Doctoral Thesis

Distributed Computation of Graph Structures by Mobile Agents

A thesis submitted to the Indian Statistical Institute in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

by

Prabhat Kumar Chand

under the supervision of

Dr. Anisur Rahaman Molla

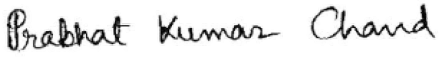
Cryptology and Security Research Unit,
Indian Statistical Institute,
203 B.T. Road, Kolkata - 700108

July 2026

Dedicated to my family and my teachers.

Declaration of Authorship

I, **Prabhat Kumar Chand**, a student of Cryptology and Security Research Unit, of the Ph.D. program of Indian Statistical Institute, Kolkata, hereby declare that the investigations presented in this thesis are based on my works and, to the best of my knowledge, the materials contained in this thesis have not previously been published or written by any other person, nor it has been submitted as a whole or as a part for any degree/diploma or any other academic award anywhere before.



Prabhat Kumar Chand
Cryptology and Security Research Unit,
Indian Statistical Institute, Kolkata 203 Barrackpore Trunk Road,
Kolkata, West Bengal, India - 700108

Acknowledgements

I would like to express my heartfelt gratitude to the individuals who have profoundly influenced my journey toward this milestone. This remarkable voyage has been truly transformative—shaping my character, career, and outlook. Alongside moments of triumph, it has offered valuable lessons drawn from setbacks and uncertainties. This research journey has instilled in me enduring values such as patience, perseverance, and the ability to transform critical feedback into meaningful improvements and clarity of thought. Though the path was often uncertain, the experiences, support, and—above all—the guidance I received have been indispensable.

I consider myself deeply fortunate to have been mentored by Dr. Anisur Rahaman Molla. I extend my sincere thanks to him for his constant support, wise counsel, and unwavering encouragement. His passion for research and distinctive approach to teaching have been truly inspiring, instilling in me a strong sense of purpose and motivation. His steady guidance and deep knowledge have played a pivotal role throughout my research journey and in the completion of this thesis. More than just a supervisor, he has often been a friend—someone who stood by me not only through the highs and lows of research but also alongside the challenges in my personal life. It has been an extraordinary privilege, and I shall remain forever grateful for his invaluable mentorship.

I am deeply grateful to my collaborators, whose guidance and support have significantly shaped the course of my doctoral research. I owe special thanks to my immediate senior Dr. Manish Kumar (currently a Postdoctoral Researcher at IIT Madras), who has been an enduring source of support and mentorship. From him, I learned the early fundamentals of academic writing, and his continued help throughout my Ph.D. has been invaluable. I am equally thankful to Dr. Sumathi Sivasubramaniam (currently an Assistant Professor at Thiagarajar College of Engineering, Madurai), whose encouragement, constructive feedback, and assistance in framing and refining different parts of our papers gave me much-needed confidence in my early research years. I am especially indebted to Prof. Apurba Das (currently an Assistant Professor at BITS Pilani, Hyderabad), who introduced me to the problems of triangle and butterfly counting, along with their numerous applications. His insights gave a new direction to my thesis, and his approachability, enthusiasm, and consistent support have made working with him a truly enriching experience. I consider myself fortunate to have collaborated with such generous and brilliant researchers.

Additionally, I am highly indebted to my friends and lab seniors at ISI Kolkata—Abhijit, Adri, Amit Da, Anirban, Aniruddha Da, Anjan Da, Anup, Avishek Da, Bibhuti Da, Diptendu Da, Faizan, Gourav, Krishnakanta, Laltu Da, Laxmikanta, Mustafizur Da, Nirupam, Rahul, Rakesh, Samir Da, Sarbendu Da, Shankar Da, Srijani, Subha, Subhadip Da, Subhra Di, Suman, Sushanta Da—and many others who have, at various times, extended their support, guidance, and friendship. I remain truly grateful for every moment we have shared during this journey.

I am also deeply grateful to my teachers at DAV Public School, DVC MTPS, Bankura, where I spent 14 formative years of my life. The care, affection, and encouragement I received there played a transformative role in shaping who I am today. In fact, I spent the first 25 years of my life in the nurturing environment of the Mejia Thermal Power Station (MTPS) Colony, up to the point of joining my Ph.D. It would be unjust not to acknowledge the exceptional quality of education and environment I was fortunate to receive during those years—not only at school but also from several generous elder mentors in the colony. I am equally indebted to many friends from the colony, and to some juniors whom I had the privilege of mentoring before I began my doctoral journey. This place holds a special and permanent place in my heart, for the quality of life, learning, and community it provided me. I am also grateful to my teachers at Bankura Christian College, where I completed my graduation, and at the University of Burdwan, where I pursued my post-graduation.

Now, to mention the most important people in my life—my family. I can never fully express how deeply grateful I am to my Baba and Maa. Their lifelong sacrifices, unwavering love, and quiet resilience have been the foundation of everything I have achieved. Their values, strength, and dedication in raising me have shaped not only my character but also my ability to persevere through life's challenges. Whatever I am today is because of them. I am also deeply thankful to my wife, Priya, whose patience, love, and steady support have brought immense strength and balance to my life during the most demanding phases of this journey. Her belief in me has meant more than words can convey. I am equally grateful to my sister, Dr. Pallabi Chand, for her affection, enthusiasm, and constant cheerfulness, which have been a source of joy and emotional strength throughout this journey. I also extend my heartfelt thanks to my extended family, whose blessings, encouragement, and quiet presence in my life have provided comfort, reassurance, and a sense of belonging that I will always cherish.

Finally, I would like to express my sincere gratitude to all those who may have been unintentionally omitted but have played a role in my journey. To everyone who crossed my path and showed me kindness, thank you from the bottom of my heart.

Prabhat Kumar Chand

Abstract

This thesis investigates how mobile agents with no centralised control can be employed in anonymous networks to perform efficient distributed graph computations. The network is modelled as a simple, undirected, anonymous graph with n nodes and m edges, where nodes are memoryless and indistinguishable, and edges represent bidirectional communication links or traversal paths for the agents. The mobile agents are uniquely identifiable, possess limited local memory, and operate under a *local communication model*, in which communication is restricted to agents co-located at the same node. Under this computational model, we explore how mobile agents can collaborate effectively to solve global network problems—including dispersion in the presence of crashes, the construction of spanning trees, the identification of dominating sets, and the analysis of sub-graph hierarchies. We study the time complexity and memory usage per agent required to solve the above problems.

The first contributory chapter addresses the fault-tolerant dispersion problem, aiming to evenly spread mobile agents across an anonymous graph in the presence of crash faults, from two initial configurations: rooted, where all the agents start at a single node, and arbitrary, where agents are initially scattered across the graph in multiple clusters. This chapter explores how dispersion can be achieved under both settings, ensuring that each node eventually hosts at most one functional agent despite crashes. The next chapter presents the problem of computing dominating sets using mobile agents, where two different algorithms are introduced: one for computing minimal dominating sets in $O(m)$ time when the agents are gathered at a single node, and another for scenarios where agents start from multiple clusters. Additionally, an $\ln \Delta$ approximation algorithm for the minimum dominating set problem is provided, where Δ is the maximum degree of the graph. The subsequent chapter focuses on subgraph analytics using mobile agents dispersed across the nodes of a graph. We present algorithms for triangle counting (3-cycles) in general graphs and butterfly counting (4-cycles) in bipartite graphs. The triangle counting framework extends to related problems such as truss decomposition, triangle centrality, and local clustering coefficient. These methods enable the distributed identification of cohesive structures and dense subgraphs, with butterfly counting being of relevance to bipartite graphs commonly found in social network analysis and recommendation systems. The final chapter focuses on constructing tree structures, specifically BFS trees and minimum spanning trees, using mobile agents. These algorithms, which assume minimal prior knowledge, improve upon existing methods by achieving better time complexity and optimal memory usage. Throughout the thesis, these graph problems are explored through the lens of the mobile agent framework, focusing on minimising the time complexity of the algorithms and memory usage per agent.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Over the past few decades, distributed computing using mobile entities has emerged as an area of significant interest within the computer science research community. This field investigates how multiple computational entities—operating without centralised control—can collaborate to solve global problems through local interactions. These entities, often referred to as robots, agents, or mobile entities, typically operate under various constraints such as limited communication, bounded memory, and restricted mobility. They are commonly deployed in environments modelled as networks or spatial domains, where the structure of the environment itself influences the nature of computation.

Research on distributed computation using mobile entities can be broadly classified into two major streams. The first is robot swarms, which typically involve large numbers of simple, identical physical robots operating in continuous Euclidean spaces. These systems draw inspiration from natural swarms (like ants or bees) and emphasise collective behaviour emerging from simple local rules. The second stream focuses on mobile agents, a more generalised framework that encompasses both physical robots and software agents. These agents are well-suited for operation in discrete environments, such as graphs, where the nodes represent processing units or spatial locations, and edges represent communication links or paths of movement. In such graph-based environments, agents navigate, interact, and compute while being constrained by the topology of the underlying network. The study of mobile agents in discrete spaces bridges distributed computing, algorithm design, and robotics, and continues to offer a rich landscape for both theoretical exploration and practical innovation.

In this thesis, we focus on autonomous agents operating in discrete environments modelled as graphs. The underlying graphs considered here are *anonymous*, meaning that their nodes do not carry any unique labels or identifiers. This assumption reflects natural security-conscious scenarios where nodes may deliberately withhold identity-related information or where identifiers are inherently unavailable. The agents operate on arbitrary, simple, connected graphs and are tasked with solving a range of classical graph problems, such as spanning tree construction, dominating set construction, and subgraph detection. A key constraint in this setting is the *local communication model*, where agents can communicate or exchange information only when they occupy the same node. These restrictions pose significant algorithmic challenges, particularly when agents must coordinate and compute using minimal memory and limited interaction. This thesis investigates how such constraints can be overcome through efficient distributed al-

gorithms that minimise time complexity and reduce memory usage per agent. The details of our model assumptions are provided in Section 1.1.2. As the scope of distributed systems continues to expand, the study of mobile agents in graph-based environments has gained increasing relevance, with applications spanning large-scale wireless sensor networks, decentralised internet protocols, multi-core architectures, and autonomous robotic systems. A central challenge in this domain is the design of local interaction rules that lead to globally correct, consistent, and efficient behaviour, even in the presence of uncertainty or anonymity. The problems explored in this thesis are reexamined through the lens of mobile agents to account for the additional difficulties posed by mobility, locality, and the absence of global knowledge, thereby contributing both to theoretical understanding and practical methodologies for distributed computation.

The thesis starts with the problem of dispersion—a foundational coordination task where the goal is to spread $k \leq n$ agents, initially positioned arbitrarily across the nodes of an n-node anonymous graph, such that each agent occupies a distinct node. First formulated by Augustine and Moses Jr. [9], the dispersion problem has evolved into a rich area of study due to its potential scope of applications to graph exploration, load balancing, coverage, etc. Dispersion is not only an interesting problem in its own right but also serves as a building block for more advanced distributed algorithms developed later in this thesis. In fact, in most of our works, dispersion helps to distribute n agents into n nodes and allows us to simulate message passing techniques from classical distributed systems into our mobile agent model. The first part of this work addresses the challenge of dispersion in the presence of crash faults—an adversarial setting where up to f agents may fail during execution. We show that it is still possible to solve dispersion efficiently under such failures, even when the agents begin in an arbitrary configuration scattered across multiple clusters. The algorithms presented are time-efficient and memory-optimal, requiring only $O(\log(k + \Delta))$ bits of memory per agent, and significantly improve upon existing bounds in the literature.

The thesis then moves to the problem of constructing dominating sets—a key concept in graph theory, computer networks and distributed computing. Dominating sets have important applications in wireless networks, such as routing, clustering, and backbone formation. While classical distributed algorithms for dominating sets assume message-passing capabilities or global knowledge, our agent-based approach assumes neither. We develop new algorithms that use graph exploration and dispersion to identify dominating nodes in both rooted and arbitrary configurations. Along the way, we discover that our minimal dominating sets also satisfy the properties of maximal independent sets, another classical and important structure in computer science. In addition, we also present an approximation algorithm that achieves a logarithmic-factor guarantee

with respect to the optimal solution.

Apart from coordination and coverage, agents can also be tasked with performing subgraph analytics—an area vital to network science. In the next part of the thesis, we investigate the problem of counting triangles and butterflies in a graph. These patterns play a foundational role in understanding the structure and function of networks. Triangle counts are used to compute clustering coefficients, identify dense subgraphs via truss decomposition, and evaluate node importance through triangle centrality and local clustering coefficient calculations. Butterfly counting, particularly relevant in bipartite graphs, is commonly used in community detection, fraud identification, and recommendation systems. We present algorithms that allow agents to compute local and global counts of such structures efficiently. These techniques are not only theoretically interesting but also applicable in practice—for instance, in drone-based surveillance systems that monitor crowd interactions, or in wireless sensor networks that need to assess redundancy and resilience.

The final part of the thesis focuses on constructing two fundamental tree structures—Breadth-First Search (BFS) trees and Minimum Spanning Trees (MST) used in a wide range of graph algorithms and network coordination tasks. Constructing such structures in the agent-based setting introduces several challenges, particularly in anonymous graphs where agents have no access to global identifiers or topology. Issues such as identifying or simulating a root node, synchronising agent movements, and distributing agents across levels of the tree pose key challenges and are carefully addressed. We develop algorithms that handle these challenges under a variety of initial conditions, ranging from fully clustered to fully dispersed agent placements, and even allow for flexibility in the number of agents used. Our MST construction notably improves upon earlier results by achieving a lower round complexity while preserving optimal memory usage per agent. Moreover, the MST structure helps in efficient leader election and enables a refined BFS construction, both accomplished without requiring any prior knowledge of global graph parameters.

Throughout the thesis, we see how seemingly simple mobile agents, when endowed with the right algorithms, can solve a wide range of classical problems in anonymous graphs, often under tight resource constraints and without central coordination. From fault-tolerant dispersion and dominating set construction to triangle and butterfly counting, and finally, to BFS and MST formation, this work presents a unified exploration of distributed computation through the coordinated motion of autonomous agents. The results are not only of theoretical importance but also suggest practical techniques for deploying agents in challenging real-world settings such as disaster response, social network analysis, and wireless sensor deployment, which showcases the application potential of mobile agents in distributed graph computation.

1.1 Preliminaries

1.1.1 Notations and Definitions

This section summarises the principal notations and definitions used throughout the thesis. For each chapter, we separately summarise the important problem-specific definitions, additional notations, and the corresponding agent variables used in the algorithms.

General Notations	
Symbol	**Description**
G	An arbitrary, anonymous, simple, connected undirected graph representing the communication network.
n	Number of nodes in the graph G.
m	Number of edges in the graph G.
Δ	Maximum degree among all nodes in G.
$\delta(v)$	Degree of node v in graph G.
D	Diameter of graph G.
k	Number of mobile agents in graph G.
λ	Maximum agent identifier (ID) assigned to an agent.
$\mathcal{R}$	Set of all mobile agents.
r_i	Mobile agent with identifier (ID) i.
$\|A\|$	Cardinality (size) of the set A.

Table 1.1: General graph and system notations used throughout the thesis.

Chapter 2: Crash-Tolerant Dispersion Algorithms for Mobile Agents

Dispersion. Given $k \leq n$ agents, initially placed arbitrarily on a graph of n nodes, the agents must reposition themselves autonomously to reach a configuration where each node has at most one agent on it.

Fault-Tolerant Dispersion. Given $k \leq n$ agents, up to f of which are faulty (which may fail by crashing), initially placed arbitrarily on a graph of n nodes, the non-faulty agents, i.e., the agents that have not crashed, must reposition themselves autonomously to reach a configuration where each node has at most one non-faulty agent on it.

Additional Notations: Chapter 2	
Symbol	**Description**
R_c	Rooted configuration from which all agents initially start.
f	Number of crash-faulty agents.
ℓ	Number of initial agent clusters.
List of Agent Variables Used – Refer Table 2.2	

Table 1.2: Additional notations used in Chapter 2.

Chapter 3: Algorithms for Finding Dominating Set using Mobile Agents

Dominating Set. For a graph $G = (V, E)$, a set $D \subseteq V$ is called a dominating set if every node $v \in V \setminus D$ has at least one neighbour in D.

Minimal Dominating Set. A dominating set D is called minimal if no proper subset of D is itself a dominating set.

Minimum Dominating Set. A dominating set $D^\star$ is called a minimum dominating set if it has the minimum possible cardinality among all dominating sets of G.

Approximate Minimum Dominating Set. A dominating set D is called an α-approximate minimum dominating set if $|D| \leq \alpha |D^\star|$, where $D^\star$ is a minimum dominating set and α is the approximation ratio.

Maximal Independent Set (MIS). A set $I \subseteq V$ is called an independent set if no two distinct nodes in I are adjacent in G. An independent set I is called a maximal independent set if no proper superset of I is an independent set.

Additional Notations: Chapter 3	
Symbol	**Description**
$\mathcal{D}$	Dominating set of graph G.
$\mathcal{D}^\star$	Minimum dominating set of graph G.
α	Approximation ratio with respect to the minimum dominating set.
List of Agent Variables Used – Refer Table 3.2	

Table 1.3: Additional notations used in Chapter 3.

Chapter 4: Discovery of Dense Subgraph Structures with Triangles and Butterflies

Triangle. A triangle in a graph $G = (V, E)$ is a set of three distinct nodes that are pairwise adjacent (a $3-$cycle).

Support. For a graph $G(V, E)$, the support of an edge $e \in E$ is the number of triangles in G that contain e.

k-Truss. A k-truss is defined as the largest subgraph T_k of $G(V, E)$ in which every edge has support at least $k - 2$ with respect to T_k.

trussness. The trussness of an edge e is the maximum value k such that e belongs to T_k but does not belong to T_{k+1}.

Triangle Centrality. The triangle centrality of a node v, denoted by $TC(v)$, is a triangle-based centrality measure that evaluates the structural importance of v according to its participation in triangles and the concentration of triangle structures around it.

Local Clustering Coefficient. For a node $v \in V$ with degree $\delta(v) \geq 2$, let $\mathbf{T}(v)$ denote the number of triangles containing v. The local clustering coefficient of v, denoted by $LCC(v)$, is defined as $LCC(v) = \frac{2\mathbf{T}(v)}{\delta(v)(\delta(v)-1)}$, which measures the fraction of pairs of neighbours of v that are adjacent to each other, quantifying the local density of triangle structures around v.

Butterfly. For a bipartite graph $G((A, B), E)$, a 2×2 biclique on (A, B) is called a butterfly.

Additional Notations: Chapter 4	
Symbol	**Description**
$\mathbf{T}(v)$	Number of triangles containing node v.
$\mathbf{T}(G)$	Total number of triangles in graph G.
$TC(v)$	Triangle centrality of node $v \in G$.
$LCC(v)$	Local clustering coefficient of node $v \in G$.
T_k	The k-truss subgraph of graph G.
A, B	Two partitions of a bipartite graph.
List of Agent Variables Used – Refer Table 4.2	

Table 1.4: Additional notations used in Chapter 4.

Chapter 5: Constructing Tree Structures in Anonymous Graphs with Mobile Agents

Breadth-First Search (BFS) Tree. A BFS tree rooted at a node r is a spanning tree in which the distance of every node from r in the tree is equal to its shortest-path distance from r in the original unweighted graph.

Minimum Spanning Tree (MST). A minimum spanning tree of a weighted graph is a spanning tree whose total edge weight is minimum among all spanning trees of the graph.

Fragment. A fragment is a connected component or partial tree formed during the execution of the MST construction algorithm.

Minimum Weight Outgoing Edge (MWOE). For a fragment F, an edge connecting a node of F to a node outside F having minimum weight among all such outgoing edges is called the Minimum Weight Outgoing Edge.

Leader Election. Leader election is the process of selecting a unique agent that acts as the coordinator for the entire system.

Additional Notations: Chapter 5	
Symbol	**Description**
p_{uv}	Port number at node u leading to node v.
T	Constructed BFS tree or MST.
T_k	Fragment/tree formed during MST construction.
$w_{u,v}$	Dynamically computed virtual weight of edge (u, v).
$treelabel$	Identifier associated with a fragment/tree.
$treelevel$	Level associated with a fragment during MST merging.
List of Agent Variables Used – Refer Table 5.2	

Table 1.5: Additional notations used in Chapter 5.

The thesis also uses certain local symbols and temporary variables that are defined contextually within specific sections to support particular algorithms, proofs, or intermediate computations. We now proceed to describe the general computational model that forms the foundation for the work throughout this thesis. While this model serves as the baseline framework in most chapters, certain variations are introduced depending on the problem setting under consideration. These deviations are explicitly stated in the corresponding chapters. In particular, the modifications may involve changes to the underlying graph structure, such as the use of bipartite graphs for butterfly counting in Chapter 4, or changes in assumptions regarding the agents themselves, such as the consideration of crash-prone agents in Chapter 2.

1.1.2 Model

Graph: We have an underlying graph $G(V, E)$ which is connected, undirected and anonymous with $|V| = n$ nodes and $|E| = m$ edges. Unless stated otherwise, graphs are assumed unweighted. The nodes of G do not have any distinguishing identifiers or labels. The nodes do not possess any memory and hence cannot store any information. The degree of a node $v \in V$ is denoted by $\delta(v)$, and the maximum degree of G is Δ. Edges incident on v are locally labelled using port numbers in the range $[0, \delta(v) - 1]$. A single edge connecting two nodes receives independent port numbering at either end. The edges of the graph serve as *routes* through which the agents can commute. Any number of agents can travel through an edge at any given time.

Mobile Agents: We have a collection of k agents $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}$ residing on the nodes of the graph. Each agent has a unique ID and has finite bits of memory to store information (we bound their IDs to n^c, where c is an arbitrary constant). An agent cannot store the whole graph structure information within its limited memory. An agent retains its memory as long as needed, and it can be updated as required. Two or more agents can be present (*co-located*) at a node or pass through an edge in G. However, an agent is not allowed to stay on an edge. An agent can recognise the port number through which it has entered and exited a node. The agents do not have any visibility beyond their (current) location at a node. An agent at node v can only see the adjacent ports (connecting to edges) at v. Only the collocated agents at a node can sense each other and exchange information. An agent can transfer all the information stored in its memory in a single round.

