Efficient Distributed Algorithms on Random Graphs

by

Reza Fathi

A dissertation submitted to the Department of Computer Science,

College of Natural Sciences and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in Computer Science

Chair of Committee: Gopal Pandurangan, University of Houston

Committee Member: Edgar Gabriel, University of Houston

Committee Member: Jaspal Subhlok, University of Houston

Committee Member: Anisur Rahaman Molla, Indian Statistical Institute

University of Houston

December 2019

Acknowledgment

I am eternally grateful for being supervised by Professor Gopal Pandurangan and his invaluable guidance during my PhD research. He sees beyond the horizon and can easily connect the dots. I also extend my deepest thanks to Professor Ernst Leiss for his believing in me, his patience and advice during my research under his guidance. I feel blessed for being provided the chance to stand on the shoulders of giants!

I also would like to thank the rest of my dissertation committee members: Dr. Edgar Gabriel, Dr. Jasphal Sublhok, and Dr. Anisur Rahaman Molla for their insightful comments and feedback.

I am extremely grateful to my collaborators Dr. Anisur Rahaman Molla, Mohsen Amini Salehi, Morteza Mehrnoush, and Puya Ghazizadeh. I am grateful for my dear friend Nguyen Dinh Pham for all the discussions, collaborations, and moments we had. I also extend my thanks to Dr. Soumyottam Chatterjee, Michele Scquizzato, and Robert Gmyr for our discussions.

EFFICIENT DISTRIBUTED ALGORITHMS ON RANDOM GRAPHS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Reza Fathi

December 2019

Abstract

This dissertation focuses on two prominent graph problems: finding Hamiltonian cycles and detecting communities in graphs. Both of them are NP-hard problems on general graphs but can admit efficient solutions in *random graphs*. In this dissertation, we present efficient distributed algorithms for the above two problems in random graphs.

First, we present fast and efficient randomized distributed algorithms to find Hamiltonian cycles in random graphs. In particular, we design and analyze a randomized distributed algorithm for the classical $G(n, p)$ random graph model, with number of nodes n and $p = \frac{c \ln n}{n^\delta}$ (for any constant $0 < \delta \leq 1$ and for a suitably large constant $c > 0$), that finds a Hamiltonian cycle with high probability in $\tilde{O}(n^\delta)$ rounds.[1] Our algorithm works in the (synchronous) CONGEST model (i.e., only $O(\log n)$-sized messages are communicated per edge per round) and its computational cost per node is sublinear (in n) per round and is fully-distributed (each node uses only $o(n)$ memory and all nodes' computations are essentially balanced). Our algorithm improves over the previous best known result in terms of both the running time as well as the edge sparsity of the graphs where it can succeed; in particular, the denser the random graph, the smaller is the running time.

Second, we present a distributed algorithm for community detection in the *stochastic block model* (also called *planted partition model*), a widely-studied and canonical random graph model for community detection and clustering. Designing effective algorithms for community detection is an important and challenging problem in *large-scale* graphs, studied extensively in the literature. Various solutions have been proposed, but many of them are centralized with expensive procedures (requiring full knowledge of the input graph) and have a large running time. Our algorithm called *CDRW(Community Detection by Random Walks)* is based on random walks, is localized and lightweight, and is easy to implement. A novel feature of the algorithm is that it uses the concept of *local mixing time* to identify the community around a given node. We also present experimental results for our CDRW algorithm that validate our theoretical analysis.

[1] The notation $\tilde{O}$ hides a polylog(n) factor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs [17] appear when there is a set of entities and there are connections or interactions between them. A graph is represented by $G(V, E)$ where V and E are the set of nodes and edges, respectively. Nodes represent entities and edges show the interaction between them. For example, in the Web graph, nodes represent webpages and edges are hyperlinks between them. In social networks such as Facebook, nodes represent people and edges represent friendship between them. In collaboration networks such as DBLP, nodes represent authors and two authors are connected if they coauthor a paper (see Stanford Large Network Dataset Collection [66]). In some other datasets, data points can be connected based on their location, interest, or occupations. The edges between nodes can be directed or undirected. A graph is called directed if all its edges are directed, otherwise it is undirected. In this dissertation, we consider undirected graphs and propose distributed algorithms to solve two fundamental graph problems called finding Hamiltonian cycles (see Chapter 3) and detecting communities (see Chapter 4).

The traditional study of graph algorithms was focused on proposing algorithms to find solutions to a specific problem on a given graph which usually work on somewhat small graphs ($|V|$ is relatively small). Indeed, much of those algorithms are centralized and work

on small general graphs. Nowadays we face large and complex graphs such as the World Wide Web, social networks like Facebook, and many others. These large scale graphs cannot be processed fast and efficiently by using centralized algorithms. Hence, devising distributed algorithms that run on large general graphs has been a focus in recent years [58, 83, 84].

When the size of a graph is large, we are also interested in the statistical properties of graphs. *Random Graphs* serve as models to capture certain properties of real-world graphs. While not all random graph models capture all properties of real-world graphs accurately, they serve as useful models that provides insight on the properties of graphs. Therefore, random graphs have applications in all areas in which real-world networks are encountered. One important application in computer science is they can be used to design and rigorously analyze algorithms. Questions such as computing shortest paths, minimum spanning tree (MST), or Hamiltonian cycle (HC) emerge. HC is the first problem we study in this dissertation [22]. It is clear that any algorithm designed for general graphs works also on random graphs while the opposite is not always true. Surprisingly, much of the works on distributed algorithms are on general graphs, but only a few on random graphs.

Erdős and Rényi introduced random graphs in [37]. They considered the dynamical process in which we start with the empty graph on n vertices, and at each step of $\binom{n}{2}$ steps add a random edge from edges not already in the graph. Two most popular random graph models (both known as Erdős-Rényi random graphs) are $G(n, m)$ and $G(n, p)$. In the former, the total number of randomly chosen edges is fixed to be m, while in the latter the probability of choosing an edge is fixed to be p. Random graphs such as $G(n, p)$ and its variants and generalizations (e.g., the Chung-Lu model [25]) have been used extensively to model and analyze real-world networks. Here, we focus on the $G(n, p)$ random graph model [37], a popular and well-studied model of random graphs with a long history in the study of graph algorithms (see e.g., [16, 47] and the references therein).

One remarkable property of the $G(n, p)$ model is that if p is above a certain threshold,

then with high probability (whp)[1], a Hamiltonian cycle (HC) exists. The Hamiltonian cycle is a cycle in a graph that passes through each node exactly once. More precisely, it is known that, with high probability, for n sufficiently large, there exists a HC in $G(n,p)$ if $p \geq \frac{c \ln n}{n}$, for any constant $c > 1$ [80]; in fact, not one, but it can be shown that an exponential number of Hamiltonian cycles exist [51, 30] for p above this threshold [2] It is worth noting that the above threshold for p is (essentially) the same as the threshold for connectivity of a $G(n,p)$ random graph.

Finding Hamiltonian cycles (or paths) in graphs (networks) is one of the fundamental graph problems. The solution has many applications such as building a token ring in wireless networks. Therefore, much research has been done to give an efficient algorithm to find an HC in a graph (if one exists). Since it is known that Hamiltonian cycles exists in $G(n,p)$ random graphs, there has been work in devising efficient algorithms for *finding* Hamiltonian cycles in these graphs. This is a non-trivial task, even though as mentioned earlier that there are exponential number of HCs present (see Chapter 3).

The second problem we study in this dissertation is community detection [38]. It is a prominent problem in machine learning, data mining, and network science. In the context of networks, community structure refers to the occurrence of groups of nodes that are more densely connected internally than with the rest of the network. Nodes of a community often show similar properties, functionalities, or behaviors. For example, communities in social networks show some similar interests, and citation networks form communities by research topic.

Finding an underlying community structure in a network is important for a number of reasons:

[1]Throughout, by "with high probability (whp)", we mean a probability at least $1-1/n^c$, for some constant $c > 0$, where n is the number of nodes.

[2]Actually, the "real" threshold for Hamiltonian cycles is $p \geq \frac{\ln n + \ln \ln n + \omega(1)}{n}$, if one wants to show the existence of HC asymptotically almost surely [16]. We use a slight larger threshold, since we want algorithms that succeed to find a HC whp.

- It helps us to visualize large scale graphs. We can show communities as meta-nodes and then can zoom in and out on its map.

- Identifying community sub-structures within a network can provide insight into how network function and topology affect each other. Such insight can be useful to improve some algorithms on graphs such as spectral clustering or rumor spreading.

- Statistical properties of communities often differ from the average of the network. We can highlight important features inside a network by considering those communities' statistics.

- It can play a role in predicting behavior of dynamic networks of future changes in a graph. It can also be used in recommendation systems. For example, it is plausible to recommend people from the same community to become friends if they are not.

The size and density of communities in a network vary. They also can overlap if a node belongs to multiple communities. For example this happens in social networks where a person is belonging to multiple communities. A person may belong to a community based on her/his school group and another one based on her/his business group and so on, or a researcher is publishing in multiple research topics such as both in computer security and distributed algorithms. In this dissertation, we study detecting non-overlapping communities. In the latter example, a researcher will be detected to belong to her/his hottest research topic. Another factor that makes the task harder is not knowing the number of communities in the network beforehand. Thus, to find the optimal communities, each possible combination in the search space should be considered which makes the problem $\mathcal{NP}$-hard in general (see Chapter 4).

1.1 The Distributed Computing Model

In distributed computing we have a network of processors (machines) modeled as a graph. The nodes of the graph model the processors and the edges the communication links between them. We consider each machine as a processing unit (call it P_i) which has a unique ID and its own memory. The input data is partitioned on the machines. All the processors run the same algorithm though they may go different directions (branches) based on some attributes such as their id. They run the algorithm on their local data and then exchange data with each other. This exchange of data lets the algorithm to generate the final global solution. We consider this local computation free because it is negligible compared to the communication cost.

There are two ways how processes may proceed compared to each other. They can run in synchronous or asynchronous mode. In the former, the network has access to a global clock. All the processes start at the same time and proceed in *rounds*. A round for a process P_i is as follows:

- P_i receives messages from its neighbors.
- P_i runs a round of execution of the algorithm locally. The algorithm utilizes its local and received data during its local computation.
- P_i sends messages to other machines.

In this dissertation we work with synchronous models. In the asynchronous model there are no rounds and each process proceeds at its own speed. It is a more realistic model but is difficult to design and analyze distributed algorithms. Indeed it is easier to design, debug, and analyze algorithms in a synchronous model. We note that choosing a synchronous model is not necessarily an impractical or unrealistic approach because there exist mechanisms, called

synchronizers, that enable one to run any synchronous algorithm on a asynchronous model with some overhead on the complexity [86].

Bandwidth capacity: We can classify distributed computing models into two categories, called *LOCAL* and *CONGEST* models. They are categorized based on the size of a message allowed to be exchanged over an edge of the network per round. In the former model, there is no restriction on the size of a message being exchanged. This is unrealistic because we have bandwidth limitations in reality. The cost (delay, energy, and time) of message exchange is not free which means we like to reduce the size of a message as much as we can. In the CONGEST model a message is limited to be of size $O(\log n)$ where n is the number of processes in the network. We note that $\log n$ bits are necessary to encode the ID of processes in the network of size n. In this dissertation we work with the latter, the CONGEST model.

1.2 Complexity Measures

The efficiency of distributed algorithms is traditionally measured by their time and message complexities. Both complexity measures crucially influence the performance of a distributed algorithm and it is of interest to keep both of them as small as possible. But, usually most of the distributed algorithms need to make a *trade-off* for the two metrics.

In the synchronous model, time is measured by the number of clock ticks called rounds. In other words, a round is the time in which all processing nodes in the network finish all three steps of receiving messages, local execution, and sending messages. In an asynchronous model, when running a distributed algorithm, different nodes might take a different number of rounds to finish. In that case, the maximum time (round) needed over all nodes is taken as the time complexity.

Message complexity is the total amount of messages exchanged by all the processes in the

network during the execution of the algorithm. Since the exchange of messages is expensive, it can be the dominant cost for most of the distributed algorithms and can affect the running time of the algorithm hugely. So it is of interest to keep it as small as possible. In the *Congest* model, the size of each message is small ($O(\log n)$ bits, where n is the size of network). This is very important for some applications such as communication networks and Internet of Things (IOT) where the small size messages mean less energy consumption for communication.

1.3 Random Graphs

The theory of random graphs was founded by Erdős-Rényi [37] after Erdős discovered that probabilistic methods help studying problems in graph theory. The methods have been used in calculating an proximate value of some properties with an appropriate probability distribution. In this dissertation, we work on two type of random graphs called $G(n, p)$ and $G(n, p, q)$, where n is the number of nodes in the graph. Both p and q are fixed probability values between 0 and 1. It was proven that when those probabilities are above a specific threshold, then the random graph is connected whp [37, 13, 16]. We work on connected random graphs throughout this dissertation.

1.3.1 Erdős-Rényi Random Graphs

The process of generating a $G(n, p)$ random graph starts with an empty graph, having n isolated vertices. Those n vertices can have at most $\binom{n}{2}$ edges. Each possible edge is selected and added to the graph with probability p. Indeed, each edges is selected independently and uniformly at random [37]. Therefore each node, in expectation, has $(n-1)p$ edges. $G(n, p)$ (sometime we call them G_{np}) graphs have interesting and useful properties and are a well studied graph model [16, 47] with lots of applications. We utilize some of those

properties in order to design efficient algorithms that run on random graphs. For instance, some important properties such as the connectivity threshold or the diameter based on values of p are well studied. It was shown that $p = \frac{\log n}{n}$ is the connectivity threshold for random graphs [37, 13, 16]. Chung and Lu [24] showed the diameters of $G(n, p)$ graphs are almost surely $\frac{\log n}{\log np}$ when $np \to \infty$. It was also shown that $G(n, p)$ random graphs have good expansion properties [54]. An expander graph is a sparse graph that has strong connectivity properties. It means every subset of the vertices that is not too large has a large boundary. Informally, a graph is a good expander if it has low degree and high expansion parameters.

1.3.2 Stochastic Block Models

Another type of random graph we worked on is the Stochastic Block Model (SBM) [52, 1]. It is a random graph model with cluster structures. In this dissertation, we study a special type of it called the Planted Partition Model (PPM) [21, 33, 20, 29, 19] which is usually used as a benchmark. It has a separable community structure where the structures or blocks are the same size. We call it $G(n, p, q)$ or G_{npq}. Given the number of partitions, r, nodes are divided into r parts. Each node exclusively belongs to a single part. We build random graphs $G(n/r, p)$ on each partition. Then we add edges between each pair of nodes with probability q if they do not belong to the same partition. In order to have a partition or community structure, q should be reasonably smaller than p ($q \ll p$). PPM graphs are extensively used as a canonical model to study clustering and community detection.

1.4 Contributions of this Dissertation

Here we highlight our results for the two problems we studied.

1. We present fast and efficient distributed algorithms for the fundamental Hamiltonian

cycle problem in random graphs, $G(n,p)$. Our algorithms finds a HC with high probability and runs in time significantly faster than the prior work of [67] as well as works for all ranges of p; in particular, the denser the graph, the faster will be our algorithms. Our distributed algorithms that run on random graphs are themselves randomized (i.e., they make random choices during the course of the algorithm) and hence the high probability bounds are both with respect to the random input and the random choices of the algorithm.

Our first distributed algorithm (DHC1) runs on $G(n,p)$ random graphs when $p = \frac{c \log n}{\sqrt{n}}$ and $c \geq 86$. It finds a Hamiltonian cycle, if one exists, in time $\tilde{O}(\sqrt{n})$ rounds whp. Our second algorithm (DHC2) works for $p = \frac{c \ln n}{n^{\delta}}$, for any fixed constant $\delta \in (0,1)$, and for a suitably large constant c, and runs in truly sublinear time, $\tilde{O}(n^{\delta})$ rounds. (Our algorithm will also work for $\delta = 1$, with running time $\tilde{O}(n)$.) Both algorithms work in the CONGEST model and are fully distributed, i.e., no node (or a few nodes) does all the computation (since the memory size of each node is restricted to be $o(n)$. Our second algorithm (DHC2) is fully-distributed and runs in truly sublinear time, $\tilde{O}(\frac{1}{p})$. We also present a conceptually simpler upcast algorithm in Section 3.4 with the same running time, but it is not fully-distributed, and does *not* achieve *load-balancing*.

2. We present a novel distributed algorithm, called *CDRW* (*Community Detection via Random Walks*) for detecting communities in the PPM model[29].[3] Our algorithm is based on the recently proposed *local mixing* paradigm [75] (see Section 4.3 for a formal definition) to detect community structure in *sparse* (bounded-degree) graphs. Informally, a local mixing set is one where a random walk started at some node in the set mixes well *with respect to this set.* The intuition in using this concept for community detection is that since a community is well-connected, it has good expansion within the community and hence a random walk started at a node in the community mixes

[3]Throughout this paper, we use the terms Stochastic Block Model (SBM) and Planted Partition Model (PPM) interchangeably.

well within the community. The notion of "mixes well" is captured by the fact that the random walk reaches close to the stationary distribution when *restricted to the nodes in the community subset* [75]. Since the main tool for this algorithm uses random walks which are local and lightweight, it is easy to implement this algorithm in a distributed manner. We will analyze the performance of the algorithm in two distributed computing models, namely the standard CONGEST model of distributed computing [86] and the k-machine model [58], which is a model for large-scale distributed computations. We show that CDRW can be implemented efficiently in both models (cf. Theorem 25 and Section 4.4.2). The k-machine model implementation is especially suitable for large-scale graphs and thus can be used in community detection in large SBM graphs. In particular, we show that the round complexity in the k-machine model (cf. Section 4.4.2) scales quadractically (i.e., k^{-2}) in the number of machines when the graph is sparse and it scales linearly (i.e., k^{-1}) in general.

As is usual in community detection, a main focus is analyzing the effectiveness of the algorithm in finding communities. We present a rigorous theoretical analysis that shows that the CDRW algorithm can accurately identify the communities in the PPM, which is a popular and widely-studied random graph model for community detection analysis [2]. A PPM model (cf. Section 4.2) is a parameterized random graph model which has a built-in community structure. Each community has high expansion within the community and forms a low conductance subset (and hence relatively fewer edges go outside the community); the expansion, conductance, and edge density can be controlled by varying the parameters. CDRW does well when the number of intra-community edges is much larger than the number of inter-community edges (these are controlled by the parameters of the model). Our theoretical analysis (cf. Theorem 25 for the precise statement) quantitatively characterizes when CDRW does well vis-a-vis the parameters of the model. Our results improve over previous distributed algorithms

that have been proposed for the PPM model ([27]) both in the number of communities that can be provably detected as well as range of parameters where accurate detection is possible; they also improve on previous results that provably work only on dense PPM graphs [62] (details in Section 2). CDRW correctly identifies the communities provided $q = o(p/(r \log(n/r)))$, where r is the number of communities. It takes $O(r \times \text{polylog } n)$ rounds for CDRW to terminate and hence it is quite fast when r is relatively small.

Chapter 2

Background

Random graphs have been studied thoroughly, see [16, 43, 44, 4, 76, 93] and references in them. A typical random graph is called an Erdős-Rény random graph [36]. In the Erdős-Rényi random graph model with n nodes, also known as the G_{np} (or $G(n,p)$) model, each of $\binom{n}{2}$ possible edges is present in the graph independently with probability p (see [14] for definitions and notations in random graphs). The main focus in those works is on the structure of random graphs. For instance, Chung and Lu [24] considered the diameter of random graphs for various ranges of p close to the phase transition point for connectivity. For a disconnected random graph, they considered the largest diameter of its connected component as the diameter of the graph. Interestingly they showed that when a random graph is sparse and close to its connectivity threshold ($p = \frac{\log n}{n}$), its diameter is almost surely $O(\log n)$ when $np \to \infty$.

