Local Computation Algorithms

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Shai Vardi

The research for this thesis has been carried out at Tel-Aviv University under the supervision of Prof. Ronitt Rubinfeld

March 2011
Acknowledgements

I would like to thank:

My thesis advisor, Prof. Ronitt Rubinfeld for being super awesome in so many ways.

Ning Xie for his wonderful ideas and contributions and for always being available on Skype.

My colleagues Rita, Reut and Aviad for their time and insights.

Mom and Dad for allowing me to find my own way.

Shiri for making everything not complicated.

And of course the ants. Thanks, ants. Thants.
Abstract

For input $x$, let $F(x)$ denote the set of outputs that are the “legal” answers to a computational problem $F$. Suppose $x$ and members of $F(x)$ are so large that there is not time to read them in their entirety. We propose a model of local computation algorithms which, for a given input $x$, support queries by a user to values of specified locations $y_i$ in a legal output $y \in F(x)$. When more than one legal output $y$ exists for a given $x$, the local computation algorithm should output in a way that is consistent with at least one such $y$. Local computation algorithms are intended to distill the common features of several concepts that have appeared in various algorithmic subfields, including local distributed computation, local algorithms, locally decodable codes, and local reconstruction.

We develop a technique which under certain conditions can be applied to construct local computation algorithms that run in polylogarithmic time and space. The technique is based on known constructions of small sample spaces of $k$-wise independent random variables and Beck’s analysis in his algorithmic approach to the Lovász Local Lemma. We apply this technique to maximal independent set computations, scheduling radio network broadcasts, hypergraph coloring and satisfying $k$-SAT formulas.
## Contents

1 Introduction and related work 5  
1.1 Introduction ........................................ 5  
1.2 Related work ....................................... 7  
  1.2.1 Local algorithms in distributed networks ........... 7  
  1.2.2 The Parnas Ron reduction ......................... 8  
  1.2.3 Sublinear optimization algorithms on graphs ....... 8  
  1.2.4 Locally decodable codes and reconstruction problems 10  
1.3 Organization ....................................... 10  

2 The model and preliminaries 11  
2.1 Local Computation Algorithms: the model ................ 11  
  2.1.1 Model Definition .................................. 11  
  2.1.2 Relationship with other distributed and parallel models 12  
  2.1.3 Locality-preserving reductions ................... 13  
2.2 Preliminaries ........................................ 13  
  2.2.1 $k$-wise independent random variables ............. 14  

3 Maximal Independent Set 15  
3.1 Overview of the algorithm ............................ 15  
3.2 Phase 1: Simulating Luby’s parallel algorithm ........ 16  
3.3 Replacing true randomness with pseudorandomness ...... 20  
3.4 Phase 2: Greedy search in the connected component .... 20  

4 Radio Networks 21  
4.1 A local computation algorithm for ISC ................ 22  
  4.1.1 Phase 1 algorithm ............................... 23  
  4.1.2 Phase 2 algorithm ............................... 24  
4.2 Discussions ....................................... 24  

5 Hypergraph Two-Coloring 25  
5.1 Our main result .................................... 25  
5.2 Overview of the coloring algorithm .................... 26  
5.3 Phase 1 coloring .................................. 27  
5.4 Phase 2 coloring .................................. 27  
5.5 Phase 3 coloring .................................. 30  

6 $k$-CNF 31
7 Concluding Remarks and Open Problems 33
A Proof of the Parnas Ron reduction 34
Chapter 1

Introduction and related work

1.1 Introduction

Classical models of algorithmic analysis assume that the algorithm reads an input, performs a computation and writes the answer to an output tape. On massive data sets, such computations may not be feasible, as both the input and output may be too large for a single processor to process. One way people have dealt with this problem is to consider what happens when more resources are available; this approach has led to parallel and distributed models of computation. When no extra resources are available, (and we find ourselves confined to time and/or space sublinear in the size of the input), another line of research considers asking for weaker notion of success – approximate answers.

In this work, we consider the scenario in which only specific parts of the output \( y = (y_1, \ldots, y_m) \) are needed at any point in time. For example, if \( y \) is a description of a maximal independent set (MIS) of a graph such that \( y_i = 1 \) if \( i \) is in the MIS and \( y_i = 0 \) otherwise, then it may be sufficient to determine \( y_i \) for a small number of vertices. To this end, we define a local computation algorithm, which supports queries such that for each query by the user to a specified location \( i \), the local computation algorithm quickly outputs \( y_i \). For problem statements that allow for more than one possible output for an input \( x \), as is often the case in combinatorial search problems, the local computation algorithm must answer in such a way that is consistent with any past or future answers (in particular, there must always be at least one allowable output that is consistent with all the answers of the local computation algorithm). For a given problem, the hope is that the complexity of a local computation algorithm is proportional to the amount of the solution that is requested by the user. Local computation algorithms are especially adapted to computations of good solutions to combinatorial search problems (cf. [PR07 MR06 NO08]). For example, for the problem of finding a maximal independent set (MIS) of a very large graph, on input a vertex in the graph, the processor should be able to quickly determine whether that vertex is in the MIS, in such a way that is consistent with all past and future queries.

Local computation algorithms are a formalization that describes concepts that are ubiquitous in the literature, and arise in various algorithmic subfields, including local distributed computation, local algorithms, locally decodable codes and local reconstruction models. We subsequently describe the relation between these models in more detail, but for now, suffice it to say that the models have some differences in the computations that they apply to (for example, whether they apply to function computations, search problems or approximation problems),...
the access to data that is allowed (for example, distributed computation models and other local
algorithms models often specify that the input is a graph in adjacency list format), and the re-
quired running time bounds (whether the computation time is required to be independent of the
problem size, or whether it is allowed to have computation time that depends on the problem
size but has sublinear complexity). The model of local computation algorithms described here
attempts to distill the essential common features of the aforementioned concepts into a unified
paradigm.

After formalizing our model, we develop a technique that is specifically suitable for con-
structing polylogarithmic time local computation algorithms. This technique is based on Beck’s
analysis in his algorithmic approach to the Lovász Local Lemma (LLL) [Bec91], and uses the
underlying locality of the problem to construct a solution. All of our constructions must process
the data somewhat so that Beck’s analysis applies, and we use two main methods of performing
this processing.

For maximal independent set computations and scheduling radio network broadcasts, we
use a reduction of Parnas and Ron [PR07]. this reduction shows that, given a distributed net-
work in which the underlying graph has degree bounded by a constant $d$, and a distributed
algorithm (which may be randomized and only guaranteed to output an approximate answer)
using at most $t$ rounds, the output of any specified node of the distributed network can be sim-
ulated by a single processor with query access to the input with $O(d^{t+1})$ queries. For our first
phase, we apply this reduction to a constant number of rounds of Luby’s maximal independent
set distributed algorithm [Lub86] in order to find a partial solution. After a large independent
set is found, in the second phase, we use the techniques of Beck [Bec91] to show that the
remainder of the solution can be determined via a simple polynomial algorithm on sublinear
subproblems.

For hypergraph coloring and $k$-SAT, we show that Alon’s parallel algorithm for finding
solutions guaranteed by the Lovász Local Lemma [Alo91], can be used to give access to poly-
logarithmic sequential time queries regarding the coloring of a given node, or the setting of a
given variable. For most of the queries, this is done by simulating the work of a processor and
its neighbors within a small constant radius. For the remainder of the queries, we use Beck’s
analysis to show that the queries can be solved via the brute force algorithm on very small
subproblems.

Note that parallel $O(\log n)$ time algorithms do not directly yield local algorithms under the
reduction of Parnas and Ron [PR07]. Thus, we do not know how to apply our techniques to all
problems with fast parallel algorithms, including certain problems in the work of Alon whose
solutions are guaranteed by the Lovász Local Lemma and which have fast parallel algorithms
[Alo91]. Recently Moser and Tardos [Mos09, MT10] gave, under some slight restrictions,
polynomial algorithms which find solutions of all problems for which the existence of a solu-
tion is guaranteed by the Lovász Local Lemma [Bec91, Alo91]. It has not yet been resolved
whether one can construct local computation algorithms based on these more powerful algo-

rithms.
1.2 Related work

Our focus on local computation algorithms is inspired by many existing works, which explicitly or implicitly construct such procedures. These results occur in a number of varied settings, including distributed systems, coding theory, and sublinear time algorithms.

1.2.1 Local algorithms in distributed networks

Local computation algorithms are a generalization of local algorithms, which for graph theoretic problems represented in adjacency list format, produce a solution by adaptively examining only a constant sized portion of the input graph near a specified vertex.

Local algorithms have received much attention in the distributed computing literature under a somewhat different model, in which the number of rounds is bounded to constant time, but the computation is performed by all of the processors in the distributed network [NS95, MNS95]. In both our model and the distributed local model, one processor examines a small neighborhood of a vertex. Therefore, although the distributed model is different from our own (sequential) model, previous work on local distributed algorithms is highly relevant to our work in terms of both techniques and results. Several local distributed algorithms are in fact trivially implementable as local computation algorithms, while all constant-time distributed algorithms yield constant-time local computation algorithms via the Parnas-Ron reduction [PR07], upon which we will expand later in this chapter.

Naor and Stockmeyer [NS95] and Mayer, Naor and Stockmeyer [MNS95] investigate the question of what can be computed under these constant-time bounded distributed networks, and show that there are nontrivial problems under these constraints. Naor and Stockmeyer [NS95] investigate locally checkable labeling (LCL) problems, where the legality of a labeling can be checked in constant time (e.g. coloring). Two of their results that are of particular relevance to this paper are: 1) There are non-trivial LCL algorithms with constant-time distributed algorithms (an example is weak coloring on graphs of odd degree, where weak coloring means that every vertex has at least one neighbor colored differently from it), and 2) Adding randomization cannot make an LCL problem local - that is, if there is an LCL problem which can not be solved by a deterministic local algorithm, then it can not be solved by a randomized local algorithm. The Naor-Stockmeyer weak coloring algorithm takes one round to produce a legal weak coloring. This trivially yields a local computation algorithm, as any vertex can calculate its own color in one round. Mayer, Naor and Stockmeyer [MNS95] extend the work of [NS95] and look at distributed networks with faults. They also solve the color reduction problem in constant time - given a coloring of the graph with $c$ colors, recolor each vertex with one of $O(d^2 \log \log c)$ colors.

