applied Mathematics* **54** (1994) 67-76.

[40] R. Uehara, Tractable and Intractable Problems on Generalized Chordal Graphs, *IEICE Technical Report, COMP98-83* (1999) 1-8.

[41] L. Valiant, A Bridging Model for Parallel Computation, *Communications of the ACM* **33** (1990) 103-111.

[42] V. Vazirani, *Approximation Algorithms*, Springer, New York, NY, 2001.

## Appendix A - Some Technical Lemmas

**Lemma 22** *Let $\alpha, \beta$ be constants with $\alpha > \beta > 0$, and let $X$ be a discrete random variable taking on values from the set $\{1, 2, \ldots, n\}$. If $\boldsymbol{E}[X] \geq \alpha n$, then*

$$\Pr[X \geq \beta n] \geq \alpha - \beta.$$

**Proof:** Let $p = \Pr[X \geq \beta n]$. Now,

$$\alpha n \ \leq \ \mathbf{E}[X] \ \leq \ pn + (1-p)\beta n$$

So $p \geq \alpha - \beta$ ■

**Lemma 23 (Chernoff Bounds)** *Let $X_1, X_2, \ldots, X_n$ be independent 0-1 random variables, and define $p_i$ for $1 \leq i \leq n$ as $p_i = \Pr[X_i = 1]$. Let $\mu = \sum_i p_i$, the expectation of the sum of these variables. Then, for any $\delta$ such that $0 < \delta \leq 1$,*

$$\Pr[X > (1+\delta)\mu] < \left\{ \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right\}^{\mu}$$

*and*

$$\Pr[X < (1-\delta)\mu] < e^{-\mu\delta^2/2}$$

For a good proof of the Chernoff Bounds, see [31], Chapter 4. In particular, we point out the following simple corollary of the Chernoff Bounds:

**Corollary 24** *Let $X_1, X_2, \ldots, X_n$ be independent 0-1 random variables and define $p_i$ for $1 \leq i \leq n$ as $p_i = \Pr[X_i = 1]$. Let $\mu = \sum_i p_i$. If $\mu = \Omega(\log n)$ then*

$$\Pr[|X - \boldsymbol{E}[X]| > \mu/2] = o(1)$$