Here we only highlight works on the distributed algorithms on random graphs. Although there is a profound knowledge about the structure of random graphs, there are few works on the design and analysis algorithms on random graphs [67, 63, 22, 99, 98]. Levy et al. [67] proposed a distributed algorithm to find Hamiltonian cycle in random graphs in $O(n^{\frac{3}{4}+\epsilon})$

rounds when $p = \omega(\frac{\sqrt{\log n}}{n^{1/4}})$. Krzywdzinski and Rybarczyk [63] proposed a distributed algorithm for coloring a random graph with $18np$ colors in $O(\ln \ln \frac{1}{p})$ rounds. Chatterjee et al. [22] proposed an algorithm to find Hamiltonian cycle in a random graph in $O(n^{\delta})$ rounds when $p \geq \frac{c \log n}{n^{\delta}}$ where $c > 1$ and $0 \leq \delta \leq 1$. Turau [99] proposed a distributed algorithm to find a Hamiltonian cycle in random graphs in $O(\log n)$ rounds for $p = \tilde{\Omega}(\frac{1}{\sqrt{n}})$. Notice that $\tilde{\Omega}$ notation hides poly-logarithmic factors in n. Turau in [98] also proposed an efficient distributed algorithm to build a routing structure for publish/subscribe systems in wireless networks.

In the following two subsections we highlight some of the most relevant related works to each subsection accordingly.

2.1 Hamiltonian Cycles

A random graph has a Hamiltonian cycle with high probability when it is dense enough (p, the existence probability of each edge, is big enough) [96, 17]. Pósa in [90] proved that a large Erdős-Rényi random graph ($n \to \infty$) has a Hamiltonian cycle with probability close to 1 when $p = \frac{c \log n}{n}$ and $c > 3$. Korshunov [61] improved Pósa's bound for p by showing that if a random graph has $\frac{1}{2} n \log n + \frac{1}{2} \log \log n + f(n)n$ edges and $f(n) \to \infty$, then the graph has a Hamiltonian cycle. Komlós and Szemerédi [60] then completely solved the problem by showing that a random graph with $\frac{1}{2} n \log n + \frac{1}{2} \log \log n + cn$ edges has a Hamiltonian cycle with probability $\exp \exp (-2c)$ when $n \to \infty$. Indeed they proved that if the number of edges is big enough, it *ensures* that there are no vertices of degree less than 2 in the graph.

Let us highlight some prominent sequential algorithms for finding a Hamiltonian cycle in random graphs. Bollobás et al. [18] suggested a sequential deterministic algorithm to find a HC in time $O(n^{3+o(1)})$ with high probability. Angluin and Valiant [7] proposed an algorithm to find a HC in $O(n \log^2 n)$ time when the random graph has at least $cn \log n$ edges where

c is a large constant ($c \geq 36$). Then Frieze and Haber [42] proposed an algorithm that succeeds whp in $O(n^{1+o(1)})$ time when the graph is sparse ($m = cn$ for sufficiently large c). The restriction is that the minimum degree of nodes should be at least three. Recently Alon and Krivelevich [5] improved the work in [42] to run in $O((1+o(1))n/p)$ time when $p \geq \frac{72}{\sqrt{n}}$. Thomason's [97] algorithm finds a Hamiltonian path between any two nodes in cnp^{-1} time when $p \geq \frac{12}{n^3}$; it shows such a path does not exist if there is not a Hamiltonian path in the graph. It is seen that when the random graph gets denser(p gets a bigger value), we get more efficient solutions to solve the problem. In other words, the real challenge emerges when the graph is sparse and as close to its connectivity threshold ($p = \frac{c \log n}{n}$, $c > 1$) as possible.

There are few distributed algorithms to find HC in random graphs. Frieze [45] suggested a parallel algorithm which constructs a HC in expected time $O((\log \log n)^2)$ using $n \log^2 n$ processes. Later MacKenzie and Stout [69] improved it by proposing a parallel algorithm which runs in expected time $\Theta(\log^* n)$ time using $\frac{n}{\log^* n}$ processes. Levy et al. [67] introduced a distributed algorithm which runs in $O(n^{3/4+\epsilon})$ rounds when $p = \omega(\sqrt{\log n}/n^{1/4})$. Our algorithm [22] runs in $O(n^{\delta})$ rounds when $p \geq \frac{c \log n}{n^{\delta}}$ where $c \geq 86$ and $0 < \delta \leq 1$. Turau [99] proposed a faster algorithm which runs in $(\log n)$ rounds when $p = \tilde{\Omega}(1/\sqrt{n})$ where $\tilde{\Omega}$ hides poly-logarithmic factors in n.

2.2 Community Detection

There has been extensive work on community detection in graphs, see, e.g., the surveys [1, 3, 39, 40]. Here we focus mainly on related works in distributed community detection and in the SBM, especially the G_{npq} model.

Dyer and Frieze [33] show that if $p > q$ then the minimum edge-bisection is the one that separates the two classes and present an algorithm that gives the bisection in $O(n^3)$ expected time. Jerrum and Sorkin improved this bound for some range of p and q by using simulated

annealing. Further improvements and more efficient algorithms were obtained in [29, 77]. We note that all the above algorithms are centralized and based on expensive procedures such as simulated annealing and spectral graph computations: all of them require the full knowledge of the graph.

The work of Clementi et al. [27] is notable because they present a distributed protocol based on the popular Label Propagation approach and prove that, when the ratio p/q is larger than n^b (for an arbitrarily small constant $b > 0$), the protocol finds the right planted partition in $O(\log n)$ time. Note however that they consider only two communities in their PPM model. We also note that this ratio can be significantly weaker compared to the ratio (for identifying all the r communities) derived in our Theorem 25 which is $p/q = O(r \log(n/r)$ where r is the number of communities (which can be much smaller compared to n, the total number of vertices). Also our algorithm works for any number of communities.

Random walks have been successfully used for graph processing in many ways. Community detection algorithms use the statistics of random walks to infer structure of graphs as clusters or communities. The *Netwalk* algorithm [101] defined a proximity measure between neighboring vertices of a graph. Initialized each node as a community, it merges two communities with the lowest proximity index into a community in iterations. But it is an expensive method running in $O(n^3)$ time complexity. Similarly, *Walktrap* by Pons and Latapy [89] defined a distance measure between nodes using random walks. Then they merged similar nodes into a community in iterations. The logic behind their method is that random walks get trapped inside densely connected parts of a graph which can capture communities. Their algorithm is centralized and has an expensive run-time of $O(mn^2)$ in worst case.

Some works uses linear dynamics of graphs to perform basic network processing tasks such as reaching self-stabilizing consensus in faulty distributed systems [12, 79] or Spectral Partitioning [32, 65, 88]. They work on connected non-bipartite graphs. Becchetti et al. [10] define averaging dynamics, in which each node updates its value to the average of its

neighbors, in iterations. They partition a graph into two clusters using the sign of last updates. Another interesting research by Becchetti et al. [11] used random walks to average values of two nodes when randomly any two nodes meet and showed that it ends in detecting communities. The convergence time of the averaging dynamics on a graph is the mixing time of a random walk [95]. These methods work well on graphs with good expansion [54] and are slower on sparse cut graphs. The *Label Propagation Algorithm (LPA)* [91] is another updating method which converges to detecting communities by applying majority rule. Each node initially belongs to its own community. At each iteration, each node joins a community having a majority among its neighbors, applying a tie-breaking policy. Recently Kothapalli et al. provided a theoretical analysis for its behavior [62] on *dense* PPM graphs ($p = \Omega(1/n^{1/4})$ and $q = O(p^2)$). In comparison, our algorithm works even for the more challenging case of sparse graphs ($p = \Omega(\log n/n)$, i.e., the near the connectivity threshold). A major drawback of the LPA algorithm is the lack of a convergence guarantee. For example, it can run forever on a bipartite graph where each part gets a different label (each community is specified by a label).

Some other random walk methods use seed node expansion to detect community structures. Starting from a small set of "seed" nodes (e.g., a single node or a connected cluster of nodes), they grow a community around the seed set by using a goodness function. They may use a greedy method [72] or an optimization function [8] to improve. Optimization techniques are the natural way to search for clusters and communities of graphs. In this method, a quality function (that captures the "goodness" of clusters) is defined and then its extreme for an optimal output is searched. For example, Newman and Girvan uses modularity as their optimization function [78]. Their aim is to maximize the modularity function that results in detecting stronger community structures. The method in [26] grows the size of community by adding nodes which improve the modularity most; it stops when reaching a favored community size. Recently Hollocou et al. [53] applied clustering methods to find

seed nodes and then find communities from those nodes. Some other methods [6, 59, 100] use a small seed set instead of one node for their detection. Unlike those methods, our CDRW algorithm uses the local mixing property of a graph [31, 74, 75] to detect community structure around a seed node.

Although we uses the notion of local mixing time introduced in [75], there are substantial differences. In [75], the authors consider only the local mixing time which is essentially the existence of a mixing set of certain size, but *not the set of nodes* where the random walk mixes. The computation of the local mixing set is more challenging. A key idea of our work is to use this notion to identify communities. For this, the algorithm and the approach of [75] have to be modified substantially.

Chapter 3

Hamiltonian Cycles

Finding Hamiltonian cycles (or paths) in graphs (networks) is one of the fundamental graph problems. A Hamiltonian cycle (HC) is a cycle in the graph that passes through each node exactly once. The decision problem is NP-complete [48] (in fact, it is one of Karp's six basic NP-complete problems) and hence unlikely to have a polynomial time algorithm in the sequential setting. In this dissertation, we focus on the distributed computation of Hamiltonian cycles (or paths) in a (undirected) graph. In particular, our goal is to find a *fast*, *efficient*, and *fully* distributed algorithm for the Hamiltonian cycle problem. By "fast", we mean running in a small number of *rounds* (ideally, sublinear in n, where n is the number of nodes in the network). By "efficient", we mean that only small-sized messages (say, at most $O(\log n)$-sized messages) are exchanged per edge per round, and the per-round computation per node should also be small, i.e., sublinear in n. The latter means that the local (i.e., "within node") computation is also efficient. By "fully-distributed", we (informally) mean that no one node (or a small set of nodes) does all the non-trivial (local) computation and all the local computations are (more or less) balanced (formally we enforce this by assuming that each node's memory is limited to $o(n)$).

Since the HC problem is NP-complete, there is not much hope of achieving a fast and

efficient distributed algorithm (even if we allow polynomial time local computation per round and even without caring whether it is fully-distributed or not) in arbitrary graphs, even if we allow polynomial number of rounds (since the total local computation time over all nodes is at most polynomial). However, the problem is reasonable and, yet challenging, when we consider random graphs, where efficient sequential algorithms (nearly linear time) for computing Hamiltonian cycles are known.

Despite the importance of the Hamiltonian cycle problem, there has been only some previous work in the distributed setting. The work of Das Sarma et al. [94] (see also [35]) showed an important lower bound for the HC problem for general graphs in the CONGEST model of distributed computing [87] (described in detail in Section 4.2), a standard model where there is a bandwidth restriction on the edges (typically, only $O(\log n)$-sized messages are allowed per edge per round, where n is the graph/network size). They showed that any deterministic algorithm (this was extended to hold even for *randomized* algorithms in [35]) needs at least $\tilde{\Omega}(D+\sqrt{n})$ rounds, where D is the graph diameter[1]. Note that this lower bound holds even if every node's local computation is free (i.e., there is no restriction on the within node computation cost in a round — this is the usual assumption in the CONGEST model [87]). It is important to note that this lower bound is for general graphs; more precisely, it holds for a family of graphs constructed in a special way.

Somewhat surprisingly, no non-trivial upper bounds are known for the distributed HC problem in the CONGEST model. A trivial upper bound in the CONGEST model is $O(m)$ where m is the number of edges of the graph (cf. Section 4.2). It is not known if one can get a $(O(D) + o(n))$-round algorithm or even a $(O(D) + o(m))$-round algorithm for HC in general graphs, where D is the graph diameter (note that D is a lower bound [94]). In this dissertation, we show that we can obtain significantly faster (truly sublinear in n) algorithms, i.e., running in time $O(n^{\delta})$ rounds (where $0 < \delta < 1$) in random graphs.

[1]The notation $\tilde{\Omega}$ hides a $1/\text{polylog}n$ factor.

We focus on the $G(n,p)$ random graph model [37], a popular and well-studied model of random graphs with a long history in the study of graph algorithms (see e.g., [16, 47] and the references therein). Random graphs such as $G(n,p)$ and its variants and generalizations (e.g., the Chung-Lu model [25]) have been used extensively to model and analyze real-world networks. In the $G(n,p)$ random graph model, there are n nodes and the probability that an edge exists between any two nodes is p (independent of other edges). A remarkable property of the $G(n,p)$ model is that if p is above a certain threshold, then with high probability (whp)[2], a Hamiltonian cycle (HC) exists. More precisely, it is known that, with high probability, for n sufficiently large, there exists a HC in $G(n,p)$ if $p \geq \frac{c \ln n}{n}$, for any constant $c > 1$ [80]; in fact, not one, but it can be shown that exponential number of Hamiltonian cycles exist [51, 30] for p above this threshold [3]. It is worth noting that the above threshold for p is (essentially) the same as the threshold for connectivity of a $G(n,p)$ random graph.

Since it is known that Hamiltonian cycles exists in $G(n,p)$ random graphs, there have been works on devising efficient algorithms for *finding* Hamiltonian cycles in these graphs. This is a non-trivial task, even though as mentioned earlier that there is an exponential number of HCs present. Angluin and Valiant [7], in a seminal paper (see also [73]), gave a sequential algorithm to find a HC in a $G(n,p)$ graph that runs in $O(n(\log n)^2)$ time, when $p \geq \frac{c \ln n}{n}$, for some sufficiently large constant (say $c \geq 36$). This is essentially the best possible as far as the sequential running time is concerned as it is almost linear. The algorithm of Angluin and Valiant is randomized. Bollobás, Fenner, and Frieze [18] give a deterministic sequential algorithm for finding Hamilton cycles in random graphs (in the related $G(n,M)$ random graph model, which is a uniform distribution over all graphs on n vertices and M edges),

[2]Throughout, by "with high probability (whp)", we mean a probability at least $1-1/n^c$, for some constant $c > 0$, where n is the number of nodes.

[3]Actually, the "real" threshold for Hamiltonian cycles is $p \geq \frac{\ln n + \ln \ln n + \omega(1)}{n}$, if one wants to show the existence of HC asymptotically almost surely [16]. We use a slight larger threshold, since we want algorithms that succeed to find a HC whp.

but the running time is essentially $O(n^4)$ and succeeds with high probability (in graphs where the number of edges is above the threshold of existence of Hamiltonian cycle). In the context of parallel algorithms, MacKenzie and Stout in [69] proposed a parallel algorithm which uses $O(\frac{n}{\log * n})$ processes and runs in $O(\log^* n)$ time. In the *distributed setting*, the only prior work we are aware of is the work of Levy et al. [67] which gives a distributed algorithm to find a HC in $O(n^{\frac{3}{4}+\epsilon})$ time when $p = \omega(\frac{\sqrt{\log n}}{n^{\frac{1}{4}}})$.

In this dissertation, we propose a *fast, efficient, and fully decentralized* (as defined earlier) distributed algorithm that finds a HC with high probability and runs in time significantly faster than the prior work of [67] as well as works for all ranges of p; in particular, the denser the graph, the faster will be our algorithms. Our distributed algorithms that run on random graphs are themselves randomized (i.e., they make random choices during the course of the algorithm) and hence the high probability bounds are both with respect to the random input and the random choices of the algorithm.

We give a brief overview of our results. In Section 3.3, we give two fast (truly sublinear in n), efficient and fully decentralized algorithms. The first algorithm is a bit simpler; it works for $p \geq \frac{c \ln n}{\sqrt{n}}$ and runs in $\tilde{O}(\sqrt{n})$ rounds. The second algorithm works for $p = \frac{c \ln n}{n^\delta}$, for any fixed constant $\delta \in (0, 1)$, and for a suitably large constant c, and runs in $\tilde{O}(n^\delta)$ rounds. (Our algorithm will also work for $\delta = 1$, with running time $\tilde{O}(n)$.) Both algorithms work in the CONGEST model and are fully distributed, i.e., no node (or a few nodes) does all the computation (since the memory size of each node is restricted to be $o(n)$ — cf. Section 4.2). In contrast, in Section 3.4, we present a (conceptually) simple *upcast* algorithm that uses a fairly generic "centralized" approach. In this algorithm, each node samples $\Theta(\log n)$ random edges among all its incident edges and upcasts it to a central node (which is the root of a Breadth First Tree) which locally computes a HC and then broadcasts the HC edges back to the respective nodes by downcast. Note that, in this approach, all the non-trivial (local) computation is done at a central node and hence the algorithm is not fully distributed (some

node needs at least $\Omega(n)$ memory), although the algorithm works in the CONGEST model. We show that this algorithm also runs in time $\tilde{O}(n^{\delta})$ rounds for $p = \frac{c \ln n}{n^{\delta}}$.

3.1 Distributed Computing Model

We model the communication network as an undirected, unweighted, connected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. Every node has limited initial knowledge. Specifically, we assume that each node is associated with a distinct identity number (e.g., its IP address). At the beginning of the computation, each node v accepts as input its own identity number and the identity numbers of its neighbors in G. We also assume that the number of nodes and edges, i.e. n and m (respectively), are given as inputs. (In any case, nodes can compute them easily through broadcast in $O(D)$, where D is the network diameter.) The nodes are only allowed to communicate through the edges of the graph G. We assume that the communication occurs in synchronous *rounds*. (In particular, all the nodes wake up simultaneously at the beginning of round 1, and from this point on the nodes always know the number of the current round.) We will use only small-sized messages. In particular, in each round, each node v is allowed to send a message of size $O(\log n)$ bits through each edge $e = (v, u)$ that is adjacent to v.[4] The message will arrive at u at the end of the current round. This is a widely used standard model known as the *CONGEST model* to study distributed algorithms (e.g., see [87, 82]) and captures the bandwidth constraints inherent in real-world computer networks.

We focus on minimizing the *running time*, i.e., the number of *rounds* of distributed communication. Note that the computation that is performed by the nodes locally is "free", i.e., it does not affect the number of rounds; however, as mentioned earlier, we will only

[4]Our algorithms can be easily generalized if $\mathbb{B}$ bits are allowed (for any pre-specified parameter $\mathbb{B}$) to be sent through each edge in a round. Typically, as assumed here, $\mathbb{B} = O(\log n)$, which is the number of bits needed to send a node id in an n-node network.

perform sublinear (in n) cost computation locally at any node.

We note that in the CONGEST model, it is rather trivial to solve a problem in $O(m)$ rounds, where m is the number of edges in the network, since the entire topology (all the edges) can be collected at one node and the problem solved locally. The goal is to design faster algorithms. Our algorithms work in the *CONGEST* model of distributed computing. We note that our bounds are non-trivial in the CONGEST model.[5]

In Section 3.3, we consider fully-distributed algorithms, where there is a restriction on the amount of memory each node can have: each node is allowed only $o(n)$ memory. This restriction, in effect, rules out "centralized" approaches such as collecting global information at one particular node and then locally solving the problem. In our fully-distributed algorithms, each node's (local) computation is more or less balanced. Fully-distributed algorithms are quite useful, since they can be be efficiently converted to work in other distributed models for Big Data computing such as the k-machine model [58] as well as MapReduce [56].

In Section 3.4, we consider algorithms where we don't have any restriction on the memory size at any node nor do we restrict the local computation cost to be sublinear (note that this restriction turns out to be not so important for the bounds that we obtain, as one can run sublinear cost local computation over a sublinear number of rounds). However, the algorithms still follow the CONGEST model (i.e., there is a bandwidth restriction).

We make a note on the output of our distributed algorithms: at the end, each node will know which of its incident edges belong to the HC (exactly two of them).

[5]In contrast, in the LOCAL model — where there is no bandwidth constraint — all problems can be trivially solved in $O(D)$ rounds by collecting all the topological information at one node.

3.2 Other Related Work

There are several algorithms for finding a HC in random graphs (both $G(n,p)$ and its closely related variant $G(n,M)$ random graphs), e.g., we refer to the survey due to Frieze [46]. There also have been works on parallel algorithms for finding Hamiltonian cycles in $G(n,p)$ random graphs. Frieze [45] proposed two algorithms for EREW-PRAM machines: the first uses $O(n \log n)$ processors and runs in $O(\log^2 n)$ time, while the second one uses $O(n \log^2 n)$ processors and runs ins $O((\log \log n)^2)$ time. MacKenzie and Stout [69] gave an algorithm for Arbitrary CRCW-PRAM machines that operates in $O(\log^* n)$ average time and requires $O(n/\log^* n)$ processors. All these parallel algorithms assume p is a constant.