In his seminal paper, Linial [Lin92] shows that not all LCL problems can be solved in constant time. In particular, he shows a lower bound of $\Omega(\log^* n)$ for distributed 3-coloring an $n$-cycle, proving that the ring-coloring algorithm of Cole and Vishkin [CV86] is asymptotically optimal. Of particular interest to our work is his extension of the result to show the same lower bound to finding a MIS on an $n$-cycle. As a cycle is a 2-regular graph, the lower bound holds for all regular graphs (with degree $\geq 2$). He also shows a recoloring to $O(d^2 \log c)$ in constant time.

Several more recent works investigate local (distributed) algorithms for various problems, including coloring, maximal independent set, dominating set and others: Kuhn et al. [KMNW05]
show how to compute a maximal independent set on growth-bounded graphs in $O(\log d \log^* n)$, with message size $O(\log n)$. (A graph $G$ is $f$-growth-bounded if there is a function $f(r)$ such that every $r$-neighborhood in $G$ contains at most $f(r)$ independent vertices. Their results apply to any growth-bounded graph.) Schneider and Wattenhofer [SW08] show how to find a MIS on growth-bounded graphs in $O(\log^* n)$, matching the lower bound given by Linial. Kuhn, Moscibroda and Wattenhofer [KMW04] give lower bounds to several distributed problems. Of particular relevance to this paper are their lower bounds on the construction of maximal independent sets. They show a lower bound of $\Omega(\frac{\log d \log \log d}{\log \log d})$ for graphs of bounded degree $d$.

Other relevant papers give local algorithms to approximate capacitated maximum dominating sets [KM07], approximate general covering and packing problems using linear programming [KMW06], find a legal multicoloring of a graph [Kuh09], and coloring graphs of bounded degree $d$ with $O(d)$ colors $d$ [KW06, BE09, BE10, SW10].

1.2.2 The Parnas Ron reduction

In [PR07], Parnas and Ron show a simple reduction from local (constant-time) distributed approximation algorithms for the vertex cover problem to sublinear approximation algorithms for this problem. This reduction can be easily modified to reduce general labeling distributed algorithms to local computation algorithms. We present the proof of the following theorem along with the reduction in the appendix (A).

Theorem 1.2.1. ([PR07]) Let $G = (V, E)$ be a distributed network with $n$ nodes and degree at most $d$. Let $D$ be a deterministic distributed algorithm that computes a labeling $D(G)$ in $k$ rounds. Then it is possible, for any node $v \in V$ to compute $D(v)$ in time and space complexity $O(d^k)$ using a single processor, where the algorithm uses only neighbor and degree queries.

Using this theorem, it is easy to transform any of the constant-time distributed algorithms mentioned in Section 1.2.1 to constant-time local computation algorithms. Distributed algorithms that run in $\Omega(\log n)$ cannot be transformed to local computation algorithms using this reduction, as $d^\Omega(\log n)$ is $\Omega(n)$. Note that the parallel algorithms of Luby [Lub86], Alon [Alon91] and Moser and Tardos [MT10] do not automatically yield local algorithms via this transformation since their parallel runtimes are $t = \Omega(\log n)$.

A similar reduction for graphs of bounded average degree is also given in [PR07]. This reduction is then applied to give a sublinear-time approximation for the vertex cover problem on sparse graphs.

1.2.3 Sublinear optimization algorithms on graphs

Sparse graphs

There has been much recent interest among the sublinear time algorithms community in devising local algorithms for problems on constant degree graphs and other sparse optimization problems. The goals of these algorithms have been to approximate quantities such as the optimal vertex cover, maximal matching, maximum matching, dominating set, sparse set cover, sparse packing and cover problems [PR07, KMW06, MR06, NO08, HKNO09, YYI09].

\[1\] A multicoloring of a graph consists of coloring each vertex with a subset of colors
One feature of these algorithms is that they show how to construct an oracle which, for each vertex, returns whether it part of the solution whose size is being approximated – for example, whether it is in the vertex cover or maximal matching. Their results show that this oracle can be implemented in time independent of the size of the graph (depending only on the maximum degree and the approximation parameter). However, because their goal is only to compute an approximation of their quantity, they can afford to err on a small fraction of their local computations. Thus, their oracle implementations give local computation algorithms for finding relaxed solutions to the the optimization problems that they are designed for. For example, Marko and Ron (MR06) study distance testing - estimating the distance from a property (such as having a vertex cover of size C). The distance is measured in the number of edges that need to be added to or removed from a graph G for it to have that property. They show sublinear algorithms for distance testing for connectivity, triangle freeness, vertex cover and other problems. Constructing a local computation algorithm using the oracle designed for estimating the size of the vertex cover [PR07] [MR06] [NO08] yields a vertex cover whose size is guaranteed to be only slightly larger than what is given by the 2-approximation algorithm being emulated – namely, by a multiplicative factor of at most 2 + δ (for any δ > 0). It is interesting to note that Marko and Ron construct a local computation algorithm using the oracle designed for estimating the size of a vertex cover, based on Luby’s algorithm for maximal independent set ([Lub86]). Their algorithm can be easily adapted to compute a maximal independent set - in fact this adaptation is a Parnas-Ron reduction of Luby’s algorithm for k = Θ(log d/δ) rounds, where δ is the error parameter. This yields a large independent, but not necessarily a maximal independent set. We use this adaptation in this paper to compute a large maximal independent set, which we then expand to a maximal independent set using a simple greedy algorithm in Chapter 3.

PageRank

Recently, local algorithms have been demonstrated to be applicable to computations on the web graph. In [JW03, Ber06, SBC+06, ACL06, ABC+08], local algorithms are given which, for a given vertex v in the web graph, computes an approximation to v’s personalized PageRank vector and compute the vertices that contribute significantly to v’s PageRank. In these algorithms, evaluations are made only to the nearby neighborhood of v, so that the running time depends on the accuracy parameters input to the algorithm, but there is no running time dependence on the size of the webgraph. Local graph partitioning algorithms, presented in [ST04, ACL06], find subsets of vertices whose internal connections are significantly richer than their external connections. The running time of these algorithms depends on the size of the cluster that is output, which can be much smaller than the size of the entire graph.

Dense graphs

Though most of the previous examples are for sparse graphs or other problems which have some sort of sparsity, local computation algorithms have also been provided for problems on dense graphs. The property testing algorithms of [GGR98] use a small sample of the vertices (a type of a coreset) to define a good graph coloring or partition of a dense graph. This approach yields local computation algorithms for finding a large partition of the graph and a coloring of the vertices which has relatively few edge violations.
1.2.4 Locally decodable codes and reconstruction problems

The applicability of local computation algorithms is not restricted to combinatorial problems. One algebraic paradigm that has local computation algorithms is that of *locally decodable codes* [KT00], described by the following scenario: Suppose \( m \) is a string with encoding \( y = E(m) \). On input \( x \), which is close in Hamming distance to \( y \), the goal of locally decodable coding algorithms is to provide quick access to the requested bits of \( m \), by looking at a limited (sublinear) number of bits of \( x \).

More generally, the *reconstruction* models described in [ACCL08, CS06, SS10] describe scenarios where a string that has a certain property, such as monotonicity, is assumed to be corrupted at a relatively small number of locations. Let \( P \) be the set of strings that have the property. The reconstruction algorithm gets as input a string \( x \) which is close (in \( L_1 \) norm), to some string \( y \) in \( P \). For various types of properties \( P \), the above works construct algorithms which give fast query access to locations in \( y \).

1.3 Organization

The rest of the paper is organized as follows. In Chapter 2, we present our computation model. Some preliminaries and notations that we use throughout the paper appear in Section 2.2. We then give local computation algorithms for the maximal independent set problem and the radio network broadcast scheduling problem in Chapter 3 and Chapter 4, respectively. In Chapter 5, we show how to use the parallel algorithmic version of the Lovász Local Lemma to give local computation algorithms for finding the coloring of nodes in a hypergraph. Finally, in Chapter 6, we show how to find settings of variables according to a satisfying assignment of a \( k \)-CNF formula.
Chapter 2

The model and preliminaries

2.1 Local Computation Algorithms: the model

We present our model of local computation algorithms for sequential computations of search problems, although computations of arbitrary functions and optimization functions also fall within our framework.

2.1.1 Model Definition

We write \( n = |x| \) to denote the length of the input.

Definition 2.1.1. For input \( x \), let \( F(x) = \{ y \mid y \text{ is a valid solution for input } x \} \). The search problem is to find any \( y \in F(x) \).

In this paper, the descriptions of both \( x \) and \( y \) are almost always assumed to be very large.

Definition 2.1.2 ((\( t, s, \delta \))-local algorithms). Let \( x \) and \( F(x) \) be defined as above. A \((t(n), s(n), \delta(n))\)-local computation algorithm is a (randomized) algorithm which implements query access to an arbitrary \( y \in F(x) \) and satisfies the following: \( A \) gets a sequence of queries \( i_1, \ldots, i_q \) for any \( q > 0 \) and after each query \( i_j \) it must produce an output \( y_{i_j} \) satisfying that the outputs \( y_{i_1}, \ldots, y_{i_q} \) are substrings of some \( y \in F(x) \). The probability of success over all \( q \) queries must be at least \( 1 - \delta(n) \). \( A \) has access to a random tape and local computation memory on which it can perform current computations as well as store and retrieve information from previous computations. We assume that the input \( x \), the local computation tape and any random bits used are all presented in the RAM word model, i.e., \( A \) is given the ability to access a word of any of these in one step. The running time of \( A \) on any query is at most \( t(n) \), which is sublinear in \( n \), and the size of the local computation memory of \( A \) is at most \( s(n) \). Unless stated otherwise, we always assume that the error parameter \( \delta(n) \) is at most some constant, say, \( 1/3 \). We say that \( A \) is a strongly local computation algorithm if both \( t(n) \) and \( s(n) \) are upper bounded by \( \log^c n \) for some constant \( c \).

Definition 2.1.3. Let SLC be the class of problems that have strongly local computation algorithms.

\(^1\)If we only require that the time used by an algorithm \( A \) be bounded (and not the space), we say that mcA is a \((t(n), \delta(n))\)-local computation algorithm.
Note that when $|F(x)| > 1$, the $y$ according to which $A$ outputs may depend on the previous queries to $A$ as well as any random bits available to $A$. Also, we implicitly assume that the size of the output $y$ is upper-bounded by some polynomial in $|x|$. The definition of local-computation algorithms rules out the possibility that the algorithms accomplish their computation by first computing the entire output. Analogous definitions can be made for the bit model. In principle, the model applies to general computations, including function computations, search problems and optimization problems of any type of object, and in particular, the input is not required by the model to be in a specific input format.