Communication Model: We consider a synchronous system where the agents are synchronised to a common clock and the local communication model, where only co-located agents (i.e., agents at the same node) can communicate among themselves.

Time Cycle: Each agent r_i, on activation, performs a $Communicate - Compute - Move$ (CCM) cycle as follows.

- **Communicate:** r_i may communicate with other agents at the same node.
- **Compute:** Based on the gathered information and subsequent computations, r_i may perform all manner of computations within the bounds of its memory.
- **Move:** r_i may move to a neighbouring node using the computed exit port.

Starting Configuration: We assume three different starting configurations for the k agents.

- **Rooted Initial Configuration:** All the k agents start from a designated node called the $root$ node.
- **Arbitrary Initial Configuration:** The agents are placed arbitrarily in small clusters across different nodes of the graph and represent the most generalised starting configurations for the agents.
- **Dispersed Initial Configuration:** In this configuration, each node of the graph G has at least one agent stationed in it at the start. For $k = n$, there is exactly one agent at each node of the graph.

Complexity Measures: An agent can perform the CCM task in one time unit, called *round*. The *time complexity*[1] of an algorithm is the number of rounds required to achieve the goal. The *memory complexity* is the number of bits required by each agent to execute the algorithm.

1.2 Literature Review

In this section, we present a concise literature review highlighting the related works corresponding to each contributory chapter. A more detailed discussion of the relevant literature is provided within the respective chapters.

Chapter 2 - Crash-Tolerant Dispersion Algorithms for Mobile Agents In Chapter 2, we study crash-tolerant dispersion algorithms for mobile agents. The dispersion problem was first introduced by Moses Jr. *et al.* [9], who provided two algorithms for arbitrary graphs—one requiring $O(\log n)$ memory per agent with a time complexity of $O(mn)$, and another requiring $O(n \log n)$ memory and running in $O(m)$ rounds. Since then, several works have extended and improved dispersion algorithms under various models and assumptions [72, 68, 111, 76, 92, 35]. Recently, Sudo *et al.* [116], proposed near-optimal algorithms: one that reduces the time complexity from

[1] Alternatively called the *round complexity*.

$O(\min(m, k\Delta))$ to $O(k \log \min k, \Delta)$ for rooted configurations, and to $O(k \log k \log \min k, \Delta)$ for arbitrary ones, both using only $O(\log(k + \Delta))$ bits of memory per agent. They also introduced a time-optimal $O(k)$-round algorithm for the rooted case requiring $O(\Delta + \log k)$ bits of memory. More recently, Kshemkalyani *et al.* [70] presented optimal $O(k)$-round dispersion algorithms for both rooted and arbitrary configurations using $O(\log(k + \Delta))$ memory per agent, along with a time-optimal (within an $O(\log k)$ factor) asynchronous algorithm with $O(k \log k)$ rounds and $O(\log(k + \Delta))$ memory. The crash-tolerant version of dispersion was studied by Pattanayak *et al.* [99], who considered a rooted initial configuration where some agents may be crash-prone. Their algorithm tolerates an arbitrary number of crashes, completes in $O(f \cdot \min(m, k\Delta))$ rounds, and uses $O(\log(k + \Delta))$ bits of memory per agent. This was later improved in [100], where two algorithms were proposed under different memory settings: the first achieves $O(\min(m, k\Delta))$ rounds using $O(k \log(k + \Delta))$ memory per agent, and the second runs in $O(\min(m, k\Delta)(l + f))$ rounds using only $O(\log(k + \Delta))$ bits of memory. Both approaches handle rooted ($l = 1$) and arbitrary configurations, without requiring any prior knowledge of graph parameters.

Chapter 3 - Algorithms for Finding Dominating Set using Mobile Agents The first efficient distributed implementation of the dominating set problem in the CONGEST model was studied in [60] by Jia *et al.* Their randomized algorithm, a refinement of the greedy approach from [85], runs in $O(\log n \log \Delta)$ rounds and achieves a $\ln(\Delta)$-approximate dominating set in expectation, with at most a constant number of messages exchanged between any pair of nodes. Sultanik *et al.* [117] addressed a variant of the art gallery problem—equivalent to finding a minimal dominating set in visibility graphs—via a distributed algorithm that completes in a number of rounds proportional to the graph's diameter and returns a solution within a constant factor of optimal with high probability. Kuhn and Wattenhofer [78] proposed LP-based approximation algorithms that compute a dominating set of expected size $(k\Delta^{2/k} \log \Delta)$ times optimal in $O(k^2)$ rounds, where k is a tunable parameter. Each node transmits $O(k^2\Delta)$ messages of size $O(\log \Delta)$; setting k as a constant yields the first constant-round, non-trivial approximation algorithm. Lower bounds were established by Kuhn *et al.* [77], showing that achieving even a polylogarithmic approximation for minimum dominating set or vertex cover requires at least $\Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ and $\Omega\left(\sqrt{\frac{\log \Delta}{\log \log \Delta}}\right)$ rounds, respectively, in the message-passing model. More recently, [38] presented deterministic approximation algorithms for minimum dominating sets in CONGEST, achieving an approximation factor of $(1 + \epsilon)(1 + \log(\Delta + 1))$, with runtimes of $O(2^{O(\sqrt{\log n \log \log n})})$ and $O(\Delta, \text{polylog}(\Delta) + \text{polylog}(\Delta) \log^\star n)$ for $\epsilon > 1/\text{polylog}(\Delta)$. They also extended their techniques to obtain a $\ln(\Delta)$-approximate connected dominating set.

Chapter 4 - Agent-Based Discovery of Dense Subgraph Structures with Triangles and Butterflies A wide range of distributed and parallel algorithms have been proposed for triangle enumeration across various computational models, including distributed memory, shared memory, multi-core architectures, and message-passing interfaces (MPI). Arifuzzaman *et al.* [6] introduced PATRIC, an MPI-based distributed memory parallel algorithm for triangle counting in massive networks. Shun *et al.* [112] developed multi-core parallel algorithms for both exact and approximate triangle computations, scaling efficiently to graphs with billions of nodes and edges. In a follow-up work, Arifuzzaman *et al.* [7] proposed two MPI-based algorithms—one optimising runtime using overlapping partitions and load balancing, and another minimising memory via non-overlapping partitions. Ghosh *et al.* [51] introduced a node-based MPI triangle counting framework, TriC, for both shared and distributed-memory settings, which was later refined in [50]. Comprehensive surveys and additional works on triangle enumeration can be found in [122, 19, 17, 120, 81, 8, 52, 118, 97]. Similarly, butterfly counting in bipartite graphs has received extensive attention in both sequential and parallel contexts. Sanei-Mehri *et al.* [107] presented efficient exact and approximate algorithms, while Wang *et al.* [125] enhanced performance using node prioritisation. Zhou *et al.* [136] extended butterfly analysis to uncertain bipartite graphs. Sarıyüce and Pinar [108] introduced the tip and wing decompositions for dense subgraph discovery. On the parallel front, Shi and Shun [110] developed multicore algorithms, Xu *et al.* [133] proposed GPU-based methods, and Weng *et al.* [129] designed distributed algorithms suitable for dynamic graph streams.

Chapter 5 - Constructing Tree Structures in Anonymous Graphs with Mobile Agents Some of the earliest studies on distributed breadth-first search (BFS) date back to Cheung [28] and Gallager *et al.* [46, 47] in the 1980s. In [47], the authors considered asynchronous point-to-point communication networks modelled by undirected graphs and proposed two BFS algorithms. The first has communication complexity $O((E + V^{1.5}) \log V)$ (later improved in [11]) and time complexity $O(V^{1.5} \log V)$. The second, a recursive version, achieves $O(E \cdot 2^{\sqrt{\log V \log\log V}})$ messages and $O(V \cdot 2^{\sqrt{\log V \log\log V}})$ rounds. Makki [89] later improved upon this by proposing a parallel BFS with enhanced synchronisation, achieving a $2l$ round algorithm (where l is the height of the BFS tree) using only $2(V - 1)$ messages. Chow *et al.* [30] implemented an experimental BFS on massive graphs (billions of edges) distributed across machines. Awerbuch *et al.* [12, 13] proposed the *STRIP-EXPLORE* algorithm, enabling a single agent to explore graphs via piecemeal BFS in $O(E + V^{1.5})$ rounds. Palanisamy *et al.* [96] developed a multi-agent strategy where the graph is partitioned into clusters, and each of the k agents performs BFS in its region sequentially, yielding a total runtime of $O(k(V + E))$. In [75], Kshemkalyani *et al.* employed BFS traversal

to solve the *dispersion* problem, where $k \leq n$ agents are distributed to distinct nodes. Early distributed Minimum Spanning Tree (MST) construction was studied by Spira [114], focusing on communication complexity. This led to the classical GHS algorithm [45], which builds an MST in $O(n \log n)$ rounds using $O(n \log n + m)$ messages, each carrying one weight and three bits. Subsequent work improved the time complexity to $O(n)$ [10] and later to $O(\sqrt{n} \log^\star n + D)$ [48], with a lower bound of $O(\frac{\sqrt{n}}{\log n} + D)$ established in [101]. In the agent-based setting, [69] presented an MST algorithm using n agents with time complexity $O(m + n \log n)$. If agents begin dispersed, each requires $O(\max(\Delta, \log n) \log n)$ bits of memory; for arbitrary starting positions, the requirement increases to $O(n \log n)$ bits.

1.3 Contribution and Organisation of the Thesis

In this section, we outline the central problems addressed in the thesis, summarise the main contributions and results, and describe the overall structure of the thesis. The subsequent chapters are organised as follows.

Chapter 2 - Crash-Tolerant Dispersion Algorithms for Mobile Agents. This chapter addresses the problem of *dispersion* in the presence of crash faults, where mobile agents must spread out such that at most one agent occupies each node of an anonymous graph. Moreover, throughout this thesis, dispersion serves as a foundational primitive for several higher-level coordination problems as discussed later, which provides a convenient and unified framework that enables localised computation and structured coordination under the local communication model. While prior works assume that all agents are fault-free, real-world systems must tolerate failures. This chapter introduces algorithms that handle up to f crash faults under two distinct initial configurations: *rooted* and *arbitrary*. In the rooted setting, where all agents start from a single node, we present an algorithm that achieves fault-tolerant dispersion in $O(k^2)$ rounds, improving on existing results. In the arbitrary configuration, where agents begin in ℓ separate clusters, we propose an algorithm that completes dispersion in $O((f+\ell) \cdot \min(m, k\Delta, k^2))$ rounds, assuming the agents know the parameters m, k, f, ℓ and Δ. Both algorithms are memory-efficient, requiring only $O(\log(k + \Delta))$ bits per agent.

Chapter 3 - Algorithms for Finding Dominating Set using Mobile Agents. It explores the problem of computing dominating sets in a graph using mobile agents. This chapter presents two algorithms for computing a *minimal dominating set*: the first runs in $O(m)$ rounds when all agents

Reference	Initial Config.	Knowledge	Time Complexity (Rounds)	Memory
[99]	Rooted	None	$O(f \cdot \min(m, k\Delta))$	$O(\log(k+\Delta))$
[100]	Arbitrary	None	$O(\min(m, k\Delta))$	$O(k\log(k+\Delta))$
[100]	Arbitrary	None	$O(\min(m, k\Delta)(\ell+f))$	$O(\log(k+\Delta))$
Section 2.4	Rooted	None	$O(k^2)$	$O(\log(k+\Delta))$
Section 2.5	Arbitrary	m, Δ, ℓ, k, f	$O((f+\ell)\min(m, k\Delta, k^2))$	$O(\log(k+\Delta))$

Table 1.6: Summary of crash-fault-tolerant dispersion algorithms for anonymous graphs. A detailed comparison is provided in Table 2.1 of Chapter 2.

start from a single node, and the second operates in $O(\ell\Delta\log(\lambda)+n\ell+m)$ rounds when agents begin from ℓ arbitrary clusters, where m is the number of edges, Δ is the maximum degree, and λ is the maximum ID-length. In addition, we propose a distributed $\ln(\Delta)$-approximation algorithm for computing a *minimum dominating set*, which completes in $O(n\Delta\log(\lambda))$ rounds.

Problem	Reference	Knowledge	Time Complexity (Rounds)	Memory	Initial Config.
Minimal Dominating Set	Section 3.4.1	–	$O(m)$	$O(\log n)$	Rooted
	Section 3.4.2	λ, Δ, n, ℓ	$O(\ell\Delta\log\lambda+n\ell+m)$	$O(\log n)$	Arbitrary
Approximate Minimum Dominating Set	Section 3.5	Δ, λ, n	$O(n\Delta\log\lambda)$	$O(\log n)$	Dispersed

Table 1.7: Summary of our results on computing dominating sets using mobile agents. Details are provided in Table 3.1 of Chapter 3.

Chapter 4 - Agent-Based Discovery of Dense Subgraph Structures with Triangles and Butterflies. In this chapter, we present distributed algorithms for n mobile agents operating on anonymous graphs to solve triangle counting and its extensions, including truss decomposition, triangle centrality, and local clustering coefficient. Our algorithms run in $O(\Delta\log\lambda)$ rounds for local triangle computations and $O(D\Delta\log\lambda)$ rounds for global counts, using only $O(\Delta\log n)$ bits of memory per agent. Additionally, we propose efficient agent-based algorithms for butterfly counting in bipartite graphs, including leader election and spanning tree construction in $O(n\log\lambda)$ rounds—techniques that extend naturally to general graphs. Butterfly counting at each node is completed in $O(\Delta)$ rounds, while the total butterfly count for the graph is computed in $O\big(\Delta+\min|A|,|B|\big)$ rounds.

Chapter 5 - Constructing Tree Structures in Anonymous Graphs with Mobile Agents. Breadth-First Search (BFS) and Minimum Spanning Tree (MST) constructions are fundamental

Problem	Reference	Knowledge	Time Complexity (Rounds)	Memory
Node-Based Triangle Counting	Section 4.5.2	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$
Edge-Based Triangle Counting	Section 4.5.2	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$
Total Triangle Counting	Section 4.5.3	Δ, λ, D	$O(D\Delta \log \lambda)$	$O(\Delta \log n)$
Truss Decomposition	Section 4.6.1	Δ, λ, D	$O(m\Delta \log \lambda + mD)$	$O(\Delta \log n)$
Triangle Centrality	Section 4.6.2	Δ, λ, D	$O(\Delta \log \lambda)$ or $O(D\Delta \log \lambda)$	$O(\Delta \log n)$
Local Clustering Coefficient	Section 4.6.3	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$
Node-Based Butterfly Counting	Section 4.11	λ	$O(\Delta)$	$O(\Delta \log n)$
Total Butterfly Counting	Section 4.11	λ	$O(\Delta + \min\{\|A\|, \|B\|\})$	$O(\Delta \log n)$

Table 1.8: Summary of our results on triangle and butterfly analysis using mobile agents. Other details are provided in Table 4.1 of Chapter 4. In all of these problems, the agents start at a dispersed configuration.

primitives in distributed graph algorithms. In this chapter, we study these problems in the same agent-based network, where n autonomous mobile agents operate on an anonymous n-node, m-edge graph G in synchronous rounds. The agents start from a *dispersed* configuration, with exactly one agent located at each node, and communicate only when they meet at the same node. We first present an MST construction algorithm, which improves over previous works and elects a unique leader agent. The algorithm runs in $O(n \log n + \Delta \log^2 n)$ rounds using $O(\log n)$ memory per agent, where Δ is the maximum degree of G. We then present BFS constructions for both known-root and root-free settings. When a root is already known, the agents construct a BFS tree in $O(D\Delta)$ rounds using $O(\log n)$ memory per agent, where D is the diameter of G. In the root-free setting, a leader is first obtained through the MST construction and is then used as the root for BFS. In this case, the overall BFS construction takes $O(\min(D\Delta, m \log n) + n \log n + \Delta \log^2 n)$ rounds using $O(\log n)$ memory per agent. In this chapter, we assume that each agent knows λ, the maximum agent identifier, and no other graph parameter is required to be known a priori.

Conclusion and Future Works 6 In this chapter, we provide a summary of the thesis's main contributions and highlight several interesting problems for future research.

Problem	Reference	Knowledge	Time Complexity (Rounds)	Memory
Leader Election	Section 5.4	λ	$O(n \log n + \Delta \log^2 n)$	$O(\log n)$
	Kshemkalyani *et al.* [69]	–	$O(m)$	$O(\log^2 n)$
	Kshemkalyani *et al.* [71]	n, Δ	$O(n \log^2 n + D\Delta \log n)$	$O(\log n)$
MST Construction	Section 5.4	λ	$O(n \log n + \Delta \log^2 n)$	$O(\log n)$
	Kshemkalyani *et al.* [69]	–	$O(m + n \log n)$	$O(\max(\Delta, \log n) \log n)$
	Kshemkalyani *et al.* [71]	n, Δ	$O(m + n \log^2 n + D\Delta \log n)$	$O(\Delta \log n)$
BFS Construction	Section 5.5	λ	$O(n \log n + \Delta \log^2 n + \min(m \log n, D\Delta)$	$O(\log n)$
	Section 5.5	root	$O(\min(m \log n, D\Delta))$	$O(\log n)$

Table 1.9: Comparison of our results with major related works for leader election, MST construction, and BFS construction using mobile agents. A more detailed discussion is provided in Chapter 5.

1.4 List of Publications

1. **Fault-Tolerant Dispersion of Mobile Robots**

   (a) **Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam.** *Fault-Tolerant Dispersion of Mobile Robots.* In: Algorithms and Discrete Applied Mathematics. Proceedings of the 9th International Conference, CALDAM 2023, Gandhinagar, India, February 9–11, 2023. Lecture Notes in Computer Science, vol 13756. Springer, Cham, pp. 28–40. https://doi.org/10.1007/978-3-031-25211-2_3

   (b) **Prabhat Kumar Chand, Manish Kumar, Sumathi Sivasubramaniam, and Anisur Rahaman Molla.** *Fault-Tolerant Dispersion of Mobile Robots. Discrete Applied Mathematics*, Elsevier, Vol. 377, pp. 299–313, 2025. DOI: 10.1016/j.dam.2025.06.068.

   Chapter 2 is based on these works.

2. **Prabhat Kumar Chand, Anisur Rahaman Molla, and Sumathi Sivasubramaniam.** *Run for Cover: Dominating Set via Mobile Agents.* In: Georgiou, K., Kranakis, E. (eds) Algorithmics of Wireless Networks. ALGOWIN 2023. Lecture Notes in Computer Science, vol 14061. Springer, Cham. https://doi.org/10.1007/978-3-031-48882-5_10

   Chapter 3 is based on this work.

3. **Agent-Based Triangle Counting and Its Applications in Anonymous Graphs.**

   (a) **Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla.** *Agent-Based Triangle Counting and Its Applications in Anonymous Graphs (Extended Abstract).* In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), pp. 2186–2188. https://dl.acm.org/doi/abs/10.5555/3635637.3663102

   (b) **Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla.** *Agent-Based Triangle Counting: Unlocking Truss Decomposition, Triangle Centrality, and Local Clustering Coefficient.* arXiv preprint, 2025. https://arxiv.org/abs/2402.03653 (A fully extended and improved version of the extended abstract, submitted to ACM/IEEE Transactions on Networking)

   Chapter 4 is based on this work.

4. **Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla.** *Brief Announcement: Distributed Butterfly Analysis using Mobile Agents.* In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2025)*, Portland, OR, USA, July 28–August 1, 2025, pp. 623–627. ACM, 2025. DOI: 10.1145/3694906.3743334.

   Chapter 4 is based on this work.

5. **Prabhat Kumar Chand, Manish Kumar, and Anisur Rahaman Molla.** *Agent-Driven BFS Tree in Anonymous Graphs with Applications.* In: Proceedings of the 12th International Conference on Networked Systems (NETYS 2024), Rabat, Morocco, pp. 67–82. https://doi.org/10.1007/978-3-031-67321-4_4

   Chapter 5 is based on this work.

6. **Prabhat Kumar Chand, Manish Kumar, and Anisur Rahaman Molla.** *Computing Tree Structures in Anonymous Graphs via Mobile Agents.* In *Stabilization, Safety, and Security of Distributed Systems – 27th International Symposium (SSS 2025)*, Kathmandu, Nepal, October 9–11, 2025, *Lecture Notes in Computer Science*, pp. 111–127. Springer, 2025. DOI: 10.1007/978-3-032-11127-2_11.

   Chapter 5 is based on this work.

Chapter 2

Crash-Tolerant Dispersion Algorithms for Mobile Agents

In this chapter, we study the *dispersion* problem, where $k \leq n$ mobile agents must spread across an n-node anonymous graph so that at most one agent occupies each node. We consider this problem in the presence of crash faults, aiming to achieve dispersion regardless of the initial placement of the agents. In a crash-fault setting, up to $f \leq k$ agents may fail arbitrarily by crashing, thereby losing all stored information and becoming incapable of communication. We solve the dispersion problem under crash-fault conditions by analysing two different initial configurations: (i) the rooted configuration, and (ii) the arbitrary configuration. In the rooted case, all agents start at a single node. In contrast, the arbitrary configuration allows the agents to be initially distributed across the graph in $\ell < k$ arbitrary clusters. For the rooted case, we design an algorithm that achieves dispersion in the presence of faulty agents in $O(k^2)$ rounds, improving upon the prior results by [99, 100]. For the arbitrary configuration, we present an algorithm that solves dispersion in $O((f+\ell) \cdot \min(m, k\Delta, k^2))$ rounds. Both our algorithms use $O(\log(k+\Delta))$ bits of memory per agent.