With regard to distributed algorithms, as mentioned earlier, the only prior work we are aware of is the work of Levy et al. [67] which gives a fully distributed algorithm to find a HC in $O(n^{\frac{3}{4}+\epsilon})$ time when $p = \omega(\frac{\sqrt{\log n}}{n^{\frac{1}{4}}})$. Their algorithm (based on the algorithm of MacKenzie and Stout [69]) works in three phases: finding an initial cycle, finding $\sqrt{n}$ disjoint paths, and finally patching paths into the cycle to build the HC. Our fully distributed algorithms (Section 3.3) follow a different and a simpler approach and are significantly faster, while working for all ranges of p above the HC threshold.

3.3 Fully-Distributed Algorithms

In this section, we give two fast, efficient, fully-distributed algorithms for the Hamiltonian cycle problem. The first algorithm, in Section 3.3.1, is a distributed algorithm for the case of $p = \frac{c \ln n}{\sqrt{n}}$ (throughout, c will be a large enough constant, say bigger than 54) and runs in time $\tilde{O}(\sqrt{n})$ rounds w.h.p. (In fact, the algorithm will work for any $p \geq \frac{c \ln n}{\sqrt{n}}$, but for simplicity we will fix $p = \frac{c \ln n}{\sqrt{n}}$.). This algorithm works in the CONGEST model and is fully distributed, i.e., each node's local computation memory is $o(n)$ and the computation cost per node per

round is also $o(n)$. This algorithm is somewhat simpler, contains some of the main ideas, and is also useful in understanding the second algorithm. The second algorithm, in Section 3.3.2, is more general, works for $p = \frac{c \ln n}{n^\delta}$, for any $0 < \delta \leq 1$ and runs in $\tilde{O}(n^\delta)$ rounds. Both algorithms have two phases; while the first phase is similar for both algorithms, the second phase for the second algorithm is more involved.

Before we go into the details of our algorithms, we will give the main intuition. Our algorithm is inspired by the well-studied *rotation* algorithm (rotation is a simple operation described in Section 3.3.1) that was used by Angluin and Valiant to develop a fast sequential algorithm for the $G(n,p)$ random graph for $p \geq \frac{c \ln n}{n}$ (for some suitably large constant c, say $c > 36$). However, this algorithm seems inherently sequential, since it tries to extend the cycle one edge at a time; hence the running time under this approach is at least $\Omega(n)$. To get a sublinear time, we follow a two-phase strategy which works in somewhat denser graphs, i.e., $p = \frac{c \ln n}{n^\delta}$, for any $0 < \delta < 1$. In Phase 1, we partition the graph into *disjoint* random subgraphs each of size (approximately) $\Theta(n^\delta)$ (there will be $\Theta(n^{1-\delta})$ subgraphs). The intuition behind this partition is that each subgraph will have a HC of its own (of length equal to the size of the subgraph) whp, since it satisfies the threshold for Hamiltonian cycle (note that $p = \frac{c \ln n}{n^\delta}$). We use a distributed implementation of the rotation algorithm to find the Hamiltonian (sub)cycles independently in each of subgraphs — this takes time essentially linear in the size of the subgraphs, i.e., $\tilde{O}(n^\delta)$. In Phase 2, we stitch the cycles without taking too much additional time, i.e., in $\tilde{O}(n^\delta)$ time. When $p = \frac{c \ln n}{\sqrt{n}}$, the case is special, since the number of subgraphs and the size of each subgraph are balanced, so the stitching can be done by essentially implementing a modification of Phase 1 as follows. Take two adjacent nodes from each subgraph cycle and find a Hamiltonian cycle between the chosen nodes (this has to be done carefully, so that it can be combined with the subgraph cycles to form a HC over all the nodes). Since $p = \frac{c \ln n}{\sqrt{n}}$, and the number of chosen nodes is $\Theta(\sqrt{n})$, whp a HC exists between the chosen nodes and we can find it using a strategy

similar to Phase 1. For general p, we note that we cannot just simply stitch as described above, since p is much smaller than the needed threshold. Hence, we do the stitching in stages, as described in Section 3.3.2.

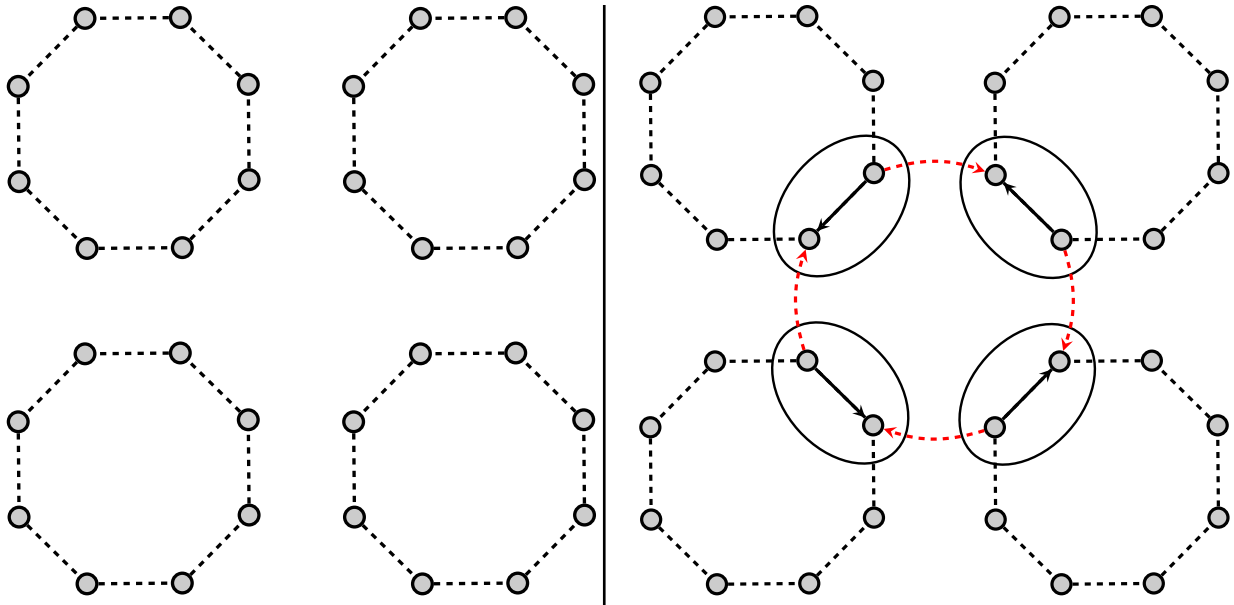
3.3.1 The Algorithm for $p = \frac{c \ln n}{\sqrt{n}}$

Our first algorithm, called the *Distributed Hamiltonian Cycle Algorithm 1 (DHC1)*, works for a random graph $G(n, p = \frac{c \ln n}{\sqrt{n}})$, where c is a suitably large constant.

3.3.1.1 High-Level Description of DHC1

Given a random graph $G(n, \frac{c \ln n}{\sqrt{n}})$, our algorithm works in two phases. In Phase 1, the graph is partitioned into $\sqrt{n}$ subgraphs G_i, each of $\Theta(\sqrt{n})$ nodes. Then each subgraph constructs its own Hamiltonian cycle C_i, independently in parallel. In Phase 2, the algorithm finds a Hamiltonian cycle connecting $C_1, \cdots, C_{\sqrt{n}}$. This is done as follows: for each C_i, pick only one edge $e_i = (v_i, u_i)$, call this a hypernode (edges inside oval shapes in Figure 3.1). Consider the graph G' of $\sqrt{n}$ hypernodes e_i, a hypernode uses u_i as the *incoming* port, and v_i as the *outgoing* port. In other words, we only look at the edges (v_j, u_i) and (v_i, u_j) for any pair $e_i \neq e_j$. The algorithm constructs a Hamiltonian cycle in G' which is easy to see completes the Hamiltonian cycle in G (see Figure 3.1).

In Phase 1 (as well as in Phase 2, for constructing a HC in G'), the cycles are constructed locally: each node becomes aware of its predecessor and successor after the construction. For convenience, each node also maintains an index of its position in the cycle. The resulting Hamiltonian cycle is hierarchical. Each node maintains its index *subcyc* in the subgraph cycle. In Phase 2, if a node is part of a hypernode, it maintains an extra index *hypcyc* in the cycle constructed in Phase 2. When traversing the cycle, if a node has a *hypcyc* link, follow it, otherwise follow the *subcyc* link.



(a) Phase1 (b) Phase2

Figure 3.1: Algorithm DHC1 builds HC in two phases. Phase 1 constructs $\sqrt{n}$ sub HCs in parallel. Phase 2 combines all sub HCs by building a HC over the graph of hyper nodes.

We next describe the distributed algorithm for constructing a HC in the $\sqrt{n}$-sized subgraph. This distributed algorithm which we call *Distributed Rotation Algorithm (DRA)* is based on the well-known randomized algorithm for finding a Hamiltonian cycle that uses so called *rotation* steps [73] (see Figure 3.2).

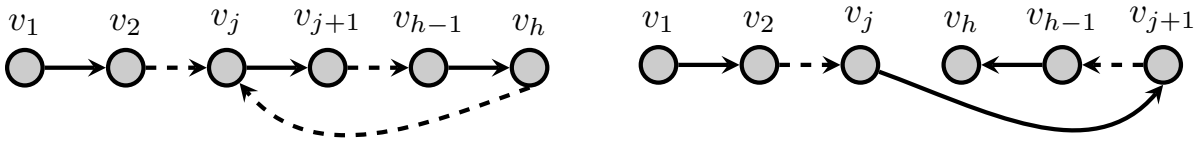


Figure 3.2: Path Rotation: Extending from the head node (v_h), we encounter a node (v_j) on the path. The right side shows the rotated path.

3.3.1.2 The DRA Algorithm

Consider a graph G with n nodes. We construct a Hamiltonian path $v_1, v_2, \cdots, v_n$; if there is an edge connecting v_n and v_1, then we have a Hamiltonian cycle. We will grow the path sequentially by a simple randomized algorithm. For a path $v_1, \cdots, v_h$, let v_h be the *head*.

Algorithm 1 Distributed Rotation Algorithm (DRA) Algorithm

```
1: function DRA(G(V, E), cycindex)  ▷ code for each node v ∈ V, use cycindex for path index
2:     Init
3:         v.unused ← all edges to neighbors
4:         v.cycindex ← 0
5:         only one v becomes head, v.cycindex ← 1
6:     while v.unused ≠ ∅ do
7:         if v is head then
8:             (v, u) ← random edge from v.unused
9:             v.unused ← v.unused − {(v, u)}
10:            send to u: progress(pos = v.cycindex)
11:        OnReceive message progress(pos)
12:            if pos = |V| and v.cycindex = 1 then return Success
13:            v.unused ← v.unused − {(sender, v)}
14:            if v.cycindex = 0 then                          ▷ first time visiting v
15:                become head: v.cycindex ← pos + 1
16:            else                                            ▷ v is already on the path
17:                broadcast: rotation(h = pos, j = v.cycindex)
18:        OnReceive message rotation(h, j)
19:            if j < v.cycindex ≤ h then
20:                v.cycindex ← h + j + 1 − v.cycindex
21:                if v.cycindex = h then
22:                    v becomes head
       return
```

Initially, we choose a random v_1 which is also the initial *head*. The *head* picks a random edge (v_h, u), say, which has not previously been used.

If $u \notin \{v_1, \cdots, v_h\}$, add node u to the path and set it as the new *head*. If u is some v_j, then we *rotate* the path: $v_1, \cdots, v_j, v_{j+1}, \cdots, v_h$ becomes $v_1, \cdots, v_j, v_h, v_{h-1}, \cdots, v_{j+1}$ and v_{j+1} is the new head. The rotation can be implemented by just a renumbering: for v_i, where $j + 1 \leq i \leq h$, reassign $i \leftarrow h + j + 1 - i$. In a distributed setting, we can implement an efficient procedure: v_j broadcasts the values h and j then every node can renumber itself accordingly. Notice that the required time for broadcast is the diameter D of the graph, and we will give bounds for D in the analysis.

The Distributed Rotation Algorithm (DRA) is given in Algorithm 1, where we initialize

Algorithm 2 Distributed Hamiltonian Cycle Algorithm 1 (DHC1)

```
1: function DHC1(G(V, E))
2:     Init
3:         n ← |V|
4:         foreach v ∈ V:
5:             v.subcyc ← 0, v.hypcyc ← 0
6:     Phase 1
7:         v.color ← random[1, ··· , √n]
8:         Let G_i(V_i, E_i) be the subgraph with nodes in color i
9:         foreach G_i:
10:            C_i ← DRA(G_i, cycindex = subcyc)
11:    Phase 2
12:        foreach C_i:
13:            pick a random u_i ∈ C_i
14:            v_i ← predecessor(u_i)
15:            hypernode_i ← [u_i, v_i]
16:            G′: graph of all hypernode_i, edges: all pairs (v_j, u_k), j ≠ k
17:        C′ ← DRA(G′, cycindex = hypcyc)
18:    return
```

the algorithm by assigning any one node to be the *head*. The DHC1 algorithm pseudocode is given in Algorithm 2. Notice that we initialize 2 position indexes for each node, and construct the (overall) Hamiltonian cycle by multiple calls to Algorithm 1.

3.3.1.3 Analysis

We first state the main theorem, which gives the probability of success and the expected runtime of the DHC1 algorithm.

Theorem 1. *For a $G(n,p)$ with $p = \frac{c \ln n}{\sqrt{n}}$ with $c \geq 86$, the DHC1 algorithm successfully builds a Hamiltonian cycle with probability $(1 - O(\frac{1}{n}))$, in $O(\sqrt{n}\frac{\ln^2 n}{\ln \ln n})$ rounds.*

The next theorem describes the performance of the Distributed Rotation Algorithm (DRA), a key subroutine of the DHC1 algorithm. This result will be used in both Phase 1 and Phase 2 of DHC1 to bound its runtime. To simplify the analysis, we will state the run time in this theorem in terms of the number of *steps*, where each step is one rotation or

growing the path by one node. For the moment, we ignore the cost of broadcast, which we will later account for in the main theorem.

Theorem 2. *Given a $G(n,p)$ graph where $p \geq 86\frac{\ln n}{n}$, the DRA algorithm constructs a Hamiltonian Cycle in $7n \ln n$ steps with probability of success $1 - O(\frac{1}{n^3})$.*

Proof. We follow the approach as described in [73] which we refer to for more details. The main idea is to relate the algorithm to a *coupon collector* process, where the goal is to collect n different coupons and in each step the probability of collecting a particular coupon is $1/n$ (independent of other coupons) and it is known that all coupons can be collected in $O(n \ln n)$ steps whp. Here, the n coupons represent the n nodes and collecting all the coupons is analogous to building a HC. Since the rotation algorithm does not give uniform $1/n$ probability, to apply the coupon collector model, we relax the analysis as follows.

We consider a relaxed algorithm such that every node has equal probability of $\frac{1}{n}$ to be chosen in every step of growing the path (this relaxation is described in [73]). Note that, in fact, the algorithm is more efficient in choosing a new node. We will not restate all the details here, except for the key technique. Remember that the edge probability is p, and this implies a dependency between two nodes. Under the relaxed algorithm, let each node has a list of edges, called "unused" edges, which is selected independently at random, with probability q. The technical part is how to convert p to q, such that the "unused" edges are a subset of the true edges. All the subtleties can be found in [73], for convenience, we cite q here: $q = 1 - \sqrt{1-p} \geq p/2$. We are now ready for the proof, where we want to improve the analysis of [73]. In particular, by allowing a larger runtime, but still in $O(n \ln n)$, we can reduce the failure probability to $O(1/n^3)$. This technique can be extended to achieve failure probability in $O(1/n^\alpha)$, with a given constant α.

The relaxed algorithm has two scenarios of failure:

- $\mathcal{E}_1$: The algorithm runs for $7n \ln n$ steps while no unused edges in any vertex become

empty, and fails to construct a Hamiltonian cycle.

- $\mathcal{E}_2$: At least one vertex runs out of unused edges during $7n \ln n$ steps.

For event $\mathcal{E}_1$, equal probability of $1/n$ yields that the probability of not seeing a node after $4n \ln n$ steps is

$$\left(1 - \frac{1}{n}\right)^{4n \ln n} \leq \frac{1}{n^4}.$$

Using a union bound, the probability of failure to meet all n nodes after $4n \ln n$ steps is $O\left(\frac{1}{n^3}\right)$.

Now, in order to close the cycle, the head needs to visit the tail, which happens with probability $\frac{1}{n}$. After $3n \ln n$ steps, the probability of failure to complete the cycle is at most:

$$\left(1 - \frac{1}{n}\right)^{3n \ln n} \leq \frac{1}{n^3}.$$

In total, $Pr(\mathcal{E}_1) \leq \frac{2}{n^3} = O\left(\frac{1}{n^3}\right)$.

For event $\mathcal{E}_2$, we break it into two sub events:

- $\mathcal{E}_{2.1}$: At least $21 \ln n$ edges are removed from at least one node during $7n \ln n$ steps.
- $\mathcal{E}_{2.2}$: At least one node has fewer than $21 \ln n$ edges in its initial unused list.

Consider $\mathcal{E}_{2.1}$ and look at a node v. Let X be the number of edges removed at v during $7n \ln n$ steps. We have $E[X] = \frac{1}{n} * 7n \ln n = 7 \ln n$. Using Chernoff's bound,

$$\begin{aligned} Pr(X \geq 21 \ln n)) &= Pr(X \geq (1+2)7 \ln n) \\ &\leq \left(\frac{e^2}{3^3}\right)^{7 \ln n} \leq \left(\frac{1}{e^{4/7}}\right)^{7 \ln n} = O\left(\frac{1}{n^4}\right). \end{aligned}$$

Using a union bound, $Pr(\mathcal{E}_{2.1}) = O\left(\frac{1}{n^3}\right)$.

Consider $\mathcal{E}_{2.2}$. Let Y be the initial number of edges in the unused edges list of a node. We have $E[Y] = q(n-1) \geq (43\frac{\ln n}{n})(n-1) \geq 42 \ln n$. Using Chernoff's bound:

$$Pr(Y \leq 21 \ln n) = Pr(Y \leq (1 - \frac{1}{2})42 \ln n)$$
$$\leq \exp\left(-\frac{\left(\frac{1}{2}\right)^2 42 \ln n}{2}\right) = O\left(\frac{1}{n^4}\right).$$

Using a union bound for n nodes, $Pr(\mathcal{E}_{2.2}) = O\left(\frac{1}{n^3}\right)$.

Union over the failure events, the failure probability is less than: $Pr(\mathcal{E}_1) + Pr(\mathcal{E}_{2.1}) + Pr(\mathcal{E}_{2.2}) = O(\frac{4}{n^3})$. □

Having analyzed the DRA algorithm, we return to the discussion of our DHC1 algorithm.

Analysis of Phase 1: Each subgraph G_i uses the DRA algorithm to independently construct (in parallel) its Hamiltonian cycle C_i. Because each subgraph performs the algorithm independently, this phase is fully parallelized, and the expected runtime will be the expected runtime of the largest subgraph. For the failure probability, we can simply use a union bound. We state the following result for Phase 1.

Lemma 3. *For a $G(n,p)$ with $p \geq \frac{c \ln n}{\sqrt{n}}$ where $c \geq 86$, Phase 1 of the algorithm succeeds with probability $1 - O(1/n)$, in $O(\sqrt{n} \ln n)$ steps.*

To prove Lemma 3, we will show that each partition has a size of $\Theta(\sqrt{n})$ and is sufficiently dense for the success of the DRA algorithm. In particular, we introduce the following:

Definition 1. *Let $\mathcal{A}$ be the event that all partitions have size $a\sqrt{n}$, where $a \in [\frac{1}{2}, \frac{3}{2}]$.*

Lemma 4. *DHC1 algorithm in Phase 1 (line 5) partitions nodes such that event $\mathcal{A}$ happens with probability at least $1 - O(\frac{1}{n})$.*

Proof. Consider any single color. Let X be a random variable representing the number of nodes with that color. Let $X_i, i = 1, \cdots, n$ be indicator random variables of values $0, 1$:

$X_i = 1$ if node i chooses that color, $X_i = 0$ otherwise. By linearity of expectation, we have $E[X] = E[\sum X_i] = \sum E[X_i] = n\frac{1}{\sqrt{n}} = \sqrt{n}$.