The model presented here is intended be more general. Indeed, while designing our model, one of the considerations was to ensure that the models reviewed in Section 1.2, which share many similarities, fall within it. Indeed, sublinear-time local-query models, including but not limited to optimization problems, page-rank approximations, locally decodable codes and reconstruction models can be described within our model. Our model differs from other local computation models in the following ways. First, queries and processing time have the same cost. Second, the focus is on problems with slightly looser running time bound requirements – polylogarithmic dependence on the length of the input is desirable, but sublinear time in the length of the input is often nontrivial and can be acceptable. Third, the model places no restriction on the ability of the algorithm to access the input, as is the case in the distributed setting where the algorithm may only query nodes in its neighborhood (although such restrictions may be implied by the representation of the input). As such, the model may be less appropriate for certain distributed algorithms applications.

**Definition 2.1.4** (Query oblivious). We say an LCA $A$ is **query order oblivious** (query oblivious for short) if the outputs of $A$ do not depend on the order of the queries but depend only on the input and the random bits generated by the random tape of $A$.

### 2.1.2 Relationship with other distributed and parallel models

A challenge that immediately arises is characterizing the problems to which the local-computation algorithm model applies. We start by noting the relationship between problems solvable with local computation algorithms and those solvable with fast parallel or distributed algorithms.

We first observe the relationship between local computation algorithms and problems computable by low depth bounded fan-in circuits.

**Fact 2.1.5.** If $F$ is computable by a circuit family of depth $t(n)$ and fan-in bounded by $d(n)$, then $F$ has a $d(n)^{t(n)}$-local computation algorithm.

**Corollary 2.1.6.** $NC^0 \subseteq SLC$.

In this paper we show solutions to several problems in $NC^1$ via local computation algorithms. However, this is not always possible as:

**Proposition 2.1.7.** $NC^1 \not\subseteq SLC$.

**Proof.** Consider the problem $n$-XOR, the XOR of $n$ inputs. This problem is in $NC^1$. However, no sublinear time algorithm can solve $n$-XOR because it is necessary to read all $n$ inputs. □

In this paper, we give techniques which allow one to construct local computation algorithms based on algorithms for finding certain combinatorial structures whose existence is guaranteed.
by constructive proofs of the LLL in [Bec91, Alo91]. It seems that our techniques do not extend to all such problems. An example of such a problem is Even cycle in a balanced digraph: find an even cycle in a digraph whose maximum indegree is not much greater that the minimum outdegree. Alon [Alo91] shows that, under certain restriction on the input parameters, the problem is in \( NC^1 \). The local analogue of this question is to determine whether a given edge (or vertex) is part of an even cycle in such a graph. It is not known how to solve this quickly.

### 2.1.3 Locality-preserving reductions

In order to understand better which problems can be solved locally, we define locality-preserving reductions, which capture the idea that if problem \( B \) is locally computable, and problem \( A \) has a locality-preserving reduction to \( B \) then \( A \) is also locally computable.

**Definition 2.1.8.** We say that \( A \) is \((t(n), s'(n))\)-locality-preserving reducible to \( B \) via reduction \( H : \Sigma^* \rightarrow \Gamma^* \), where \( \Sigma \) and \( \Gamma \) are the alphabets of \( A \) and \( B \) respectively, if \( H \) satisfies:

1. \( x \in A \iff H(x) \in B \).

2. \( H \) is \((t(n), s'(n), 0)\)-locally computable; that is, every word of \( H(x) \) can be computed by querying at most \( t(n) \) words of \( x \).

**Theorem 2.1.9.** If \( A \) is \((t(n), s'(n))\)-locality-preserving reducible to \( B \) and \( B \) is \((t(n), s(n), \delta(n))\)-locally computable, then \( A \) is \((t(n) \cdot t'(n), s(n) + s'(n), \delta(n))\)-locally computable.

**Proof.** As \( A \) is \((t'(n), s'(n))\)-locality-preserving reducible to \( B \), to determine whether \( x \in A \), it suffices to determine if \( H(x) \in B \). Each word of \( H(x) \) can be computed in time \( t'(n) \) and using space \( s'(n) \), and we need to access at most \( t(n) \) such words to determine whether \( H(x) \in B \). Note that we can reuse the space for computing \( H(x) \). \( \square \)

### 2.2 Preliminaries

All logarithms in this paper are to the base 2. Let \( \mathbb{N} = \{0, 1, \ldots \} \) denote the set of natural numbers. Let \( n \geq 1 \) be a natural number. We use \( [n] \) to denote the set \( \{1, \ldots, n\} \).

Unless stated otherwise, all graphs are undirected. Let \( G = (V, E) \) be a graph. The *distance* between two vertices \( u \) and \( v \) in \( V(G) \), denoted by \( d_G(u, v) \), is the length of a shortest path between the two vertices. We write \( N_G(v) = \{u \in V(G) : (u, v) \in E(G)\} \) to denote the neighboring vertices of \( v \). Furthermore, let \( N^+_G(v) = N(v) \cup \{v\} \). Let \( d_G(v) \) denote the degree of a vertex \( v \). Whenever there is no risk of confusion, we omit the subscript \( G \) from \( d_G(u, v) \), \( d_G(v) \) and \( N_G(v) \). For a subset \( S \subseteq V \), we define \( G|_S \) to be the induced subgraph of \( G \) on \( S \).

The celebrated Lovász Local Lemma plays an important role in our results. We use the simple symmetric version of the lemma.

**Lemma 2.2.1** (Lovász Local Lemma [EL75]). Let \( A_1, A_2, \ldots, A_n \) be events in an arbitrary probability space. Suppose that the probability of each of these \( n \) events is at most \( p \), and suppose that each event \( A_i \) is mutually independent of all but at most \( d \) of other events \( A_j \). If \( ep(d + 1) \leq 1 \), then with positive probability none of the events \( A_i \) holds, i.e.,

\[
\Pr[\cap_{i=1}^n \bar{A}_i] > 0.
\]
Several of our proofs use the following graph theoretic structures, based on [Bec91]:

**Definition 2.2.2.** Let $G = (V, E)$ be an undirected graph. Define $W \subseteq V(G)$ to be a 3-set if the pairwise distances of all vertices in $W$ are each at least 3 and the graph $G^* = (W, E^*)$ is connected, where $E^*$ is the set of edges between each pair of vertices whose distance is exactly 3 in $G$.

**Definition 2.2.3.** Let $G = (V, E)$ be an undirected graph. Define $E^*$ to be the set of edges between each pair of vertices whose distance is exactly 3 in $G$. A 3-tree $T = (W, E')$ is a graph such that $W$ is a 3-set of $G$ and $E'$ is a subset $E^*$ such that $T$ is a tree.

### 2.2.1 $k$-wise independent random variables

Let $1 \leq k \leq n$ be an integer. A distribution $D : \{0, 1\}^n \to \mathbb{R}^\geq 0$ is $k$-wise independent if restricting $D$ to any index subset $S \subset [n]$ of size at most $k$ gives rise to a uniform distribution. A random variable is said to be $k$-wise independent if its distribution function is $k$-wise independent. Recall that the support of a distribution $D$, denoted $\text{supp}(D)$, is the set of points at which $D(x) > 0$. We say a discrete distribution $D$ is symmetric if $D(x) = 1/|\text{supp}(D)|$ for every $x \in \text{supp}(D)$. If a distribution $D : \{0, 1\}^n \to \mathbb{R}^\geq 0$ is symmetric with $|\text{supp}(D)| \leq 2^m$ for some $m \leq n$, then we may index the elements in the support of $D$ by $\{0, 1\}^m$ and call $m$ the seed length of the random variable whose distribution is $D$. We will need the following construction of $k$-wise independent random variables over $\{0, 1\}^n$ with small symmetric sample space.

**Theorem 2.2.4 ([ABIS86]).** For every $1 \leq k \leq n$, there exists a symmetric distribution $D : \{0, 1\}^n \to \mathbb{R}^\geq 0$ of support size at most $n^{\lceil k/2 \rceil}$ and is $k$-wise independent. That is, there is a $k$-wise independent random variable $x = (x_1, \ldots, x_n)$ whose seed length is at most $O(k \log n)$. Moreover, for any $1 \leq i \leq n$, $x_i$ can be computed in space $O(k \log n)$.

Note that if an LCA $A$ does not require total independence and $k$-wise independence suffices, then $A$ can store a seed of length $O(k \log n)$ to generate a random variable of length $n$. In general, if $A$ runs in $\ell$ rounds, then $A$ only needs to store $O(\ell k \log n)$ random bits for all the randomness used for the entire computation. Furthermore, we notice that as long as $k$ is polylogarithmic in $n$, the seed is also polylogarithmic in $n$, and, consequently, so is the space requirement of $A$. 

14
Chapter 3

Maximal Independent Set

An independent set (IS) of a graph $G$ is a subset of vertices such that no two vertices are adjacent. An independent set is called a maximal independent set (MIS) if it is not properly contained in any other IS. It is well-known that a sequential greedy algorithm finds an MIS $S$ in linear time: Order the vertices in $G$ lexicographically - $1, 2, \ldots, n$ (the order can be arbitrary) and initialize $S$ to be the empty set; for $i = 1$ to $n$, if vertex $i$ is not adjacent to any vertex in $S$, add $i$ to $S$. The MIS obtained by this algorithm is call the lexicographically first maximal independent set (LFMIS). Cook \cite{Coo85} showed that deciding if vertex $n$ is in the LFMIS is $P$-complete with respect to logspace reducibility. On the other hand, fast randomized parallel algorithms for MIS were discovered in 1980’s \cite{KW85, Lub86, ABI86}. The best known distributed algorithm for MIS runs in $O(\log^* n)$ rounds with a word-size of $O(\log n)$ \cite{GPS88}. By Theorem 1.2.1 this implies $\log n \cdot d^{O(\log^* n)}$ local computation algorithm. In this section, we give a query oblivious and parallelizable local computation algorithm for MIS based on Luby’s algorithm as well as the techniques of Beck \cite{Bec91}, and runs in time $\text{poly}(d) \cdot \log n \cdot d^{O(d \log d)}$.