This chapter is based on the following works:

Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, Sumathi Sivasubramaniam. *Fault-Tolerant Dispersion of Mobile Robots.* In: Algorithms and Discrete Applied Mathematics. Proceedings of the 9th International Conference, CALDAM 2023, Gandhinagar, India, February 9–11, 2023. Lecture Notes in Computer Science, vol. 13756. Springer, Cham, pp. 28–40. https://doi.org/10.1007/978-3-031-25211-2_3

Prabhat Kumar Chand, Manish Kumar, Sumathi Sivasubramaniam, and Anisur Rahaman Molla. *Fault-Tolerant Dispersion of Mobile Robots. Discrete Applied Mathematics*, Elsevier, Vol. 377, pp. 299–313, 2025. DOI: 10.1016/j.dam.2025.06.068.

2.1 Introduction

The dispersion of autonomous mobile agents to spread them out evenly in a region is a problem of significant interest in distributed robotics, e.g. [53, 54]. Initially, this problem was formulated by Augustine and Moses Jr. [9] in the context of graphs. They defined the problem as follows: Given any arbitrary initial configuration of $k \leq n$ agents positioned on the nodes of an n-node anonymous graph, the agents reposition autonomously to reach a configuration where each agent is positioned on a distinct node of the graph. Mobile agents dispersion has various real-world and practical applications, such as the relocation of self-driving electric cars (agent) to recharge stations (nodes) (finding a free or empty charging station, assuming that the cars have smart devices to communicate with each other), exploration (to visit each node of the graph in minimum possible time), scattering (spread out in an equidistant manner in symmetric graphs like rings), load balancing (nodes send or receives loads, and distributes them evenly among the nodes), covering, and self-deployment can all be explored as mobile agent dispersion problems [68, 72, 74, 75].

The problem has been extensively studied in different graphs with varying assumptions since its conceptualisation [92, 68, 74, 72, 73, 75, 76, 93, 94, 95]. We continue the study about the trade-off between memory requirement and time to solve the dispersion problem. In [99], the authors have studied the problem of dispersion in a setup where some of these mobile agents are prone to crash faults. Whenever an agent crashes, it loses all its information immediately, as if the agent had vanished from the network. This makes the problem more challenging and brings the problem to a more practical level, where, quite naturally, some agents may be faulty, and one could expect them to crash at any moment. With the limited resources that these agents have, it is a challenging job to figure out how they can coordinate with each other and perform large, distributed tasks. We have continued to study the efficacy of the problem in the same faulty environment. We have studied the dispersion problems with the rooted and arbitrary configuration of the agent with faulty setups and have been able to provide two fault-tolerant algorithms [25].

In our work, we mainly discuss two problems of mobile agent dispersion in the presence of crash-fault: i) rooted configuration and ii) arbitrary configuration. In the rooted case, all agents are placed together at a single node (called the $root$) at the start. On the other hand, the arbitrary configuration is a general configuration where the agents are initially placed in some $\ell < k$ clusters arbitrarily across the graph. Our first algorithm works for the rooted configuration by using depth-first search (DFS) traversal and improves the round complexity from $O(f \cdot \min(m, k\Delta))$ rounds in [99] to $O(k^2)$ rounds (in the worst case, improvement is from cubic to quadratic). The

second algorithm for the arbitrary graph is an entirely new result whose complexity depends upon the factors like the number of faulty agents (f), number of agent clusters (ℓ), total number of edges in the graph (m), number of agent (k) and the maximum degree of the graph (Δ). In this case, we have a round complexity of $O((f+\ell)\cdot \min(m, k\Delta, k^2))$. Both of our algorithms maintain an optimal level of memory requirement for each agent, requiring $O(\log(k+\Delta))$ bits per agent.

2.1.1 Challenges and Techniques

Both of our algorithms, Algorithm 1 and Algorithm 2 have Depth First Search (DFS) as their foundation. Since the nodes themselves are indistinguishable, navigation is done via the agent that settles on the nodes. Furthermore, the agents cannot communicate between themselves unless they are in the same node. In a crash setting, this causes immediate problems in navigation. Faulty agents, when they crash at inappropriate times, can also create endless cycles or can increase the number of clusters on the arbitrary initial configuration. In this chapter, we have tried to solve the problem by overcoming the said challenges, minimising the time complexity and keeping an optimal memory requirement for each agent.

For the rooted initial configuration, we perform a DFS on G from the $root$ vertex. Agents from the $root$ are released one by one as they explore the graph sequentially. The first agent from the $root$ settles down at the $root$ and sets the minimum port number (that is yet unexplored) available at the $root$ as its current direction pointer (cdr). The next agent follows the cdr of the previous agent and reaches a new node, where it settles and sets its own cdr pointer. Continuing in a similar way, the succeeding agent builds upon the DFS and continues to explore the graph. While exploring the graph, the agent can complete the exploration of a certain part of the graph via a particular vertex when we set the backtrack value of the agent in the particular vertex to 1. This ensures no agent further visits this part of the graph unnecessarily. In our algorithm, the i^{th} agent is sent only after $3i$ rounds have elapsed. The i^{th} agent uses the $3i$ rounds to explore and, if needed, return to the $root$ (the agent returns to the $root$ if it does not find a new node to settle after $2i$ rounds). In such a case, the agent keeps exploring new edges before it breaks out into a new path and settles at a new node there. During the algorithm, if any agent crashes, the succeeding agent from the $root$, which follows the previously set DFS path, corrects any inconsistency in the pointers of the settled agent caused due to crashing. We claim that each such crash could only extend the number of rounds by an $O(k)$, at-most (Lemma 2.2) and eventually the algorithm (Algorithm 1) completes in $O(k^2)$ rounds.

The arbitrary configuration required a different approach, however. At the start, we have $\ell < k$ clusters of agents at ℓ different nodes of the graph. Our algorithm (Algorithm 2) runs in phases, where each phase consists of $\min(m, k\Delta, k^2)$ rounds. At the start of each phase, each cluster begins a $counter$ that counts down from $\min(m, k\Delta, k^2)$. At the start of the phase, each cluster C_i begins exploring the network in parallel using a DFS approach. Unlike the rooted configuration, individual agents do not explore and return, but the entire cluster moves together. Whenever a cluster encounters a new (empty) node in the network, the agent with the current highest ID in the cluster settles and updates its flag variables accordingly. Each time the cluster moves through an edge to a different node, the counter is decreased by 1. When the counter becomes zero, all flags are reset. After that, each cluster starts exploring the network with its current node as a point of origin. This continues until all agents in the cluster settle, or the algorithm ends. The algorithm completes in $O((\ell + f)\min(m, k\Delta, k^2))$ rounds with each agent requiring an optimal $O(\log(k + \Delta))$ bits of memory.

2.2 Related Works

The problem of dispersion was first introduced in [9] by Moses Jr. *et al.* , where they solved the problem for different types of graphs. They had given a lower bound of $\Omega(\log n)$ on the memory of each agent (later, made more specific with $\Omega(\log(\max(k, \Delta)))$ in [72]) and of $\Omega(D)$ on the time complexity, for any deterministic algorithm on arbitrary graphs. They also proposed two algorithms on arbitrary graphs, one requiring $O(\log n)$ memory and running for $O(mn)$ time, while the other needing a $O(n \log n)$ memory and having a time complexity of $O(m)$.

Kshemkalyani and Ali [68] provided several algorithms for both synchronous and asynchronous models. In the synchronous model, they solved the dispersion problem in $O(\min(m, k\Delta))$ rounds with $O(k \log \Delta)$ memory. For the asynchronous cases, they proposed several algorithms, one particularly requiring $O(\Delta^D)$ rounds and $O(D \log \Delta)$ memory, while another requiring $O(\max(\log k, \log \Delta))$ memory and having a time complexity of $O((m - n)k)$. In a later work, Kshemkalyani *et al.* , in [72] improved the time complexity to $O(\min(m, k\Delta) \log k)$ keeping the memory requirement to $O(\log n)$, while requiring that the agent know the parameters m, n, k, Δ beforehand. In subsequent work, [111] kept the time and memory complexity of [72] intact while dropping the requirement of the agent to have prior knowledge of m, k, Δ. Kshemkalyani and Sharma [76] improved the time complexity to $O(\min(m, k\Delta))$. Works of [92] and [35] used randomisation, which helped to reduce the memory requirement for each agent. Recently, there have been two significant advancements in fault-free dispersion problems. Sudo *et*

al. [116] improved the time complexity of dispersion from $O(\min(m, k\Delta))$ to $O(k \log \min\{k, \Delta\})$ in the rooted initial configuration, and to $O(k \log k \log \min\{k, \Delta\})$ in the arbitrary configuration, both using only $O(\log(k + \Delta))$ bits of memory per agent. They also proposed an $O(k)$ (time-optimal) algorithm for the rooted configuration that requires $O(\Delta + \log k)$ bits of memory per agent. Quite recently, Kshemkalyani *et al.* [70] presented an optimal $O(k)$-round algorithm for dispersion in both rooted and arbitrary configurations, while maintaining the memory usage at $O(\log(k + \Delta))$ bits per agent. Furthermore, in the asynchronous setting, they proposed an algorithm with time complexity $O(k \log k)$ and memory complexity $O(\log(k + \Delta))$ per agent, which is time-optimal within an $O(\log k)$ factor.

In [73], Kshemkalyani *et al.* , studied the problem in the *Global Communication Model*, in which the agents can communicate with each other irrespective of their positions in the graph[1]. The authors obtained a time complexity of $O(k\Delta)$ rounds when $O(\log(k + \Delta))$ bits of memory were allowed at each agent. Whereas, when agent were allowed $O(\Delta + \log k))$ bits, the number of rounds reduced to $O(\min(m, k\Delta))$. Both were for an arbitrary initial configuration of agents. They also used BFS traversal techniques for investigating the dispersion problem. The BFS traversal technique yielded a time of $O((D + k)\Delta(D + \Delta))$ rounds with $O(\log D + \Delta \log k)$ bits of memory at each agent, using global communication, for an arbitrary starting configuration of agents. Here, D denotes the diameter of the graph. The problem was also studied on *dynamic* graphs in [74], [2], [87]. *Graph Exploration*, which is a related problem, has also been intensively studied in literature [14] [33] [37] [43]

The dispersion problem has also been recently studied for configurations with a faulty agent. In [90], Molla *et al.* , considered the problem for anonymous rings, tolerating weak Byzantine faults (agents that behave arbitrarily but cannot change their IDs). They gave three algorithms **(i)**, the first one being memory-optimised, requiring $O(\log n)$ bits of memory, $O(n^2)$ rounds and tolerating up to $n - 1$ faults. **(ii)** the second one is time optimized with $O(n)$ rounds, but require $O(n \log n)$ bits of memory, tolerating up-to $n-1$ faults. **(iii)** the third one runs in $O(n)$ rounds and $O(\log n)$ memory but cannot tolerate more than $[\frac{n-4}{17}]$ faulty agents. In [95], the authors proposed several algorithms for dispersion, with some of them tolerating strong Byzantine agents (agents that behave arbitrarily and can tweak their IDs as well). Their algorithms are mainly based on the idea of gathering the agents at a $root$ vertex, using them to construct an isomorphic map of G and finally dispersing them over G according to a specific protocol. However, their algorithms take exponential rounds for a strong Byzantine agent starting from an arbitrary configuration. For the rooted configuration, their algorithm takes $O(n^3)$ rounds but tolerates no more than $[n/4 - 1]$

[1]In the *Local Communication Model*, agents can communicate with each other only when they are at the same node.

strong Byzantine agents.

Dispersion under *crash faults* in arbitrary graphs has been studied by Pattanayak *et al.* in [99] and [100]. In [99], they considered the problem for a team of agents starting from a rooted configuration, where some agents may be crash-prone. Their algorithm tolerates an arbitrary number of crashes, completes in $O(f \cdot \min(m, k\Delta))$ rounds, and uses $O(\log(k + \Delta))$ bits of memory per agent. Later, in [100], they improved the time complexity under different memory settings. The first solves dispersion in $O(\min(m, k\Delta))$ rounds with $O(k \log(k + \Delta))$ bits of memory per agent, while the second achieves $O(\min(m, k\Delta)(\ell + f))$ rounds using $O(\log(k + \Delta))$ bits. Both algorithms handle rooted ($\ell = 1$) and arbitrary configurations, and the agents do not need prior knowledge of any graph parameters. Despite these recent developments, our algorithms remain significant due to their optimal memory usage and better time complexity in the rooted setting, particularly when $\Delta > k$. A detailed comparison with these works is provided in Table 2.1. Apart from arbitrary graphs, fault-tolerant dispersion has also been studied in grid graphs in [16, 15].

Algorithm	Initial Config.	Crash Handling	Time Complexity (Rounds)	Memory
Fault-Free Dispersion				
[116]	Rooted	No	$O(k \log \min\{k, \Delta\})$	$O(\log(k + \Delta))$
[116]	Arbitrary	No	$O(k \log k \log \min\{k, \Delta\})$	$O(\log(k + \Delta))$
[70]	Arbitrary	No	$O(k)$	$O(\log(k + \Delta))$
Crash-Fault Dispersion				
[99]	Rooted	Yes	$O(f \cdot \min(m, k\Delta))$	$O(\log(k + \Delta))$
[100]	Arbitrary	Yes	$O(\min(m, k\Delta))$	$O(k \log(k + \Delta))$
[100]	Arbitrary	Yes	$O(\min(m, k\Delta) \cdot (\ell + f))$	$O(\log(k + \Delta))$
Theorem 2.1	Rooted	Yes	$O(k^2)$	$O(\log(k + \Delta))$
Theorem 2.2	Arbitrary	Yes	$O((f + \ell) \cdot \min(m, k\Delta, k^2))$	$O(\log(k + \Delta))$

Table 2.1: Summary of results for the dispersion of $k \leq n$ agents, including up to $f \leq k$ crash-prone agents, on anonymous graphs with m edges and maximum degree Δ. The parameter ℓ denotes the initial number of agent clusters. Theorem 2.2 assumes the knowledge of m, Δ, k, f and ℓ.

2.2.1 Our Results

We consider a team of $k \leq n$ mobile agents placed on an arbitrary, undirected simple graph, consisting of n anonymous, memory-less nodes and m edges. The ports at each node are labelled. The agents have unique IDs and a restricted amount of memory (measured in number of $bits$). These agents have some computing capability and can communicate with other agents,

only when they are at the same node. We consider two different starting scenarios, based on the initial configuration of the agent. When the agents start from a single node, we call the configuration $rooted$, otherwise, we call it an $arbitrary$ configuration. We further assume that $f \leq k$ faulty agents in the network are prone to crash at any point in time. Our first algorithm for the rooted configuration crucially uses a depth-first search (DFS) traversal and improves the round complexity from $O(f \cdot \min(m, k\Delta))$ [99] rounds to $O(k^2)$ [25]. The second algorithm for the arbitrary configuration is an entirely new result whose complexity depends upon the factors: the number of faulty agents(f), number of agent clusters (ℓ), total number of edges in the graph (m), number of agents (k) and the maximum degree of the graph (Δ). In this case, the round complexity is $O((f+\ell) \cdot \min(m, k\Delta, k^2))$ [25]. The results are summarised in the following two theorems:

Theorem 2.1 (Crash Fault with Rooted Initial Configuration) *Consider any rooted initial configuration of $k \leq n$ mobile agent, out of which $f \leq k$ may crash, positioned on a single node of an arbitrary, anonymous n-node graph G having m edges, in synchronous setting* DISPERSION *can be solved deterministically in $O(k^2)$ rounds with $O(\log(k+\Delta))$ bits memory at each agent, where Δ is the maximum degree of the graph.*

Theorem 2.1 improves over the algorithm in [99] (in the worst case, improvement is from cubic to quadratic) that takes $O(f \cdot \min(m, k\Delta))$ rounds for f faulty agents. The theorem also matches the optimal memory bound ($\Omega(\log(\max(k, \Delta)))$ [72]) with $O(\log(k+\Delta))$ bit memory and can handle any number of crashes.

Theorem 2.2 (Crash Fault with Arbitrary Initial Configuration) *Consider any arbitrary initial configuration of $k \leq n$ mobile agent, out of which $f \leq k$ may crash and positioned on $\ell < k$ nodes of an arbitrary and anonymous n-node graph G having m edges, in synchronous setting* DISPERSION *can be solved deterministically in $O((f+\ell) \cdot min(m, k\Delta, k^2))$ rounds with $O(\log(k+\Delta))$ bits memory at each agent.*

Theorem 2.2 solves the dispersion for an arbitrary configuration with optimal memory per agent. The time complexity matches the one conjectured by Pattanayak *et al.* [99]. When f, ℓ and Δ are constants, the time complexity matches the lower bound of $\Omega(k)$. Moreover, the algorithm can handle any number of faulty agents. The results are summarised in the Table 2.1.

2.3 Model Specifications and Problem Formulation

For this problem, we use the general model as described in Section 1.1.2, with the only difference being in the crash fault assumptions detailed below. We assume that the number of agents, denoted by k, is less than the number of nodes in the graph, which is n. For this work, we assume that the agents start at two different starting configurations.

Crash Fault Model: The agents are not fault-proof, and a faulty agent can $crash$ at any time during the execution of the algorithm. Such $crashes$ are not recoverable, and once an agent $crashes$, it immediately loses all the information stored in itself, as if it were not present at all. Further, a crashed agent is not visible or sensible to other agents. We assume there are f faulty agents such that $f \leq k \leq n$.

Starting Configuration: We assume two different starting configurations for the k agents.

- **Rooted Initial Configuration:** All the k agents start from a designated node called the $root$ node.
- **Arbitrary Initial Configuration:** The agents are placed arbitrarily in small clusters across different nodes of the graph and represent the most generalised starting configurations for the agents.

Given a simple, anonymous, port-labelled, connected graph G with n memory-less nodes and m edges with maximum degree Δ. Consider a team of $k \leq n$ mobile agents residing arbitrarily on the nodes of the graph, with some of the agents being prone to crashes. We want to devise algorithms such that each agent, which eventually remains active (some of the agents can crash during the algorithm), repositions itself to a distinct node of G and remains stationary thereafter, i.e., no two active agents occupy a single node at the conclusion of the algorithm. Below, we formally state the problem of fault-tolerant dispersion.

Definition 2.1 (Fault-Tolerant Dispersion)**.** *Given $k \leq n$ agents, up to f of which are faulty (which may fail by crashing), initially placed arbitrarily on a graph of n nodes, the non-faulty agents, i.e., the agents that are not yet crashed, must re-position themselves autonomously to reach a configuration where each node has at most one (non-faulty) agent on it.*

Before proceeding to the main sections, we provide a table depicting the agent variables used in this chapter, in Table 2.2.

Agent Variables Used in Chapter 2		
Variable	**Values or Type**	**Description**
$r_i.ID$	Unique ID	ID of an agent.
$r_i.parent$	Port / $null$	Port through which r_i entered and settled at its current node.
$r_i.cdr$	Port number	Current direction pointer used to continue DFS traversal.
$r_i.B$	$0/1$	Backtracking status of the settled agent.
$r_i.settled$	$0/1$	Indicates whether the agent has permanently settled.
$r_i.state$	*forward*, *backtrack*	Current traversal state of the moving agent/cluster.
$r_i.cid$	Cluster ID	Stores the identifier of the cluster to which the agent belonged when it settled. The cluster ID is defined as the highest agent ID within that cluster at the beginning of the phase.
$r_i.counter$	Integer	Stores the phase counter value, initialized to $\min(m, k\Delta, k^2)$ at the beginning of each phase.
$r_i.priority$	Cluster ID	Maintains the priority associated with the settled agent's cluster. Initially, it is equal to the cluster ID (cid), but it may be updated if the agent is discovered by another cluster with a higher priority.

Table 2.2: List of agent variables used in this chapter. List of notations on Table 1.2 (Introduction)

2.4 Crash-Fault Dispersion from Rooted Initial Configuration

In this section, we present a deterministic algorithm that disperses the agent from a single-source (rooted configuration) in the crash fault setting. Our goal is to minimise the round complexity as well as keep the memory of the agent low (in bits).

2.4.1 Algorithm

In a crash fault setup, due to crashes, it becomes challenging to explore the graph. Typically, dispersion algorithms rely on the agents themselves to keep track of the paths during exploration. The presence of a crashed agent in this instance may lead to an endless cycle. Therefore, our goal is to ensure the dispersion of the mobile agents despite the presence of faulty agents.

In the rooted configuration, to manage the presence of faults, we avoid exploring the graph together with all the agents. That is, the graph is explored sequentially such that each agent r_i ($1 \leq i \leq k$) does not begin exploring the graph until the previous agent r_{i-1} is guaranteed to

have settled. During exploration, whenever an agent r_i finds an empty node, it settles down at that spot. Let us call this algorithm Rooted-Crash-Fault-Dispersion. Below, we explain the algorithm in detail.

Functionality: For simplicity, let us assume that the agent's ID lies in the range of $[1,\ k]$. Otherwise, the agent can map their IDs from the actual range to the range $[1,\ k]$, since the IDs are distinct. We denote the rooted configuration by R_c. We slightly abuse notation and use R_c to indicate both the $root$ and the initial gathering of the agents. An agent at R_c traverses the graph via DFS (Depth First Search) approach, where the decision of which edge to traverse first is based on the port numbers. The process proceeds in increasing order of IDs, starting with the agent with the minimum ID at R_c. R_c then sends each agent to explore the graph via DFS.

Let the agent with the current minimum ID at R_c be r_i. Then r_i begins to explore the graph via DFS (starting with the minimum port number at R_c). Once it leaves R_c, it has $3i$ rounds within which it can either i) settle at the first empty node it finds or ii) return to R_c if it does not find an empty node to settle within $2i$ rounds. More specifically, r_i has at most i rounds to traverse till the point traversed by the previous agent r_{i-1} (let us say to the node v_{i-1}); then i rounds to explore new edges of G and settle if a new edge leads to an empty node or otherwise return to v_{i-1} (specifically, r_i uses $i/2$ rounds for exploring new edges from v_{i-1} and $i/2$ rounds to return to v_{i-1}, in case, no new empty node is found); and another i rounds to go back to R_c for reporting, if it does not get to settle. If r_i reports to R_c within $3i$ rounds, then R_c ensures that it does not release the agent with the next lowest ID, say r_{i+1}. r_i needs to traverse at most $(i-1)$ edges to explore the path traversed by r_{i-1} and similarly will require at most i rounds to return to the base R_c. As r_i requires i rounds to report at the R_c, therefore, r_i explores the graph only for the first $2i$ rounds. Notice that an agent will not traverse a distance of more than $(i+1)$, before that, there will be an empty edge at a distance (distance from the root) of $(i+1)$ and the agent will settle down there. If r_i did not find the empty node within $2i$ rounds, then it starts to traverse towards R_c. In this way, r_i reports to R_c within $3i$ rounds so that R_c does not send another agent to explore the graph. If r_i reports back to R_c, it re-sends r_i to explore the graph once again, this time allocating it a $3(i+1)$ round window. In this way, any r_i traverses the graph until it finds an empty node. Note that in our process, we ensure that there are no two agents that are exploring the graph at the same time.