In order to show that X is concentrated around its expectation, $\frac{1}{2}E[X] \leq X \leq \frac{3}{2}E[X]$, we apply Chernoff's bound:

$$Pr(|X - \sqrt{n}| \geq \frac{1}{2}\sqrt{n}) \leq 2e^{\frac{-(\frac{1}{2})^2\sqrt{n}}{3}} = 2e^{\frac{-\sqrt{n}}{12}}.$$

With $\sqrt{n}$ partitions, by union bound, we have:

$$Pr(\neg\mathcal{A}) \leq \sqrt{n} \times 2e^{\frac{-\sqrt{n}}{12}} = O\left(\frac{1}{n}\right).$$

□

Lemma 5. *When event $\mathcal{A}$ happens, Phase 1 succeeds with probability $1 - O(\frac{1}{n})$.*

Proof. By Lemma 4, each partition has size of $a\sqrt{n}$, where $\frac{1}{2} \leq a \leq \frac{3}{2}$. Consider a partition with n' vertices as a random graph with probability p'. It is easy to show that $p' \geq 86 \ln n'/n'$, as follows. The probability for the presence of an edge in this partition is the same as in the original graph. We have:

$$p' = p \geq 86\frac{\ln n}{\sqrt{n}} = 86\frac{\ln \frac{n'^2}{a^2}}{\frac{n'}{a}} = 86a\frac{2\ln n' - \ln a^2}{n'}.$$

When $1/2 \leq a < 1$, then $p' \geq 86a\frac{2\ln n'}{n'} \geq 86\frac{\ln n'}{n'}$.
When $1 \leq a \leq 3/2$, then $p' \geq 86a\frac{2\ln n'}{2an'} = 86\frac{\ln n'}{n'}$, using the fact that $x - y > \frac{x}{2z}$, for x sufficiently large and small constants y, z such that $z > 1$.

Applying Theorem 2, the probability of failing for this partition is $O(\frac{1}{(\sqrt{n})^3})$. Using a union bound, the probability of failure in Phase 1 is at most: $\sqrt{n} \times O(\frac{1}{(\sqrt{n})^3}) = O(\frac{1}{n})$. □

Proof of Lemma 3. By Lemma 5 and Lemma 4, the probability of failure for Phase 1 of is $O(\frac{1}{n})$.

For the runtime of this phase, we apply Theorem 2. Consider a partition of size $a\sqrt{n}$, the runtime is $7a\sqrt{n}\ln(a\sqrt{n})$. Each partition executes Algorithm 1 in parallel, the runtime is dominated by the largest partition. Since $a \leq \frac{3}{2}$, the runtime of Phase 1 is $O(\sqrt{n}\ln n)$. □

Analysis of Phase 2: In this phase, we apply the DRA algorithm on the G' graph of hypernodes. We only need to show that G' is dense enough to apply Theorem 2. We have the following lemma.

Lemma 6. *For a $G(n,p)$ with $p \geq \frac{c\ln n}{\sqrt{n}}$ where $c \geq 86$, Phase 2 of the DHC1 algorithm succeeds with probability $O\left(1-\frac{1}{n^{\frac{3}{2}}}\right)$, in $O(\sqrt{n}\ln n)$ steps.*

Proof. The graph G' constructed according to the algorithm is a random graph with $n' = \sqrt{n}$ and the edge probability p'. Consider a pair $(e_i = [v_i, u_i], e_j = [v_j, u_j])$ of hypernodes; by construction, the probability to have an edge between them is $p' = 1-(1-p)^2 \geq p$, where p is the probability for an edge between two nodes in the original G graph.

$$p' \geq p \geq 86\frac{\ln n}{\sqrt{n}} > 86\frac{\ln n'}{n'}.$$

Applying Theorem 2 this phase succeeds with probability $O\left(1-\frac{1}{n^{\frac{3}{2}}}\right)$ in $O(\sqrt{n}\ln n)$ steps. □

Proof of Theorem 1. The proof of the main theorem then follows trivially, by Lemma 3 and Lemma 6. The probability of success is:

$$O\left(1-\frac{1}{n}\right)O\left(1-\frac{1}{n^{3/2}}\right) = O\left(1-\frac{1}{n}\right).$$

The number of steps in each phase is: $O(\sqrt{n}\ln n)$. In the worst case, consider we have broadcast in every step, then, the number of rounds is the number of steps multiplied by $O(D)$ where D is the diameter of the graph executing the DRA algorithm. In both Phase 1 and Phase 2, the graphs are random graphs under the model $G(n', p')$ where $p' \geq 86\ln n'/n'$, and $n' = \Theta(\sqrt{n})$. By [24], the diameter of these graphs is $\Theta(\frac{\ln n'}{\ln\ln n'}) = \Theta(\frac{\ln n}{\ln\ln n})$.

Therefore, the number of rounds is bounded by:

$$O\left(\sqrt{n}\frac{(\ln n)^2}{\ln \ln n}\right).$$

□

3.3.2 The Algorithm for $p = \frac{c \ln n}{n^\delta}$

We proved that for a $G(n,p)$ with $p = \frac{c \ln n}{\sqrt{n}}$, the DHC1 algorithm 2 finds a Hamiltonian cycle in $\tilde{O}(\sqrt{n})$ times. It is natural to ask the question: what is the performance on sparser graphs? Consider a $G(n,p)$ random graph where $p = O(\frac{c \ln n}{n^\delta})$, for any $\delta \in (0,1)$. If we divide the graph into $n^{1-\delta}$ partitions, each of size n^δ, then Phase 1 of the DHC1 algorithm will work. However, Phase 2 will not, since the graph of hypernodes is too sparse, under the threshold required for the presence of an Hamiltonian cycle in Phase 2.

We present a general Algorithm 3 called DHC2 that finds a Hamiltonian cycle in random graphs $G(n,p)$ where $p = O(\frac{c \ln n}{n^\delta})$, where c is a suitably large constant. This algorithm also has two phases. Phase 1 is essentially a generalization of Phase 1 of DHC1, with $n^{1-\delta}$ partitions. In Phase 2 of DHC2, we recursively merge pairs of two disjoint cycles (in parallel) until the final cycle is formed. Figure 3.3 depicts these merging steps. It follows that the algorithm constructs the final Hamiltonian cycle if it always successes in merging. We will show that this probability is very high. But let's first describe the merging procedure.

To merge the cycles, we define a rule for pairing them, then describe the merging by finding a "bridge" between a pair of two cycles, as explained below. Let's have the cycles indexed by colors: $HC_1, HC_2, ..., HC_{n^{1-\delta}}$. The pairing rule is to match two consecutive cycles, from left to right: $(HC_1, HC_2), \cdots, (HC_{2k+1}, HC_{2k+2}), \cdots$, at most one cycle will be left out. Each pair merges independently in parallel, then every node (thus every cycle, including the one left out), updates its respective color: $color \leftarrow \lceil color/2 \rceil$. Therefore, the next merge step can progress with the same pairing rule, and every cycle is aware of its pair in

Algorithm 3 Distributed Hamiltonian Cycle Algorithm 2 (DHC2). Code for $v \in G(V, E)$.

1: **Phase** 1
2: Run phase 1 of algorithm 2, using $n^{1-\delta}$ colors
3: **Phase** 2
4: **for** $i = 1 \cdots \lceil \log n^{1-\delta} \rceil$ **do**
5: **if** *v.color* is *odd* **then** ▷ v is an active node
6: send message *verify*(*succ*(v)) to all its neighbors with color *v.color* + 1
7: **OnReceive** $\cup\{$*verified*$(u, u')\}$
8: Select the smallest (u, u'), construct candidate bridge: *candidate* $\leftarrow ((v, u'), (u,$ *succ*$(v)))$
9: Broadcast *candidate* within v's partition
10: **if** *candidate* = *min*($\cup$*candidates*) **then**
11: Send message *buildBridge* to u
12: Broadcast *Renumbring* inside HC
13: **OnReceive** message *verify*(u) ▷ only passive nodes receive this type message
14: ask *succ*(v) and *pred*(v) if they have u as their (*v.color* $-$ 1) neighbor
15: if *succ*(v) (or *pred*(v)) confirmed, set u' to *succ*(v) (or *pred*(v)) , reply to sender: *verified*(v, u')
16: **OnReceive** message *buildBridge*
17: Broadcast *Renumbring* HC
18: *v.color* $\leftarrow \lceil$ *v.color*$/2 \rceil$

all steps. It is clear that we need $\lceil \log(n^{1-\delta}) \rceil = O(\log(n))$ merge steps. To merge two cycles, we need to pick one "bridge" between them. Let $e_i = (v_i, u_i) \in HC_i$ and $e_j = (v_j, u_j) \in HC_j$ where (HC_i, HC_j) is a pair. If there are two edges (v_i, v_j) and (u_i, u_j) or two edges (v_i, u_j) and (u_i, v_j) in $G(n, p)$, then we say (e_i, e_j) is a bridge of (HC_i, HC_j). The idea is, that each node can check if it is part of a bridge, in parallel. Then within HC_i and HC_j, each node broadcasts the discovered bridge. This is done so as to choose one unique bridge per pair (since there may be more than one bridge per pair). Each cycle chooses the smallest bridge (say, based on the IDs of the bridge nodes). Once a bridge is chosen, for example, merging is done by each node independently updating its *cycindex*, and updating *color* (as mentioned above) for the next merging step. For efficiency, in a pair, only the cycle with smaller *color* will initiate the process, as shown in Algorithm 3.

Also, to avoid cluttering Algorithm 3, we did not specify the renumbering process. This

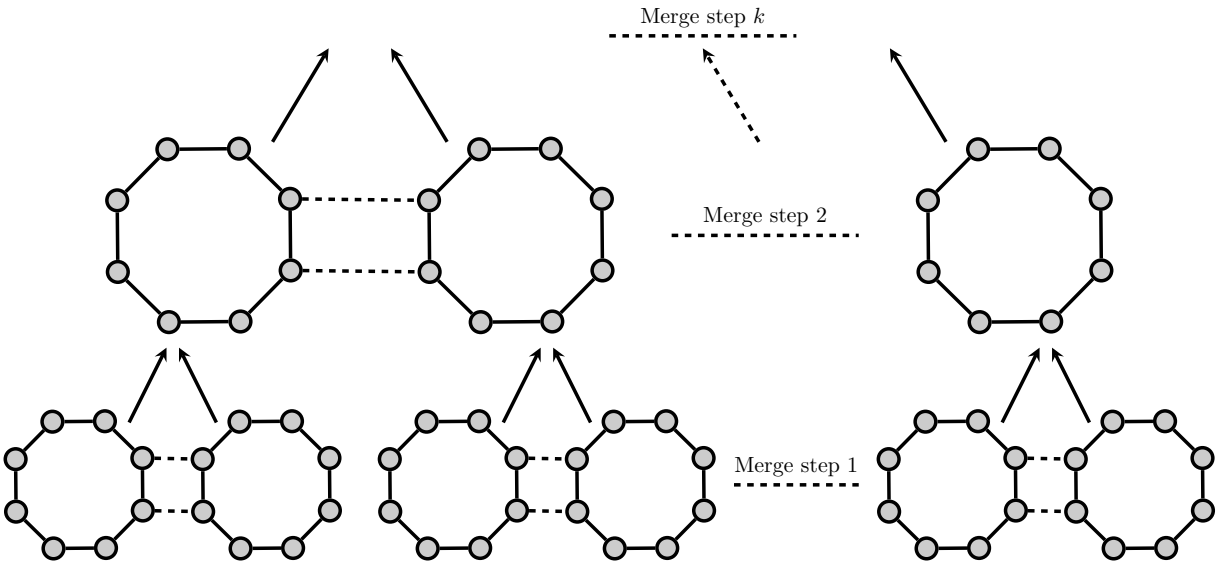


Figure 3.3: Phase 2 of the DHC2 algorithm: Merging pairs of cycles in a tree-like fashion. There are $O(\log n)$ merge steps; in each step, all HC pairs merge in parallel. The figure also shows how two cycles are merged into a larger cycle by choosing two bridge edges.

is trivial, given the bridge and the size of the two cycles. Initially, each cycle performs a broadcast, so that its member nodes get to know the cycle size. Then, this information is attached to the bridge building message. From here onward, every node can keep track of the size of the cycle that it is part of until the merging process is finished.

Lemma 7. *Phase 1 of the DHC2 algorithm succeeds in $O(n^\delta \ln n)$ steps, with probability at least $1 - O(\frac{1}{n})$.*

Proof. Similar to Lemma 4, it is easy to see that all partitions have size concentrated around the expected size, which is $\Theta(n^\delta)$. Consider a single color; let X be the random variable of the size of the corresponding partition. Let X_i be indicator random variables: $X_i = 1$ if node i chooses that color, 0 other wise. By linearity of expectation we have $E[X] = \frac{n}{n^{1-\delta}} = n^\delta$. Chernoff's bound gives

$$Pr(|X - n^\delta)| \geq \frac{1}{2}n^\delta) \leq 2e^{-n^\delta/12}.$$

By a union bound, all $n^{1-\delta}$ partitions have sizes in $\Theta(n^\delta)$, with probability

$$O\left(n^{1-\delta}2e^{-n^\delta/12}\right) = O\left(\frac{1}{n}\right).$$

Consider a partition with size: $n' = \Theta(n^\delta)$, as a random graph with edge probability p'. We have

$$p' = p = O\left(\frac{\ln n}{n^\delta}\right) = O\left(\frac{1}{\delta}\frac{\ln n'}{n'}\right) = O\left(\frac{\ln n'}{n'}\right).$$

By Theorem 2, note that we can reduce the probability of failure to $O\left(\frac{1}{n^{2-\delta}}\right)$ by increasing the number of steps by some factor of $(2-\delta)$. Thus, the number of required steps is $O((2-\delta)n' \ln n') = O(n^\delta \ln n)$.

Using union bound for $n^{1-\delta}$ partitions, the probability of failure is bounded above by

$$n^{1-\delta} \times O\left(\frac{1}{n^{2-\delta}}\right) = O\left(\frac{1}{n}\right).$$

□

To prove that Phase 2 of the DHC2 algorithm succeeds, we will first show the probability of success for merge step 1.

Lemma 8. *The merging of $\frac{np}{\ln n} = n^{1-\delta}$ Hamiltonian cycles in the first merging step of Phase 2 will be successful, with very high probability.*

Proof. Consider two partitions with two cycles C, C', each with expected size n^δ. Fix an edge e in C; the probability that e has a bridge to a fixed edge in C' is at least p^2. Consider the set S' of all non-adjacent edges in C', such that $|S'|$ is maximal. The probability that e does not have any bridge to C' is at most the probability that e does not have any bridge to S'

$$\begin{aligned}(1-p^2)^{n^\delta/2} &= O\left(\left(1 - \frac{(\ln n)^2}{(n^\delta)^2}\right)^{n^\delta/2}\right) \\ &= O\left(\left(e^{-(\ln n)^2}\right)^{1/(2\sqrt{n^\delta})}\right) \\ &= O\left(n^{-\frac{\ln n}{2n^{\delta/2}}}\right).\end{aligned}$$

Consider the set S of all non-adjacent edges in C, such that $|S|$ is maximal, the probability that all edges in S has no bridge to C' is

$$O\left(\left(n^{-\frac{\ln n}{2n^{\delta/2}}}\right)^{n^\delta/2}\right) = O\left(n^{-n^{\delta/2}\ln n}\right).$$

The above is the bound for the probability that C and C' fail to merge. We have $n^{1-\delta}/2$ pairs to merge, thus, union bound gives the failure probability

$$\frac{n^{1-\delta}}{2} \times O\left(n^{-n^{\delta/2}\ln n}\right) = O\left(n^{-n^{\delta/2}\ln n+1-\delta}\right).$$

□

Lemma 9. *Phase 2 of the HHC algorithm is successful with very high probability which is* $1 - o(1/n)$.

Proof. Observe that after merging, the sizes of the Hamiltonian cycles increase, thus, in successive merge steps, the probability of failure becomes smaller than that in the first step. Using Lemma 8, with $O(\log n)$ merge steps, union bound of the failure of Phase 2 is

$$O\left(\ln n \cdot n^{-n^{\delta/2}\ln n+1-\delta}\right) = o\left(\frac{1}{n}\right).$$

□

Theorem 10. *The DHC2 algorithm succeeds with probability* $1 - O(\frac{1}{n})$ *in* $\tilde{O}(n^\delta)$ *steps.*

Proof. By Lemma 7 and Lemma 9, the probability that the DHC2 algorithm succeeds is

$$\left(1 - O\left(\frac{1}{n}\right)\right)\left(1 - o\left(\frac{1}{n}\right)\right) = 1 - O\left(\frac{1}{n}\right).$$

To find the time complexity, we proceed similarly to the analysis of DHC1 algorithm: first calculate the number of steps, then consider the number of rounds required for broadcast.

In Phase 1, the size of a subgraph is $n' = \Theta(n^\delta)$, and the edge probability is $p' = O(\ln n'/n')$, and by [24], the diameter is $O(\frac{\ln n'}{\ln \ln n'}) = O(\frac{\ln n}{\ln \ln n})$.

In Phase 2, each merging takes constant number of rounds, and the broadcast time depends on the diameter of a subgraph. Observe that after each merging, we have a larger subgraph, while the edge probability is fixed, thus relative to the size, this larger subgraph is denser. Therefore we can bound the diameter of the merged subgraphs by the diameter of subgraphs in the first level, which is $O(\frac{\ln n}{\ln \ln n})$.

The number of rounds for our DHC2 algorithm is then

$$\begin{aligned} &O\left(n^{\delta} \ln n \frac{\ln n}{\ln \ln n}\right) + O\left(\ln n \frac{\ln n}{\ln \ln n}\right) \\ &= O\left(\frac{n^{\delta} (\ln n)^2}{\ln \ln n}\right). \end{aligned}$$

□

3.4 The Upcast Algorithm: A Centralized Approach

In this section we consider what perhaps is the simplest and most obvious strategy of all — we collect "sufficiently large" number of edges at some pre-designated root and then leave it to the root to compute a Hamiltonian cycle, see Algorithm 4.

Algorithm 4 The Upcast Algorithm

1: Elect a leader, call it v. This step takes $O(D)$ rounds.
2: Construct a BFS tree rooted at v, and call it $\mathcal{B}$. This step takes $O(D)$ rounds.
3: All nodes except v sample some $c' \log n$ of their adjacent edges (for a sufficiently large constant c') — independently and randomly — and send the sampled edges to v via the BFS tree constructed in the previous step.
4: The root v computes a Hamiltonian cycle locally and downcasts it to the rest of the nodes in G. This step takes essentially the same number of rounds as the previous (upcast) step.

The main technical challenge in the analysis is showing that the upcast can be done in time $\tilde{O}(1/p)$. This is done by showing that in a BFS tree in a random graph, the sizes of the subtrees rooted at every node are balanced (i.e., essentially the same size) whp. This

ensures that the congestion at each node during upcast is balanced and is $\tilde{O}(1/p)$.

3.4.1 Analysis for the Special Case when $p = \Theta(\frac{\log n}{\sqrt{n}})$.

Let D be the diameter of $G = (V, E)$. Then Corollary 7 in [13] implies that

Fact 1. $D = 2$ when $p = \Theta(\frac{\log n}{\sqrt{n}})$.

Thus Steps 1 and 2 take $O(1)$ time in total. We claim that Step 3 in the algorithm takes $O(\sqrt{n}\log^2 n)$ rounds with high probability.

For $i \geq 0$, let L_i be the nodes at level i in the BFS tree $\mathcal{B}$. That is, $L_0 = \{v\}$, $L_1 = \{w \in V \mid (v, w) \in E\}$, and $L_2 = \{w \in V \mid dist(v, w) = 2\}$. We note that $L_0 \cup L_1 \cup L_2 = V$ by dint of Fact 1.