Our local computation algorithm is partly inspired by the work of Marko and Ron \cite{MR06}. They simulate the distributed algorithm for MIS of Luby \cite{Lub86} in order to approximate the minimum number of edges one need to remove to make the input graph free of some fixed graph $H$. In addition, they show that similar algorithm can also approximate the size of a minimum vertex cover. We simulate Luby’s algorithm for a constant number of rounds (polynomial in the maximal degree of the graph) to find an independent set, and then run a simple greedy algorithm on the remaining vertices to expand this independent set to a maximal independent set. This idea was inspired by Beck’s approach to hypergraph 2-coloring \cite{Bec91}, and its application to problems such as MIS (and Radio Networks - see Chapter 4) appears to be new.

3.1 Overview of the algorithm

Let $G$ be an undirected graph on $n$ vertices and with maximum degree $d$. On input a vertex $v$, our algorithm decides whether $v$ is in a maximal independent set using two phases. In Phase 1, we simulate Luby’s parallel algorithm for MIS \cite{Lub86} via the reduction of \cite{PR07}. That is, in each round, $v$ tries to put itself into the IS with some small probability. It succeeds if none of its neighbors also tries to do the same. We run our Phase 1 algorithm for $O(d \log d)$ rounds.

\footnote{It is easy to check that the greedy algorithm readily yields a simpler LCA. However, such an LCA is not query oblivious, requires linear space to store answers to previous queries and can not answer queries in parallel.}
It turns out that, after Phase 1, most vertices have been either added to the IS or removed from the graph due to one (or more) of their neighbors being in the IS. Our key observation is that – following a variant of the argument of Beck [Bec91], almost surely, all the connected components of the surviving vertices after Phase 1 have size at most $O(\log n)$. This enables us to perform the greedy algorithm for the connected component $v$ lies in.

As we show in Section 3.3, we can replace the random variable in $\{0, 1\}^n$ in each round by a random variable that is only $O(d^2 \log n)$-wise independent. By Theorem 2.2.4, the seed length of such random variable is $O(d^2 \log^2 n)$. As Phase 1 runs in $O(d \log d)$ rounds, the total space required for the random (seed) tape is therefore at most $\text{poly}(d) \cdot \log^2 n$.

Our main result in this section is the following.

**Theorem 3.1.1.** Let $G$ be an undirected graph with $n$ vertices and maximum degree $d$. Then there is a $(d^O(d \log d), \text{poly}(d) \cdot \log n, \text{poly}(d) \cdot \log^2 n, 1/n)$-local computation algorithm which, on input a vertex $v$, decides if $v$ is in a maximal independent set. Moreover, the algorithm will give a consistent MIS for every vertex in $G$.

### 3.2 Phase 1: Simulating Luby’s parallel algorithm

Figure 3.1 illustrates Phase 1 of our local computation algorithm for Maximal Independent Set. Our algorithm is a variation on Luby’s algorithm for $r = O(d \log d)$ rounds. At the end of the algorithm, every vertex $v$ will be in one of three possible states:

- “in MIS” — $v$ is in the MIS;
- “not in MIS” — one of $v$’s neighbors is selected to be in the MIS and $v$ is therefore not;
- or
- “⊥” — $v$ is not in either of the previous states.

Initially, every vertex is in state “⊥”. Once a vertex enters state “in MIS” or “not in MIS” it remains in that state until the end of Phase 1. For the purpose of analysis of the algorithm, we also initialize every vertex to be uncolored. A vertex becomes colored purple when it chooses itself and none of its neighbors choose themselves in any one round. A vertex is in state “in MIS” if it has been added to the independent set (IS) or “not in MIS” if one of its neighbors has been added to the IS. When a neighbor is in state “⊥” and colors itself purple, it is marked “in MIS” and added to the MIS and all of its neighbors are marked “not in MIS”. At the end of Phase 1, all vertices in states “in MIS” or “not in MIS” are deleted from the graph.

The subroutine $\text{MIS}(v, i)$ returns the state of a vertex $v$ in round $i$. In each round, vertex $v$ “chooses” itself to be in the MIS with probability $1/2d$. At the same time, all its neighboring vertices also flip random coins to decide if they should “choose” themselves.\(^2\) If $v$ is the only vertex in $N^+(v)$ that is chosen in that round, it is colored purple, and if it is unmarked, we add $v$ to the MIS (“select” $v$) and mark all the neighbors of $v$ as “deleted”. However, the state of $v$ in round $i$ is determined not only by the random coin tosses of vertices in $N^+(v)$ but also by these vertices’ states in round $i - 1$. Therefore, to compute $\text{MIS}(v, i)$, we need to recursively call $\text{MIS}(u, i - 1)$ for every $u \in N^+(v)$. Notice that whether a vertex is “selected” or not in

---

\(^2\)We store all the randomness generated by each vertex in each round so that our answers will be consistent. However, we generate the random bits only when the state of corresponding vertex in that round is requested.
any round is dependent on whether or not it is marked, and therefore on previous rounds (and possibly on vertices far away from it). However, whether or not a vertex is colored purple in any round is independent of previous rounds, and in fact independent of the coin flips of any vertices at distance greater than two from it. By induction, the total running time of simulating \( r \) rounds is \( d^{O(r)} = d^{O(d \log d)} \).

If after Phase 1 no vertices remain (that is, all vertices are either “selected” or “deleted”), then the resulting independent set is a maximal independent set. Our main observation is, after simulating Luby’s algorithm for \( O(d \log d) \) rounds, we are already almost surely not far from a maximal independent set. Specifically, if vertex \( v \) returns “⊥” after Phase 1 of the algorithm, we call it a surviving vertex, and consider the subgraph induced on the surviving vertices. Following a variant of Beck’s argument [Bec91], we show that, almost surely, no connected component of surviving vertices is larger than \( \text{poly}(d) \cdot \log n \).

Let \( A_v \) be the event that vertex \( v \) is a surviving vertex. Note that event \( A_v \) depends on the random coin tosses of \( v \) and \( v \)'s neighborhood of radius \( r \) made during the first \( r \) rounds, where \( r = O(d \log d) \). To get rid of the complication caused by this dependency, we consider another set of events. Let \( B_v \) be the event that vertex \( v \) is never colored purple during all \( r \) rounds.

**Claim 3.2.1.** \( A_v \subset B_v \) for every vertex \( v \).

**Proof.** If a vertex is colored purple, then it is either unmarked, in which case it is either “selected” or will be “deleted”. In either case, if a vertex is colored purple at any point, it will not be a surviving vertex. \( \square \)

As a simple corollary, we have

**Corollary 3.2.2.** For any vertex set \( W \subset V(G) \), \( \Pr[\cap_{v \in W} A_v] \leq \Pr[\cap_{v \in W} B_v] \).

**Definition 3.2.3** (dependency graph). A graph \( H \) on the vertices \( V(G) \) is called a dependency graph for \( \{B_v\}_{v \in V(G)} \) if for all \( v \) the event \( B_v \) is mutually independent of all \( B_u \) such that \( (u, v) \notin H \).

**Claim 3.2.4.** The dependency graph \( H \) has maximum degree \( d^2 \).

**Proof.** Since for every vertex \( v \), \( B_v \) depends only on the coin tosses of \( v \) and vertices in \( N(v) \) in each of the \( r \) rounds, the event \( B_v \) is independent of all \( B_u \) such that \( d_G(u, v) \geq 3 \). The claim follows as there are at most \( d^2 \) vertices at distance 1 or 2 from \( v \). (Note that \( B_v \) is not independent of \( B_u \) when \( d_G(u, v) = 2 \), as their mutual neighbor’s choice affects both \( v \) and \( u \).) \( \square \)

**Claim 3.2.5.** For every \( v \in V \), the probability that \( B_v \) occurs is at most \( 1/2d^6 \).

**Proof.** The probability that vertex \( v \) is chosen in round \( i \) is \( \frac{1}{2d} \). The probability that none of its neighbors is chosen in this round is \( (1 - \frac{1}{2d})^d \geq (1 - \frac{1}{2d}) \geq 1/2 \). Since the coin tosses of \( v \) and vertices in \( N(v) \) are independent, the probability that \( v \) is selected in round \( i \) is at least \( \frac{1}{2d} \cdot \frac{1}{2} = \frac{1}{4d} \). We get that the probability that \( B_v \) happens is at most \( (1 - \frac{1}{4d})^{2d \log d} \leq \frac{1}{2d^6} \). \( \square \)

Now we are ready to prove the main lemma for our local computation algorithm for MIS.

**Lemma 3.2.6.** After Phase 1, with probability at least \( 1 - 1/n^2 \), all connected components of the surviving vertices are of size at most \( O(\text{poly}(d) \cdot \log n) \).
**Maximal Independent Set: Phase 1**

Input: a graph $G$ and a vertex $v \in V$

Output: \{“true”, “false”, “⊥”\}

Let $r = 24d \log d$

(a) If $\text{MIS}(v, r) =$ “in MIS”
   return “true”
(b) Else if $\text{MIS}(v, r) =$ “not in MIS”
   return “false”
(c) Else
   return “⊥”

$\text{MIS}(v, i)$

Input: a vertex $v \in V$ and a round number $i$

Output: \{“in MIS”, “not in MIS”, “⊥”\}

1. If $i = 0$ return “⊥”.
2. $v$’s state = $\text{MIS}(v, i - 1)$.
3. For every $u$ in $N(v)$
   If $\text{MIS}(u, i - 1) =$ “in MIS”
   $v$’s state = “not in MIS”.
4. $v$ chooses itself independently with probability $\frac{1}{2d}$
   If $v$ chooses itself
   (i) For every $u$ in $N(v)$
    $u$ chooses itself independently with probability $\frac{1}{2d}$
   (ii) If $v$ doesn’t have a chosen neighbor
    a) $v$ colors itself purple.
    b) If $v$’s state == “⊥”,
    $v$’s state = “in MIS”.
5. return $v$’s state.

Figure 3.1: Local Computation Algorithm for MIS: Phase 1
The proof is based on that of Beck [Bec91].

Before we prove this lemma, we give an overview of the proof. For any set of \( u \) vertices that are connected (under no assumptions of maximal degree – we may assume a complete graph), there are at most \( 4^u \) different possible “shapes” of trees on these vertices. For each possible shape, fix the root. Now find all possible configurations of vertices that can make this tree, given the root. We notice that no possible tree has been unaccounted for. We show that the possibility of any one tree of size \( O(\log n) \) existing is very small, so small that even a union bound gives us a negligible probability of any one tree existing. The trees that we show have a low probability of existing are trees in a graph \( G' \) derived from our original graph \( G \), in which two vertices are connected if the distance between them is three. Finally, we notice that a connected component in \( G \) cannot be much larger than a tree in \( G' \) (and must contain one, as if it is connected, then there are vertices of distance three from each other in it).