To maintain the protocol, each r_i maintains the following fields. Its ID (r_i), a parent pointer $(r_i.parent)$ that represents the edge it traversed, a current direction pointer $(r_i.cdr)$ which indicates the direction it is required to follow. And finally, a backward traversal value $(r_i.B)$ which

is initially 0, and is set to 1 once the backward traversal is complete. Here, our procedure performs the traditional DFS protocol, but one-by-one, that is, the agent does not explore the graph simultaneously. In the following subsection, we give a detailed procedure for the DFS procedure.

2.4.1.1 DFS Traversal Procedure

Let the agent positioned initially at $root$ R_c be denoted by $R_c = \{r_1, r_2, \ldots, r_k\}$, where r_i is agent with ID i. For the *rooted configuration*, each agent stores the following four variables :

1. $r_i.parent$: the port number through which the agent r_i has entered a new empty node and has settled. Initially, it is assigned to $null$.
2. $r_i.cdr$: the current direction of a settled agent r_i. A settled agent sets cdr as the minimum available port number; however, as the algorithm progresses and more nodes are explored, the agent i can change its cdr value, if needed.
3. $r_i.B$: a binary variable denoting backtrack status, initially assigned to 0, takes the value 1, if and only if every sub-graph accessible through each of r_i's child edges has been explored, and now, no new part of G could be explored through the node containing r_i.
4. $r_i.settled$: a binary variable, initially assigned 0, takes the value 1, if and only if, r_i has settled at a particular node of G

Update procedure: In the first round, the agent r_1 assigns $r_1.settled \leftarrow 1$ and sets $r_1.cdr \leftarrow 1$, the minimum port number available at R_c. In the next round, the agent r_2 at R_c reads and exits through $r_1.cdr$ to land at a new empty node (say w) and consequently sets $r_2.settled \leftarrow 1$. Assume that r_2 arrived at w through port p_w. r_2 writes $r_2.parent \leftarrow p_w$ and if $r_2.cdr \leq deg(w)$, $r_2.cdr \leftarrow r_2.cdr + 1$, if port $r_2.cdr + 1 \neq p_w$, else $r_2.cdr \leftarrow r_2.cdr + 2$. In the third phase, the incoming agent r_3, following the cdr pointers to w, can now decide to march forward or backtrack based on the following conditions:

- **forward** : if ($p_w = r_2.parent$ or $p_w =$ old value of $r_2.cdr$) and (there is at least one port at w that has not been taken yet). The agent r_3 exit w through port $r_2.cdr$
- **backtrack** : if ($p_w = r_2.parent$ or $p_w =$ old value of $r_2.cdr$) and (all the ports of w have been taken already). Then, the agent r_3 exits w through port $r_2.parent$. In such case, r_3 also sets the backtrack value of $r_2.B \leftarrow 1$

There is another condition, denoting the onset of a cycle, under which choosing the backtrack phase is in order. When an agent r_j enters x through p_x and agent r is settled at x,

- **backtrack** : if ($p_x \neq r.parent$ and $p_x \neq$ old value of $r.cdr$). The agent r_j exits x through port p_x, and no variables of r are altered.

Each agent explores the graph one by one and eventually settles. In the i^{th} phase, an agent r_i has $3 \cdot i$ number of rounds, in which, it can either settle down or explore $i/2$ new edges of G (it takes i rounds for r_i to come to r_{i-1}, takes additional i rounds to explore $i/2$ new edges and return to r_{i-1} and further i rounds to return at R_c, following the $parent$ pointers).

Now, coming back to the main algorithm, we explain the **Decision** part.

Decision: If r_i encounters an unexpected child, r_u, i.e., a child whose parent and current pointer direction are set in the inappropriate direction w.r.t. the perspective of r_i, it realises that r_u has replaced an agent that has previously crashed. In such a situation, r_i changes (resets) the pointer variables of r_u appropriately (see, Figure 2-1 and Algorithm 1) and the algorithm continues.

In this way, the algorithm continues till all the agents from R_c have settled one by one into the nodes of G. A pseudo-code for the algorithm is provided in Algorithm 1.

Lemma 2.1. *In the non-faulty setup, round complexity is* $O(k^2)$.

Proof. In a non-faulty setup, each agent behaves robustly, and there are no crashes. Therefore, after the backtracking flag is set on a node, an edge is not traversed again during the DFS traversal. In traversing a graph from R_c, two kinds of situations may arise: either an agent r_i reaches an empty node after $O(i)$ edge traversals, or it traverses $O(i^2)$ edges. In the first case, there is an empty node at a distance of $O(i)$. Therefore, r_i settles at the empty node after $O(i)$ rounds. If such a situation arises repeatedly, then the algorithm takes $O(1)+O(2)+\cdots+O(k) = O(k^2)$ rounds. In the second case, there might be a situation such that r_i traverses $O(i^2)$ edges to find the empty node and only encounters previously settled nodes (at most $i(i-1)/2$ edges). More precisely, $i/2$ new edges are traversed in $3i$ rounds. Notice that an agent will traverse only earlier traversed nodes at a distance $(i+1)$, if not, then there will be an empty edge at a distance (distance from the root) of $(i+1)$ and the agent will settle down there. Therefore, r_i covers $O(i^2)$ edges in $O(i^2)$ rounds and future agent (i.e., agent having ID r_j; $\forall\ j > i$) will not traverse these edges again. Hence, we can conclude that the non-faulty setup takes $O(k^2)$ rounds in the given model. □

Lemma 2.2. *In the faulty setting, a crashed agent may bring about an extra cost of* $O(k)$ *rounds in comparison to the non-faulty setting.*

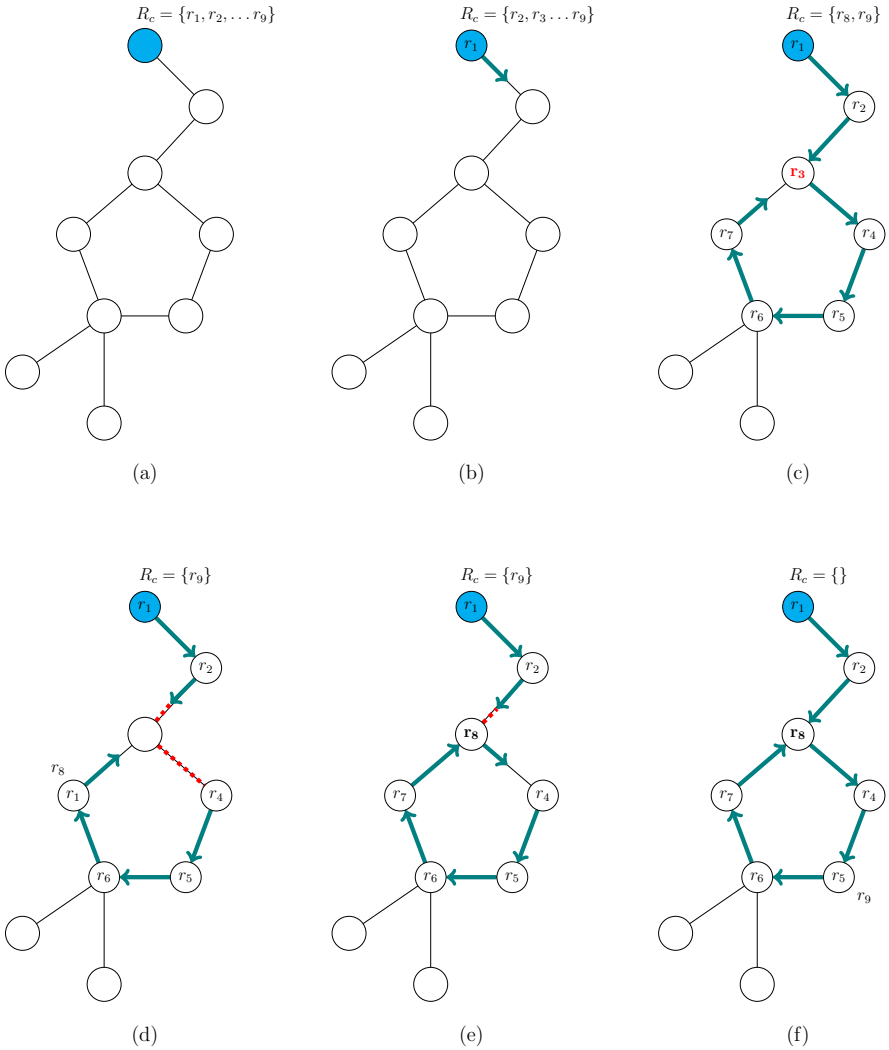


Figure 2-1: All agents are initially positioned in a rooted configuration at the root node R_c (Fig. (a)). The algorithm begins with the agent having the minimum ID, r_1, settling at R_c and assigning the cdr pointer, represented by the directed teal edge in Fig. (b). As the algorithm progresses, using these cdr pointers, the remaining agents establish their positions accordingly, where a faulty agent r_3 (marked in red) has also settled (Fig. (c)). In Fig. (d), as exploration continues, r_3 crashes while r_8 is still in search of an empty node. This failure results in r_3 losing all its pointers and stored information (depicted by the dotted red edges representing lost connections). Consequently, r_8 settles at the node previously occupied by r_3. Although its cdr pointer remains valid, its parent pointer is misdirected. Finally, in Fig. (f), agent r_9 corrects these parent pointers as it proceeds with its exploration in search of an unoccupied node.

Algorithm 1 ROOTED-CRASH-FAULT-DISPERSION

Require: An anonymous network of n nodes where f agent are faulty such that $f \leq k \leq n$. k is the number of agents.
Ensure: Agents Dispersion.

1: Each agent r_i maintains its ID, parent pointer, current direction pointer and the backward traversal pointer as $\langle r_i, r_i.parent, r_i.cdr, r_i.B \rangle$, respectively. Initially, $r_i.cdr$ is the same, i.e., the minimum port number, $r_i.B = 0$ for all the k agents at the rooted configuration. $r_i.B = 1$ indicates that the backtrack is done for that settled agent, while $r_i.B = 0$ represents backtracking is remaining.
2: **for** $7k^2$ rounds **do** ▷ As stated in Lemma 2.5
3: The graph is traversed from (R_c) via DFS (Depth First Search) by sending the current minimum ID agent (r_i) based on the current direction pointer at R_c after every $3i$ round. If agent r_i reports to R_c within $3i$ rounds, then R_c resends r_i until there is no report regarding r_i. ▷ DFS described in Section 2.4.1.1
4: Each r_i settles down at the first empty node it finds and sets all the attributes accordingly. ▷ See functionality for more detail.
5: **if** r_i does not find an empty node in $2i$ rounds **then**
6: r_i proceed to reports R_c. ▷ r_i reports within $3i$ rounds.
7: **else**
8: r_i settles at the empty node.
9: **end if**
10: **if** r_i encounters an unexpected parent for r_u $(u < i)$ **then** ▷ See in decision.
11: r_i resets the parent pointer, $r_u.parent$ and current direction pointer, $r_u.cdr$ based on minimum port available.
12: **end if**
13: **end for**
14: All the non-faulty agents settle at a unique node.

Proof. When an agent crashes, it loses all its internal information and its ability to interact with others, effectively disappearing from the network. A crashed agent is no longer recognised by any other agent in the system. A crash may occur in one of the following three scenarios:

Case (i): **Before being dispatched from R_c:** An agent may crash while still waiting at the root R_c, before beginning its exploration.

Case (ii): **During exploration:** An agent may crash while actively exploring the graph within its allocated number of rounds.

Case (iii): **After settling:** An agent may crash after it has successfully settled at a node.

We analyse each of these cases separately in terms of their impact on the overall round complexity:

Case (i): If an agent crashes while at R_c, its absence is immediately detectable by the other agent waiting at R_c. In such a situation, the next agent in order (i.e., the agent with the next smallest ID) simply begins its execution when its scheduled round window starts. Since no exploration had begun, this crash does not cause any deviation from the original schedule. Thus, there is no extra round cost incurred in this case.

Case (ii): Suppose the ith agent crashes during its exploration. Recall that each agent is allotted a $3j$-round window (where $j \geq i$ is the number of times an agent (new or exploratory) has started its exploration from R_c). If the agent does not return to R_c within its allocated time window, the system infers that it has crashed. Consequently, the next agent in line (i.e., r_{i+1}) begins its exploration at the start of the $3(j+1)$ round window. Therefore, the crash is detected within the existing round budget, and no additional rounds are wasted beyond what was already allocated. Hence, no extra round complexity is incurred in this case either.

Case (iii): Now consider the case where an agent r_i successfully settles at a node v_i, but subsequently crashes. As a result, the node v_i becomes vacant, and all local information maintained by r_i—such as direction pointers, parent references, and backtracking status—is lost. This loss poses a challenge for subsequent exploration. Depending on the timing of the crash, when the next agent r_{i+1} begins its DFS traversal, it may either reach v_i via the now-stale cdr pointer pointing to the empty node, or it may encounter v_i while backtracking to R_c. In either case, r_{i+1} settles at v_i, potentially with incorrectly set $parent$ and cdr pointers. Since the DFS traversal relies on port numbers and backtracking is coordinated through the $parent$ and cdr pointers, the corruption or absence of this information causes inconsistencies in the traversal structure. As a result, the next agent in the sequence, r_{i+2}, may be forced to re-explore parts of the graph that were previously discovered by r_i. In the worst-case scenario, r_i might have explored up to $i-2$ distinct edges before settling. Consequently, r_{i+2} would need to retrace and rediscover these portions of the graph and correct these pointers on r_{i+1}; incurring an additional cost of $O(i)$ rounds.

Since i can be as large as k, the total additional cost due to such a fault can be at most $O(k)$ rounds. □

Lemma 2.3. *There is (at most) one agent moving (neither settled at its respective node, nor at the rooted configuration R_c) at any instance.*

Proof. Proof by contradiction, let us suppose there exist two agents in moving condition, say r_i and r_{i+1}. Also, assume r_i started before, r_{i+1}. Now, as r_i has not settled, r_i reports to R_c every $3i$

rounds (Line 5 of algorithm Rooted-Crash-Fault-Dispersion). But if r_i reports every $3i$ rounds, then R_c does not release the next agent, which is contradictory to our assumption. □

Lemma 2.4. *A loop or cycle may be formed by the current direction pointer ($cdr\ pointer$). The algorithm* Rooted-Crash-Fault-Dispersion *successfully avoids any loop during dispersion.*

Proof. During the execution of the algorithm, a loop or cycle may be formed if an agent r_i crashes at a node v_i, then the current direction pointer ($cdr\ pointer$) is set by the upcoming agent r_{i+1} with the lowest port and that lowest port might have been traversed earlier. Therefore, a loop is formed (as shown in Figure 2-1, see A and B). From Lemma 2.3, we know that only one agent is moving at any instance, say r_{i+1}. Therefore, r_{i+2} (the next agent) starts after r_{i+1} settles. If r_{i+2} encounters any agent with an unexpected $cdr\ pointer$ then r_{i+1} changes the $cdr\ pointer$ appropriately (Line 10 of the algorithm Rooted-Crash-Fault-Dispersion and the Figure 2-1, see C). Thus, loops are avoided in the network. □

Lemma 2.5. *The algorithm* Rooted-Crash-Fault-Dispersion *takes at most $7k^2$ rounds and $O(\log(k+\Delta))$ bits memory.*

Proof. In case of round complexity, a non-faulty set-up from Lemma 2.1, the total number of rounds is $3(1+2+\cdots+k) < 3k^2$ (in the best case where r_i finds the empty node within $3i$ rounds). Additionally, an agent can traverse at most $i/2$ new edges in $3i$ rounds (in a particular phase) without settling down on an empty node (in the worst case). Therefore, round complexity for $k(k-1)/2$ edges in the non-faulty setup is $< 3k^2$. Moreover, from Lemma 2.2, we know that the extra cost incurred for the f agents crashing is at most fk. Hence, overall round complexity is at most $3k^2 + 3k^2 + k^2 = 7k^2$.

In case of memory complexity, each agent stores its ID, which takes $O(\log k)$ bit space. Along with that, the parent pointer and current direction pointer take $O(\log \Delta)$ bits of memory each. While the backward pointer takes a single bit. Therefore, the memory complexity is $O(\log(k+\Delta))$. □

From the above discussion, we conclude the following result.

Theorem 2.1. *Consider any rooted initial configuration of $k \leq n$ mobile agent, out of which $f \leq k$ may crash, positioned on a single node of an arbitrary, anonymous n-node graph G having m edges, in synchronous setting* Dispersion *can be solved deterministically in $O(k^2)$ time with $O(\log(k+\Delta))$ bits memory at each agent, where Δ is the maximum degree of the graph.*

The rooted crash-tolerant dispersion algorithm completes in $O(k^2)$ rounds using $O(\log(k + \Delta))$ bits of memory per agent. This improves the earlier crash-tolerant bound of $O(f \cdot \min(m, k\Delta))$ in the worst case, where the dependence on the number of faulty agents may make the running time much larger.

It is also useful to compare this bound with the best-known results for fault-free dispersion. In the synchronous fault-free setting, dispersion can be solved in optimal $O(k)$ rounds using $O(\log(k + \Delta))$ bits of memory per agent [70]. Before this optimal result, Sudo et al. [116] gave an $O(k \log \min\{k, \Delta\})$-round algorithm with $O(\log(k + \Delta))$ bits of memory, and also an $O(k)$-round rooted algorithm using larger $O(\Delta + \log k)$ bits of memory. These results show that, without faults, the rooted dispersion problem admits near-linear or linear-time solutions.

In contrast, the present chapter studies the crash-fault setting. Here, a faulty agent may crash during the execution, lose all its stored information, and break the consistency of the DFS-based navigation structure. The $O(k^2)$ term in our algorithm mainly comes from the sequential release of agents from the root and the possible need to repair inconsistencies caused by crashes. Therefore, the fault-free $O(k)$ bound is not directly comparable with our crash-tolerant setting, but it does suggest that the gap between $\Omega(k)$ and $O(k^2)$ under crash faults. The $O(k^2)$ bound may not be optimal; however, improving it would likely require a more parallel exploration strategy or a faster method for detecting and repairing the effects of crashed agents in anonymous, memoryless graphs with only local communication.

2.5 Crash-Fault Dispersion for Arbitrary Initial Configuration

In this configuration setting, the agents are distributed across the graph in clusters such that there are $C = \{C_1, \ldots, C_\ell\}$ groups of agents at ℓ different nodes at the start, such that $\sum_i C_i = k$. The goal of the dispersion is to ensure that the agents are dispersed among the graph nodes such that each node has at most one agent. In this setting, we assume that the agents are aware of k, f, Δ, ℓ and m.

2.5.1 High-Level Idea:

Our protocol runs in phases, in which each phase consists of $\min(m, k\Delta, k^2)$ rounds. At the start of each phase, each cluster begins a $counter$ that counts down from $\min(m, k\Delta, k^2)$. Each

cluster C_i then begins exploring the network simultaneously via the traditional DFS algorithm (in the trivial case of a singleton cluster consisting of only one agent, it considers itself dispersed). Unlike the rooted configuration, individual agents do not explore and return; the entire cluster moves together. Whenever a cluster encounters a new (empty) node in the network, the agent with the current least ID in the cluster settles and sets its pointers appropriately. At the end of each round, the counter is decreased by 1. When the counter becomes zero, it signals the end of the phase, and all flags are reset, i.e., all pointers become null, including the pointers of already settled agents. After that, each cluster starts exploring the network with its current node as a point of origin. This continues until all agents in the cluster settle. Details of this procedure can be found in the Algorithm 2.

2.5.2 Details:

There are two main parts to the protocol, i) exploration, ii) encounter. Exploration deals with the general procedure involved in exploring the graph. While encounter deals with the details involved in agents from different clusters meeting.

Let's begin with all the information stored at an agent. Each agent r in a cluster C_i consists of the following pointers cid, $parent$, cdr, $priority$, $counter$ and B (backtrack). When an agent decides to settle at a node, the $parent$ pointer keeps track of the port through which it entered the node. Similarly, the cdr pointer is used to keep track of the port through which a cluster leaves the node in which it is settled. And of course, the B of the backtrack pointer keeps track of the backtrack status of its DFS. Now, we explain below the three additional variables used by the agent in this case.

(i) cid: cid denotes the ID of the cluster it belonged to when an agent settles. cid of a cluster C_i is determined at the start of the phase, and is the ID of the agent with the highest ID.

(ii) $counter$: $counter$ is set to $\min(m, k\Delta, k^2)$ at the beginning of each phase. Note that since all agents set the counter at the beginning of the phase simultaneously, the counter has the same value across all agents.

(iii) $priority$: The $priority$ pointer of a settled agent keeps track of its priority in various clusters. Originally, this is simply the cid of the cluster it was part of; that is, the priority of a cluster is simply its cid. However, an agent's priority may change if a higher-priority cluster discovers it and updates its priority pointer. In our work, priority is decided by the cluster's ID; that is, between two clusters, the cluster with the higher cid has higher precedence.