Lemma 11. *$c(1-\delta_1)(1-\delta_2)\sqrt{n}\log n \leq |L_1| \leq c(1+\delta_1)\sqrt{n}\log n$ with high probability for any fixed constants $\delta_1, \delta_2 \in (0, 1)$.*

Proof. As $p = \frac{c\log n}{\sqrt{n}}$, $E[|L_1|] = (n-1)p = \frac{c(n-1)\log n}{\sqrt{n}} = c\sqrt{n}\log n - o(1) \implies (1-\delta_2)c\sqrt{n}\log n \leq E[|L_1|] \leq c\sqrt{n}\log n$, for any fixed constant δ_2 in $(0, 1)$. A simple application of Chernoff's bound gives us

$$\begin{aligned} &Pr(|L_1| \geq c(1+\delta_1)\sqrt{n}\log n) \\ &\leq \exp(-\frac{\delta_1^2 \cdot c(1-\delta_2)\sqrt{n}\log n}{3}) \\ &= n^{-\frac{\delta_1^2 \cdot c(1-\delta_2)\sqrt{n}}{3}}. \end{aligned}$$

Similarly,

$$\begin{aligned}
&Pr(|L_1| \leq c(1-\delta_1)(1-\delta_2)\sqrt{n}\log n) \\
&\leq \exp(-\frac{\delta_1^2 \cdot c(1-\delta_2)\sqrt{n}\log n}{2}) \\
&= n^{-\frac{\delta_1^2 \cdot c(1-\delta_2)\sqrt{n}}{2}}.
\end{aligned}$$

□

Lemma 12. *$n - (1 + c(1+\delta_1)\sqrt{n}\log n) \leq |L_2| \leq n - (1 + c(1-\delta_1)(1-\delta_2)\sqrt{n}\log n)$ with high probability for any fixed constants $\delta_1, \delta_2 \in (0,1)$.*

Proof. Follows directly from Fact 1 and Lemma 11. □

For $w \in L_1$, let $\Gamma_{\mathcal{B}}(w)$ be the set of children of w in the BFS tree $\mathcal{B}$. Then

Lemma 13. *$(1-\delta_3)(n-(1+c(1+\delta_1)\sqrt{n}\log n))p \leq |\Gamma_{\mathcal{B}}(w)| \leq (1+\delta_3)(n-(1+c(1-\delta_1)(1-\delta_2)\sqrt{n}\log n))p$ with high probability for any fixed constants $\delta_1, \delta_2, \delta_3 \in (0,1)$.*

Proof. Similar to that of Lemma 11. □

Lemma 14. *$c(1-\delta_3)(1-\delta_4)(1-\delta_5)\sqrt{n}\log n \leq |\Gamma_{\mathcal{B}}(w)| \leq c(1+\delta_3)(1+\delta_4)\sqrt{n}\log n$ with high probability for any fixed constants $\delta_3, \delta_4, \delta_5 \in (0,1)$.*

Proof. Simplifying Lemma 13. □

Since the "high probability" in Lemma 14 is actually exponentially high [6] (please refer to the proof of Lemma 11), we can take a union bound over all $w \in L_1$, and get the following lemmas.

Lemma 15. *The following statement holds with high probability: For all $w \in L_1$, $c(1-\delta_3)(1-\delta_4)(1-\delta_5)\sqrt{n}\log n \leq |\Gamma_{\mathcal{B}}(w)| \leq c(1+\delta_3)(1+\delta_4)\sqrt{n}\log n$ for any fixed constants $\delta_3, \delta_4, \delta_5 \in (0,1)$.*

[6] that is $\geq 1 - \frac{1}{n^{\mathrm{poly}(n)}}$.

Lemma 16. *The upcast process takes at most* $\frac{b}{\mathbb{B}} \cdot (c' \log n + cc'(1+\delta_3)(1+\delta_4)\sqrt{n}\log^2 n)$ *rounds, where* $\mathbb{B}$ *is the bandwidth of the network, each edge is encoded in* b *bits, and* $0 < \delta_3, \delta_4 < 1$ *are fixed constants.*

Proof. Follows directly from Lemma 15. □

Usually we would have $b = \Theta(\log n)$ and $\mathbb{B} = \Theta(\log n)$, and that gives us the main result of this section:

Theorem 17. *The Upcast algorithm solves the distributed Hamiltonian Cycle problem in* $G(n,p)$ *random graphs in* $O(\sqrt{n}\log^2 n)$ *rounds, when* $p = \Theta(\frac{\log n}{\sqrt{n}})$. *Both the success probability and the running time hold with high probability.*

3.4.2 Analysis for the General Case when $p = \Theta(\frac{\log n}{n^{1-\delta}})$ for some constant $\delta \in (0,1)$

Let D be the diameter of the graph $G = (V, E)$. Let K be the smallest integer such that $K\delta \geq 1$, i.e., $K \stackrel{\text{def}}{=} \lceil \frac{1}{\delta} \rceil$. Then Klee and Larman showed that [58]

Fact 2. $Pr(D(G) = K) \to 1$ *as* $n \to \infty$, *when* $p = \frac{c \log n}{n^{1-\delta}}$ *for some positive constant* c.

Thus Steps 1 and 2 in the upcast algorithm take $O(1)$ time in total. We claim that Step 3 takes $O(\frac{\log n}{p}) = O(n^{1-\delta})$ rounds.

In a graph G, we denote by $\Gamma_k(x)$ the set of vertices in G at distance k from a vertex x

$$\Gamma_k(x) \stackrel{\text{def}}{=} \{y \in G \mid dist(x,y) = k\}.$$

We define $\mathcal{N}_k(x)$ to be the set of vertices within distance k of x

$$\mathcal{N}_k(x) \stackrel{\text{def}}{=} \bigcup_{i=0}^{k} \Gamma_i(x).$$

We can adapt Lemma 3 in [24] to show that

Lemma 18. *For any constant $\epsilon > 0$, with probability at least $1 - \frac{1}{n^3}$, we have*

1. $|\Gamma_i(x)| \leq (1+\epsilon)(np)^i$, $\forall 1 \leq i \leq D$.
2. $|\mathcal{N}_i(x)| \leq (1+2\epsilon)(np)^i$, $\forall 1 \leq i \leq D$.

Lemma 18 basically says that the BFS tree $\mathcal{B}$ is essentially balanced. Hence an upcast algorithm would take $O(\frac{b}{\mathbb{B}} \cdot (1+\epsilon)^D \cdot \frac{n \log n}{d_v})$ rounds, where δ is any fixed positive constant, $n = |\mathcal{B}|$, and d_v is the degree of the root v. As $D = K = \lceil \frac{1}{\epsilon} \rceil$ is a constant, this implies a time complexity of $O(\frac{n \log n}{d_v})$. But d_v is concentrated around np with high probability. Thus an upcast algorithm would take $O(\frac{n \log n}{np}) = O(\frac{\log n}{p})$ rounds with high probability. That is the main theorem of this section:

Theorem 19. *The Upcast algorithm solves the distributed Hamiltonian Cycle problem in $G(n,p)$ random graphs in $O(\frac{\log n}{p}) = O(n^{1-\delta})$ rounds, when $p = \Theta(\frac{\log n}{n^{1-\delta}})$ for some constant $\delta \in (0,1)$. Both the success probability and the running time hold with high probability.*

3.4.3 Distributed Verification of a Hamiltonian Cycle

We use a distributed MST algorithm in [49] in order to verify a Hamiltonian cycle. First, we set weight zero for each edge on Hamiltonian cycle and none zero value for other edges. If the MST algorithm builds a minimum spanning tree successfully having the sum of the edges zero, we check whether the two end nodes are connected by an edge of weight zero or not (an edge belonging to the Hamiltonian cycle).

Table 3.1: Comparison of different distributed algorithms to find HC in $G(n,p)$ random graphs.

	Method	p	Time
1	Angluin, et al.[7]	$\frac{c\log n}{n}, c \geq 36$	$O(n(\log n)^2)$
2	Levy, et al. [67]	$O\left(\frac{\sqrt{\log n}}{n^{\frac{1}{4}}}\right)$	$O\left(n^{3/4+\epsilon}\right)$
3	**DHC1**	$\frac{c\ln n}{\sqrt{n}}, c \geq 86$	$O(\sqrt{n}\frac{\ln^2 n}{\ln\ln n})$
4	**DHC2**	$O(\frac{\ln n}{n^\delta})$	$O\left(\frac{n^\delta(\ln n)^2}{\ln\ln n}\right)$
5	Turau [99]	$\frac{(\log n)^{\frac{3}{2}}}{\sqrt{n}}$	$O(\log n)$

3.5 Summary

We presented efficient distributed algorithms for the fundamental Hamiltonian cycle problem in random graphs. Table 3.1 compares and highlights our achievement compared to relevant methods that runs on $G(n,p)$ random graphs. Our algorithms are labeled $DHC1$ and $DHC2$. Turau in [99] proposed an improved algorithm recently.

Chapter 4

Community Detection

Finding communities in networks (graphs) is an important problem and has been extensively studied in the last two decades, e.g., see the surveys [1, 27, 39, 40] and other references in Section 2. At a high level, the goal is to identify subsets of vertices of the given graph so that each subset represents a "community." While there are differences in how communities are defined exactly (e.g., subsets defining a community may overlap or not), a uniform property that underlies most definitions is that there are more edges connecting nodes *within* a subset than edges connecting to *outside* the subset. Thus, this captures the fact that a community is somehow more "well-connected" with respect to itself compared to the rest of the graph. One way to express this property formally is by using the notion of *conductance* (defined in Section 4.3) which (informally) measures the ratio of the total degree going out of the subset to the total degree among all nodes in the subset. A community subset will have generally low conductance (also sometime referred to as a "sparse" cut). Another way to measure this is by using the notion of *modularity* [78] which (informally) measures how well connected a subset is compared to the random graph that can be embedded within the set. A vertex subset that has a high modularity value can be considered a community according to this measure.

Designing effective algorithms for community detection in graphs is an important and challenging problem. With the rise of massive graphs such as the Web graph, social networks, biological networks, finding communities (or clusters) in large-scale graphs efficiently is becoming even more important [27, 34, 40, 50, 78]. In particular, understanding the community structure is a key issue in many applications in various complex networks including biological and social networks (see e.g., [27] and the references therein).

Various solutions have been proposed (cf. Section 2), but many of them are centralized with expensive procedures (requiring full knowledge of the input graph) and have a large running time [27]. In particular, the problem of detecting (identifying) communities in the *stochastic block model (SBM)* [1, 27] has been extensively studied in the literature (cf. Section 2). The stochastic block model (defined formally in Section 4.2), also known as the *planted partition model (PPM)* is a widely-used random graph model in community detection and clustering studies (see e.g., [1, 23, 57]). Informally, in the PPM model, we are given a graph G on n nodes which are partitioned into a set of r communities (each is a random graph on n/r vertices) and these communities are interconnected by random edges. The total number of edges within a community (intra-community edges) is typically much larger than the number of edges between communities (inter-community edges). The main goal is to devise algorithms to identify the r communities that are "planted" in the graph. Several algorithms have been devised for this model, but as mentioned earlier, they are mostly centralized (with some exceptions — cf. Section 2) with large running time. There are few distributed algorithms (see e.g., [27, 62]) but either they are shown to work only when the number of communities is small (typically 2) [27]) or when the communities are very dense [62]). In particular, to the best of our knowledge, (prior to this work) there is no rigorous analysis of community detection algorithms in distributed large-scale graph processing models such as MapReduce and the Massively Parallel Computing (MPC) model [56], and the k-machine model [58].

4.1 Our Contributions

In this dissertation, we present a novel distributed algorithm, called *CDRW* (*Community Detection via Random Walks*) for detecting communities in the PPM model [29].[1] Our algorithm is based on the recently proposed *local mixing* paradigm [75] (see Section 4.3 for a formal definition) to detect community structure in *sparse* (bounded-degree) graphs. Informally, a local mixing set is one where a random walk started at some node in the set mixes well *with respect to this set.* The intuition in using this concept for community detection is that since a community is well-connected, it has good expansion within the community and hence a random walk started at a node in the community mixes well within the community. The notion of "mixes well" is captured by the fact that the random walk reaches close to the stationary distribution when *restricted to the nodes in the community subset* [75]. Since the main tool for this algorithm uses random walks which are local and lightweight, it is easy to implement this algorithm in a distributed manner. We will analyze the performance of the algorithm in two distributed computing models, namely the standard CONGEST model of distributed computing [87] and the k-machine model [58], which is a model for large-scale distributed computations. We show that CDRW can be implemented efficiently in both models (cf. Theorem 25 and Section 4.4.2). The k-machine model implementation is especially suitable for large-scale graphs and thus can be used in community detection in large SBM graphs. In particular, we show that the round complexity in the k-machine model (cf. Section 4.4.2) scales quadractically (i.e., k^{-2}) in the number of machines when the graph is sparse and it scales linearly (i.e., k^{-1}) in general.

As is usual in community detection, a main focus is analyzing the effectiveness of the algorithm in finding communities. We present a rigorous theoretical analysis that shows that the CDRW algorithm can accurately identify the communities in the PPM, which is a

[1] Throughout this dissertation, we use the terms stochastic block model (SBM) and planted partition model (PPM) interchangeably.

popular and widely-studied random graph model for community detection analysis [1]. A PPM model (cf. Section 4.2) is a parameterized random graph model which has a built-in community structure. Each community has high expansion within the community and forms a low conductance subset (and hence relatively fewer edges go outside the community); the expansion, conductance, and edge density can be controlled by varying the parameters. CDRW does well when the number of intra-community edges is much larger than the number of inter-community edges (these are controlled by the parameters of the model). Our theoretical analysis (cf. Theorem 25 for the precise statement) quantitatively characterizes when CDRW does well vis-a-vis the parameters of the model. Our results improve over previous distributed algorithms that have been proposed for the PPM model ([27]) both in the number of communities that can be provably detected as well as range of parameters where accurate detection is possible; they also improve on previous results that provably work only on dense PPM graphs [62] (details in Section 2).

We also present extensive simulations of our algorithm in the PPM model under various parameters. Experimental results on the model validate our theoretical analysis; in fact our experiments show that CDRW works relatively well in identifying communities even under less stringent parameters.

4.2 Model, Definitions, and Preliminaries

4.2.1 Graph Model

We describe the *stochastic block model (SBM)* [52], a well-studied random graph model that is used in community detection analysis. Before that we recall the *Erdős-Rényi* random graph model [37], a basic random graph model, that the SBM model builds on. In the Erdős-Rényi random graph model, also known as the G_{np} model, each of $\binom{n}{2}$ possible edges

is present in the graph independently with probability p. The G_{np} random graph has close to uniform degree (the expected degree of a node is $(n-1)p$) and a well-known fact about $G(n,p)$ is that if $p = \Theta(\log n/n)$ the graph is connected with high probability[2] and its degree is concentrated at its expectation. In the SBM model, the vertices are partitioned into r disjoint community sets (r is a parameter). Let $c(u)$ denote the community that node u belongs to. If two nodes u and v belong to the same community (i.e., $c(u) = c(v)$), they connect independently with probability $p_{c(u)}$ (independent of other nodes). If they belong to different communities, they connect independently with probability $p_{c(u),c(v)}$. A SBM model has a separable community structure [1] if it has a higher value of intra- than inter-community connectivity probability. A commonly used version of SBM model called *Planted Partition Model (PPM)* and denoted by G_{npq} [28, 52] is usually used as a benchmark. A symmetric G_{npq} with r blocks is composed of r exclusive set of nodes in $\mathcal{C} =< C_1, C_2, \ldots, C_r >$, where $\cup_{i=1}^{i=r} C_i = V$, $|C_i| = |C_j|$ and $C_i \cap C_j = \emptyset$ when $i \neq j$. In G_{npq}, each possible edge $e(u,v)$ is independently present with probability p if both ends of u and v belong to the same block C_i, otherwise with probability q. Figure 4.1 shows a PPM of 5 blocks ($r = 5$), each containing 200 nodes. For the PPM model, each of the r communities induce a n/r-vertex subgraph which is a $G(n/r, p)$ random graph. The goal of community detection in the PPM (or SBM) model is to identify the r community sets. This problem has been widely studied [1].

4.2.2 Notion of a Community

While there is not a universally accepted definition for communities [40], in this dissertation, we use a reasonable definition that is either the same or closely related to what is used in the literature (cf. see Section 2). Informally, we define a community as a subset of nodes ($C_i \subset V$) where nodes inside the subset are "more connected" compared to nodes outside of the community. In other words, for each $v \in C_i$ it is $\sum_{u \in C_i} I_{e(v,u)} > \sum_{u \in V/C_i} I_{e(v,u)}$

[2]Throughout this dissertation, by "with high probability" we mean, "with probability at least $1 - 1/n$."

(a) A PPM graph shown without its ground-truth communities.

(b) Redrawing of the same PPM graph showing the communities.

Figure 4.1: Both of the above graphs are the same PPM graph. The graph size is $n = 1000$, the number of communities is $r = 5$, the existence probability of intra(inter) community edges is $p = \frac{1}{20}(q = \frac{1}{1000})$ as in [1, Figure 1]. We highlight ground truth communities in different colors in Figure 4.1b.

where I is an indicator function and returns 1 if an edge $e(v, u)$ exists, otherwise 0. In this dissertation, we look for non-overlapping communities of an undirected graph $G(V, E)$. Let $\mathcal{C} = \{C_1, C_2, ..., C_r\}$; we call it non-overlapping if $\cup C_i = V$ and $C_i \cap C_j = \emptyset$ for all $i \neq j$. For the SBM, the communities are the r subsets of the vertices that induce a random graph. In particular, for the PPM model, each of the r communities induce a n/r-vertex subgraph which is a $G(n/r, p)$ random graph. The goal of community detection in the PPM (or SBM) model is to identify the r community sets. This problem has been widely studied [1].

4.2.3 Distributed Computing Models

We consider two distributed computing models and analyze the complexity of the algorithm's implementation under both the models.

CONGEST model. This is a standard and widely-studied model of distributed computing [81], which captures the bandwidth constraints inherent in real-world networks. The

distributed network is modeled as an undirected and unweighted graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, where nodes represent processors (or computing entities) and the edges represent communication links. In this dissertation, G will be a graph belonging to the PPM model. Each node is associated with a distinct label or ID (e.g., its IP address). Nodes communicate through the edges in synchronous rounds; in every round, each node is allowed to send a message of size at most $O(\log n)$ bits to each of its neighbors. The message will arrive to the receiver nodes at the end of the current round. Initially every node has limited knowledge; it only knows its own ID and its neighbors IDs. In addition, it may know additional parameters of the graph such as n, m, D (where D is the diameter). The two standard complexities of an algorithm are the time and message complexity in the CONGEST model. While time complexity measures the number of rounds taken by the algorithm, the message complexity measures the total number of messages exchanged during the course of the algorithm.

k-machine model. The k-machine model (a.k.a. Big Data model) is a model for *large-scale* distributed computing introduced in [58] and studied in various papers [9, 55, 58, 83, 85]. In this model, the input graph (or more generally, any other type of data) is distributed across a group of $k \geq 2$ machines that are pairwise interconnected via a communication network. The k machines jointly perform computations on an arbitrary n-vertex, m-edge input graph (where typically $n, m \gg k$) distributed among the machines (randomly or in a balanced fashion). The communication is point-to-point via message passing. Machines do not share any memory and have no other means of communication. The computation advances in synchronous rounds, and each link is assumed to have a bandwidth of B bits per round, i.e., B bits can be transmitted over each link in each round; unless otherwise stated, we assume $B = O(\log n)$ (where n is the input size) [83, 85]. The goal is to minimize the *round complexity*, i.e., the number of *communication rounds* required by the computation.[3]

[3]The communication cost is assumed to be the dominant cost – which is typically the case in Big Data computations — hence the goal of minimizing the number of communication rounds [92].

Initially, the entire graph G is not known by any single machine, but rather partitioned among the k machines in a "balanced" fashion, i.e., the nodes and/or edges of G are partitioned approximately evenly among the machines. We assume a *vertex-partition* model, whereby vertices, along with information of their incident edges, are partitioned across machines. Specifically, the type of partition that we will assume throughout is the *random vertex partition (RVP)*, that is, each vertex of the input graph is assigned randomly to one machine. (This is the typical way used by many real systems, such as Pregel [70], to initially distribute the input graph among the machines.) If a vertex v is assigned to machine M_i we say that M_i is the *home machine* of v. A convenient way to implement the RVP model is through hashing: each vertex (ID) is hashed to one of the k machines. Hence, if a machine knows a vertex ID, it also knows where it is hashed to. It can be shown that the RVP model results in (essentially) a *balanced* partition of the graph: each machine gets $\tilde{O}(n/k)$ vertices and $\tilde{O}(m/k + \Delta)$ edges, where Δ is the maximum degree.