**Proof.** Note that we may upper bound the probability that all vertices in \( W \) are surviving vertices by the probability that all the events \( \{B_v\}_{v \in W} \) happen simultaneously:

\[
\Pr[\text{all vertices in } W \text{ are surviving vertices}] = \Pr[\bigcap_{v \in W} A_v] \leq \Pr[\bigcap_{v \in W} B_v].
\]

Let \( S \) be the set of all vertices not selected after Phase 1 of our algorithm. Let \( U \) be an independent set in \( H \). It is immediate that

\[
\Pr[U \subseteq S] \leq \left( \frac{1}{2d^3} \right)^{|U|}.
\]

Let \( V^T \) be a 3-set on \( G \). \(|V^T| = u\). We bound the maximum number of 3-trees on \( G \). The number of non-isomorphic trees on \( u \) vertices is bounded by \( 4^u \) [Lov93]. Therefore, there at most \( 4^u \) possible trees on \( V^T \). Now fix one, \( T \). Label the vertices of \( T \) \( 1, 2, \ldots, u \), such that for \( j > 1 \), \( j \) is adjacent to some \( i < j \). Consider the number of sets of vertices \( v_1, v_2, \ldots, v_u \) which can correspond to this tree. There are \( n \) choices for \( v_1 \). As \( T \) is a 3-tree, the distances between adjacent vertices are exactly 3 on \( G \), and therefore the distance between \( v_1 \) and \( v_2 \) in \( G \) is exactly 3. Therefore, there are at most \( d^3 \) possible vertex choices for \( v_2 \) (given \( v_1 \) is fixed,) and so at most \( n(d^3)^{u-1} \) possible vertex combinations for \( T \). and as there can be at most \( 4^u \) \( T \)'s, there are at most \( n(4d^3)^u \) possible 3-trees on \( G \). For any particular such 3-tree, the probability that none of its vertices were selected after Phase 1 is at most \( \left( \frac{1}{2d^3} \right)^u \). Using a union bound, we see that the expected number of 3-trees of size \( u \) in which no vertices were selected in Phase 1 is

\[
\frac{n(4d^3)^u}{(2d^3)^u} < \frac{n}{d^u}
\]

And so, for \( u = c_1 \log n \), we get that the expected number of 3-trees of size \( u \) is at most \( 1/n^2 \). By Markov’s inequality, with probability at least \( 1 - 1/n^2 \), there is no 3-tree of size larger than \( c_1 \log n \).

We now want to bound the size of a connected component. We note that each (sufficiently large) connected component contains a 3-tree. A maximal 3-tree \( T \) in a component \( C \) must

---

3It is enough to count non-isomorphic trees. For each “shape” of a tree, we then count all the number of ways it can appear in the graph.
have the property that every vertex $v_i \in C$ lies within distance two of a $v_j \in T$. There are less than $d^2$ vertices within distance two from any $v_j \in T$, and so if $C$ contains a 3-tree of size $u$, the size of $C$ is at most $d^2u$. We get that with probability at most $1/n^2$,

$$c_1 \log n > |T| > |C|/d^2$$

$$\Rightarrow |C| < c_2 \log n$$

We get that, with probability $1 - 1/n^2$, the size of the largest connected component of vertices that were not selected in Phase 1 is $O(\log n)$.  

### 3.3 Replacing true randomness with pseudorandomness

A simple while useful observation allows us to reduce the space complexity of the local algorithm from linear to polylogarithmic. Specifically, we can replace the totally independent random bits tossed by the $n$ vertices in each round by a random variable from $\{0, 1\}^n$ that is only $k$-wise independent. When $k = c_2(d^2 + 1) \log n = O(d^2 \log n)$, (where $c_2$ is the constant used in the proof of Lemma 3.2.6), then all the analyses of Phase 1 still hold. Note that we still require that the random bits in each round are independent of previous rounds, i.e., if the algorithm runs in $t$ rounds, then we generate $t$ independent copies of $k$-wise independent random variables.

By Theorem 2.2.4 we can store $t = O(d \log d)$ random seeds each of length $O(d^2 \log n)$ to generate all the randomness needed to simulate Luby’s algorithm for every vertex in the graph. In Claim 3.2.5 the random bits in each round are only required to be $d$-wise independent (each vertex’s choice is independent of its neighbors’). In the proof of Lemma 3.2.6, to make all the events $\{B_v\}_{v \in W}$ completely independent, we only need the random variable to be $(d^2 + 1)|W|$-wise independent, since by Claim 3.2.4 each random variable $B_v$ depends on the random bits of at most $d^2 + 1$ vertices. We showed that $|W| = O(\text{poly}(d) \log n)$. In other words, we do not need to store any answer to previous queries but instead store only the random seeds the algorithm generated on the random tape. In this way, using $O(\text{poly}(d) \cdot \log^2 n)$ space, we still can quickly compute the answer to any query on the fly that is consistent with the random seeds.

### 3.4 Phase 2: Greedy search in the connected component

If $v$ is a surviving vertex after Phase 1, we perform Phase 2 of the algorithm. In this phase, we first explore $v$’s connected component, $C(v)$, in the graph induced on $G$ by all the vertices in state “⊥”. If the size of $C(v)$ is larger than $c_2 \log n$ for some constant $c_2(d)$ depending only on $d$, we abort and output “Fail”. Otherwise, we perform the simple greedy algorithm described at the beginning of this section to find the MIS in $C(v)$. The running time for Phase 2 is at most $O(|C(v)|) \leq \text{poly}(d) \cdot \log n$. Therefore, the total running time of our local computation algorithm for MIS is $d^{O(d \log d)} \cdot \text{poly}(d) \cdot \log n$. 

20
Chapter 4

Radio Networks

For the purposes of this section, a radio network is an undirected graph $G = (V, E)$ with one processor at each vertex. The processors communicate with each other by transmitting messages in a synchronous fashion to their neighbors. In each round, a processor $P$ can either receive a message, send messages to all of its neighbors, or do nothing. We will focus on the radio network that is referred to as a Type II network in [Alo91]: $P$ receives a message from its neighbor $Q$ if $P$ is silent, and $Q$ is the only neighbor of $P$ that transmits in that round. Our goal is to check whether there is a two-way connection between each pair of adjacent vertices. To reach this goal, we would like to find a schedule such that each vertex in $G$ broadcasts in one of $t$ rounds and $t$ is as small as possible.

Definition 4.0.1 (Broadcast function). Let $G = (V, E)$ be an undirected graph. We say $F_r : V \rightarrow [t]$ is a broadcast function for the network $G$ if the following hold:

1. Every vertex $v$ broadcasts once and only once in round $F_r(v)$ to all its neighboring vertices;
2. No vertex receives broadcast messages from more than one neighbor in any round;
3. For every edge $(u, v) \in G$, $u$ and $v$ broadcast in distinct rounds.

Let $\Delta$ be the maximum degree of $G$. Alon et. al. [ABNLP89] [ABNLP92] show that the minimum number of rounds $t$ satisfies $t = \Omega(\Delta \log \Delta)$. Furthermore, Alon [Alo91] gives an $NC_1$ algorithm that computes the broadcast function $F_r$ with $t = \Theta(\Delta \log \Delta)$. Here we give a local computation algorithm for this problem, i.e. given a degree-bounded graph $G = (V, E)$ (with maximum degree $\Delta$) in the adjacency list form and a vertex $v \in V$, we output the round number in which $v$ broadcasts in logarithmic time. Our solution is consistent in the sense that all answers our algorithm outputs to the various $v \in V$ agree with some broadcast scheduling function $F_r$.

Definition 4.0.2 (Square graph). Let $G^{1,2}$ be the “square graph” of $G$; that is, $u$ and $v$ are connected in $G^{1,2}$ if and only if their distance in $G$ is either one or two.

The other model, Type I radio network, is more restrictive: A processor $P$ receives a message from its neighbor $Q$ in a given round only if $P$ is silent, $Q$ transmits and $P$ chooses to receive from $Q$ in that round.

---

1The other model, Type I radio network, is more restrictive: A processor $P$ receives a message from its neighbor $Q$ in a given round only if $P$ is silent, $Q$ transmits and $P$ chooses to receive from $Q$ in that round.
Notice that \( G^{1,2} \) is identical to the dependency graph \( H \) from Chapter 3. Our algorithm is based on finding an independent set cover of \( G^{1,2} \) which simulates Luby’s Maximal Independent Set algorithm \([Lub86]\). Note that if we denote the maximum degree of \( G^{1,2} \) by \( d \), then \( d \leq \Delta^2 \).

**Definition 4.0.3 (Independent Set Cover).** Let \( H = (V, E) \) be an undirected graph. A collection of vertex subsets \( \{S_1, \ldots, S_t\} \) is an independent set cover (ISC) for \( H \) if these vertex sets are pairwise disjoint, each \( S_i \) is an independent set in \( H \) and their union equals \( V \). We call \( t \) the size of ISC \( \{S_1, \ldots, S_t\} \).

**Fact 4.0.4.** If \( \{S_1, \ldots, S_t\} \) is an ISC for \( G^{1,2} \), then the function defined by \( F_r(v) = i \) iff \( v \in S_i \) is a broadcast function.

**Proof.** First note that, since the union of \( \{S_i\} \) equals \( V \), \( F_r(v) \) is well-defined for every \( v \in G \). That is, every \( v \) broadcasts in some round in \([t]\), hence both directions of every edge are covered in some round. As \( v \) can only be in one IS, it only broadcasts once. Second, for any two vertices \( u \) and \( v \), if \( d(u, v) \geq 3 \), then \( N(u) \cap N(v) = \emptyset \). It follows that, if in each round all the vertices that broadcast are at least 3-apart from each other, no vertex will receive more than one message in any round. Clearly the vertices in an independent set of \( G^{1,2} \) have the property that all the pairwise distances are at least 3 (in \( G \)).

The following is a simple fact about ISCs.

**Fact 4.0.5.** For every undirected graph \( H \) on \( n \) vertices with maximum degree \( d \), there is an ISC of size at most \( d \). Moreover, such an ISC can be found by a greedy algorithm in time at most \( O(dn) \).