2.5.2.1 Exploration:

As mentioned before, as long as a cluster is non-empty, at the beginning of each phase, each cluster begins exploring the graph via the traditional DFS until the cluster is empty or it encounters an agent from a higher priority cluster (more on this in the encounter section). In each phase, each agent in any cluster C_i sets its cid and $priority$ to the highest ID in the cluster, and its counter to $\min(m, k\Delta, k^2)$. We consider the node in which C_i is at the start of the phase to be its root. C_i then follows the traditional DFS format for exploration. It leaves the node via the smallest unexplored port. If the node is empty, the agent with the current minimum ID, say r, sets its *parent* and cdr pointers and settles at the node.

The update function for the cdr pointer is exactly the same as the one in the rooted case, i.e., it follows the traditional DFS procedure, except that all the agents in the cluster move through the exit port. Here, the agents use a similar DFS traversal technique as explained in DFS Procedure 2.4.1.1. In the arbitrary case, the whole group moves together except for the agents that have settled. Whenever an exit port is calculated for moving, every (unsettled) agent in the group moves out through the port into the neighbouring node.

All agents in C_i decrease their counter by one, and C_i leaves through the port in $r.cdr$. If a cluster ever finds itself returning to a node with an agent r from its own cluster and it has exhausted all of the ports in which r has settled, then it sets r's backtrack flag. Once a phase has finished, if the cluster is non-empty, it resets all flags and counters and begins DFS once again. During exploration, if the cluster C_i reaches a node u whose degree is k, then they use BFS to explore the neighbourhood of u and settle the agent of C_i in at most $O(k)$ rounds. However, here we have not explored what happens if an agent from a cluster C_i meets an agent from C_j. That brings us to the next important part of the protocol, the encounters. Details of the exploration part of a cluster can be found in Algorithm 4.

2.5.2.2 Encounter

This section explains the *encounter* mechanism used during the dispersion process (see Algorithm 3). An encounter occurs when a cluster meets a settled agent from a different cluster or a different cluster. In such situations, the next step in the exploration is determined based on a predefined priority rule. We now describe the details of priority assignment and how it is updated during encounters.

Priority Upgradation: Initially, each agent's $priority$ is set to the cid (cluster ID) of its cluster, which is defined as the ID of the highest-ID agent in that cluster. Consider a cluster C_i where some agents have already settled, while the remaining are still exploring the graph. During this exploration, the following three scenarios may arise (see Fig. 2-2):

Case (i): The cluster encounters a settled agent from another cluster with a lower priority.

Case (ii): The cluster encounters a settled agent from another cluster with a higher or equal priority.

Case (iii): The cluster encounters another cluster that is actively exploring.

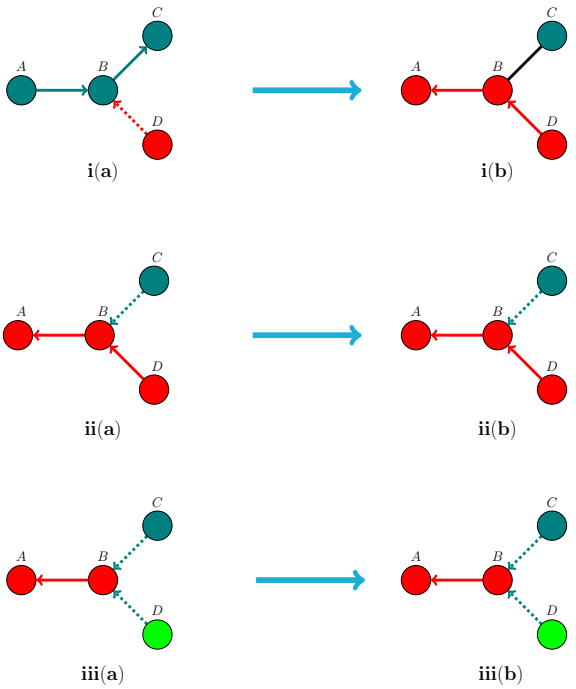


Figure 2-2: **Interaction between different types of clusters**. In Fig i(a), a high-priority cluster (in red) encounters a lower-priority agent (in teal) at node B. In such a case, the higher priority cluster immediately takes over the lower priority agent (Fig i(b)). Conversely, if a lower priority cluster encounters a higher priority agent (Fig ii(a)), the lower priority cluster halts and waits at node B until its priority succeeds the priority of the agent it has encountered at node B (Fig ii(b)). Similarly, if two clusters of lower priority (depicted in teal and green) meet a higher priority agent at node B (Fig iii(a)), they merge and wait until its priority becomes higher than that of the agent in B (Fig iii(b)).

Let C_i be the exploring cluster with priority p_i, and suppose it encounters a settled agent r with priority p_j at node v. If $p_i > p_j$ (case (i)), the higher-priority cluster C_i takes over and treats r as if it were an unsettled agent from its own cluster. It performs the following actions:

- Resets all of r's variables,
- Sets $r.priority \leftarrow p_i$,
- Assigns $r.parent$ and $r.cdr$ as it would when settling a new agent at an empty node.

This effectively assimilates the encountered agent into the dispersion process of C_i.

If $p_i \leq p_j$ (case (ii)), then the lower-priority cluster C_i halts its exploration for the remainder of the current phase. Each agent in the cluster begins a local countdown and resumes exploration at the start of the next phase.

In case (iii), where two or more exploring clusters meet at node v, they merge and adopt the highest priority among them. Each agent r_i in the merged cluster updates its priority as $r_i.priority \leftarrow \max\{r.priority : r \in v\}$, waits for the current phase countdown to complete, and resumes exploration as part of the new merged cluster in the next phase.

At the end of every phase, agent update their priorities as follows:

- A singleton agent (i.e., one that is alone at a node) resets all of its variables, considers itself to have $settled$, and sets its $priority \leftarrow null$. Unless it is encountered by another cluster, the agent remains at its location, and these values do not change.
- an agent r_i that finds itself at a node v with other agents (i.e., part of a cluster) sets its cluster ID as $r_i.cid \leftarrow \max\{r.ID : r \in v\}$ and updates its priority accordingly as $r_i.priority \leftarrow r_i.cid$. Any previously settled agent at v is also reset, and its variables are re-initialised.

The resetting of variables at the end of a phase does not interfere with the correctness of the dispersion protocol. Each phase is sufficiently long—requiring $O(m, k\Delta, k^2)$ rounds—allowing even the largest cluster (up to k agents) to complete dispersion. Thus, even if an agent's internal variables—specifically its $parent$, cdr, or backtracking status (B)—are reset at the phase boundary, the cluster still has sufficient rounds within the same phase to recompute these values from scratch and complete its dispersion correctly. While some redundant exploration may occur due to the resetting of internal variables, all such activity is confined within the $O(m, k\Delta, k^2)$ round window. Therefore, it does not affect the asymptotic complexity of the algorithm.

Importantly, the number of clusters is non-increasing from one phase to the next. A cluster may (i) successfully complete dispersion, (ii) survive to continue exploration in the next phase, or (iii) merge with a higher-priority cluster. In all cases, the number of clusters either remains unchanged or decreases. Intuitively, allowing $(\ell + f)$ phases—where ℓ is the initial number of clusters and f is the number of crash-faults - ensures that each cluster gets an opportunity to complete dispersion in a crash-free environment. We show that by the end of $(\ell + f)$ phases, dispersion is guaranteed.

Algorithm 2 ARBITRARY-CRASH-FAULT-DISPERSION

Require: An anonymous network of n nodes where f agent are faulty such that $f \leq k \leq n$. k is the number of agents. The agent are distributed across the network in $C = \{C_1, C_2, \ldots, C_\ell\}$ clusters such that $\sum_i |C_i| \leq k$. The value of f, ℓ, m, k and Δ are known to the agent.

Ensure: agent' Dispersion.

Each agent r belonging to a cluster C_i maintains the following pointers. $r.cid$, $r.priority$, $r.parent$, $r.cdr$ and $r.counter$. Initially $r.counter$ is set to min$(m, k\Delta, k^2)$,where Δ is the maximum degree of the network.

$j = 0$.

1: **while** $j \leq (\ell + f)$ **do**
2: $counter = \min(m, k\Delta, k^2)$
3: **while** $counter > 0$ **do**
4: All non-empty clusters C_i perform Explore(C_i).
5: $counter = counter - 1$.
6: **end while**
7: Reset all pointers to null. Set $r.counter = \min(m, k\Delta, k^2)$ across all r.
8: $j = j + 1$.
9: **end while**

Lemma 2.6. *The effects of an agent crash, that is, the time delay caused by the presence of a crash, are limited to the phase in which it occurs. After that, it ceases to have an effect.*

Proof. Since at the end of every phase, all agent reset their flags, including the parent and cdr pointers, previously explored paths are equivalent to new unexplored paths in the current phase, as their pointers are set by the currently exploring clusters. Hence, previous phases do not have any impact on the DFS running in the current phase. □

Lemma 2.7. *Let C_i be the cluster with the highest priority in Phase j. C_i is guaranteed dispersion by the end of j if j is fault-free.*

Proof. From Lemma 2.6 we know that crashes in previous rounds do not have an effect on exploration in the current phase. And, in the absence of faults during the phase itself, we see that

Algorithm 3 ENCOUNTER(C_i)

Require: A non-empty cluster of agent C_i

if node is not empty and contains an agent r_p **then**
  if $r_p.cid = C_i.cid$ **then**
    if $r_p.B$ is set **then**
      Return through $r.parent$.
    else
      Continue to explore. Update $r_p.cdr$ to the minimum unexplored port. Move through $r_p.cdr$. ▷ See Section 2.4.1.1 for detailed procedure.
    end if
  end if
  if $r_p.cid \neq C_i.cid$ **then**
    if r_p has higher priority **then**
      Wait for $counter$ to become zero.
    else
      $r_p.cid = C_i.cid$.
      $r_p.priority = C_i.priority$.
      $r_p.parent =$ port through which C_i entered.
      Continue to explore. Update $r_p.cdr$ to the minimum unexplored port. Move through $r_p.cdr$. ▷ See Section 2.4.1.1 for detailed procedure.
    end if
  end if
  if $r_p.cid = null$ **then** ▷ r_p's flags have been reset.
    $r_p.cid = C_i.cid$.
    $r_p.priority = C_i.priority$
    $r_p.parent =$ port through which C_i entered.
    $r_p.cdr$ is set to the minimum unexplored port, and the cluster then moves through $r_p.cdr$. ▷ See Section 2.4.1.1 for detailed procedure.
  end if
end if
if node is not empty and contains clusters $C_s \subset C$ that are not C_i **then**
  if C_i has highest priority **then**
    Explore (C_i).
  else
    Let C_j be the cluster on the node with the highest priority.
    C_i merges with all remaining clusters in C_s/C_j. It takes the priority of the highest priority cluster in C_s/C_j.
    The clusters wait until the start of the next phase to begin exploration.
  end if
end if

Algorithm 4 EXPLORE(C_i)

Require: A non-empty cluster of agent C_i
1: **if** node is empty **then**
2: Settle agent with current lowest ID in C_i (say r).
3: $r.priority \leftarrow r.cid$.
4: $r.counter \leftarrow counter$.
5: $r.parent \leftarrow$ port through which r entered the node.
6: $r.cdr$ is the minimum port that hasn't been explored so far.
7: **end if**
8: **if** node is not empty **then**
9: Perform Encounter(C_i).
10: **end if**

C_i exploration is equivalent to a rooted single cluster exploration of the network presented in Section 2.4.1.1. Thus, it is able to complete its dispersion using DFS without any delays or interference from other clusters, which takes less than $O(\min(m, k\Delta, k^2))$ rounds to complete. □

Lemma 2.8. *Each cluster $C_i \in C$ is guaranteed to have at least a single fault-free phase in which it has the highest priority.*

Proof. Quite trivially, since there are $(\ell + f)$ phases, each cluster is guaranteed at least one phase in which no faults occur, and in which they are the highest priority. □

Lemma 2.9. *At the end of $(\ell + f)$ phases, all clusters are guaranteed to have dispersed.*

Proof. This follows directly from Lemmas 2.7 and 2.8. Each cluster is guaranteed to have at least one fault-free phase in which it has the highest priority. From 2.6 we know in that phase there is guaranteed dispersion. Hence, in $(\ell + f)$ phases, we are guaranteed to have total dispersion of all clusters. □

Thus, we have the following theorem.

Theorem 2.2. *In the synchronous setting, the crash-tolerant algorithm for the arbitrary configuration (algorithm* ARBITRARY-CRASH-FAULT-DISPERSION*) ensures dispersion of mobile agents in an arbitrary graph from an arbitrary initial configuration in $O((f + \ell) \cdot min(m, k\Delta, k^2))$ rounds with each agent requiring $O(\log(k + \Delta))$ bits of memory.*

Since the number of clusters ℓ and the number of faulty agents f are both less than the total number of agents k, we have the following remark.

Remark: 1. *If only the number of agents (k) is known and all other factors are unknown to the network, then the algorithm for arbitrary configuration takes $O(k^3)$ rounds.*

2.6 Conclusion

In this work, we studied the problem of *dispersion* for distinguishable mobile agents operating on anonymous, port-labelled, arbitrary graphs under the presence of crash faults. We presented a deterministic algorithm that solves the dispersion problem in two different initial configurations: (i) the *rooted* setting, where all agents start at a designated root node, and (ii) the *arbitrary* setting, where agents can begin at any location in the graph. Our algorithm achieves a round complexity of $O(k^2)$ in the rooted case and $O((f+\ell)\min(m, k\Delta, k^2))$ in the arbitrary case, where k is the number of agents, f is the number of faulty agents($f \leq k$), ℓ is the number of initially non-isolated agent (i.e., agent in a group of size two or more), m is the number of edges, and Δ is the maximum degree of the graph. In both settings, each agent uses only $O(\log(k+\Delta))$ bits of memory.

Several directions emerge for future research. A natural next step is to improve the upper bound or investigate lower bounds for the dispersion problem, particularly aiming to design a time-optimal algorithm that uses only $O(\log(k+\Delta))$ bits of memory per agent, for both rooted and arbitrary initial configurations. Another significant extension is to explore more severe fault models, such as the *Byzantine* fault model, where faulty agents may behave arbitrarily under the control of an adversary. This model can be further classified into the *weak Byzantine* setting, where agents cannot forge identifiers, and the *strong Byzantine* setting, where identifiers can also be forged. Additionally, it would be valuable to study dispersion in the *global communication model*, where agents can exchange information without needing to be co-located. Finally, extending this work to *heterogeneous agent systems*—where agents differ in capabilities, mobility, or communication protocols—would make the model more representative of real-world applications such as search and rescue, surveillance, or multi-platform robotic operations. For instance, in a search-and-rescue scenario, aerial and ground agents may be equipped with different sensors and actuation capabilities, and understanding the challenges of autonomous coordination in such systems would be an intriguing research direction.

2.7 Dispersion as a Primitive for Later Chapters

In this chapter, we studied dispersion as an independent problem in the presence of crash faults. However, dispersion also plays an important role throughout the rest of this thesis. Several later chapters begin from a dispersed configuration, which provides an organised starting point before performing more involved graph computations. Once the agents are dispersed, each node

is occupied by an agent that can act as its local representative. Information that would normally be stored at a node in a classical distributed algorithm can instead be maintained by the corresponding agent. Similarly, communication between neighbouring nodes can be realised through agent movement, local meetings, and information exchange. This enables graph computations to be carried out in a manner similar to message-passing systems while preserving the restrictions of the mobile-agent model.

At the same time, a dispersed starting configuration does not directly simulate the message-passing model. The agents remain mobile, the nodes are anonymous and memoryless, and communication is possible only when agents meet at the same node. As a result, even after dispersion is achieved, the algorithms developed in later chapters require additional coordination mechanisms that are specific to the mobile-agent setting. To the best of our knowledge, several of the problems studied later in this thesis are investigated for the first time in the mobile-agent model. These include dominating set computation, triangle and butterfly counting, truss decomposition, BFS tree construction, and MST construction. Although these problems have been extensively studied in classical graph algorithms and distributed message-passing systems, solving them using locally communicating mobile agents requires new algorithmic techniques.

In many of these settings, starting from a dispersed configuration simplifies both the design and analysis of the algorithms. Moreover, the cost of obtaining dispersion is often asymptotically smaller than the cost of the graph computation that follows, and therefore does not dominate the overall complexity. The role of dispersion also differs across chapters. In some cases, it is necessary because computations must be performed locally at many nodes, while in others, it mainly serves as a convenient design choice that simplifies coordination.

Alternative approaches without full dispersion may also be possible. One natural direction is to study settings where the number of agents is smaller than the number of nodes, allowing an agent to periodically visit and serve multiple nodes over time. Such oscillating agents may emulate several local representatives, although this would require additional coordination and would likely increase the complexity. We briefly discuss this direction in Chapter 6. Thus, in this thesis, dispersion serves both as an independent coordination problem and as a useful starting point for more advanced graph computations using mobile agents.

Chapter 3

Algorithms for Finding Dominating Set using Mobile Agents

In this work, we study the problem of finding small dominating sets of a graph G using mobile agents. The dominating set problem is fundamental in the field of mobile agents as it opens up a way for solving various coordination and coverage problems, e.g., guarding, covering, facility location, transport routing, etc. In this work, we first present two algorithms for computing a *minimal dominating set*: (i) an $O(m)$ rounds algorithm if the agents start from a single node (i.e., gathered initially), (ii) an $O(\ell\Delta\log(\lambda)+n\ell+m)$ rounds algorithm, if the agents start from multiple nodes (i.e., positioned arbitrarily), where m is the number of edges and Δ is the maximum degree of G, ℓ is the number of clusters of the agents initially and λ is the maximum ID-length of the agents. Then we present a $\ln(\Delta)$ approximation algorithm for the *minimum* dominating set which takes $O(n\Delta\log(\lambda))$ rounds.

This chapter is based on the following work:

Prabhat Kumar Chand, Anisur Rahman Molla and Sumathi Sivasubramaniam *Run for Cover: Dominating Set via Mobile Agents.* In: Georgiou, K., Kranakis, E. (eds) Algorithmics of Wireless Networks. ALGOWIN 2023. Lecture Notes in Computer Science, vol 14061. Springer, Cham. https://doi.org/10.1007/978-3-031-48882-5_10

3.1 Introduction

Research on autonomous mobile agents has become an area of significant interest in the field of distributed computing. As autonomous agents become part of everyday life (in the form of agents, drones, self-driving cars, etc.), the area becomes more and more relevant. On the other hand, the dominating set problem is a well-researched classical graph problem. A dominating set D of a graph $G=(V,E)$ is a subset of the nodes V such that for any $v\notin D$, v has a neighbour in D. Finding a dominating set has several practical applications. For example, in wireless communications, the dominating set of the underlying network is useful in finding efficient routes

within ad-hoc mobile networks. They are also useful in problems such as *document summarising*, and for designing *secure systems* for *electrical grids*, to mention a few.

Covering problems, such as vertex covers, maximal independent sets (MIS), and dominating sets, also have real-world applications in the field of mobile agents. For example, a minimum dominating set can form the charge stations or parking places for autonomous mobile agents. Maximal independent sets can be used to solve the same but also ensure that any two agents are not close in real life. Both MIS and dominating sets can find a place in the guarding problem, i.e., placing mobile agents such that agents guard (cover) an entire polygon. In the mobile agent setting, MIS has been explored in [36, 103, 63]. However, in these works, the agents have some amount of visibility (they have some knowledge of the graph) while ours have zero knowledge. While both MIS and dominating sets are of great interest, in this work, we limit ourselves to solving the dominating set problem in the field of mobile agents. We note, however, that for the minimal dominating set case, our produced dominating sets are also maximal independent sets.

While the problem is a classic problem in graph theory, several attempts have been made to solve it in distributed computing research as well [60, 78, 77, 38]. In the field of mobile agents, to the best of our knowledge, this is the first work to solve the dominating set problem. Our algorithms rely on the ideas used for *dispersion* to achieve a solution for the dominating set problem. Dispersion, first introduced by the authors in [9], has been well studied for various configurations of agents (see [92, 68, 72, 75]). The problem is briefly: on an n node graph G, in which there is a configuration of agents $\mathcal{R}, |\mathcal{R}| \leq n$, we want to ensure that there is at most one agent on each node. We take advantage of the fact that most dispersion protocols involve the exploration of G, and develop algorithms for calculating dominating sets. In the next subsection, we mention our major results.

3.1.1 Our Results:

In this work, we show how to compute a minimal dominating set and an approximate minimum dominating set on an anonymous graph using mobile agents. Let G be a connected and anonymous graph of n nodes and m edges with maximum degree Δ. Suppose n agents (with distinct IDs) are distributed arbitrarily over the nodes of G. We develop algorithms for the agents to work collaboratively and identify small dominating sets of G. In particular, our results (Table 3.1) are:

- an $O(m)$ rounds algorithm for the agents to compute a minimal dominating set on G when all agents are gathered at a single node initially. The set of dominating nodes also forms an MIS.

- an $O(\ell\Delta\log(\lambda)+n\ell+m)$ rounds algorithm for the agents to compute a minimal dominating set on G when the agents are placed arbitrarily at ℓ nodes initially.
- an $O(n\Delta\log(\lambda))$ rounds algorithm for a $\ln(\Delta)$ approximation solution to the minimum dominating set on G, where λ is the maximum ID-length of the agents.

All our algorithms require that the agents have at most $O(\log(n))$ bits of memory. In a recent work in [66], the authors solved a related problem called the Distance-2-Dispersion problem (Details in Section 3.1.2) which also produces a MIS in special cases (when the number of agents is greater than the number of nodes in the graph) in $O(m\Delta)$ rounds with $O(\log(\Delta))$ bits of memory in each agent.

Problem	**Knowledge**	**Time Complexity (Rounds)**	**Memory**	**Initial Configuration**	**Remarks**
Our Results					
Minimal Dominating Set (Theorem 3.1)	–	$O(m)$	$O(\log n)$	Rooted	Dominating set also forms a MIS
Minimal Dominating Set (Theorem 3.3)	λ, Δ, n, ℓ	$O(\ell\Delta\log\lambda+n\ell+m)$	$O(\log n)$	Arbitrary	Dominating set also forms a MIS
Approximate Minimum Dominating Set (Theorem 3.4)	Δ, λ, n	$O(n\Delta\log\lambda)$	$O(\log n)$	Dispersed	$\ln(\Delta)$-approximation ratio

Table 3.1: Summary of our results on computing dominating sets using n mobile agents in n-node anonymous graphs.