Note that we can also assume an alternate partitioning model, the *random edge partition (REP)* model, where each edge of G is assigned independently and randomly to one of the k machines. The results in the random vertex partition model can be related to the random edge partition model [83].

At the end of the computation, for the community detection problem, the community that each vertex belongs to will be output by some machine.

4.3 Random Walk Preliminaries and Local Mixing Set

Our algorithm is based on the mixing property of a random walk in a graph. We use the notion of local mixing set of a random walk, introduced in [75], to identify communities in a graph. Let us define random walk preliminaries, local mixing time, and local mixing set as defined in [75]. Given an undirected graph and a source node s, a *simple random walk* is

defined as follows: in each step, the walk goes from the current node to a random neighbor; i.e., from the current node u, the probability of moving to node v is $\Pr(u,v) = 1/d(u)$ if $(u,v) \in E$, otherwise $\Pr(u,v) = 0$, where $d(u)$ is the degree of u. Let $\mathbf{p}_t(s)$ be the probability distribution at time t starting from the source node s. Then $\mathbf{p}_0(s)$ is the initial distribution with probability 1 at the node s and zero at all other nodes. The $\mathbf{p}_t(s)$ can be seen as the matrix-vector multiplication between $(A)^t$ and $\mathbf{p}_0(s)$, where A is the transpose of the transition matrix of G. Let $p_t(s,v)$ be the probability that the random walk be in node v after t steps. When it's clear from the context we omit the source node from the notations and denote it by $p_t(v)$ only. The stationary distribution (a.k.a. steady-state distribution) is the probability distribution which doesn't change anymore (i.e., it has converged). The stationary distribution of an undirected connected graph is a well-defined quantity which is $\left(\frac{d(v_1)}{2m}, \frac{d(v_2)}{2m}, \ldots, \frac{d(v_n)}{2m}\right)$, where $d(v_i)$ is the degree of node v_i. We denote the stationary distribution vector by $\boldsymbol{\pi}$, i.e., $\pi(v) = d(v)/2m$ for each node v. The stationary distribution of a graph is fixed irrespective of the starting node of a random walk; however, the number of steps (i.e., time) to reach to the stationary distribution could be different for different starting nodes. The time to reach to the stationary distribution is called the *mixing time* of a random walk with respect to the source node s. The mixing time corresponding to the source node s is denoted by τ_s^{mix}. The mixing time of the graph, denoted by τ^{mix}, is the maximum mixing time among all (starting) nodes in the graph.

Definition 3. *($\tau_s^{mix}(\epsilon)$–mixing time for source s and $\tau^{mix}(\epsilon)$–mixing time of the graph) Define $\tau_s^{mix}(\epsilon) = \min\{t : ||\mathbf{p}_t - \boldsymbol{\pi}||_1 < \epsilon\}$, where $||\cdot||_1$ is the L_1 norm. Then $\tau_s^{mix}(\epsilon)$ is called the ϵ-near mixing time for any ϵ in $(0,1)$. The mixing time of the graph is denoted by $\tau^{mix}(\epsilon)$ and is defined by $\tau^{mix}(\epsilon) = \max\{\tau_v^{mix}(\epsilon) : v \in V\}$. It is clear that $\tau_s^{mix}(\epsilon) \leq \tau^{mix}(\epsilon)$.*

We sometimes omit ϵ from the notation when it is understood from the context. For any set $S \subseteq V$, we define $\mu(S)$ as the *volume* of S, i.e., $\mu(S) = \sum_{v \in S} d(v)$. Therefore, $\mu(V) = 2m$ is the volume of the vertex set. The *conductance* of the set S is denoted by $\phi(S)$ and defined

by $\phi(S) = \frac{|E(S,V\setminus S)|}{\min\{\mu(S),\mu(V\setminus S)\}}$, where $E(S, V \setminus S)$ is the set of edges between S and $V \setminus S$. The conductance of the graph G is $\Phi_G = \min_{S\subseteq V} \phi(S)$.

Let us define a vector $\boldsymbol{\pi}_S$ over the set of vertices S as follows: $\pi_S(v) = d(v)/\mu(S)$ if $v \in S$, and $\pi_S(v) = 0$ otherwise.

Notice that $\boldsymbol{\pi}_V$ is the stationary distribution $\boldsymbol{\pi}$ of a random walk over the graph G, and $\boldsymbol{\pi}_S$ is the restriction of the distribution $\boldsymbol{\pi}$ on the subgraph induced by the set S. Recall that we defined $\mathbf{p}_t$ as the probability distribution over V of a random walk of length t, starting from some source vertex s. Let us denote the restriction of the distribution $\mathbf{p}_t$ over a subset S by $\mathbf{p}_t\restriction_S$ and define it as: $p_t\restriction_S(v) = p_t(v)$ if $v \in S$ and $p_t\restriction_S(v) = 0$ otherwise.

It is clear that $p_t\restriction_S$ is not a probability distribution over the set S as the sum could be less than 1.

Informally, having a *local mixing set*, with respect to a source node s, means that there exists some (large-enough) subset of nodes S containing s such that the random walk probability distribution becomes close to the stationary distribution restricted to S (as defined above) quickly. The time that a random walk mixes locally on a set S is called as *local mixing time* which is formally defined below.

Definition 4. *(Local Mixing Set and Local Mixing Time)*
Consider a vertex $s \in V$. Let $\beta \geq 1$ be a positive constant and $\epsilon \in (0, 1)$ be a fixed parameter. We first define the notion of local mixing in a set S. Let $S \subseteq V$ be a fixed *subset containing s of size at least n/β. Let $\mathbf{p}_t\restriction_S$ be the restricted probability distribution over S after t steps of a random walk starting from s and $\boldsymbol{\pi}_S$ be as defined above. Define the* mixing time with respect to set S *as $\tau_s^S(\beta,\epsilon) = \min\{t : ||\mathbf{p}_t\restriction_S - \boldsymbol{\pi}_S||_1 < \epsilon\}$. We say that the random walk* locally mixes *in S if $\tau_s^S(\beta,\epsilon)$ exists and well-defined. (Note that a walk may not locally mix in a given set S, i.e., there exists no time t such that $||\mathbf{p}_t\restriction_S - \boldsymbol{\pi}_S||_1 < \epsilon$; in this case we can take $\tau_s^S(\beta,\epsilon)$ to be ∞.)*

The local mixing time with respect to s is defined as $\tau_s(\beta,\epsilon) = \min_S \tau_s^S(\beta,\epsilon)$, where the minimum is taken over all subsets S (containing s) of size at least n/β, where the random walk starting from s locally mixes. A set S where the minimum is attained (there may be more than one) is called a local mixing set*. The local mixing time of the graph, $\tau(\beta,\epsilon)$ (for given β and ϵ), is $\max_{v\in V} \tau_v(\beta,\epsilon)$.*

From the above definition, it is clear that $\tau_s(\beta,\epsilon)$ always exists (and is well-defined) for every fixed $\beta \geq 1$, since in the worst-case, it equals the mixing time of the graph; this happens when $|S| = n \geq n/\beta$ (for every $\beta \geq 1$). We note that, crucially, in the above definition of local mixing time, the *minimum* is taken over subsets S of size at least n/β, and thus, in many graphs, local mixing time can be substantially smaller than the mixing time when $\beta > 1$ (i.e., the local mixing can happen much earlier in some set S of size $\geq n/\beta$ than the mixing time).

4.4 Algorithm for Community Detection

We design a random walk based community detection algorithm (cf. Algorithm 5). Given a graph and a node, the algorithm finds a community containing the node in a distributed fashion. We show the efficiency and effectiveness of the algorithm both theoretically and experimentally on random graphs and stochastic block model.

Outline of the Algorithm. We use the concept of *local mixing set*, introduced by Molla and Pandurangan [75], to identify community in a graph. A local mixing set of a random walk is a subset of the vertex set where the random walk probability mixes fast, see the formal definition in Section 4.2. Intuitively, a random walk probability mixes fast over a subset where nodes are well-connected among themselves. The idea is to use the concept of local mixing set to identify a community — a subset where nodes are well-connected inside

Algorithm 5 COMMUNITY-DETECTION-BY-RANDOM-WALKS (CDRW)

Input: An undirected graph $G = (V, E)$.
Output: Set of Detected Communities $\mathcal{C}^{\mathcal{D}}$.

$\mathcal{C}^{\mathcal{D}} \leftarrow \{\}$; $pool \leftarrow V$
while $pool \neq \emptyset$ **do** ▷ There exist nodes not assigned to any communities yet
  $s \leftarrow$ pick a random node from $pool$
  s computes a BFS tree of depth $O(\log n)$ via flooding
  Set $R = \log n$, $p_0(s) = 1$, and $p_0(u) = 0$ for all other nodes u.
  for $\ell = 1, 2, 3, \ldots, O(\log n)$ **do** ▷ Length of the random walk
    Each node u whose $p_{\ell-1}(u) \neq 0$, does the following in parallel:
      (i) Send $p_{\ell-1}(u)/d(u)$ to all the neighbors $v \in N(u)$.
      (ii) Compute the sum of the received values from its neighbors and set it to $p_\ell(u)$.
    for $|S| = R, (1 + 1/8e)R, (1 + 1/8e)^2R, \ldots, n$ **do**
      Each node u computes the difference $x_u = |p_\ell(u) - \frac{d(u)}{\frac{2m}{n}|S|}|$ locally
      s computes the sum of $|S|$ smallest x_u values using binary search method discussed in the detail description of the algorithm.
      s checks if the sum is less than $1/2e$, i.e., if $\sum_{\{|S| \, smallest \, x_u\}} x_u < \frac{1}{2e}$.
      If "true", then s checks for the next size of the mixing set
      Else, s sets S_ℓ to be the largest set S which satisfies the mixing condition. s broadcasts an indicator message to all the nodes via BFS tree. The nodes whose x_u value gives the $|S|$ smallest values belong to the largest mixing set S_ℓ.
    s checks the community condition: **if** $\frac{|S_\ell|}{|S_{\ell-1}|} < (1 + \delta)$ **Then** Break the **for**-loop. ▷ $\delta = \Phi_G$
  $C^s \leftarrow S_{\ell-1}$; $\mathcal{C}^{\mathcal{D}} \leftarrow \mathcal{C}^{\mathcal{D}} \cup \{C^s\}$; $pool \leftarrow pool \setminus C^s$;
Return $\mathcal{C}^{\mathcal{D}}$

the set and less-connected outside. That is, if a random walk starts from a node inside a community, its probability distribution is likely to mix fast inside the community nodes and with fewer probability will go outside of the set. Thus the high level idea of our approach is to perform a random walk from a given source node and find the largest subset (of nodes) where the random walk probability mixes quickly. We extend the distributed algorithm from [75] to find a largest mixing set in the following way. In each step of the random walk, we keep track the size of the largest mixing set. When the size of the largest mixing set is not increasing significantly with the increase of the length of the random walk, we stop and output the largest mixing set as the community containing the source node.

Algorithm in Detail. Given an undirected graph $G(V, E)$, our algorithm randomly selects

a node s and outputs a community set $C^s \subseteq V$ containing s. It maintains a set, called a *pool* which contains all the remaining nodes of V excluding the nodes in C^s. Then another random node gets selected from the set *pool* and we compute the community containing that node in G, and so on. This way all the different communities are computed one by one. The *pool* set is initialized as V in the beginning. The algorithm stops when the *pool* set becomes empty.

Now we describe how the algorithm computes the community set C^s in G from a given node s. The algorithm performs a random walk from the source node s and computes the probability distribution $\mathbf{p}_\ell$ at each step ℓ of the random walk. The probability distribution $\mathbf{p}_\ell$ starting from the source node s is computed locally by each node as follows: Initially, at round 0, the probability distribution is as follows: at node s, $p_0(s,s) = 1$ and at all other nodes u, $p_0(s,u) = 0$. At the start of a round ℓ, each node u sends $p_{\ell-1}(s,u)/d(u)$ to its $d(u)$-neighbors and at the end of the round ℓ, each node u computes $p_\ell(s,u) = \sum_{v \in N(u)} p_{\ell-1}(s,v)/d(v)$. This local flooding approach essentially simulates the probability distribution of each step of the random walk starting from a source node. Moreover, this deterministic flooding approach can be used to compute the probability distribution $\mathbf{p}_\ell$ of length ℓ from the previous distribution $\mathbf{p}_{\ell-1}$ in *one round* only, simply by resuming the flooding from the last step. The full algorithm can be found as Algorithm 1 in [75]. Then at each step ℓ, our algorithm computes a largest mixing set S_ℓ. The largest mixing set S_ℓ is computed as follows: Each node u knows its $p(u) = p_\ell(s,u)$ value. The algorithm gradually increases the size of a candidate local mixing set S starting from size 1.[4] First each node u locally calculates its x_u value as $x_u = |p_\ell(u) - \frac{d(u)}{\mu'(S)}|$, where $\mu'(S) = \frac{2m}{n}|S|$ is the average volume of the set S. Note that any node u can compute $\mu'(S)$ when it knows the "size" $|S|$ and hence can compute x_u locally. However, it's difficult to compute $\mu(S)$ unless it knows the set S (i.e., the nodes in S) and the degree distribution of the nodes in S. Computing nodes in S and their degree

[4] In the pseudocode we assume the size of each community is at least $\log n$.

distribution is expensive in terms of time. That's why we consider $\mu'(S)$ instead $\mu(S)$ in the localized algorithm.

Then the source node s collects the $|S|$ smallest of x_u values and checks if their sum is less than $1/2e$ (mixing condition). For this each node may sends its x_u to the source node s via upcasting through a BFS tree rooted at s. (A BFS tree is computed from s at the beginning of the algorithm). However, the upcast may take $\Omega(n)$ time in the worst case due to the congestion in the BFS tree. A better approach is used in [75], which is to do a binary search on $\{x_u \,|\, u \in V\}$, as follows: All the nodes send $x_{\min}$ and $x_{\max}$ (the minimum and maximum respectively among all x_u) to the root s through a convergecast process (e.g., see [87]). This will take time proportional to the depth of the BFS tree. Then s can count the number of nodes whose x_u value is less than $x_{\text{mid}} = (x_{\min} + x_{\max})/2$ via a couple of broadcast and convergecast. In fact, s broadcasts the value x_{mid} to all the nodes via the BFS tree and then the nodes whose x_u value is less than x_{mid} (say, the *qualified nodes*), reply back with 1 value through the convergecast. Depending on whether the number of qualified nodes is less than or greater than $|S|$, the root updates the x_{mid} value (by again collecting $x_{\min}$ or $x_{\max}$ in the reduced set) and iterates the process until the count is exactly $|S|$. Note that there might be multiple nodes with the same x_u value. We can make them distinct by adding a 'very' small random number to each of the x_u such that the addition doesn't affect the mixing condition. The detailed approach and analysis can be found in [75].

Once the node s gets $|S|$ smallest x_us, it checks if their sum is less than $1/2e$. If true, then these nodes u whose sum value is less than $1/2e$ form a candidate mixing set whose size is $|S|$. Then we increase the set size and check if there is a larger mixing set. If the mixing condition is not satisfied, then there is no mixing set of size $|S|$. The algorithm iterates the checking process a few more times by increasing the size of S and checking if there is a mixing set of larger size. If not, then the algorithm stops for this length ℓ and stores the largest mixing set at s. This way, the algorithm finds the largest mixing set S_ℓ at the ℓ^{th} step

of the random walk. Note that we can increase the candidate mixing set size by 1 each time. This will increase the time complexity of the algorithm by a factor of the "size of the largest mixing set". Instead we increase the size of the mixing set by a factor of $(1 + 1/8e)$ in each iteration. This will only add a factor of $O(\log n)$ to the time complexity. The reason why we increase by a factor of $(1 + 1/8e)$ instead of doubling is discussed in [75] (see, Lemma 3 in [75]). The correctness of the all the above tests is also analyzed in [75].

Then the algorithm checks if the size of the largest mixing set S_ℓ at step ℓ increases more significantly than the mixing set $S_{\ell-1}$ in the previous step $(\ell - 1)$. This is checked locally by the source node as the source node has the information of the largest mixing set of the current and previous steps. If the size doesn't increase by a factor $(1 + \delta)$, i.e., if $S_\ell < (1 + \delta)S_{\ell-1}$, then the algorithm stops and outputs $S_{\ell-1}$ as the community set C^s. Otherwise, the algorithm increases the length by 1 and checks for $S_{\ell+1}$. The parameter δ is chosen to be the conductance of the graph Φ_G which essentially measures the vertex expansion of the graph.

4.4.1 Analysis

We analyze the algorithm and show that it correctly identifies communities in the planted partition model (PPM) – G_{npq} graphs. The G_{npq} graph is formed by connecting several communities of G_{np} graphs (see the definition in Section 4.2). Let us first analyze Algorithm 5 on the random graph G_{np}. We then extend the analysis to the stochastic block model G_{npq}.

On G_{np} Graphs. Suppose the algorithm is executed on the standard random (almost regular) graph $G_{np} = (V_1, E_1)$, defined in Section 4.2. Since G_{np} is an expander graph, the random walk starting from any node mixes over the vertex set V_1 very fast, in fact, in $O(\log n)$ steps. Given any node s, we show that our algorithm computes the community C^s as the complete vertex set V_1. More precisely, we show that the size of the largest mixing set

increases on a higher rate (than the considered threshold) after each step of the random walk, when the length of the walk is $o(\log n)$. Since $O(\log n)$ is the mixing time of G_{np}, the random walk probability reaches the stationary distribution after $c \log n$ steps, for a sufficiently large constant c.

Let $p_t(u)$ be the probability that the walk is at u after t steps (starting from a source node s). It is known that in a regular graph $p_t(u)$ is bounded by

$$\frac{1}{n} - \lambda_2^t \leq p_t(u) \leq \frac{1}{n} + \lambda_2^t \tag{4.1}$$

where λ_2 is the second largest eigenvalue (absolute value) of the transition matrix of G_{np}[5]. Hence, the above bound on the probability distribution p_t holds in G_{np} graphs. It is further known that in a random d-regular graph, the second largest eigenvalue is bounded by [41]

$$\frac{1}{\sqrt{d}} \leq \lambda_2 \leq \frac{1}{\sqrt{d}} + o(1). \tag{4.2}$$

Let B_ℓ be the set of nodes that are within the distance ℓ from the s. The distance is measured by the hop distance between nodes. Let's call B_ℓ a ball of radius ℓ centered at s. We now show that after ℓ steps of the random walk, the largest mixing set is $B_{\ell/2}$ in a G_{np} graph.