**Proof.** We repeatedly apply the greedy algorithm that finds an MIS in order to find an ISC. Recall that the greedy algorithm repeats the following until the graph has no unmarked vertex: pick an unmarked vertex \( v \), add it to the IS and mark off all the vertices in \( N(v) \). Clearly each IS found by the greedy algorithm has size at least \( \frac{n}{d+1} \). To partition the vertex set into an ISC, we run this greedy algorithm to find an IS which we call \( S_1 \), and delete all the vertices in \( S_1 \) from the graph. Then we run the greedy algorithm on the new graph again to get \( S_2 \), and so on. After running at most \( d \) rounds (since each round reduces the maximum degree of the graph by at least one), we partition all the vertices into an ISC of size at most \( d \) and the total running time is at most \( O(dn) \).

Our main result in this section is a local computation algorithm that computes an ISC of size \( O(d \log d) \) for any graph of maximum degree \( d \). On input a vertex \( v \), our algorithm outputs the index \( i \) of a vertex subset \( S_i \) to which \( v \) belongs, in an ISC of \( H \). We will call \( i \) the round number of \( v \) in the ISC. By Fact 4.0.4, applying this algorithm to graph \( G^{1,2} \) gives a local computation algorithm that computes a broadcast function for \( G \).

### 4.1 A local computation algorithm for ISC

Our main result for computing an ISC is summarized in the following theorem.
**Theorem 4.1.1.** Let $H$ be an undirected graph on $n$ vertices with maximum degree $d$. Then there is a $(\text{poly}(d) \cdot \log n, \text{poly}(d) \cdot \log^2 n, 1/n)$-local computation algorithm which, on input a vertex $v$, computes the round number of $v$ in an ISC of size at most $O(d \log d)$. Moreover, the algorithm will give a consistent ISC for every vertex in $H$.

On input a vertex $v$, our algorithm computes the round number of $v$ in two phases. In Phase 1 we simulate Luby’s algorithm for MIS [Lub86] for $O(d \log d)$ rounds. In each round, $v$ tries to put itself in the independent set generated in that round. That is, $v$ chooses itself with probability $1/2d$ and if none of its neighbors choose themselves, then $v$ is selected in that round and we output that round number for $v$. As we show shortly, after Phase 1, most vertices will be assigned a round number. We say $v$ survives if it is not assigned a round number. We consider the connected component containing $v$ after one deletes all vertices that do not survive from the graph. Following an argument similar to that of Beck [Bec91], almost surely, all such connected components of surviving vertices after Phase 1 have size at most $O(\log n)$. This enables us, in Phase 2, to perform the greedy algorithm on $v$’s connected component to deterministically compute the round number of $v$ in time $O(\log n)$.

### 4.1.1 Phase 1 algorithm

Phase 1 of our local computation algorithm for computing an ISC is shown in Figure 4.1.

For every $v \in V$, let $A_v$ be the event that vertex $v$ returns “⊥”, i.e. $v$ is not selected after $r$ rounds. We call such a $v$ a surviving vertex. After deleting all $v$ that do not survive from the
graph, we are interested in bounding the size of the largest remaining connected component. Clearly event $A_v$ depends on the random coin tosses of $v$ and $v$’s neighboring vertices in all the $r$ rounds. The following claim is identical to Claim 3.2.5 in Chapter 3, we therefore omit the proof.

**Claim 4.1.2.** For every $v \in V$, the probability that $A_v$ occurs is at most $1/2d^6$.

The following observation is crucial in our local computation algorithm.

**Lemma 4.1.3.** After Phase 1, with probability at least $1 - 1/n$, all connected components of the surviving vertices are of size at most $O(\text{poly}(d) \cdot \log n)$.

**Proof.** The proof is similar to that of Lemma 3.2.6 but is only simpler: we can directly upper bound the probability

$$\Pr[\text{all vertices in } W \text{ are surviving vertices}] = \Pr[\bigcap_{v \in W} A_v]$$

by way of Beck [Bec91] without resorting to any other random process. We omit the proof.

Finally, we apply the same approach as in Section 3.3 to replace the total independent random variables with $k$-wise independent random variables, where $k = O(d^2 \log n)$. By doing so, we reduce the space complexity of our local computation algorithm to $O(\text{poly}(d) \cdot \log^2 n)$.

### 4.1.2 Phase 2 algorithm

If $v$ is a surviving vertex after Phase 1, we perform Phase 2 of the algorithm. In this phase, we first explore the connected component, $C(v)$, that the surviving vertex $v$ lies in. If the size of $C(v)$ is larger than $c_2 \log n$ for some constant $c_2(d)$ depending only on $d$, we abort and output “Fail”. Otherwise, we perform the simple greedy algorithm described in Fact 4.0.5 to partition $C(v)$ into at most $d$ subsets deterministically. The running time for Phase 2 is at most $O(\text{poly}(d) \cdot \log n)$. Since any independent set of a connected component can be combined with independent sets of other connected components to form an IS for the surviving vertices, we conclude that the total size of ISC we find is $O(d \log d + d) = O(d \log d)$.

### 4.2 Discussions

Now a simple application of Theorem 4.1.1 to $G^{1,2}$ gives a local computation algorithm for the broadcast function.

**Theorem 4.2.1.** Given a graph $G = (V, E)$ with $n$ vertices and maximum degree $\Delta$ and a vertex $v \in V$, there exists a $(\text{poly}(\Delta) \cdot \log n, \text{poly}(\Delta) \cdot \log^2 n, 1/n)$-local computation algorithm that computes a broadcast function with at most $O(\Delta^2 \log \Delta)$ rounds. Furthermore, the broadcast function it outputs is consistent for all queries to the vertices of the graph.

We note our round number bound is quadratically larger than that of Alon’s parallel algorithm [Alo91]. We do not know how to turn his algorithm into a local computation algorithm.
Chapter 5

Hypergraph Two-Coloring

A hypergraph $H$ is a pair $H = (V, E)$ where $V$ is a finite set whose elements are called nodes or vertices, and $E$ is a family of non-empty subsets of $V$, called hyperedges. A hypergraph is called $k$-uniform if each of its hyperedges contains precisely $k$ vertices. A two-coloring of a hypergraph $H$ is a mapping $c : V \to \{\text{red, blue}\}$ such that no hyperedge in $E$ is monochromatic. If such a coloring exists, then we say $H$ is two-colorable. We assume that each hyperedge in $H$ intersects at most $d$ other hyperedges. Let $N$ be the number of hyperedges in $H$. Here we think of $k$ and $d$ as fixed constants and all asymptotic forms are with respect to $N$. By the Lovász Local Lemma, when $e(d + 1) \leq 2^{k-1}$, the hypergraph $H$ is two-colorable.

Let $m$ be the total number of vertices in $H$. Note that $m \leq kN$, so $m = O(N)$. For any vertex $x \in V$, we use $\mathcal{E}(x)$ to denote the set of hyperedges $x$ belongs to. For convenience, for any hypergraph $H = (V, E)$, we define an $m$-by-$N$ vertex-hyperedge incidence matrix $\mathcal{M}$ such that, for any vertex $x$ and hyperedge $e$, $\mathcal{M}_{x,e} = 1$ if $e \in \mathcal{E}(x)$ and $\mathcal{M}_{x,e} = 0$ otherwise. A natural representation of the input hypergraph $H$ is this vertex-hyperedge incidence matrix $\mathcal{M}$. Moreover, since we assume both $k$ and $d$ are constants, the incidence matrix $\mathcal{M}$ is necessarily very sparse. Therefore, we further assume that the matrix $\mathcal{M}$ is implemented via linked lists for each row (that is, vertex $x$) and each column (that is, hyperedge $e$).

Let $G$ be the dependency graph of the hyperedges in $H$. That is, the vertices of the undirected graph $G$ are the $N$ hyperedges of $H$ and a hyperedge $E_i$ is connected to another hyperedge $E_j$ in $G$ if $E_i \cap E_j \neq \emptyset$. It is easy to see that if the input hypergraph is given in the above described representation, then we can find all the neighbors of any hyperedge $E_i$ in the dependency graph $G$ (there are at most $d$ of them) in $O(\log N)$ time.

5.1 Our main result

A natural question to ask is: Given a two-colorable hypergraph $H$, and a vertex $v$, can we quickly compute the coloring of $v$ in bounded time? Our main result in this section is, given a two-colorable hypergraph $H$ whose two-coloring scheme is guaranteed by the Lovász Local Lemma (with slightly weaker parameters), we give a local computation algorithm which answers queries of the coloring of any single vertex in $\text{polylog}N$ time, where $N$ is the number of the hyperedges in $H$. The coloring returned by our oracle will agree with some (legal) two-coloring of the hypergraph with probability at least $1 - o(1)$.

**Theorem 5.1.1.** Let $d$ and $k$ be such that there exist three positive integers $k_1, k_2$ and $k_3$ such
that the following hold:

\[ k_1 + k_2 + k_3 = k, \]
\[ 16d(d-1)^3(d+1) < 2^{k_1}, \]
\[ 16d(d-1)^3(d+1) < 2^{k_2}, \]
\[ 2e(d+1) < 2^{k_3}. \]

Then there exists a \((\text{polylog } N, 1/N)\)-local computation algorithm which, given a hypergraph \(H\) and any sequence of queries to the colors of vertices \((x_1, x_2, \ldots, x_s)\), returns a consistent coloring for all \(x_i\)’s which agrees with some two-coloring of \(H\).

### 5.2 Overview of the coloring algorithm

Our local computation algorithm imitates the parallel coloring algorithm of Alon [Alo91]. Recall that Alon’s algorithm runs in three phases. In the first phase, we randomly color each vertex in the hypergraph following some arbitrary ordering of the vertices. If some hyperedge has \(k_1\) vertices in one color and no vertices in the other color, we call it a dangerous edge and mark all the remaining vertices in that hyperedge as troubled. These troubled vertices will not be colored in the first phase. After Phase 1, we delete all hyperedges which have been assigned both colors and call the remaining hyperedges surviving edges. Now we repeat the same process for the surviving hyperedges, but this time a hyperedge becomes dangerous if \(k_1 + k_2\) vertices are colored the same and no vertices are colored by the other color. Finally, in the third phase, we do a brute-force search for a coloring in each of the connected components of the surviving vertices as they are of size \(O(\log \log N)\) almost surely.