In this work, dispersion and related organised configurations are useful because they allow agents to reason locally about domination. Since the nodes are anonymous and memoryless, an agent placed at or visiting a node can act as the local representative of that node. This helps the agents decide which nodes should belong to the dominating set and which nodes are already dominated. The algorithms in this chapter use dispersion or partial organisation as a clean starting point for developing the main ideas in the mobile agent model.

3.1.2 Related Works

3.1.2.1 Dominating Set in Distributed Computing

Finding small dominating sets is one of the most fundamental problems in graph theory and distributed computing. The dominating set problem is closely related to the set cover problem and

is known to be NP-hard [49, 64]. Several distributed algorithms have been proposed for computing dominating sets under different communication models. In [60], Jia *et al.* proposed one of the first efficient distributed implementations for the dominating set problem in the CONGEST model. Their randomised algorithm, adapted from the greedy heuristic of [85], computes a $\ln(\Delta)$-approximate dominating set in expectation within $O(\log n \log \Delta)$ rounds. Kuhn and Wattenhofer [78] developed LP-relaxation-based approximation algorithms that compute dominating sets within a factor of $(k\Delta^{2/k} \log \Delta)$ of optimal in $O(k^2)$ rounds. They also established one of the first non-trivial constant-round approximation schemes for dominating sets in distributed settings. Later, Kuhn *et al.* [77] proved lower bounds for approximating minimum dominating sets and minimum vertex covers in classical message-passing systems. They showed that even obtaining polylogarithmic approximation ratios requires at least $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ and $\Omega\left(\sqrt{\frac{\log \Delta}{\log\log \Delta}}\right)$ rounds. More recently, deterministic approximation algorithms with near-optimal guarantees were proposed in [38]. Their algorithms achieve approximation ratios of $(1+\epsilon)(1+\log(\Delta+1))$ in the CONGEST model with improved deterministic complexities. The work also studies connected dominating sets and obtains $\ln(\Delta)$-approximation guarantees. In a related direction, Sultanik *et al.* [117] studied a distributed variant of the art gallery problem equivalent to computing minimal dominating sets of visibility graphs.

3.1.2.2 Distance-2 Dispersion and Related Problems

A problem closely related to ours is the Distance-2-Dispersion (D-2-D) problem introduced by Kaur *et al.* [66]. In the D-2-D problem, agents settle on nodes such that no two settled agents occupy adjacent nodes. Additionally, an agent may settle at an already occupied node only when no valid unoccupied node remains satisfying the distance constraint. The authors showed that the D-2-D problem can be solved in $O(m\Delta)$ rounds using $O(\log \Delta)$ bits of memory per agent without requiring prior knowledge of m, n, or Δ. They further observed that when the number of agents satisfies $k \geq n$, the nodes containing settled agents form a maximal independent set of the graph.

3.1.2.3 Mobile Agents and Dispersion

The mobile-agent model differs fundamentally from classical message-passing systems because agents can communicate only through physical co-location. Consequently, many coordination tasks rely on structured exploration and movement-based communication. One important primitive in this setting is *dispersion*, where mobile agents rearrange themselves so that at most one

agent occupies each node of the graph. Dispersion was introduced in [9] and has since been extensively studied under various assumptions and initial configurations [92, 68, 72, 75, 94, 95, 99, 25]. Detailed dispersion literature is already discussed in Chapter 1 and Chapter 2. In this chapter, we primarily use dispersion as a coordination primitive that enables agents to reason locally about domination.

3.2 Model and Problem Definition

In this work, we use the same model as described in Section 1.1.2, with one key difference: the number of agents is exactly n. We study the minimal dominating set problem under two initial configurations, namely *rooted* and *arbitrary*. For a minimal dominating set, we do not assume dispersed initialisation. However, the approximation algorithm begins from a dispersed configuration.

For computing the approximate dominating set, we use the dispersed configuration as a start because they allow agents to reason locally about domination. Since the nodes are anonymous and memoryless, an agent placed at a node can act as the local representative of that node. This helps the agents decide which nodes should belong to the dominating set and which nodes are already dominated.

Definition 3.1 (Problem Definition)**.** *Consider an undirected, connected n-node simple anonymous graph G with n mobile agents placed over the nodes of G arbitrarily. Let Δ denote the maximum degree of a node in G.*

Minimal Dominating Set. *The agents, irrespective of their initial placement, rearrange and colour themselves in such a way that i) there is an agent at each node of G ii) there is a self-identified subset of agents, coloured black, on $D \subseteq G$ which forms a minimal dominating set for G.*

Approximate Minimum Dominating Set. *The agents, irrespective of how they are initially placed, rearrange and colour themselves in such a way that a dominating set for G of size at most $\alpha|D^*|$ is identified, where D^* is a minimum dominating set. Here α is the approximation ratio to the minimum dominating set.*

Our goal is to design an algorithm for solving the above problems as fast as possible and keeping α as small as $\ln(\Delta)$.

3.3 Preliminaries: DFS Traversal and Protocol MYN

In this section, we present two subroutines that we use in our algorithms. The first one is a simple depth-first search (DFS) traversal that allows the agents to explore the entire graph and disperse at the same time. The second one is a protocol that allows two agents on neighbouring nodes to meet each other if required. But, at first, we provide the list of agent variables used throughout the chapter and their main functionality.

Agent Variables Used in Chapter 3		
Variable	**Values or Type**	**Description**
$r_i.ID$	Unique ID	Distinguishes agents; also used in Protocol MYN.
$r_i.parent$	Port / $null$	Port through which r_i first entered its settled node during DFS.
$r_i.child$	Port / 0	DFS port used for the next successful movement or exploration.
$r_i.state$	*forward*, *backtrack*	Current DFS traversal state of the agent.
$r_i.settled$	$0/1$	Indicates whether the agent is permanently settled.
$r_i.treelabel$	Agent ID	Identifies the DFS tree or cluster of the agent.
$r_i.colour$	*white*, *grey*, *black*	Stores the domination status of the agent/node.
$r_i.span$	Integer pair	Span value used in the approximate dominating set algorithm.

Table 3.2: Agent variables used in this chapter. List of notations on Table 1.3 (Introduction)

3.3.1 DFS Traversal Protocol

The DFS Traversal Protocol is used frequently as a subroutine in several mobile agent problems. Our work uses the same DFS protocol as described in [76]. Assume that all the n agents are initially docked at a node, which we call the $root$ node. The high-level idea is to traverse the graph, node by node, while posting one agent at each node. The DFS protocol begins at the $root$ where the agent with the least ID settles. The remaining agents leave the $root$ via a computed exit port to land at a new node. The next smallest ID agent settles at the new node while the remaining $n-2$ agents leave the node to continue exploration. The agent group may need to backtrack when it encounters a settled agent or when all ports of a node have been explored. DFS completes when every agent settles. With the end of DFS, the agents achieve dispersion. The protocol takes $O(m)$ rounds.

We consider a collection of n agents, $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ placed initially at a single node called the $root$. For simplicity, we assume that r_1 is the lowest ID agent and r_n, the highest. The objective

of the DFS Traversal Protocol is to explore the graph in a DFS manner until all the agents are dispersed. In addition, since $|\mathcal{R}| = n$, it is ensured that G has a distinct agent stationed at each of its nodes after the end of the protocol. We recall that, for a node w, the ports are numbered from 1 to δ_w, where δ_w is the degree of w.

To execute DFS, each agent r is provisioned with the following variables:

- $parent-$ stores the port number through which r has entered an empty node and has settled (for a settled agent).
- $child-$ for an unsettled agent r it stores the port number last taken while entering/exiting a node. r, when settled, stores the port number that the other agents used for exiting a node except when they entered the node in a forward mode for a second or subsequent time. $child$ is set to 0 initially.
- $state-$ to indicate if r is in a *forward* mode or *backtrack* mode. $state$ is initially set to *forward*
- $treelabel-$ to differentiate between different DFSs arising from different clusters (meaningful only in an arbitrary initial agent configuration). $treelabel$ is the ID of the smallest agent in a specific cluster.
- $settled-$ to indicate whether an agent is settled (1) or unsettled (0)

Execution (Update Procedure)

In the first round, the agent r_1 assigns $r_1.settled \leftarrow 1$. The remaining agents $\mathcal{R} \setminus \{r_1\}$ decide on the minimum port number available at $root$ (which is 1), set $r_1.child$ and exit through port $r_1.child$ to a new empty node u. In the next round, the agent r_2 settles at u, and the remaining agents similarly exit through $r_2.child$ leaving r_2 at u. Let, at any stage of the DFS, the agents $\{r_{i+1}, r_{i+2}, \ldots, r_n\}$ arrive at a node w through port p_w with degree δ_w. Now, the agents decide on the next course of action based on their $state$ variable:

- $state = forward$: If w is empty, set $r_{i+1}.settled \leftarrow 1$, $r_{i+1}.parent = r_{i+1}.child$. The remaining agents $\{r_{i+2}, r_{i+3}, \ldots, r_n\}$ set $child \leftarrow (child + 1) \pmod{\delta_w}$ and then $r_{i+1}.child \leftarrow child$. If $child = r_i.parent$, it implies that all ports at w have been used and hence the remaining unsettled agents set $state \leftarrow$ *backtrack*. Otherwise if w is non-empty, $\{r_{i+1}, r_{i+2}, \ldots, r_n\}$ set their $state$ to *backtrack*. In both cases, the unsettled agents now move out through $child$.

- $state = backtrack$: Let r_j be the settled agent at w. The agents $\{r_{i+1}, r_{i+2}, \ldots, r_n\}$ set $child \leftarrow (child + 1) \pmod{\delta_w}$ and $r_j.child \leftarrow child$ (this updates the $child$ port on the settled agent). If $child \neq r_j.parent$ (implying an available port at w), the agents $\{r_{i+1}, r_{i+2}, \ldots, r_n\}$ switch their $state$ to *forward* and exit through $child$ port.

The protocol ends when there are no more unsettled agents remaining. Out of the m edges, each $m-(n-1)$ non-tree edge is traversed a maximum of four times (twice from either end) and the tree edges are traversed twice. Hence, the maximum number of rounds required to execute the DFS Traversal Protocol is $4(m-n+1)+2(n-1) = 4m+2n-2$. So, we have the following lemma.

Lemma 3.1. *Let G be an n-node arbitrary, connected and anonymous graph with a maximum degree Δ. Let n mobile agents with distinct IDs be placed initially at a single node. Then, the DFS Traversal Protocol disperses each of the mobile agents into n distinct nodes in $O(m)$ rounds.*

3.3.2 Protocol MYN

Since the port ordering of nodes is different, it can be tricky to ensure that two agents on neighbouring nodes meet. Also, it is difficult to time the movement of agents and guarantee that two neighbours meet without access to a global clock. Since it can be essential for two agents to pass information to each other, arranging ways to ensure such a meeting can be beneficial. Protocol MYN is helpful in ensuring that an agent is able to communicate with all its neighbours at least once when required. We use the pairing protocol Protocol MYN to ensure that during a scan for neighbours, an agent meets all its neighbours. For this, the algorithm essentially exploits the bits representing the IDs of the agents. Let λ denote the largest ID among all the n agents. Therefore, the agents use a $\log(\lambda)$ bit field to store the IDs. Protocol MYN runs in phases. Each phase consists of Δ rounds, and there are a total of $\log(\lambda)$ phases. Each phase corresponds to a bit in the field (with agents having IDs less than $\log(\lambda)$ bits padding the rest with 0s). The steps in a phase are simple, starting with the rightmost bit. If the bit is 1, the agent uses Δ rounds to visit all its neighbours. If the bit is 0, the agent waits at its node for visitors. Clearly, since all agents have unique IDs, for any two pairs of neighbouring agents, there exists at least one round in which the agents have different bits and meet.

Now, to find the neighbouring agents, an agent r_i stationed at a node u does the following.

1. Checks the rightmost bit in its ID field.

2. If the bit is 1, r_i goes to each of its neighbours one by one from u, following the ascending order of the port numbers and back. Note that, from u, there are ports numbered from 1 to δ_u, each leading to a distinct neighbour of u.

3. If r_i finds an agent r_j (or other multiple agents) in its neighbour, it can exchange required information.

4. Otherwise, if the bit id 0, the agents sits at u and waits for Δ rounds.

5. In the next phase of Δ rounds, r_i now checks its second bit from the right and repeats the same process as described in the previous steps.

6. The process continues for $\log \lambda$ phases, ensuring every bit in the ID field has been scanned. If there are no more bits remaining in $r_i.ID$ and $\log \lambda$ rounds have not been completed (implying that r_i has a smaller ID length than $\log \lambda$ bits), r_i assumes the current bit as 0 and stays back at its own node for the rest of the algorithm.

Clearly, the protocol takes $O(\log(\lambda))$ rounds to ensure that two specific neighbours meet, to ensure that an agent meets with all its neighbours, it takes no more than $O(\Delta \log(\lambda))$ rounds. Hence,

Lemma 3.2. *Protocol MYN ensures that an agent meets all its neighbours at least once and takes no more than $O(\Delta \log(\lambda))$ rounds.*

Algorithm 5 Protocol MYN

Require: A mobile agent r with $\log \lambda$ bit ID field - $b_1 b_2 ... b_{\log \lambda}$;
Ensure: r meets every other agent in its neighbourhood

1: **for** $i = \log \lambda$ to 1 **do**
2: **if** b_i is 1 **then**
3: **for** $j = 1$ to Δ **do**
4: r visits neighbour at the port δ_j. If the neighbour contains an agent, it can communicate with it.
5: **end for**
6: **else**
7: r remains at its node.
8: **end if**
9: **end for**

3.4 Algorithm for Minimal Dominating Set

In this section, we show that we can achieve a minimal dominating set for both the rooted and arbitrary initial configurations. In the rooted case, initially, all agents are gathered at a single root, while in the arbitrary case, the agents are gathered in clusters across the graph.

3.4.1 Single Source or Rooted Initial Configuration

Let us first consider the case where all the mobile agents are initially housed at a single node of the graph. We refer to such a configuration as a single source or *rooted initial configuration*. We design an algorithm for the mobile agents to work collaboratively to compute a minimal dominating set of G. In particular, agents identify a subset D of the nodes V such that D is a minimal dominating set of G. Given n agents that start from a single source, our algorithm takes $O(m)$ rounds to find such a D, where m is the number of edges in the graph. Agents are not required to know any graph parameters.

To solve the dominating set problem in the rooted initial configuration, we modify the standard DFS traversal (see Section 3.3.1) to allow the agents to simultaneously traverse and compute the dominating set D.

To ensure that the agents know if they are part of the dominating set for G in the end, each agent is *coloured* accordingly. The agents that occupy the nodes $D \subset V$ are coloured "black" to distinguish them from other mobile agents. Thus, the set of mobile agents coloured black forms the dominating set for G. To ensure proper colouring, each agent uses an additional variable $r_i.colour$ along with the ones used for executing the DFS Traversal. $r_i.colour$ is used to classify the nature of the agent with respect to the dominating set. Each agent, at a given time, has one out of the three colours - *black*, *grey* and *white*. Initially, all agents are *white* in colour. Agents change their colour to *black* once it becomes a member of the dominating set. Agents that are adjacent to a black coloured agent are coloured *grey*.

We will now describe our algorithm in detail. As mentioned before, our algorithm for creating a dominating set follows a modified version of the depth-first algorithm described in Section 3.3.1. We do so by the addition of an extra step each time an agent explores a new node, to decide an agent's colour.

As in the DFS, the collection of agents $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ start the algorithm from the $root \in V$. The agents leave the smallest ID agent r_1 at $root$, where r_1 settles and colours itself

black. In general, the agents navigate the graph via the DFS protocol with the extra step of deciding the colour of the agent that settles. This is decided as follows. Each time the agents decide to settle an agent at the node u, they visit all neighbours of u. If they find a *black* settled agent among u's neighbours, then the agent settling at u colours itself *grey*, else it colours itself *black*. However, the agent that has just arrived at a new empty node with its $parent$ coloured *black* can immediately become *grey*. The remaining agents move via the smallest port available at u to the next node according to the DFS traversal protocol (see Section 3.3.1), and the algorithm continues.

The unsettled agents scan their neighbour for *black* agents only in the *forward* phase. The algorithm stops when the last unsettled agent that started from $root$ settles and colours itself. The nodes of G, which we notate by D, that have a *black* agent placed, now form a dominating set for the graph G.

3.4.1.1 Analysis

Lemma 3.3. *Algorithm 6 ensures that each agent is associated with a distinct node in G and that the subset, D, of agents coloured black forms a minimal dominating set of G.*

Proof. During the course of the algorithm, which begins with a group of unsettled agents at a specially designated node called $root$, the agents get settled one by one in increasing order of the IDs, at some distinct node of G. The DFS protocol ensures that the entire graph is explored and each node in G contains a settled agent. Since each settled agent gets immediately coloured as black or grey, Algorithm 6 leaves no agent with the colour *white*. The *black*-coloured agents, D, form the dominating set, whereas the *grey* agents are the ones that are adjacent to one or more agents included in the dominating set. Note that an agent is coloured grey if and only if there exists a neighbour that was coloured black. Since there are no white agents at the end of the protocol, D is a dominating set of G.

We now prove that D represents a minimal dominating set for the graph G. For contradiction, let us assume that after the end of the algorithm, we can remove a *black* node (agent) r_x from D without disrupting its dominating set property. Therefore, D is still a dominating set for G without r_x. Now, since the algorithm executes sequentially, the parent and every neighbourhood of r_x is *grey* after the end of the algorithm. It leaves the node r_x uncovered by any black node. Therefore, D does not form a dominating set for G with r_x excluded. □

Lemma 3.4. *Algorithm 6 executes in $O(m)$ rounds.*

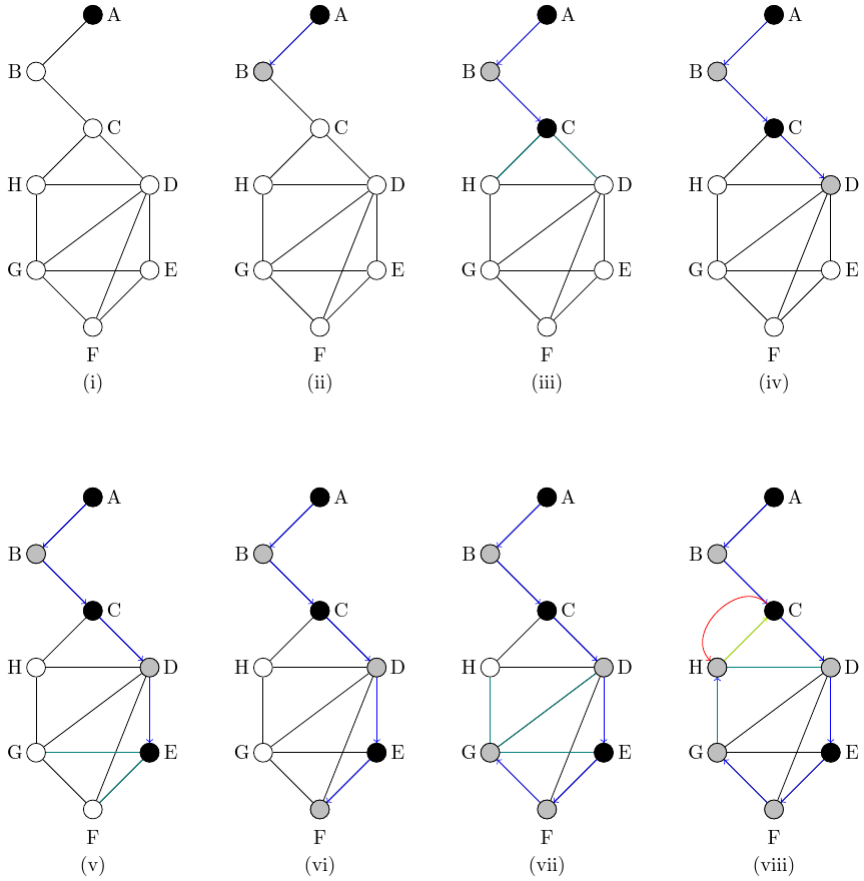


Figure 3-1: Illustration of Mobile Agents identifying a Minimal Dominating Set. The agents start at the $root$ node A. The minimum ID agent settles at A and colours itself as *black*. The rest of the agents follow the DFS and end up at node B, where the next minimum ID agent in the group settles and colours itself as *grey* (since it comes from a *black* parent). The rest of the agents now stop at the next node in the DFS, node C. The settled agent at C scans the neighbourhood (teal coloured edges) and finds no *black* agent and hence colours itself as *black*. The algorithm continues along the DFS route (blue-directed edges) shown from (iv) to (vii), settles and colours agents accordingly till it reaches the final configuration (viii). The red edge denotes a $backtrack$. After the configuration of (viii) is reached, no new node is discovered, and the algorithm stops after completing the DFS of the graph.

Algorithm 6 DOMINATING SET - ROOTED CONFIGURATION (ALGORITHM FOR AGENT r_i)

Require: An n-node anonymous graph with n mobile agents docked at $root$ node.
Ensure: Agents settle over the nodes and identify a minimal dominating set.

Let the agents $\{r_1, r_2, \ldots, r_i, \ldots, r_n\}$ be docked at a node $root$. Agent r_i maintains the following list of variables: $\langle r_i.id, r_i.parent, r_i.child, r_i.state, r_i.colour\rangle$, where $r_i.colour$ is initially set to *white*, $r_i.parent$, $r_i.child$ are set to $null$ and $r_i.state$ is set to *forward*.
while r_i is unsettled **do**
    if r_i is not the minimum ID agent among the unsettled agents **then**
        move according to DFS Traversal Protocol 3.3.1
    else
        if the current node is empty **then**
            set $r_i.settle \leftarrow 1$ and inform the other unsettled agents to stay stationary
            move to and check the colour of its parent agent and return
            if parent is black **then**
                sets $r_i.colour \leftarrow$*grey*
            else
                visits each neighbour of the current node and if there is at least one *black* in its neighbour, set $r_i.colour \leftarrow$*grey* or else set $r_i.colour \leftarrow$*black*
                inform the remaining unsettled node about the completion of colouring
            end if
        else
            move according to DFS Traversal Protocol 3.3.1 to find an empty node
        end if
    end if
end while
remains stationary at the current node for the rest of the algorithm
The nodes where each *black*-coloured agent is stationed form the dominating set D of G.