Lemma 20. *Let a random walk start from a source node s in a G_{np} graph. Then for any length ℓ which is less than the mixing time of the random walk, the largest mixing set is the ball $B_{\lfloor \ell/2 \rfloor}$ with high probability.*

Proof. Assume $\ell = o(\log n)$, since ℓ is less than the mixing time $O(\log n)$. It is known that the size of the ball B_ℓ in a random graph G_{np} is bounded by $O((np)^\ell)$ with high probability (cf. Lemma 2 in [24]). To prove the lemma we show that the random walk probability mixes inside the ball $B_{\lfloor \ell/2 \rfloor}$ and doesn't mix on the ball of radius larger than $\lfloor \ell/2 \rfloor$. Recall that the

[5]The bound follows from the standard bound $|p_t(s,u) - \pi(v)| \leq \lambda_2^t \sqrt{\pi(v)/\pi(s)}$ in general graphs [68]. In a regular graph, $\pi(v) = 1/n$ for all v. Note that G_{np} is not exactly a regular graph, but very close to regular (especially if $p = (c \log n)/n$ for a large enough constant c). It can be shown that $\pi(v) = 1/n \pm o(1/n)$ in G_{np}. For simplicity we assume that G_{np} is a regular graph as this little $\pm\epsilon$ changes in the degree or in the probability distribution doesn't affect the lemmas.

condition of locally mixing on a subset $B_{\lfloor \ell/2 \rfloor}$ is $\sum_{u \in B_{\lfloor \ell/2 \rfloor}} |p_\ell(u) - \frac{1}{|B_{\lfloor \ell/2 \rfloor}|}| \leq \frac{1}{2e}$, (since G_{np} is regular graph). Using the above bound of $p_t(u)$, λ_2 (Equ 4.1, 4.2) and $|B_{\lfloor \ell/2 \rfloor}| \leq d^{\lfloor \ell/2 \rfloor}$ (since $d = np = \Theta(\log n)$ in expectation in G_{np}) we have

$$\begin{aligned}
&\sum_{u \in B_{\lfloor \ell/2 \rfloor}} \Big| p_\ell(u) - \frac{1}{|B_{\lfloor \ell/2 \rfloor}|} \Big| \leq \sum_{u \in B_{\lfloor \ell/2 \rfloor}} \Big| \frac{1}{n} + \lambda_2^\ell - \frac{1}{|B_{\lfloor \ell/2 \rfloor}|} \Big| \\
&\leq d^{\lfloor \ell/2 \rfloor} \Big| \frac{1}{n} + \frac{1}{d^{\frac{\ell}{2}}} + o(1) - \frac{1}{d^{\lfloor \ell/2 \rfloor}} \Big| < \frac{d^{\lfloor \ell/2 \rfloor}}{n} + o(1) \\
&< \frac{1}{2e} \qquad [\text{since } \frac{d^{\lfloor \ell/2 \rfloor}}{n} = o(1) \text{ as } \ell = o(\log n)]
\end{aligned}$$

This shows that the random walk of length ℓ mixes over the nodes in $B_{\lfloor \ell/2 \rfloor}$. Now we show that it doesn't mix on B_t for $t > \ell/2$. Again from Equations 4.1 and 4.2,

$$\begin{aligned}
&\sum_{u \in B_t} \Big| p_\ell(u) - \frac{1}{|B_t|} \Big| \geq \sum_{u \in B_t} \Big| \frac{1}{n} - \lambda_2^\ell - \frac{1}{|B_t|} \Big| \\
&= \sum_{u \in B_t} \Big| \lambda_2^\ell + \frac{1}{|B_t|} - \frac{1}{n} \Big| \geq \sum_{u \in B_t} \Big| \lambda_2^\ell \Big| \quad [\text{since } |B_t| \leq n] \\
&\geq \frac{|B_t|}{d^{\frac{\ell}{2}}} \geq d > 1/2e \quad [\text{since } t > \ell/2 \text{ and } d = \log n]
\end{aligned}$$

Thus the largest mixing set is $B_{\lfloor \ell/2 \rfloor}$. □

Now we show that our algorithm outputs the full vertex set as the community in G_{np} graphs.

Lemma 21. *Given a random regular expander graph $G_{np} = (V_1, E_1)$, Algorithm 5 outputs the vertex set V_1 as a single community with high probability.*

Proof. It follows from the previous lemma that when ℓ is less than the mixing time of G_{np}, then the largest local mixing set is $B_{\lfloor \ell/2 \rfloor}$. Therefore, in each step of the random walk, the size of the mixing set is increased by a factor $\frac{|B_{\lfloor \ell/2 \rfloor}|}{|B_{\lfloor \ell/2-1 \rfloor}|} = O(d) = \Theta(\log n) > (1+\delta)$. Hence, by the condition of Algorithm 5, it doesn't stop and continue to look for a community set for the larger lengths of the random walk. This means that until the length of the random walk

reaches the mixing time of the graph G_{np}, the algorithm continues its execution. When the length reaches the mixing time, then the random walk will mix the full vertex set V_1. Then the algorithm stops and outputs V_1 as a single community set (as the size of the mixing set won't increase anymore for larger lengths). □

On G_{npq} Graphs. Let us now analyze the algorithm on the planted partition model, i.e., on a random G_{npq} graph. A random G_{npq} graph is formed by r equal size blocks $C_1, C_2, \ldots, C_r$ where each component C_i is a $G_{\frac{n}{r}p}$ random graph (see the definition in Section 4.2). We show that the algorithm correctly identifies each block as a community. Suppose the randomly selected node s belongs to some block C. The induced subgraph on C is a $G_{\frac{n}{r}p}$ graph, i.e., the nodes inside C are connected to each other with probability p. Further each node in C is connected to every node outside of C with probability q. Thus the random walk may go out of the set C at some point. We show that the probability of going out of C is very small when the length of the walk is smaller than the mixing time of $G_{\frac{n}{r}p}$ graph, which is $O(\log(n/r))$.

Lemma 22. *Given a G_{npq} graph and a node s in some block C, the probability that a random walk starting from s stays inside C is at least $1 - o(1)$ until $\ell = O(\log(n/r))$ when $q = o(\frac{p}{r\log(n/r)})$.*

Proof. We show that in each step, the probability that the random walk goes outside of C is $o(1/\log n)$. For any $u \in C$, the number of neighbors of u in C is $p|C| = pn/r$ and the number of neighbors in $\bar{C} = V \setminus C$ is $q|\bar{C}| = q(n - n/r)$ in expectation. Thus the probability that the random walk goes outside of the block C is $\frac{q(n-n/r)}{p(n/r)+q(n-n/r)} = \frac{q(r-1)}{p+q(r-1)}$. This is $o(1/\log(n/r))$ when $q = o(\frac{p}{r\log(n/r)})$. Thus in $\ell = O(\log(n/r))$ steps, the probability that walk goes outside of the block C is $o(1)$. That is the random walk stays inside C with probability at least $1 - o(1)$. □

Now we show that the random walk probability will mix over C in $O(\log(n/r))$ steps.

Lemma 23. *Given a G_{npq} graph and a node $s \in C$, a random walk starting from s will mix over the nodes in C after $\tau = O(\log(n/r))$ steps with high probability.*

Proof. We show that after $O(\log(n/r))$ steps of the walk, the amount of probability goes out of C is very little and that the remaining probability will mix inside C. The expected number of outgoing edges from any subset S of the block C is $|E(S, V \setminus C)| = q|\bar{C}||S| = q(n - n/r)(|S|)$. In each step the probability of leaving C is $\frac{|E(S,V\setminus C)|}{d|S|}$, as $d = p(n/r) + q(n - n/r)$ is the degree of a node, each edge carries $1/d|S|$ fraction of the probability. We have $\frac{|E(S,V\setminus C)|}{d|S|} = \frac{q(n-n/r)|S|}{(p(n/r)+q(n-n/r))|S|} = o(1/\log(n/r))$ for $q = o(\frac{p}{r\log(n/r)})$. Thus in $\ell = O(\log(n/r))$ steps, the probability of leaving C is $o(1)$. Hence $1 - o(1)$ fraction of the probability remains inside C and it will mix over the nodes in C after $O(\log(n/r))$ steps as shown in Lemmas 20 and 21. □

Thus it follows from the above lemma that the largest mixing set is C after $\tau = O(\log(n/r))$ steps of the random walk. Further, it is shown in Lemma 4 of [75] that the random walk keeps mixing in C until 2τ steps. In other words, C remains the largest local mixing set for at least another τ steps. Thus the size of the largest local mixing set will not increase from C in the further few steps of the walk after the mixing time τ. Hence the algorithm outputs C as a community with high probability. Since we sample the source node s from the different blocks, each time our algorithm outputs a new community until all the blocks are identified as separate communities.

The δ value measures the rate of change of the size of the largest mixing set in each step. When the largest mixing set reaches a community C, the vertex expansion becomes $\frac{|E(C,V\setminus C)|}{d|C|}$ which is the conductance of the G_{npq} graph. If the largest mixing set doesn't reach the community, the size increases at a rate higher than δ. Hence we take δ to be Φ_G in our algorithm to stop and output the community. We assume that Φ_G is given as input, or it can be computed using a distributed algorithm, e.g., [64].

Complexity of the Algorithm in the CONGEST model. Let us first analyze the *distributed time complexity* of the Algorithm 5 which computes a community corresponding to a given source node. We will focus on the CONGEST model first. The algorithm first computes a BFS tree of depth $O(\log n)$ from the source node. This takes $O(\log n)$ rounds. Note that the diameter of a G_{np} graph is $O(\log n)$; hence the BFS tree covers all the nodes in the community containing the source node. The algorithm then iterates for the length of the walk, $\ell = 1, 2, 4, \ldots, O(\log n)$. In each iteration:

- The algorithm probability distribution $\mathbf{p}_\ell$. As we discussed before, it takes $O(1)$ rounds to compute $\mathbf{p}_\ell$ from $\mathbf{p}_{\ell-1}$.
- s collects the sum of $|S|$ smallest x_us through the BFS tree using binary search method. It takes $O((depth\_BFS\_tree) \cdot \log n) = O(\log^2 n)$ rounds. This is done for all the potential candidate sets of size $(1 + 1/8e)^i|S|$, where $i = 0, 1, 2, \ldots$. It may take $O(\log n)$ rounds in the worst case. Hence the total time taken is $O(\log^3 n)$ rounds.
- Checking if the sum of differences is less than $1/2e$ and also checking the community condition is done locally at s.

Thus the total time required is $O(\log n) + O(\log n) \cdot (O(1) + O(\log^3 n))$, which is bounded by $O(\log^4 n)$.

Message Complexity of the Algorithm. Let us calculate the number of messages used by the algorithm during the execution in a G_{npq} graph. The degree of a node is $p(n/r) + q(n - n/r)$ in expectation. Hence the number of edges in the G_{npq} graph is $n^2p/r + nq(n - n/r)$. In the worst case, the the algorithm runs over all the edges in the graph. Thus the message complexity of the algorithm for computing a single community is bounded by (time complexity) $\times$ (the number of edges involved during the execution), which would be $O(\frac{n^2}{r}(p + q(r - 1)) \log^4 n)$ in expectation. That is the message complexity of Algorithm 5 is $\tilde{O}(\frac{n^2}{r}(p + q(r - 1)))$.

Therefore we have the following main result.

Theorem 24. *Consider a stochastic block model G_{npq} with r blocks, where $p = \Omega(\frac{\log n}{n})$ and $q = o(\frac{p}{r \log(n/r)})$. Given a node s in the G_{npq} graph, there is a distributed algorithm (cf. Algorithm 5) that computes the block containing s as a community with high probability in $O(\log^4 n)$ rounds and incurs $\tilde{O}(\frac{n^2}{r}(p + q(r-1)))$ messages in expectation.*

The CDRW algorithm can be used to detect all the r communities in the PPM graphs one by one. In that case the running time would be r times the time of detecting one community, which is $O(r \log^4 n)$. The message complexity in this case would be $O(n^2(p+q(r-1)) \log^4 n)$ in expectation. Thus we have the following theorem.

Theorem 25. *Given a stochastic block model G_{npq} with r blocks, where $p = \Omega(\frac{\log n}{n})$ and $q = o(\frac{p}{r \log(n/r)})$, there is a distributed algorithm (cf. Algorithm 5) that correctly computes each block as a community with high probability and outputs all the r communities in $O(r \log^4 n)$ rounds and incurs expected $\tilde{O}(n^2(p + q(r-1)))$ messages.*

4.4.2 Complexity in the k-machine Model.

As mentioned earlier, in the k-machine model, the input (SBM) graph is partitioned across the k machines according to the random vertex partition (RVP) model (cf. Section 4.2). The algorithm can be implemented in the k-machine model by simulating the corresponding CONGEST model algorithm. Note that since each vertex and its incident edges are assigned to a machine (i.e., its "home" machine — cf. Section 4.2), the machine simply simulates the code executed by the vertex in the CONGEST model. If a vertex u sends a message to its neighbor u in the CONGEST model, then the home machine of u sends the same message to the home machine of v (addressing it to v). If u and v have the same (home) machine, then no communication is incurred, otherwise there will be communication along the link that connects these two home machines. This type of simulation is detailed in

[58]. Hence one can use the Conversion Theorem (part a) of [58] to compute the round complexity of the CDRW implementation in the k-machine model which depends on the message complexity and time complexity of CDRW in the CONGEST model. If M and T are the message and time complexities (respectively) in the CONGEST model, then in the k-machine model, by the Conversion Theorem, the above simulation will give a round complexity of $\tilde{O}(M/k^2 + (\Delta T)/k)$, where Δ is the maximum degree of the graph.[6] For the SBM model, $\Delta = O(np/r + (n - n/r)q)$. Hence plugging in the message complexity and time complexity from the CONGEST model analysis, we have that the round complexity in the k-machine model is $\tilde{O}((\frac{n^2}{k^2} + \frac{n}{kr})(p + q(r - 1)))$.

4.5 Experimental Results

In this section we experimentally analyze the performance of our algorithm in the PPM model under various parameters. In particular, we show how accurately our algorithm can identify the communities in the PPM model. As an important special case, we also analyze the case when $r = 1$, i.e., there is only one community — in other words, the whole graph is a $G(n, p)$ random graph. In this case, we expect the algorithm to output the whole graph as one community.

Since in the PPM model, we know the ground-truth communities, we use the F-score metric [71] to measure the accuracy of the detected communities. Let C^D be the set of detected communities by CDRW algorithm and $C^G = \cup C_i$ be the ground-truth communities (each C_i is a ground-truth community). Let C^s be the detected community by CDRW using seed node s and C^g be the ground-truth community that seed node s belongs to. Then the *precision* is the percentage of truly detected members in detected community defined as $precision(C^s) = \frac{|C^s \cap C^g|}{|C^s|}$ and *recall* is the percentage of truly detected members from the

[6] $\tilde{O}$ notation hides a polylog n multiplicative and additive factor.

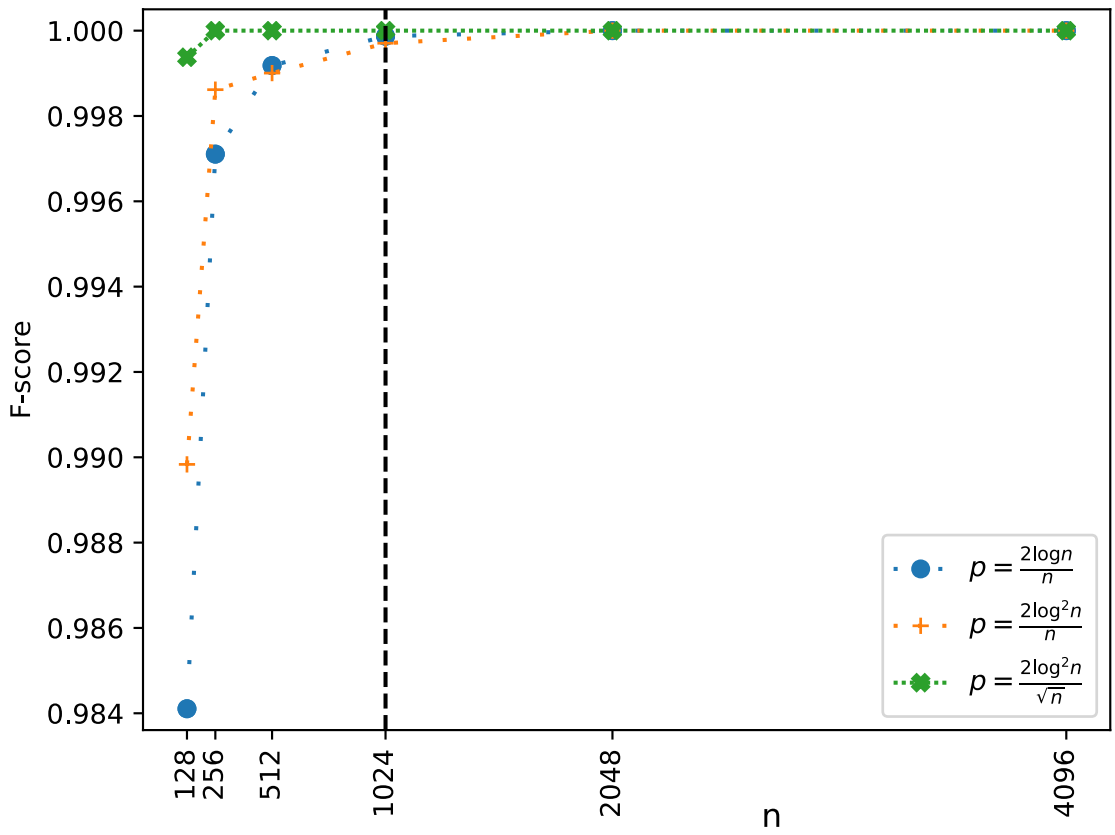


Figure 4.2: Community detection accuracy of CDRW algorithm on G_{np} random graphs. It shows that even when the graph is sparse, when p is small and as close to the connectivity threshold as possible, its accuracy is still high. The vertical line shows that when the size is big enough ($n \geq 2^{10}$), the accuracy becomes almost 1.0.

ground truth community defined as $recall(C^s) = \frac{|C^s \cap C^g|}{|C^g|}$. Both *precision* and *recall* return a high value when a method detects communities well. For example, if all the detected members belong to the ground-truth community of the seed node, then its *precision* is equal to 1.0; and if all the ground-truth community members of the seed node are included in the detected community, then its *recall* value is equal to 1.0. We utilize the F-score as our accuracy measurement metric which reflects both precision and recall of a result. The F-score of a detected community C^s is defined as: $F\text{-}score(C^s) = \frac{2 \times precision(C^s) \times recall(C^s)}{precision(C^s) + recall(C^s)}$. Then the total *F-score* is equal to the average F-score of all detected communities: $F\text{-}score = \frac{1}{|C^D|} \sum_{C^j \in C^D} F\text{-}score(C^j)$. Again a higher F-score value means a better detection of communities.

The first challenge for any community detection (CD) algorithm is detecting a random graph as a single community. This challenge becomes harder when the graph becomes sparse and it gets closer to the connectivity threshold of a random graph (i.e., $p = \frac{c \log n}{n}$, s.t. $c > 1$) [15]. In the first experiment we show that our CDRW algorithm detects almost the whole graph as a single community resulting in a high F-score accuracy value, see Figure 4.2. Figure 4.2 shows that when we increase the size of graph n, the accuracy of our algorithm increases as well. For example, for $n = 2^{10}$ the accuracy metric becomes almost 1.0, meaning that almost all the nodes of the graph are detected as a single community. It also shows that when p increases (the graph gets denser), the accuracy also increases. So in the remaining experiments on PPM graphs, we choose two lowest values of $p = \frac{c \log n}{n}$ and $p = \frac{c \log^2 n}{n}$ for generating its random parts in order to give more challenging input graphs to the CDRW algorithm.