We construct a local computation algorithm which simulates Alon’s algorithm and takes the ordering of the vertices as they are queried.

If the coloring of a vertex \(x\) can not be determined in the first phase (i.e. when we query \(x\), it is troubled), we explore the dependency graph of the hyperedges containing \(x\) and find the connected component of the surviving hyperedges to perform the second phase coloring. To ensure that all the connected components of surviving hyperedges resulting from the second phase coloring are of small sizes, we repeat the second phase colorings independently many times until the connected components sizes are small enough. If the coloring of \(x\) has not been decided by the second phase (i.e. \(x\) was marked as troubled in Phase 2 as well), then we run the third (and final) phase of coloring, in which we exhaustively search for a two-coloring for vertices in \(x\)’s connected component, which we are guaranteed is (almost surely) very small (i.e., of size at most \(O(\log \log N)\)). Following Alon’s analysis, we will show that with probability at least \(1 - 1/N\), the total running time of all these three phases for any vertex in \(H\) is \(\text{polylog } N\).

During the execution of the algorithm, each hyperedge will be in either initial, safe, unsafe-1, unsafe-2, dangerous-1 or dangerous-2 state. Vertices will be in either achromatic, red, blue, troubled-1 or troubled-2 state. If a vertex’s state is neither red nor blue, we say that it is uncolored. Initially every hyperedge is in initial state and every vertex is in achromatic state.
Phase-1 Coloring($x$)  
Input: a vertex $x \in V$  
Output: a color in \{red, blue\}  
1. If $x$ is already colored 
   Return the color of $x$  
2. If $x$ is in troubled-1 state 
   Return Phase-2 Coloring($x$)  
3. If $x$ is in troubled-2 state 
   Return Phase-3 Coloring($x$)  
4. If $x$ is in achromatic state 
   (a) Uniformly at random choose a color $c$ for $x$ from \{red, blue\}  
   (b) Update the states of all hyperedges in $\mathcal{E}(x)$  
   (c) Return color $c$  

Figure 5.1: Phase 1 coloring algorithm

5.3 Phase 1 coloring

If $x$ is already colored (that is, $x$ is in either red or blue state), then we simply return that color. If $x$ is in the troubled-1 state, we invoke Phase 2 coloring for vertex $x$. If $x$ is in the troubled-2 state, we invoke Phase 3 coloring for vertex $x$. If $x$ is achromatic, then we flip a fair coin to color $x$ red or blue with equal probability (that is, vertex $x$’s state becomes red or blue, respectively). After that, we update the status of all the hyperedges in $\mathcal{E}(x)$. Specifically, if some $E_i \in \mathcal{E}(x)$ has $k_1$ vertices in one color and no vertices in the other color, then we change $E_i$’s state from initial to dangerous-1. Furthermore, all achromatic vertices in $E_i$ will be changed to troubled-1 states. On the other hand, if both colors appear among the vertices of $E_i$, we update the state of $E_i$ from initial to safe. If none of the vertices in a hyperedge is achromatic and the hyperedge is still in initial state (that is, it is neither safe or dangerous-1), then we change its state to unsafe-1. Note that if a hyperedge is unsafe-1 then all of its vertices are either colored or in troubled-1 state, and the colored vertices are monochromatic.

Running time analysis. The running time of Phase 1 coloring for an achromatic vertex $x$ is $O(kd) = O(1)$ (recall that we assume both $k$ and $d$ are constants). This is because vertex $x$ can belong to at most $d + 1$ hyperedges, hence there are at most $k(d + 1)$ vertices that need to be updated during Phase 1. If $x$ is already a colored vertex, the running time is clearly $O(1)$. Finally, the running time of Phase 1 coloring for a troubled-1 or troubled-2 vertex is $O(1)$ plus the running time of Phase 2 coloring or $O(1)$ plus the running time of Phase 3 coloring, respectively.

5.4 Phase 2 coloring

During the second phase of coloring, given an input vertex $x$ (which is necessarily in state troubled-1), we first explore the dependency graph $G$ of the hypergraph $H$ by coloring some other vertices whose colors may have some correlation with the coloring of $x$. In doing so, we grow a connected component of surviving-1 hyperedges containing $x$ in $G$. Here, a hyperedge is called surviving-1 if it is either dangerous-1 or unsafe-1. We denote this connected component
Phase-2 Coloring($x$)
Input: a troubled-1 vertex $x \in V$
Output: a color in \{red, blue\} or FAIL
1. Start from $\mathcal{E}(x)$ to explore $G$ in order to find the connected components of all the surviving-1 hyperedges around $x$
2. If the size of the component is larger than $c_1 \log N$
   Abort and return FAIL
3. Repeat the following until a good coloring is found
   (a) Color all the vertices in $C_1(x)$
   (b) Explore the dependency graph of $G|_{\mathcal{S}_1(x)}$
   (c) Check if the coloring is good
4. Return the color of $x$ in the good coloring

Figure 5.2: Phase 2 coloring algorithm

of surviving-1 hyperedges surrounding vertex $x$ by $C_1(x)$.

Growing the connected component. Specifically, in order to find out $C_1(x)$, we maintain a set of hyperedges $\mathcal{E}_1$ and a set of vertices $V_1$. Throughout the process of exploring $G$, $V_1$ is the set of uncolored vertices that are contained in some hyperedge in $\mathcal{E}_1$. Initially $\mathcal{E}_1 = \mathcal{E}(x)$. Then we independently color each vertex in $V_1$ red or blue uniformly at random. After coloring each vertex, we update the state of every hyperedge that contains the vertex. That is, if any hyperedge $E_i \in \mathcal{E}_1$ becomes safe, then we remove $E_i$ from $\mathcal{E}_1$ and remove from $V_1$ all the vertices that are only contained in $E_i$. On the other hand, once a hyperedge in $\mathcal{E}_1$ becomes dangerous-2 (it has $k_1 + k_2$ vertices in one colour and none in the other), all the uncolored vertices in that hyperedge become troubled-2 and we skip the coloring of all such vertices. After the coloring of all vertices in $V_1$, hyperedges in $\mathcal{E}_1$ are surviving hyperedges. Then we check all the hyperedges in $G$ that are adjacent to the hyperedges in $\mathcal{E}_1$. If any of these hyperedges is not in the safe state, then we add it to $\mathcal{E}_1$ and also add all its uncolored vertices to $V_1$. Now we repeat the coloring process described above for these newly added uncolored vertices to $V_1$. This exploration of the dependency graph terminates if, either there are no more hyperedges to color, or the cumulative number of hyperedges in $\mathcal{E}_1$ is greater than $c_1 \log N$, where $c_1$ is some absolute constant. The following Lemma shows that, almost surely, the size of $C_1(x)$ is at most $c_1 \log N$.

Lemma 5.4.1 ([Alo91], [AS00]). Let $S \subseteq G$ be the set of surviving hyperedges after the first phase. Then with probability at least $1 - O\left(\frac{1}{N^2}\right)$ (over the choices of random coloring), all connected components $C_1(x)$ of $G|_S$ have size at most $c_1 \log N$.

The original lemma which appears in [Alo91] shows that the size of all connected components is smaller than $c_1 \log N$ with probability greater than $1/2$. In [AS00], the probability is shown to be $o(1)$.

Proof [Sketch]: The proof is very similar to the proof of Lemma 3.2.6. It is shown that the probability of any hyperedge to be dangerous is at most $2^{1-k_1}$, because for an edge to become dangerous, $k_1$ of its vertices must be colored the same color. For any set $U$ of size

\[That is, all the hyperedges that were addressed in Phase 2.\]
$u$ of hyperedges, the probability that they all survive is at most $(2^{1-k_1})^u$. Now we look at edges that are mutually at distance at least 4 from each other, and notice that an edge to survive it is necessary for it or one of its neighbors to be dangerous, so, as each edge $e \in U$ has at most $d$ neighbors, we get that the probability of all edges in $U$ surviving is at most $(d + 1)^u(2^{1-k_1})^u$. The number of 4-trees (defined analogously to the definition of 3-trees - see Definition 2.2.3) is bounded by $N(4d(d-1)^3)^u$, as there at at most $4^u$ non-isomorphic trees on a set of $u$ vertices, and, fixing a tree and a root, there are at most $D$ choices for each vertex if we build the tree in some order. We get that the expected number of 4-trees on $G$ is $N[d(d-1)^3(d+1)2^{1-k_1}]^u \leq N[8d(d-1)^3(d+1)2^u]u$. The bracketed term is, by assumption, less than $1/2$, and so for a large enough choice of $c_1$, we get that with probability at least $1 - O(\frac{1}{N^2})$ (over the choices of random coloring), all connected components $C_1(x)$ of $G_{|S}$ have size at most $c_1 \log N$. 

**Random coloring.** Since $C_1(x)$ is not connected to any surviving-1 hyperedges in $H$, we can color the vertices in the connected component $C_1(x)$ without considering any other hyperedges that are outside $C_1(x)$. Now we follow a similar coloring process as in Phase 1 to color the vertices in $C_1(x)$ uniformly at random and in an arbitrary ordering. The only difference is, we ignore all the vertices that are already colored red or blue, and if $k_1 + k_2$ vertices in a hyperedge get colored monochromatically, and all the rest of vertices in the hyperedge are in troubled-1 state, then this hyperedge will be in dangerous-2 state and all the uncolored vertices in it will be in troubled-2 state. Analogously we define unsafe-2 hyperedges as hyperedges whose vertices are either colored or in troubled-2 state and all the colored vertices are monochromatic. Finally, we say a hyperedge is a surviving-2 edge if it is in either dangerous-2 state or unsafe-2 state.

Let $S_1(x)$ be the set of surviving hyperedges in $C_1(x)$ after all vertices in $C_1(x)$ are either colored or in troubled-2 state. Now we explore the dependency graph of $S_1(x)$ to find out all the connected components. Another application of Lemma 5.4.1 to $G_{|S_1(x)}$ shows that with probability at least $1 - O(\frac{1}{\log^2 N})$ (over the choices of random coloring), all connected components in $G_{|S_1(x)}$ have size at most $c_2 \log \log N$, where $c_2$ is some constant. We say a Phase 2 coloring is good if this condition is satisfied. Now if a random coloring is not good, then we erase all the coloring performed during Phase 2 and repeat the above coloring and exploring dependency graph process. We keep doing this until we find a good coloring. Therefore, in expected polylog$N$ time we can color $C_1(x)$ such that each connected component in $G_{|S_1(x)}$ has size at most $c_2 \log \log N$.