Proof. The algorithm essentially executes a depth-first search, in which it settles the n agents, one by one. The depth-first search takes up $O(m)$ rounds. In addition to that, once an agent r_i settles at a specific node u, it takes a maximum of $O(\delta_i)$ rounds, where δ_i is the degree of the node u, to search for agents among its neighbours, to decide its own colour. Therefore, the algorithm takes $O(m + \sum(\delta_i)) = O(m)$ rounds to execute. □

Memory per agent. Each agent stores its ID, which takes $O(\log(n))$ bit space. Along with that, the parent and child pointers take $O(\log(\Delta))$ bits of memory each. Other variables take up a constant number of bits. Therefore, the memory complexity is $O(\log(n + \Delta)) = O(\log(n))$ bits.

Combining it with Lemma 3.3 and Lemma 3.4, we have the following theorem.

Theorem 3.1. *Let G be an n-node arbitrary, connected and anonymous graph with m edges. Let n mobile agents with distinct IDs in the range $[0, n^c]$, where c is a constant, be placed at a single node, known as the rooted initial configuration. Then, a minimal dominating set for G can be found in $O(m)$ rounds using Algorithm 6 with $O(\log(n))$ bits of memory at each agent.*

As a by-product, the Algorithm 6 computes a maximal independent set (MIS). In fact, the set D is also an MIS for G. Thus, we get the following result on MIS.

Theorem 3.2 (Maximal Independent Set)**.** *Let n mobile agents be initially placed at a single node of an n-node anonymous graph G with m edges. Then there is an algorithm (cf. Algorithm 6) for the agents to compute a maximal independent set of G in $O(m)$ rounds and with $O(\log(n))$ bits of memory per agent.*

Proof. An agent receives a *black* colour after ensuring that no neighbour is coloured black. This ensures that no two adjacent nodes host *black* agents, guaranteeing D to be an independent set. Now, let's assume that we can add a black node v to D while maintaining its independent set property. Since the agent at v was initially *grey*, it must have had a *black* neighbour at some node u. Therefore, we happen to have two adjacent nodes u and v that are occupied by *black* agents, contradicting the fact that D is an independent set. Therefore, no new black node can be added to the independent set D, making it a maximal independent set (MIS). □

3.4.2 Multi Source or Arbitrary Initial Configuration

In the multi-source (or arbitrary) case, the n agents $\{r_1, r_2, \ldots, r_n\}$ start as $\ell < n$ clusters, placed arbitrarily at ℓ different nodes of G. Once again, our algorithm for constructing dominating sets is inspired by the DFS Protocol for dispersion [76]. In this section, we show how we can achieve a minimal dominating set for such a configuration. For this algorithm, we assume that the agents are aware of the parameters $n, \Delta, \ell,$ and λ.

The agents first perform dispersion using the method described in Kshemkalyani [76][1]. Briefly, their algorithm allows each of the ℓ clusters to begin DFS independently. If a DFS meets no other

[1] A recent optimal dispersion algorithm by Kshemkalyani et al. [70] disperses n mobile agents from an arbitrary initial configuration in $O(n)$ rounds. This can improve the dispersion phase of our algorithm from $O(m)$ to $O(n)$ rounds. However, it does not asymptotically improve our overall dominating set construction from an arbitrary configuration, since the post-dispersion construction phase still takes $O(m)$ rounds. Also, as in [76], the algorithm does not provide termination detection, so agents still need to wait for a sufficiently large number of rounds before proceeding to the next phase.

DFS, then it executes to completion as in the rooted case. If not, their algorithm ensures that if two (or more) DFSs meet while dispersing, they are all merged into the DFS with the largest number of settled agents. That is, if DFS i meets DFS j and i has more settled agents at the time of meeting, then j is collapsed, and all of j's agents join in executing i's DFS. This act of gathering all of j into i is called *subsumption*. Thus, when two DFSs meet, the shorter DFS gets subsumed by the larger one. Note that this means that once dispersion is achieved, there may be several DFS trees.

Let us assume then, after the end of the algorithm, there are $\ell^* \leq \ell$ independent DFSs that never met (each with its own unique $root$). Each DFS is identified by a unique label $treelabel$ (which is the ID of the smallest agent in the tree). Our procedure to create the minimal dominating set consists of two steps (i) elect a global leader agent r^* (ii) with r^* as root, use an adaptation of Algorithm 6 to achieve a minimal dominating set. Note that the dispersion algorithm in [76] takes $O(m)$ rounds. To ensure coordination, each agent waits to begin the leader election protocol after $cn\Delta$ rounds for a sufficiently large constant c from the start of dispersion.

Leader Election: We first use the Protocol MYN (see Section 3.3) to ensure that all agents first meet their neighbours. When an agent meets an agent from a different ID, they share their $treelabel$s with one another. When an agent receives a lower $treelabel$ value, it updates its own $treelabel$ to the lower value. After Protocol MYN has been executed, each agent, starting from the leaf nodes in a DFS (say DFS i), now sends its newly updated $treelabel$ value to the root of DFS i ($root_i$) via its $parent$ pointers. The root of the DFS i, $root_i$, waits for $O(n)$ rounds as it receives different values of $treelabel$ continuously from its leaf nodes (*up-casting*). The $root_i$ now compares the minimum $treelabel$ values received from its children with its own $treelabel$ and decides on the minimum $treelabel$. It then sends it along through its children into each and every node of its, in the DFS (*down-casting*). All the agents in DFS i are now updated with the lowest $treelabel$ value. This marks the end of a phase. After the end of ℓ (although ℓ^* is sufficient, its value is unknown) such phases, it is guaranteed that each of the n agents in G now has a consistent $treelabel$ value. The final $treelabel$ value is the minimum among the ℓ^* DFSs; however, it may not represent the minimum ID agent, as it may be consumed by a larger DFS during the dispersion process happening earlier. After the end of the protocol, the agent that has the globally decided minimum $treelabel$ continues the algorithm. We identify the leader agent with the minimum $treelabel$ value as $r^\star$.

Creating Minimal Dominating Set: With $r^\star$ as a leader, it first creates a new DFS of the graph G with the node containing $r^\star$ as the $root$ node. Since the agents are dispersed across each and every node of G, the nodes of G are now distinguishable owing to the distinct IDs of the mobile

agents in the nodes. $r^\star$ starts from the root and rewrites the $parent$ and $child$ pointers of each agent as it traverses across the whole graph G in a depth-first manner. Moving to an empty node implies that $r^\star$ has arrived at the $root$. In such cases, $r^\star$ modifies its own variables accordingly. The other details regarding the DFS can be found in Section 3.3.1. After the end of this process, which takes an additional $O(m)$ rounds, the newly assigned pointers of the agents now represent a DFS of G with $r^\star$ (node) as the root node.

Initially, all the agents are coloured *white* by default. The identification of the dominating set takes place in similar lines as with the rooted case described in the previous Section 6. The algorithm begins with $r^\star$ colouring itself black. $r^\star$ then moves the next node (say, node u resided by an agent r_u) following the newly created DFS. Upon the arrival of $r^\star$ at u, r_u scans each of its neighbours to check if there is a black agent. If there are no black agents, r_u colours itself as *black*; otherwise, if there is at least one black agent in the neighbourhood of u, r_u colours itself *grey*. After r_u has changed its colour from *white*, it informs $r^\star$ as $r^\star$ now moves to the next node following the DFS. The process continues similarly with each new node of degree $O(\delta_i)$, taking $O(\delta_i)$ rounds to scan its neighbour and colour itself accordingly. The algorithm terminates as soon as $r^\star$ completes the DFS of G.

Therefore, combining the results from the previous section (Section 3.4.1), DFS traversal and Protocol MYN, we get the following main result.

Theorem 3.3. *Let G be an n-node anonymous graph with the maximum degree Δ. Let n mobile agents with distinct IDs in the range $[0, n^c]$, where c is a constant, be initially placed arbitrarily among ℓ nodes of G in ℓ clusters. Then, a minimal dominating set for G can be found in $O(\ell\Delta\log(\lambda) + n\ell + m)$ rounds using the algorithm in Section 3.4.2 with $O(\log(n))$ bits of memory per agent. λ is the maximum ID-string length of the agents.*

Proof. The algorithm in Section 3.4.2 first disperses the n agents over the n nodes using the algorithm in [76], which takes $O(m)$ rounds. We then elect a leader among the agents. There are at most ℓ DFS trees (as it starts with ℓ clusters initially). Each DFS tree uses Protocol MYN to communicate with the neighbouring DFS (if any). Once the leaf nodes receive a new communication from a different DFS, it sends the information to the root of its DFS, which may take $O(n)$ rounds. In the worst case Protocol MYN may be executed ℓ times to elect the leader. From Lemma 3.2 we know that Protocol MYN takes $O(\Delta\log(\lambda))$ rounds. Thus, the leader election requires $O(\ell(\Delta\log(\lambda) + n))$ rounds. The leader agent then finds a minimal dominating set while forming a single DFS tree using at most $O(m)$ rounds. Finally, the dominating set identification takes another $O(m)$ rounds. Therefore, the time complexity of the algorithm in Section 3.4.2 is $O(m + \ell(\Delta\log(\lambda) + n) + m) = O(m + \ell\Delta\log(\lambda) + n\ell)$ rounds. □

The running time of the multi-source algorithm is $O(\ell\Delta \log \lambda + n\ell + m)$, where ℓ is the number of initial clusters. The dependence mainly on ℓ comes from the arbitrary initial configuration, since ℓ dictates the number of phases during leader election. Therefore, the term $O(\ell(\Delta \log \lambda + n))$ incurs a fixed cost depending on ℓ, Δ, λ, and n. When the agents are divided into many small clusters, ℓ can be large. In the extreme case, $\ell = \Theta(n)$, and the term $n\ell$ may become $O(n^2)$. In such a case, even for a sparse graph where $m = O(n)$, this term dominates the rooted-case complexity, which is $O(m) = O(n)$. Although the internal information propagation within a DFS subtree can be fast in this setting (since the tree sizes are smaller), the nature of the algorithm still requires the same coordination mechanism, which is bounded by the fixed values of n for internal information propagation within a tree and ℓ for the number of phases. On the other hand, when ℓ is small, for example, constant, the running time remains close to $O(m)$ as in the rooted case, apart from the additional neighbour-meeting and coordination costs. Therefore, the algorithm is most efficient when the number of initial clusters is small, while its cost increases as the initial placement becomes more fragmented. Now, when Δ and ℓ become $\Theta(n)$, in such a case, we obtain the worst-case complexity of $O(n^2 \log(\lambda))$ rounds. On the other hand, when Δ and ℓ are constants, and the graph is sparse, the complexity becomes $O(n)$ rounds, a best-case possibility.

3.5 Algorithm for Approximate Minimum Dominating Set

In this section, we describe an approximation algorithm that gives us a dominating set with a size that is optimal within a factor of $\ln(\Delta)$ in the mobile agent setting. In the previous algorithms, our methodology was based on scanning only the 1-hop neighbourhood of an agent to assign an agent one of the two colours - *black* or *grey*. The set of *black* coloured agents formed the dominating set for G. Although these previous algorithms (*rooted*) ran in $O(m)$ rounds, the approximation ratio of the dominating set size produced with these algorithms to the optimal (minimum) dominating set could be huge. For example, if we consider a star graph with the agents starting at one of the leaves, then all leaf nodes become included in the dominating set (Figure 3-2). Although that dominating set is minimal, it is far from optimal. The optimal in the case of a star graph is the single non-leaf node at the centre. Therefore, the approximation ratio in such a case can be as large as $O(n)$. In this section, we adapt the well-known greedy distributed algorithm [85], to provide a better approximation solution for the dominating set problem in the mobile agent setting.

In [85], the authors give a greedy distributed algorithm (referred to in the paper as "*distributed*

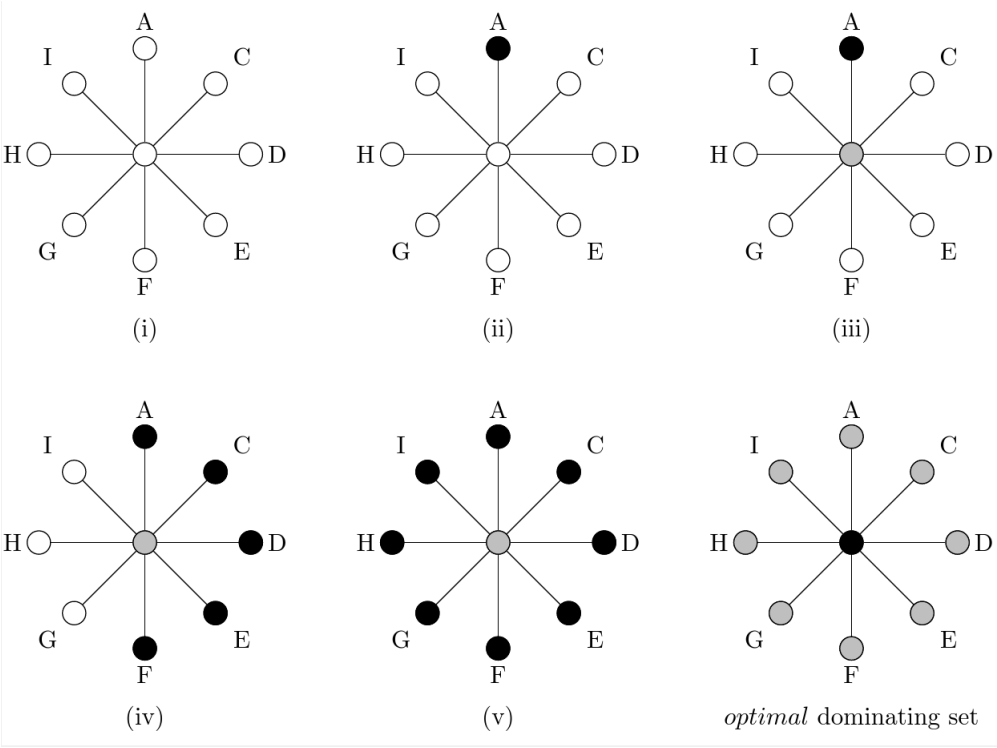


Figure 3-2: When the agents start at a leaf node (node A), our previous algorithm (Algorithm 6) produces a minimal dominating set identified by the black nodes as in (v) by executing through (ii), (iii) and (iv). The figure right next to (v) on the other hand, shows an optimal dominating set. Therefore, for an n node graph, the size of a minimal dominating set could be as bad as $O(n)$ compared to the optimal.

database coverage heuristic (DDCH)") for calculating a dominating set with an approximation ratio of $\ln \Delta$ to the optimal set size. The same is reintroduced for a synchronous communication model where the nodes can communicate with each other without moving [60]. The authors also refined the algorithm to a randomised version, obtaining a better expected running time. We essentially adopt the greedy distributed algorithm to our own model. The challenge is, of course, as follows: in the earlier distributed setting, nodes are allowed to communicate with their neighbours within the span of a round, whereas the agents in our model cannot communicate with each other unless they are at the same node. While the agents can gather information by visiting neighbouring agents, it is difficult to time the movement of agents and guarantee that two neighbours meet without access to a global clock. However, with the help of Protocol MYN introduced in the previous section, a successful adaptation of the greedy algorithm is possible. More details follow.

For this protocol, we assume that the n mobile agents of set, $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ start from a

dispersed position on graph G. That is, they are initially dispersed among the n nodes of the graph G with each agent at a distinct node of G. If the agents are arbitrarily placed over G initially, the agents could be re-positioned using the dispersion algorithm described in [76]. As in our previous sections, each agent is assigned a variable colour to keep track of its colour, which can take one of three values *white, grey* or *black*. Let $D \subset G$ be the set of black agents at the end of the protocol, and D forms the dominating set of G. A white agent is one that is not (yet) covered by any agent in the dominating set. The *grey* agents represent agents that are already covered by some agent in D. Initially, all the agents are coloured *white*.

Let $span$ of an agent r_i, $w(r_i)$, be the number of white agents in the direct $(1-hop)$ neighbourhood of r_i, including r_i itself. Each agent r_i uses an extra variable $r_i.span$ for the $span$ values. $r_i.span$ has both the ID and the span count. The basic idea of the protocol is as follows. Each agent calculates the maximum span value within its $2-hop$ neighbourhood. If r is the agent that has the maximum span within $2-hops$, then r colours itself black and informs its neighbours. To achieve this, each agent performs the following four stages (of $2\Delta\log(\lambda)$ rounds each) until there are no white agents left in the graph:

1. r_i **calculates its span** $r_i.span$**:** The Protocol MYN, described in the previous section, guarantees that r_i meets all its neighbouring agents within $O(\Delta\log(\lambda))$ rounds ("meeting" includes r_i being stationary and a neighbouring agent arriving at r_i during these $O(\Delta\log(\lambda))$ rounds). Inside the first $O(\Delta\log(\lambda))$ rounds, the agent r_i communicates with its immediate neighbours and evaluates $r_i.span$.

2. r_i **gets to know the highest** $span$ **value within its immediate neighbours**
   Inside the next $O(\Delta\log(\lambda))$ rounds, r_i communicates with its immediate neighbours to know the highest $span$ value within its neighbourhood. If $r_j.span$ is the highest span value among immediate neighbours, $r_i.span$ gets replaced by $r_j.span$. For identical $span$ values, an agent with a lower ID is selected.

3. r_i **gets to know the highest** $span$ **value within its** $2-hop$ **neighbours**
   In the next $O(\Delta\log(\lambda))$ rounds of the algorithm, r_i visits all its immediate neighbours once again to check if there are any new updated span values (possibly from a neighbour's neighbour).

4. **if** r_i **coloured itself black, then it informs its neighbours.**
   After the completion of the first three stages, r_i has the information of the highest span and the agent that has the highest span in its $2-hop$ neighbourhood. If r_i itself is the agent with the highest span, it colours itself *black*. Any agent that receives *black* colour then

takes additional $O(\Delta \log(\lambda))$ rounds to go to its neighbours and colour any *white* agent in their neighbour as *grey*. The key intuition here is that agents with larger spans dominate a larger number of currently uncovered nodes. Therefore, agents locally compete within a 2-hop neighbourhood, and only the locally strongest candidate joins the dominating set.

Note that once an agent colours itself *black*, its colour does not change. All agents then reset their $span$ values, and the next phase of $O(\Delta \log \lambda)$ begins. The algorithm runs till each agent r_i has no more *white* agents in their neighbourhood.

Algorithm 7 $\ln(\Delta)$ Optimal Dominating Set (Algorithm for agent r_i)

while there are white agents in the neighbourhood **do**
  for $i = 1$ to $O(\Delta \log \lambda)$ **do** ▷ Stage 1
    Each agent $r_i \in \mathcal{R}$ computes its span $r_i.span$ using compute span Protocol MYN.
    if $r_i.span$ is zero **then**
      Then r_i stops executing the protocol. But will provide information if requested.
    end if
  end for
  for $i = 1$ to $O(\Delta \log \lambda)$ **do** ▷ Stage 2
    Each agent $r_i \in \mathcal{R}$ communicates its span $r_i.span$ its neighbours.
  end for
  for $i = 1$ to $O(\Delta \log \lambda)$ **do** ▷ Stage 3
    Each agent $r_i \in \mathcal{R}$ computes the maximum span within 2-hops.
  end for
  if $r_i.span$ is highest among all agents in the $2 - hop$ neighbourhood **then**, ▷ Stage 4
    r_i colours itself as *black*
    for $i = 1$ to $O(\Delta \log \lambda)$ **do**
      r_i meets each neighbour and colours any *white* agent as *grey* (use Protocol MYN)
    end for
  end if
  Reset $r_i.span$ for all $r_i \in \mathcal{R}$.
end while

Lemma 3.5. *Let r_i and r_k be two agents at a distance of 2 hops from each other. Then, the value of $r_k.span$ can be communicated to r_i within $O(\Delta \log(\lambda))$ rounds.*

Proof. Since r_i and r_k are at a distance of 2 hops from each other, there exists a sequence of agents (nodes) (r_i, r_{i_1}, r_k) from r_i to r_k. In the first $O(\Delta \log(\lambda))$ rounds, r_i can communicate with r_{i_1} to get the value of $r_{i_1}.span$. In the meanwhile, r_{i_1} also collects the value of $r_k.span$ in the same $O(\Delta \log(\lambda))$ round. Therefore, in the second sub-phase of $O(\Delta \log(\lambda))$ rounds, when r_i communicates with r_{i_1} again, r_i receives the value of $r_k.span$ through r_{i_1} or the vice-versa. □

Lemma 3.6. *Algorithm 7 computes a dominating set D which is a $\ln(\Delta)$ approximation to the optimal size dominating set D^* for the graph G.*

Proof. Our protocol emulates the greedy distributed algorithm in [60] by ensuring that the node with the highest span within a two-hop neighbourhood becomes a part of the dominating set. Hence the $\ln(\Delta)$ approximation follows directly from the approximation results in [60, 127]. □

Lemma 3.7. *Algorithm 7 takes $O(n\Delta\log(\lambda))$ rounds to execute.*

Proof. There are four stages within a single while loop in Algorithm 7. The first stage starts by calculating the span of each agent, which takes $O(\Delta\log(\lambda))$ rounds. In the next two stages of $O(\Delta\log(\lambda))$ rounds, the agents communicate their $span$ to all the agents with a distance of $2-hops$. In the final stage, when an agent gets a colour *black* (the agent with the highest span in its $2-hop$ neighbourhood), it can take another $O(\Delta\log(\lambda))$ rounds to instruct its neighbouring agents to colour themselves as *grey*. So, a single while loop from $span$ calculation till colouring agents as *grey*; takes no more than $O(\Delta\log(\lambda))$ rounds. As the execution of a single while loop gives us at least one black agent in the worst case, the algorithm needs at most $O(n)$ iterations of the while loop. Thus, giving us a complexity of $O(n\Delta\log(\lambda))$ rounds. □

Theorem 3.4. *Let G be an n-node arbitrary, connected and anonymous graph with a maximum degree Δ. Let n mobile agents with distinct IDs be placed in a dispersed initial configuration. Then, a $ln(\Delta)$-approximation solution to the minimum dominating set for the graph G can be found in $O(n\Delta\log(\lambda))$ rounds using Algorithm 7 with $O(\log(n))$ bits of memory per agent, where λ is the maximum length of the ID-string of the agents.*

Proof. From Lemma 3.6, we know that Algorithm 7 provides a $ln(\Delta)$ approximation solution. And from Lemma 3.7 we know that it takes at most $O(n\Delta\log(\lambda))$ rounds to execute. Thus, the theorem. □

3.6 Conclusion and Future Work

In this work, we solved the problem of finding the dominating set of a graph G using mobile agents. When the agents start at a single source, we are able to find a minimal dominating set in $O(n\Delta)$ rounds. On the other hand, for the multi-source starting configuration, the agents obtained a minimal dominating set in $O(\ell\Delta\log(\lambda)+n\ell+n\Delta)$ rounds. Additionally, the dominating sets obtained by these two algorithms also serve as a maximal independent set for the graph G.