After showing that CDRW works well on G_{np} random graphs, now we consider PPM G_{npq} graphs. At first we fix the number of communities to two ($r = 2$) so that we can consider the effect of various values of p and q. This will show us the threshold for the ratio of $\frac{p}{q}$ where CDRW works well. As we shown in Figure 4.2, when the size of each random graph is big enough ($n \geq 2^{10}$), CDRW detects a single G_{np} community well. Therefore we set the size of G_{npq} to $n = 2^{11}$ which makes each ground-truth community big enough ($\frac{n}{r} = 2^{10}$). When considering PPM graphs with p and q, as the connectivity probability for intra- and inter-community edges, CD algorithms face hardship in detecting communities when p is small and q is relatively high. But the $\frac{p}{q}$ ratio can not be arbitrarily small because it causes the two communities to blend into each other and the graph looses its community structure. Figure 4.3 shows the accuracy of CDRW for different values of p and q. We highlight that it performs well even for sparse parted G_{npq} graphs: for $p = \frac{2 \log n}{n}$, CDRW detects the two communities with a high F-score value (more than 0.90) for $q = \frac{0.1}{n}$ and $\frac{0.6}{n}$. In other words, our CDRW algorithm works well even on sparse parted PPM graphs when the $\frac{p}{q}$ ratio is as

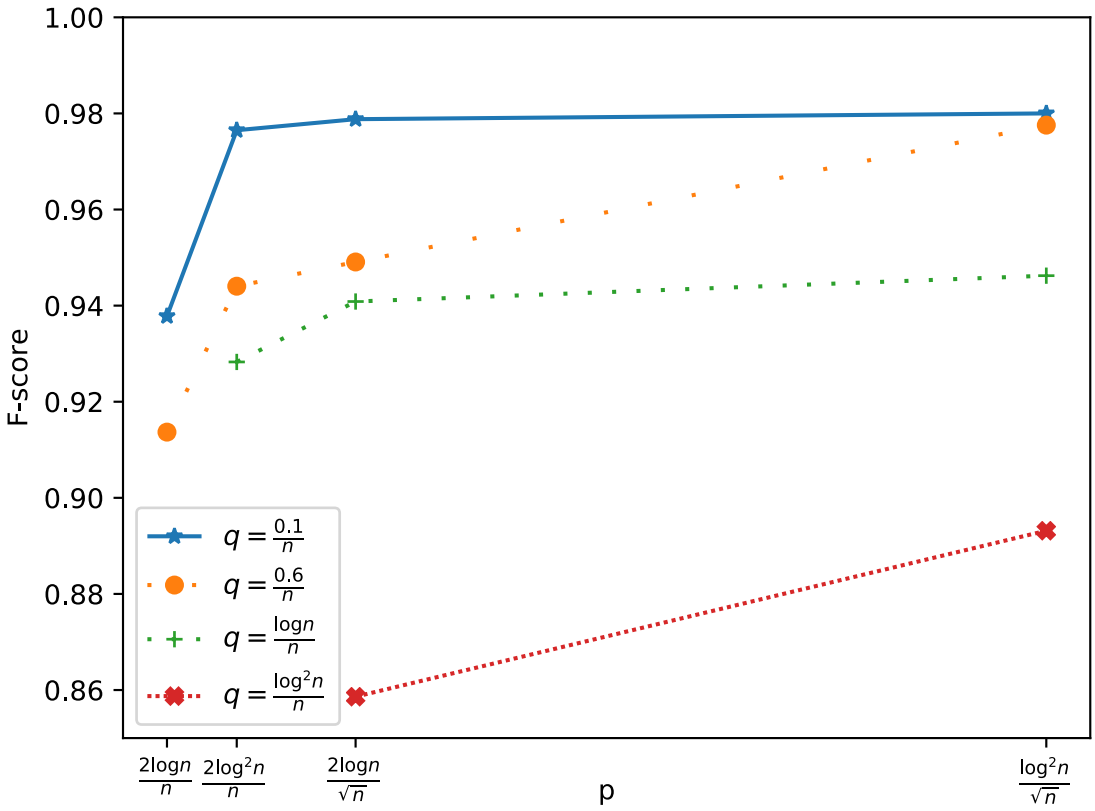
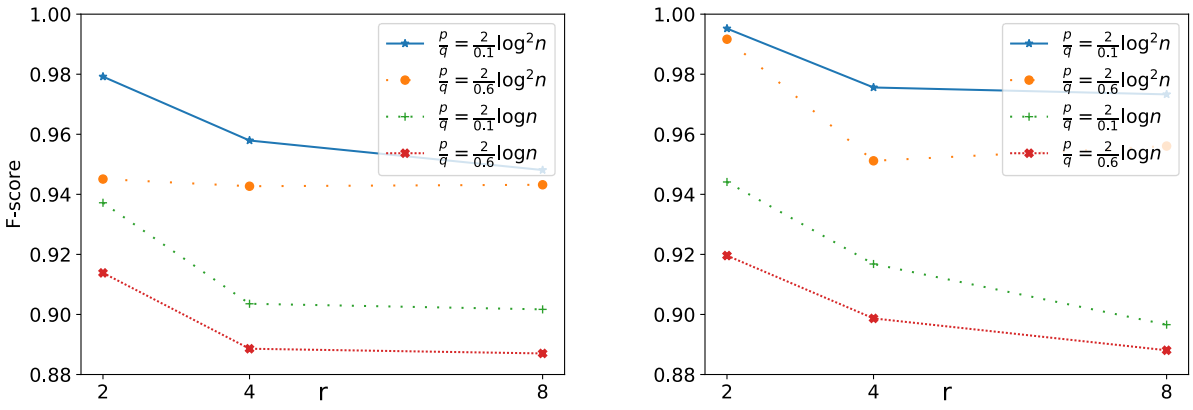


Figure 4.3: Performance of CDRW algorithm on PPM graphs when there are two parts ($r = 2$). We fixed the size of the graph to $n = 2^{11}$, each planted partition is of size 2^{10}. It shows that CDRW works well for small values of $p = \frac{2\log n}{n}$ and $p = \frac{2\log^2 n}{n}$ when q (the probability parameter for the existence of inter-community edges) is small enough.

small as $\Omega(\log n)$. Notice that when $p = \frac{2\log n}{n}$, the two ground truth communities of the PPM graph are as sparse as possible, i.e., close to its connectivity threshold. In the latter example, for instance, when $q = \frac{0.6}{n}$, a partition has in expectation $e_{in} = \binom{\frac{n}{r}}{2}p = 10230$ intra and $e_{out} = \frac{n}{r}(n - \frac{n}{r})q = 614$ inter community edges. It means the ratio of inter to intra community edges ($\frac{e_{out}}{e_{in}}$) is high, equal to 6%.

We now consider the effect of increasing the number of ground-truth communities (r) in order to see its effect on the accuracy of our CDRW algorithm, see Figure 4.4. We do it in two ways. First, we fix the size of each community to 2^{10} and vary the number of communities.



(a) $n = r \times 2^{10}$.

(b) $n = 8 \times 2^{10}$.

Figure 4.4: Varying the number of ground-truth communities to see its effect on the accuracy of our CDRW algorithm. It shows that when we increase the number of communities, the accuracy decreases slightly. This is expected because the number of inter-community edges increases. Comparing Sub-figures 4.4a and 4.4b, we see that if we fix the number of communities, then the accuracy gets higher when the size of the communities becomes larger.

The size of graph is $n = r \times 2^{10}$. Figure 4.4a shows that our CDRW algorithm works well when we reasonably increase the number of communities. Second, we fix the size of graph to a number so that the size of each community is 2^{10} when the number of communities is the biggest ($r = 8$), see Figure 4.4b. Then, when the number of communities becomes lower, the size of communities gets bigger. By comparing Sub-figures 4.4a and 4.4b, we see that when the number of communities are the same, the accuracy is higher when the size of each community is bigger.

Chapter 5

Conclusion

We presented distributed algorithms to solve two basic graph problems. First, we presented fast and efficient distributed algorithms for the fundamental Hamiltonian cycle problem in random graphs. Our algorithm (DHC2) is fully distributed and runs in truly sublinear time — $\tilde{O}(\frac{1}{p})$ — for all ranges of p; in fact, the denser the graph, the smaller the running time. We also presented a conceptually simpler upcast algorithm with the same running time, but it is not fully-distributed, and does *not* achieve *load-balancing*.

Second, we proposed a distributed algorithm, CDRW, for community detection that works well for the PPM model (G_{npq} random graph), a standard random graph model used extensively in clustering and community detection studies. Our CDRW algorithm is relatively simple and localized; it uses random walks and the mixing time property of graphs to detect communities. We provided a rigorous theoretical analysis of our CDRW algorithm on the G_{npq} random graph and characterized its performance vis-a-vis the parameters of the model. In particular, our main result is that it correctly identifies the communities provided $q = o(p/(r \log(n/r)))$, where r is the number of communities. Our CDRW algorithm takes $O(r \times \text{polylog } n)$ rounds and hence is quite fast when r is relatively small.

Our fully-distributed algorithms can be used to obtain efficient algorithms in other distributed message-passing models such as the k-machine model [58], which is a distributed model for large-scale data computation. We also believe that our presented ideas can be extended to obtain similarly fast and efficient fully distributed algorithms for other random graph models such as the $G(n, M)$ model and random regular graphs [16].

Several open questions arise from our work. For the Hamiltonian cycle problem, is it possible to show non-trivial lower bounds for the HC problem in random graphs? In particular, we conjecture that our upper bounds are essentially tight (up to polylogarithmic factors). Can we find a sublinear time, i.e., an algorithm running in $o(n)$ rounds for $p = \frac{c \ln n}{n}$, i.e., at the threshold; or show that this is not possible? Nothing non-trivial is known regarding upper bounds for general graphs. For the community detection problem, it will be interesting to study the performance of our CDRW algorithm on other graph models. It can be a starting point to design and analyze community detection algorithms that perform well in the more challenging case of real-world graphs.

Bibliography

[1] Emmanuel Abbe. Community detection and stochastic block models: recent developments. *The Journal of Machine Learning Research*, 18(1):6446–6531, 2017.

[2] Emmanuel Abbe. Community detection and stochastic block models: Recent developments. *Journal of Machine Learning Research*, 18(177):1–86, 2018.

[3] Emmanuel Abbe and Colin Sandon. Community detection in general stochastic block models: Fundamental limits and efficient algorithms for recovery. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 670–688. IEEE, 2015.

[4] Yahav Alon and Michael Krivelevich. Random graph's Hamiltonicity is strongly tied to its minimum degree. *arXiv preprint arXiv:1810.04987*, 2018.

[5] Yahav Alon and Michael Krivelevich. Finding a Hamilton cycle fast on average using rotations-extensions. *arXiv preprint arXiv:1903.03007*, 2019.

[6] Reid Andersen and Kevin J Lang. Communities from seed sets. In *Proceedings of the 15th International Conference on World Wide Web*, pages 223–232. ACM, 2006.

[7] Dana Angluin and Leslie G Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, 1979.

[8] James P Bagrow. Evaluating local community methods in networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(05):P05001, 2008.

[9] Sayan Bandyapadhyay, Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Near-optimal clustering in the k-machine model. In *Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN)*, 2018.

[10] Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Luca Trevisan. Find your place: Simple distributed algorithms for community detection. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 940–959. SIAM, 2017.

[11] Luca Becchetti, Andrea E. F. Clementi, Pasin Manurangsi, Emanuele Natale, Francesco Pasquale, Prasad Raghavendra, and Luca Trevisan. Average whenever you

meet: Opportunistic protocols for community detection. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, pages 7:1–7:13, 2018.

[12] Florence Bénézit, Patrick Thiran, and Martin Vetterli. Interval consensus: from quantized gossip to voting. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 3661–3664. IEEE, 2009.

[13] Belá Bollobás. The diameter of random graphs. *Transactions of the American Mathematical Society*, 267(1):41–52, 1981.

[14] Béla Bollobás. Random graphs. *Academic Press, New York*, 1985.

[15] Béla Bollobás. Random graphs. In *Modern graph theory*, pages 215–252. Springer, 1998.

[16] Béla Bollobás. *Random graphs*. Number 73. Cambridge University Press, New York, 2001.

[17] Béla Bollobás. *Graph theory: an introductory course*, volume 63. Springer Science & Business Media, New York, 2012.

[18] Béla Bollobás, Trevor I. Fenner, and Alan M. Frieze. An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica*, 7(4):327–341, 1987.

[19] Béla Bollobás, Svante Janson, and Oliver Riordan. The phase transition in inhomogeneous random graphs. *Random Structures & Algorithms*, 31(1):3–122, 2007.

[20] Ravi B Boppana. Eigenvalues and graph bisection: An average-case analysis. In *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, pages 280–285. IEEE, 1987.

[21] Thang Nguyen Bui, Soma Chaudhuri, Frank Thomson Leighton, and Michael Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[22] Soumyottam Chatterjee, Reza Fathi, Gopal Pandurangan, and Nguyen Dinh Pham. Fast and efficient distributed computation of Hamiltonian cycles in random graphs. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 764–774. IEEE, 2018.

[23] Peter Chin, Anup Rao, and Van Vu. Stochastic block model and community detection in sparse graphs: A spectral algorithm with optimal rate of recovery. In *Conference on Learning Theory*, pages 391–423, 2015.

[24] Fan Chung and Linyuan Lu. The diameter of sparse random graphs. *Advances in Applied Mathematics*, 26(4):257–279, 2001.

[25] Fan Chung and Linyuan Lu. *Complex Graphs and Networks (Cbms Regional Conference Series in Mathematics)*. American Mathematical Society, Boston, MA, USA, 2006.

[26] Aaron Clauset. Finding local community structure in networks. *Physical Review E*, 72(2):026132, 2005.

[27] Andrea Clementi, Miriam Di Ianni, Giorgio Gambosi, Emanuele Natale, and Riccardo Silvestri. Distributed community detection in dynamic graphs. *Theoretical Computer Science*, 584:19–41, 2015.

[28] Anne Condon and Richard M Karp. Algorithms for graph partitioning on the planted partition model. In *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*, pages 221–232. Springer, 1999.

[29] Anne Condon and Richard M Karp. Algorithms for graph partitioning on the planted partition model. *Random Structures & Algorithms*, 18(2):116–140, 2001.

[30] Colin Cooper and Alan M Frieze. On the number of Hamilton cycles in a random graph. *Journal of Graph Theory*, 13(6):719–735, 1989.

[31] Atish Das Sarma, Anisur Rahaman Molla, and Gopal Pandurangan. Fast distributed computation in dynamic networks via random walks. In *International Symposium on Distributed Computing*, pages 136–150. Springer, 2012.

[32] William E Donath and Alan J Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[33] Martin E. Dyer and Alan M. Frieze. The solution of some random NP-hard problems in polynomial expected time. *Journal of Algorithms*, 10(4):451–489, 1989.

[34] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, New York, 2010.

[35] Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Can quantum communication speed up distributed computation? In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 166–175, 2014.

[36] Paul Erdős and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[37] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[38] Reza Fathi, Anisur Rahaman Molla, and Gopal Pandurangan. Efficient distributed community detection in the stochastic block model. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019.

[39] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[40] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[41] Joel Friedman. *A proof of Alon's second eigenvalue conjecture and related problems*. American Mathematical Society, Providence, RI, USA, 2008.

[42] Alan Frieze and Simi Haber. An almost linear time algorithm for finding Hamilton cycles in sparse random graphs with minimum degree at least three. *Random Structures & Algorithms*, 47(1):73–98, 2015.

[43] Alan Frieze and Michał Karoński. *Introduction to random graphs*. Cambridge University Press, New York, 2015.

[44] Alan Frieze, Michael Krivelevich, Peleg Michaeli, and Ron Peled. On the trace of random walks on random graphs. *Proceedings of the London Mathematical Society*, 116(4):847–877, 2018.

[45] Alan M. Frieze. Parallel algorithms for finding Hamilton cycles in random graphs. *Information Processing Letters*, 25(2):111–117, 1987.

[46] Alan M Frieze. Finding Hamilton cycles in sparse random graphs. *Journal of Combinatorial Theory, Series B*, 44(2):230–250, 1988.

[47] Alan M. Frieze and Colin McDiarmid. Algorithmic theory of random graphs. *Random Struct. Algorithms*, 10(1-2):5–42, 1997.

[48] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[49] Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *Proc. of the Int'l Symp. on Princ. of Dist. Comp.(PODC)*, 2017.

[50] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[51] Roman Glebov and Michael Krivelevich. On the number of Hamilton cycles in sparse random graphs. *SIAM J. Discrete Math.*, 27(1):27–42, 2013.

[52] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983.

[53] Alexandre Hollocou, Thomas Bonald, and Marc Lelarge. Multiple local community detection. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):76–83, 2018.

[54] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[55] Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Large-scale distributed algorithms for facility location with outliers. In *The 22nd International Conference on Principles of Distributed Systems (OPODIS)*, 2018.

[56] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[57] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, 83(1):016107, 2011.

[58] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

[59] Isabel M Kloumann and Jon M Kleinberg. Community membership identification from small seed sets. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1366–1375. ACM, 2014.

[60] János Komlós and Endre Szemerédi. Limit distribution for the existence of Hamiltonian cycles in a random graph. *Discrete Mathematics*, 43(1):55–63, 1983.

[61] Aleksei Korshunov. Solution of a problem of erdős and rényi on Hamilton cycles non-oriented graphs. *Soviet Math*, (17):760–764, 1976.

[62] Kishore Kothapalli, Sriram V Pemmaraju, and Vivek Sardeshmukh. On the analysis of a label propagation algorithm for community detection. In *International Conference on Distributed Computing and Networking*, pages 255–269. Springer, 2013.

[63] K Krzywdziński and Katarzyna Rybarczyk. Distributed algorithms for random graphs. *Theoretical Computer Science*, 605:95–105, 2015.

[64] Fabian Kuhn and Anisur Rahaman Molla. Distributed sparse cut approximation. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, 2015.

[65] Jing Lei, Alessandro Rinaldo, et al. Consistency of spectral clustering in stochastic block models. *The Annals of Statistics*, 43(1):215–237, 2015.

[66] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[67] Eythan Levy, Guy Louchard, and Jordi Petit. A distributed algorithm to find Hamiltonian cycles in {G}(n, p) random graphs. In *Workshop on Combinatorial and Algorithmic Aspects of Networking*, pages 63–74. Springer, 2004.

[68] László Lovász. Random walks on graphs: A survey. *Combinatorics*, 2:1–46, 1993.

[69] Philip D MacKenzie and Quentin F Stout. Optimal parallel construction of Hamiltonian cycles and spanning trees in random graphs. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 224–229. ACM, 1993.

[70] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[71] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[72] Alan Mislove, Bimal Viswanath, Krishna P Gummadi, and Peter Druschel. You are who you know: inferring user profiles in online social networks. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, pages 251–260. ACM, 2010.

[73] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, New York, 2017.

[74] Anisur Rahaman Molla and Gopal Pandurangan. Distributed computation of mixing time. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 5. ACM, 2017.

[75] Anisur Rahaman Molla and Gopal Pandurangan. Local mixing time: Distributed computation and applications. In *Proc. of 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 743–752, 2018.

[76] Michael Molloy. The freezing threshold for k-colourings of a random graph. *Journal of the ACM (JACM)*, 65(2):7, 2018.

[77] Elchanan Mossel, Joe Neeman, and Allan Sly. Stochastic block models and reconstruction. *arXiv preprint arXiv:1202.1499*, 2012.

[78] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[79] Alex Olshevsky and John N Tsitsiklis. Convergence speed in distributed consensus and averaging. *SIAM Review*, 53(4):747–772, 2011.

[80] Edgar M. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[81] Gopal Pandurangan. *Distributed Network Algorithms*. url: https://sites.google.com/site/gopalpandurangan/dna, 2016. Accessed on February 10, 2019.

[82] Gopal Pandurangan and Maleq Khan. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook*, chapter Theory of Communication Networks. CRC Press, 2009. (Focuses on the theoretical and algorithmic underpinnings of today's wide-area communication networks, especially the Internet.).

[83] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 429–438, 2016.

[84] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(1):4, 2018.

[85] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 405–414, 2018.

[86] David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, Philadelphia, PA, USA, 2000.

[87] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[88] Richard Peng, He Sun, and Luca Zanetti. Partitioning well-clustered graphs: Spectral clustering works! In *Conference on Learning Theory*, pages 1423–1455, 2015.

[89] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. *J. Graph Algorithms Appl.*, 10(2):191–218, 2006.

[90] Lajos Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14(4):359–364, 1976.

[91] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[92] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, 2015.

[93] Oliver Riordan. Random cliques in random graphs. *arXiv preprint arXiv:1802.01948*, 2018.

[94] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

[95] Devavrat Shah. Gossip algorithms. *Foundations and Trends® in Networking*, 3(1):1–125, 2009.

[96] Eli Shamir. How many random edges make a graph Hamiltonian? *Combinatorica*, 3(1):123–131, 1983.

[97] Andrew Thomason. A simple linear expected time algorithm for finding a Hamilton path. *Discrete Mathematics*, 75(1-3):373–379, 1989.

[98] Volker Turau. A $o(\log n)$ distributed algorithm to construct routing structures for pub/sub systems. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 65–79. Springer, 2018.

[99] Volker Turau. A distributed algorithm for finding Hamiltonian cycles in random graphs in $o(\log n)$ time. In *International Colloquium on Structural Information and Communication Complexity*, pages 72–87. Springer, 2018.

[100] Joyce Jiyoung Whang, David F Gleich, and Inderjit S Dhillon. Overlapping community detection using seed set expansion. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, pages 2099–2108. ACM, 2013.

[101] Haijun Zhou and Reinhard Lipowsky. Network Brownian motion: A new method to measure vertex-vertex proximity and to identify communities and subcommunities. In *International Conference on Computational Science*, pages 1062–1069. Springer, 2004.