**Running time analysis.** Combining the analysis above with an argument similar to the running time analysis of Phase 1 coloring gives

**Claim 5.4.2.** Phase 2 coloring can be done in expected polylog$N$ time.

**Proof.** By Lemma 5.4.1 after Phase 1 is completed, with probability at least $1 - O(1/N^2)$, the sizes of all connected components are at most $c_1 \log N$. However, unlike the Alon algorithm, we do not actually carry out Phase 1 coloring for the whole hypergraph before starting Phase 2 coloring. Fortunately, the Lemma applies to an arbitrary ordering of the vertices. Once we enter Phase 2, we color uncolored vertices in $E(x)$ and add neighboring hyperedges until all the neighboring hyperedges of our connected component are in safe state. By Lemma 5.4.1, this exploration terminates almost surely after we visited at most $O(\log N)$ hyperedges, as each hyperedge intersects at most $d$ other hyperedges. Consequently, finding the connected component in the dependency graph involves updating at most $O(k \cdot \log N) = O(\log N)$ vertices (so that
Phase-3 Coloring($x$)
Input: a troubled-2 vertex $x \in V$
Output: a color in \{red, blue\}
1. Start from $\mathcal{E}(x)$ to explore $G$ in order to find the connected component of all the surviving-2 hyperedges around $x$
2. Go over all possible colorings of the connected component and color it using a feasible coloring.
3. Return the color $c$ of $x$ in this coloring.

Figure 5.3: Phase 3 coloring algorithm

they become either colored or troubled-1), as all the hyperedges are $k$-uniform. (Note that as $x$ is the first vertex queried in the connected component, it is guaranteed to be colored first and so at the end of Phase 2, $x$ is colored.)

5.5 Phase 3 coloring

In Phase 3, given a vertex $x$ (which is necessarily troubled-2), we grow a connected component which includes $x$ as in Phase 2, but of surviving-2 hyperedges. Denote this connected component of surviving-2 hyperedges by $C_2(x)$. By our Phase 2 coloring, the size of $C_2(x)$ is no greater than $c_2 \log \log N$. We then color the vertices in this connected component by exhaustive search. The existence of such a coloring is guaranteed by the Lovász Local Lemma (Lemma 2.2.1).

Claim 5.5.1. The time complexity of Phase 3 coloring is at most polylog $N$.

Proof. Using the same analysis as for Phase 2, in time $O(\log \log N)$ we can explore the dependency graph to grow our connected component of surviving-2 hyperedges. Exhaustive search of a valid two-coloring of all the vertices in $C_2(x)$ takes time at most $2^{O(|C_2(x)|)} = 2^{O(\log \log N)} = \text{polylog} N$, as $|C_2(x)| \leq c_2 \log \log N$ and each hyperedge contains $k$ vertices.

Finally, we remark that using the same techniques as those in [Alo91], we can make our local computation algorithm run in parallel and find an $\ell$-coloring of a hypergraph for any $\ell \geq 2$ (an $\ell$-coloring of a hypergraph is to color each vertex in one of the $\ell$ colors such that each color appears in every hyperedge).
Chapter 6

k-CNF

As another example, we show our hypergraph coloring algorithm can be easily modified to compute a satisfying assignment of a $k$-CNF formula, provided that the latter satisfies some specific properties.

Let $H$ be a $k$-CNF formula on $m$ Boolean variables $x_1, \ldots, x_m$. Suppose $H$ has $N$ clauses: $H = A_1 \land \cdots \land A_N$ and each clause consists of exactly $k$ distinct literals. We say two clauses $A_i$ and $A_j$ intersect with each other if they share some variable (or the negation of that variable). As in the case for hypergraph coloring, $k$ and $d$ are fixed constants and all asymptotics are with respect to the number of clauses $N$ (and hence $m$, since $m \leq kN$). Our main result is the following.

**Theorem 6.0.2.** Let $H$ be a $k$-CNF formula with $k \geq 2$. If each clause intersects no more than $d$ other clauses and furthermore $k$ and $d$ are such that there exist three positive integers $k_1, k_2$ and $k_3$ satisfying the followings relations:

\[ k_1 + k_2 + k_3 = k, \]
\[ 8d(d - 1)^3(d + 1) < 2^{k_1}, \]
\[ 8d(d - 1)^3(d + 1) < 2^{k_2}, \]
\[ c(d + 1) < 2^{k_3}, \]

then there exists a local computation algorithm that, given any sequence of queries to the truth assignments of variables $(x_1, x_2, \ldots, x_s)$, with probability at least $1 - 1/N^2$, returns a consistent truth assignment for all $x_i$’s which agrees with some satisfying assignment of the $k$-CNF formula $H$. Moreover, the algorithm answers each single query in expected $O((\log N)^c)$ time, where $c$ is some constant (depending only on $k$ and $d$).

**Proof [Sketch]:**

*Relationship to hypergraph two-coloring:* We follow a similar algorithm to that of hypergraph two-coloring as presented in Chapter 5. The algorithm is virtually identical and the only difference is that while in hypergraph two-coloring we require that there be at least one vertex with each color in each hyperedge, here we require that every clause has at least one literal in the *true* state. The analogy to hypergraph two-coloring is straightforward: each clause is

---

1 Our algorithm works for the case that each clause has at least $k$ literals; for simplicity, we assume that all clauses have uniform size.
analogous to a hyperedge; each variable is analogous to a vertex. The main difference is that a variable’s assignment can evaluate to either true or false depending on whether its literal is positive or negative. Furthermore, as stated before, we require that there is at least one literal whose assignment evaluates to $true$. This is analogous to requiring that every hyperedge has at least one variable colored red, which is a weaker requirement, therefore the analysis of the hypergraph two-coloring algorithm holds for $k$-CNF. There is one last issue we must address: in hypergraph two-coloring, each vertex is assigned a single color, while here, although each variable is assigned a single value, each variable can be present in either a positive or a negative literal, and so its evaluation may be either $true$ or $false$. Fortunately, at no point in the hypergraph two-coloring analysis did we require the assumption or use the property that $x$ is colored the same in each hyperedge. The analysis holds even if each vertex would receive a different color in each hyperedge, as the coloring is random. Therefore, the analysis for hypergraph two-coloring holds for $k$-CNF.

Algorithm and analysis outline: Every clause will be in one of the following states: initial, safe, unsafe-1, unsafe-2, dangerous-1 or dangerous-2, and every variable will be in one of the following states: unassigned, true, false, troubled-1 or troubled-2. Initially every clause is in the initial state and every variable is in the unassigned. Suppose we are asked about the value of a variable $x_i$. If $x_i$ is in the unassigned state, we randomly choose from $\{true, false\}$ with equal probabilities and assign it to $x_i$. Then we update all the clauses that contain either $x_i$ or $\bar{x}_i$ accordingly: if the clause is already evaluated to true by this assignment of $x_i$, then we mark the clause as safe; if the clause is in the initial state and is not safe yet and $x_i$ is the $k_1$th literal in the clause to be assigned a value, then the clause is marked as dangerous-1 and all the remaining unassigned variables in that clause are now in troubled-1 state. We perform similar operations for clauses in other states as we do for the hypergraph coloring algorithm. The only difference is now we have $\Pr[A_i \text{ becomes dangerous-1}] = 2^{-k_1}$, instead of $2^{1-k_1}$ as in the hypergraph coloring case (in hypergraph 2 coloring a vertex becomes dangerous if all $k_1 - 1$ subsequent colors are the same as the the first; in the case of $k$-CNF, we require that all assignments evaluate to false.) Following the same analysis, almost surely, all connected components in the dependency graph of survived-1 clauses are of size at most $O(\log N)$ and almost surely all connected components in the dependency graph of survived-2 clauses are of size at most $O(\log \log N)$, which enables us to do exhaustive search to find a satisfying assignment. □
In this paper we propose a model of local computation algorithms and give some techniques which can be applied to construct local computation algorithms with polylogarithmic time and space complexities. It would be interesting to better understand the scope of problems which can be solved with such algorithms and to develop other techniques that would apply in this setting.

Immediate examples are: Under what conditions can an algorithmic LLL be turned into an LCA? In particular, can one turn the Moser-Tardos algorithms into a local approximation ones? Under what conditions can a distributed or parallel algorithm be turned into an LCA?

Finally, although our computation solutions to hypergraph 2-coloring and $k$-CNF achieve polylogarithmic time complexity, they give no guarantees for sublinear space. We conjecture that our solutions can be modified to perform in polylogarithmic space, using a random order for the queries instead of taking the queries in the order in which they come.
Appendix A

Proof of the Parnas Ron reduction

**Theorem A.0.3.** ([PRO7]) Let \( G = (V, E) \) be a distributed network with \( n \) nodes and degree at most \( d \). Let \( D \) be a deterministic distributed algorithm that computes in \( k \) rounds a labeling \( D(G) \). Then it is possible, for any node \( v \in V \) to compute \( D(v) \) in time and space complexity \( O(d^k) \) using a single processor, where the algorithm uses only neighbor and degree queries.

**Proof.** The simple reduction is shown in Figure A.1. The important observation is that for any vertex \( v \), if we run algorithm \( D \) on the subgraph \( G_k(v) \) (the graph induced on \( G \) by the \( k \)-neighborhood of \( v \)), it makes the same decision about vertex \( v \) as if we had run \( D \) for \( k \) rounds on \( G \). This is because no information from outside \( G_k(v) \) can reach \( v \) in \( k \) rounds. Therefore \( D(v) \) can only depend on the running of \( D \) on \( G_k(v) \) in \( k \) rounds. The neighborhood \( G_k(v) \) can be constructed in time and space \( O(d^k) \) by building a BFS tree from \( v \). Then we can run \( D \) on the subgraph induced by this tree, where the degrees of the leaves is the same as in \( G \). This requires \( O(d^k) \) degree queries.

---

**Reduction Algorithm**

Input: a graph \( G \), a vertex \( v \in V \) and a distributed labeling algorithm \( D \)

Output: \( D(v) \)

1. Consider the graph \( G_k(v) \) induced by the \( k \)-neighborhood of \( v \).
2. Run algorithm \( D \) on \( G_k(v) \), where the degrees of vertices that are a distance \( k \) from \( v \) are as they are in \( G \).
3. Output \( D(v) \).

Figure A.1: Parnas-Ron reduction
Bibliography