In the last section, we described an approximation algorithm that gave us a dominating set with a size that is optimal within a factor of $\ln(\Delta)$. The approximation algorithm had a running time of $O(n\Delta \log(\lambda))$ rounds.

In future work, it would be interesting to investigate the lower bounds in terms of time, memory and number of agents being used, for the dominating set problem in the same distributed model. In our work, the dominating sets are also MIS, so it would be interesting to see if it's possible to produce dominating sets that are also connected, i.e., connected dominating sets. It would also be exciting to explore other classical graph problems, such as ruling sets, colouring, etc., and of course, given practical real-world concerns, adaptation of the algorithms for faulty agents is an important aspect to be considered for the future as well.

Chapter 4

Agent-Based Discovery of Dense Subgraph Structures with Triangles and Butterflies

In this chapter, we study triangle and butterfly counting in anonymous graphs using autonomous mobile agents—a setting motivated by scenarios where direct access to the network is restricted but decentralised exploration is possible. For general graphs, we design efficient algorithms that enable each agent to compute node- and edge-level triangle counts in $O(\Delta \log \lambda)$ rounds and collectively determine the global triangle count in $O(D\Delta \log \lambda)$ rounds. These form the basis for computing truss decomposition, triangle centrality, and local clustering coefficients. In bipartite graphs, we present the first agent-based butterfly counting framework, where agents construct a spanning tree, elect a leader, and use a local coordination technique to compute butterfly counts in $O(\Delta + \min\{|A|, |B|\})$ rounds. All protocols operate without prior knowledge of the graph and use limited memory, offering scalable solutions for decentralised network analysis in constrained environments.

This chapter is based on the following works:

1. **(a) Prabhat Kumar Chand, Apurba Das and Anisur Rahaman Molla.** *Agent-Based Triangle Counting and Its Applications in Anonymous Graphs. (Extended Abstract)* In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), pp. 2186–2188. https://dl.acm.org/doi/abs/10.5555/3635637.3663102

   (b) Prabhat Kumar Chand, Apurba Das and Anisur Rahaman Molla. *Agent-Based Triangle Counting: Unlocking Truss Decomposition, Triangle Centrality, and Local Clustering Coefficient.* arXiv preprint, 2025. https://arxiv.org/abs/2402.03653 (A fully extended and improved version of the extended abstract, currently under revision on IEEE Transactions on Networking)

2. **Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla.** *Brief Announcement: Distributed Butterfly Analysis using Mobile Agents.* In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2025)*, Portland, OR, USA, July 28–August 1, 2025, pp. 623–627. ACM, 2025. DOI: 10.1145/3694906.3743334.

4.1 Introduction

Triangle and butterfly counting are two of the most important problems in graph analytics, fundamental to revealing the structure, cohesion, and functional roles of nodes within complex networks. Triangle-based metrics—such as the clustering coefficient [128, 20], triangle centrality [84, 1, 22], and truss decomposition [123]—enable identification of tightly-knit communities and hierarchical substructures [56, 3], with applications spanning web spam detection [19], content quality estimation [40], database query optimization [17], link prediction in social networks [120], and community detection in biological systems [61]. In bipartite networks, butterfly motifs—i.e., 2×2 bicliques or 4-cycles—serve as analogues of triangles, capturing dense two-mode interactions and facilitating tasks like dense subgraph peeling [108], community detection [4], and motif-based analytics [126]. Practical uses of butterfly counting extend to fraud detection [31], recommendation systems, and social media analysis. Prior studies have developed efficient sequential and parallel algorithms for butterfly counting, including exact and approximate approaches [107], node-prioritised schemes [125], adaptations to uncertain graphs [136], decomposition-based strategies [108], and hardware-accelerated solutions using multicore [110], GPU [133], and distributed computing [129]. In this work, we study both triangle and butterfly counting through the lens of autonomous mobile agents in anonymous graphs. Our interest is motivated by applications in private or inaccessible networks, such as underwater navigation [34], military surveillance [82], drone-assisted crowd monitoring [91], social epidemiology [41], and WSN deployments in remote environments [5, 39, 58, 55]. In such settings, triangle counting by mobile agents not only enables local metric computation and global aggregation but also supports downstream tasks like truss decomposition, triangle centrality, and clustering coefficient estimation. Moreover, drones can leverage triangle structures for robust SAR communication [135, 104, 132, 130], while bipartite butterfly motifs can reveal interaction clusters in two-mode networks.

For general graphs, we present algorithms by which each agent can locally compute the number of triangles incident to its node and to its adjacent edges in $O(\Delta \log \lambda)$ rounds, and collectively compute the total number of triangles in the graph in $O(D\Delta \log \lambda)$ rounds, where Δ and D are the graph's maximum degree and diameter, respectively. These serve as building blocks to solve higher-level tasks: truss decomposition is achieved in $O(m\Delta \log \lambda + mD)$ rounds, triangle centrality of each node is computed in $O(\Delta \log \lambda)$ or $O(D\Delta \log \lambda)$ rounds depending on whether the global triangle count is precomputed, and the local clustering coefficient for all nodes is obtained in $O(\Delta \log \lambda)$ rounds. All our algorithms require $O(\Delta \log n)$ bits of memory per agent.

In bipartite graphs, agents begin by determining their partitions, collaboratively constructing a spanning tree and electing a leader in $O(n \log \lambda)$ rounds with only $O(\log \lambda)$ bits per agent. A novel local meeting mechanism enables efficient butterfly counting per node in $O(\Delta)$ rounds and computation of the total butterfly count in $O(\Delta + \min\{|A|, |B|\})$ rounds. Our techniques require no prior knowledge of the graph and naturally extend to general graphs while preserving asymptotic efficiency.

We divide this chapter into two parts: the first part focuses on triangle counting and its applications, while the second addresses butterfly counting in bipartite graphs.

Part A: Triangle Counting and Its Applications

4.2 Our Contributions

We first enumerate the triangles in the graph G and apply the triangle counting methodology to three applications: (i) *Truss Decomposition*, (ii) *Triangle Centrality*, and (iii) *Local Clustering Coefficient*. Let G be an n-node simple, undirected, anonymous and connected graph with the maximum degree Δ and diameter D. Let n mobile agents with distinct IDs with the highest agent ID λ, be placed at each of the n nodes of G in a dispersed initial configuration. Then, we solve the following problems.

1. **Triangle Counting**
    - Each agent r_i can calculate and output the number of triangles containing the node where r_i is placed on, in $O(\Delta \log \lambda)$ rounds.
    - Each agent r_i can calculate the number of triangles based on each edge of G in $O(\Delta \log \lambda)$ rounds.
    - Each agent r_i can calculate the total number of triangles in G in $O(D\Delta \log \lambda)$ rounds.

    The memory requirement for each agent is $O(\Delta \log n)$ bits.

2. **Applications**
    - **Truss Decomposition** - The *Truss Decomposition Problem* for G can be solved by the mobile agents in $O(m\Delta \log \lambda + mD)$ rounds.

- **Triangle Centrality** - The *Triangle Centrality* of each node $v \in G$ can be calculated in $O(\Delta \log \lambda)$ rounds if $\mathbf{T}(\mathbf{G})$ is known and in $O(D\Delta \log \lambda)$ rounds, if $\mathbf{T}(G)$ is unknown. $\mathbf{T}(G)$ is the total triangle count of the graph G.
- **Local Clustering Coefficient** - The *Local Clustering Coefficient* of each node $v \in G$, i.e., $LCC(v)$ can be calculated by the n agents in $O(\Delta \log \lambda)$ rounds.

The memory requirement for each agent is $O(\Delta \log n)$ bits.

A summary of our results has been provided in Table 4.1. The notations are from Section 1.1.1.

4.3 Related Works

4.3.1 Triangle Counting

Triangle counting is a well-studied graph mining problem in both sequential and parallel settings.

4.3.1.1 Sequential Algorithms

In 1985, Chiba and Nishizeki [29] proposed an algorithm for counting all triangles in a simple graph by computing intersections of neighbourhoods of adjacent nodes with time complexity $O(m^{\frac{3}{2}})$, where m is the number of edges in the graph. Other classical sequential algorithms are based on the *node-iterator* [8] and *edge-iterator* [59] techniques. In the node-iterator approach, the algorithm iterates over each node v and intersects the adjacency lists of pairs of neighbours of v. In the edge-iterator approach, the algorithm iterates over each edge and intersects the adjacency lists of its endpoints.

4.3.1.2 Parallel and Distributed Algorithms

Several distributed and parallel algorithms for triangle enumeration have been proposed for shared-memory, distributed-memory, multi-core, GPU, and MPI-based systems. In [6], the authors proposed an MPI-based distributed-memory parallel algorithm called PATRIC for counting triangles in massive networks. Shun *et al.* [112] designed scalable multi-core parallel algorithms for exact and approximate triangle counting on graphs with billions of nodes and

Problem	Knowledge	Time Complexity (Rounds)	Memory per Agent	Initial Configuration	Remarks
Triangle Counting and Applications					
Node-Based Triangle Counting	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$	Dispersed	Computes $\mathbf{T}(v)$ for each node
Edge-Based Triangle Counting	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$	Dispersed	Computes triangle count for each edge
Total Triangle Counting	Δ, λ, D	$O(D\Delta \log \lambda)$	$O(\Delta \log n)$	Dispersed	Computes $\mathbf{T}(G)$
Truss Decomposition	Δ, λ, D	$O(m\Delta \log \lambda + mD)$	$O(\Delta \log n)$	Dispersed	Computes trussness of edges
Triangle Centrality	Δ, λ, D	$O(\Delta \log \lambda)$ or $O(D\Delta \log \lambda)$	$O(\Delta \log n)$	Dispersed	Depends on whether $\mathbf{T}(G)$ is known
Local Clustering Coefficient	Δ, λ	$O(\Delta \log \lambda)$	$O(\Delta \log n)$	Dispersed	Computes $LCC(v)$ for each node
Butterfly Counting in Bipartite Graphs					
Partition Detection + Leader Election + Spanning Tree	λ	$O(n \log \lambda)$	$O(\log \lambda)$	Arbitrary	Constructs spanning tree and elects leader
Node-Based Butterfly Counting	λ	$O(\Delta)$	$O(\Delta \log n)$	Bipartite, Dispersed	Computes butterfly count per node
Total Butterfly Counting	λ	$O(\Delta + \min\{\|A\|, \|B\|\})$	$O(\Delta \log n)$	Bipartite, Dispersed	Computes total butterflies in graph

Table 4.1: Summary of the main results for triangle and butterfly counting using n-mobile agents in n-node anonymous graphs.

edges. In [7], two efficient MPI-based distributed-memory algorithms were proposed for exact triangle counting in large graphs using overlapping partitioning and load-balancing techniques. Ghosh *et al.* [51] introduced TriC, an MPI-based graph triangle counting method for shared and distributed-memory systems, which was later improved in [50]. Additional surveys and developments on triangle enumeration in different computational models may be found in [122, 19, 17, 120, 81, 8, 52, 118, 97].

4.3.2 Truss Decomposition

4.3.2.1 Sequential Algorithms

In [32], Cohen introduced trusses as a relaxation of cliques and defined a k-truss as a non-trivial connected subgraph in which every edge participates in at least $k-2$ triangles. Since then, truss decomposition has been extensively studied in graph mining and community detection [56, 27, 57, 65]. Yang *et al.* [123] proposed an improved in-memory serialized algorithm for computing all k-trusses of a graph in $O(m^{1.5})$ rounds using $O(m+n)$ memory. They also proposed I/O-efficient algorithms for handling massive graphs that do not fit into the memory of a single machine.

4.3.2.2 Parallel and Distributed Algorithms

To handle the computational and memory requirements of massive graphs, several parallel implementations for truss decomposition have been developed. In [62], the authors parallelized the serialized algorithm of [123] using concurrent-update-friendly data structures. Sariyuce *et al.* [109] employed iterative h-index computation, originally formulated in [88], for nucleus decomposition and established convergence guarantees for both synchronous and asynchronous settings. Since truss decomposition is a special case of nucleus decomposition, these results directly apply to truss computation. Voegele *et al.* [122] proposed a parallel graph-centric k-truss decomposition algorithm and studied its relation with k-core decomposition. Jian Wu *et al.* [131] further optimised both serialised and parallel truss decomposition algorithms by reducing memory usage through improved data structures and WebGraph compression techniques. In [42], the authors extended truss decomposition to probabilistic graphs and analysed convergence bounds using iterative h-index updates. Many of these works also provide extensive experimental evaluations to demonstrate scalability and efficiency.

4.3.3 Mobile Agents and Graph Analytics

Agent-based graph exploration has been studied extensively in distributed computing and mobile robotics. In [115], Sudo *et al.* studied graph exploration using a single mobile agent with whiteboards to reduce memory requirements. In [67], the authors investigated Depth-First-Search-based graph exploration and analysed the trade-off between node memory and agent memory. Dereniowski *et al.* [37] proposed collective graph exploration algorithms using multiple agents and established near-tight bounds relating exploration time and team size under local and global

communication models. A related line of research studies *dispersion*, where mobile agents spread themselves so that at most one agent occupies each node of the graph. Dispersion has been extensively studied under various assumptions and initial configurations [68, 72, 75, 94, 95, 99, 25]. A detailed literature on dispersion is already discussed in Chapter 1 and Chapter 2.

The mobile-agent model considered in this work has also been used in several recent graph-computation problems, including maximal independent set construction [103, 98], dominating set computation [26], BFS and MST construction [69], and other distributed graph analytics tasks in anonymous networks.

4.4 Model and Problem Formulation

In this work, we use the same model as described in Section 1.1.2 with two significant distinctions. First, we use exactly n agents for our algorithm, and second, the agents start at a *dispersed* initial configuration.

4.4.1 Problem Statements

Triangle Counting using Mobile Agents: Consider an undirected, simple, connected anonymous n-node graph $G = (V, E)$ and a collection $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ of n agents, each of which is initially placed distinctly at each node of G. We solve the following problems.

(a) Node-Based Triangle Counting: To count the number of triangles incident to a given node.

(b) Edge-Based Triangle Counting: To count the number of triangles based on a given edge.

(c) Total Triangle Counting: To count the total number of triangles in the graph G.

Truss Decomposition, Triangle Centrality and Local Clustering Coefficient: Consider an undirected, simple, connected anonymous n-node graph $G = (V, E)$ and a collection $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ of n agents, each of which is initially placed distinctly at each node of G. The n autonomous agents coordinate among themselves to solve the (i) *Truss Decomposition Problem* and compute (ii) *Triangle Centrality*, and (iii) *Local Clustering Coefficient* of a given node.

In this work, we study the above problems from a theoretical perspective and aim to solve them while minimising both the number of rounds and the memory per agent as much as possible.

For easy reference, we provide a table for the list of agent variables used in this chapter, for different algorithms in Table 4.2

Before presenting the triangle counting algorithm, we briefly describe the motivation for using the dispersed configuration in this work. Triangle counting is a local subgraph problem, where each node needs information about its neighbourhood and about connections among its neighbours. A dispersed configuration is therefore very natural, because each agent can act as the representative of one node and can collect the local information needed for counting triangles involving that node. Using a dispersed configuration gives a clean starting point and allows us to focus on the main difficulty of performing triangle-based computation under local communication and node anonymity.

4.5 Triangle Counting via Mobile Agents

In this section, we develop algorithms for n mobile agents that are initially dispersed among the n nodes of the graph G to enumerate the number of triangles in G. Since the nodes themselves are memory-less and indistinguishable, the algorithm relies on the memory and IDs of the mobile agents that reside on the nodes of the graph. Also, since the agents cannot communicate among themselves (unless they are at the same node), synchronising the movement of the agents is another challenge.

In our algorithm, the agents (technically representing the nodes they are sitting at) first scan their neighbourhood. Once all the information about the neighbourhood is collected, the agents now count the number of common neighbourhoods between two adjacent agents. After each agent receives that count, the sum of each such count stored at each agent is evaluated, which, when divided by three, gives us the number of triangles in G. The algorithm runs in three phases. In the first phase, the agents learn their neighbours. In the second phase, the agents check the number of common neighbours with each of their adjacent agents. In this phase, each agent r_i also counts the number of local triangles with r_i as a node and the number of triangles with (r_i, r_j) as an edge, where r_j is an adjacent agent to r_i. In the third and last phase, each agent collects the local triangle count from every other agent and counts the number of triangles in G. We now explain each phase in detail below.

Agent Variables Used in Chapter 4		
Variable	**Values or Type**	**Description**
$r_i.ID$	Unique ID	Agent ID
$r_i.edge(r_j)$	Integer	Number of triangles containing edge (r_i, r_j).
$r_i.local\_sum$	Integer	Sum of triangle contributions collected locally by r_i.
$r_i.nbd\_list$	Neighbor Set	A variable that stores the list of all neighbours of agent r_i. The size of the $r_i.nbd\_list$ list is $\Delta \cdot \log n$ bits, since it may store up to Δ entries.
$r_i.h(e)$	Integer	Current support/h-index estimate of an edge e during truss decomposition. Its final value becomes the $trussness$ of edge e
$r_i.scheduled(e)$	TRUE / FALSE	Indicates whether edge e is scheduled for the next truss update iteration.
$r_i.change$	TRUE / FALSE	Indicates whether any edge value changed in the current truss iteration.
$r_i.N(e)$	Edge set	Stores neighbouring edges participating in triangles with edge e.
$r_i.L(e)$	Integer multiset	Stores intermediate support values used for h-index computation.
$r_i.parent$	Port / Agent ID	Parent reference
$r_i.child$	Port / Agent ID	Child reference
$r_i.level$	Integer	BFS/tree level of the agent in the spanning tree.
$r_i.visitor$	$0/1$	A variable set to 1 if agent r_i is visiting another node from its home node. When r_i is at its home node, the variable `visitor` is set to 0.
$r_i.partition$	A/B	Stores the bipartite partition of agent r_i.
$r_i.completion$	Boolean	Initially set to `false`; becomes `true` once all children of r_i report completion.
$r_i.nextport$	Port Number	Stores the next port to be explored by agent r_i.
$r_i.sibling$	Port Number	Used to reduce memory overhead while maintaining child pointers.
$r_i.treelabel$	Agent ID	Stores the identifier of the spanning tree (fragment) to which agent r_i currently belongs. Initially set to $r_i.ID$.
$r_i.leader$	Boolean	Indicates whether agent r_i is currently the leader of its spanning tree.
$r_i.BN(a_i)$	Neighbor Set	Number of neighbors involving agents (nodes) (a_i, a_j) in butterfly counting
$r_i.BN(a_i, a_j)$	Integer	Number of common neighbors involving agents (nodes) (a_i, a_j)
$r_i.BN(a_i, a_j)$	Integer	Number of butterflies involving agents (nodes) (a_i, a_j)
$r_i.BN(a_i)$	Integer	Number of butterflies involving agents (nodes) (a_i)

Table 4.2: List of agent variables used in this chapter. List of notations on Table 1.4 (Introduction).

4.5.1 Phase 1: Neighbourhood Discovery (Protocol MYN)

Each node is occupied by a distinct (via their IDs) agent before the start of the algorithm, and we represent each node of G by its stationed agent. The algorithm starts with *Phase 1*, where each agent discovers and records its neighbour. This is the same neighbour-meeting protocol introduced in Chapter 3; we revisit it here with some additional details and for ease of reference. As the IDs of the agents are unknown, agents cannot synchronously start scanning the neighbourhood. With each agent executing the same algorithm, agents may not find other agents at their exact place when they move. Therefore, we need to ensure that each gets to record the correct neighbour set for it. To do this, we exploit the ID bits of the agents. We use the fact that the IDs of the agents are distinct. To this end, we state the following lemma.

Lemma 4.1. *Let r_i and r_j be two distinct agents in $\mathcal{R}$, with their ID fields being $r_i.ID$ and $r_j.ID$ respectively. Then, there exists at least one dissimilar bit in $r_i.ID$ and $r_j.ID$ with one being 0 and the other being 1.*

Let λ denote the largest ID among all the n agents. Therefore, the agents use a $\log \lambda$ bit field to store the IDs. Now, to list the neighbouring agents, an agent r_i stationed at a node u does the following. Here, $\delta(u)$ denotes the degree of a node u and Δ denotes the maximum degree of a node in G.

1. For $\log \lambda$ rounds r_i executes the following.
   (a) r_i checks the **current ID bit** in its ID from the right. (At the start, the **current ID bit** is the rightmost bit in the ID field).
   (b) In the next 2Δ rounds, r_i chooses to do one of the following two:
      - If the **current ID bit** is 0, r_i waits at its own node for 2Δ rounds.
      - If the **current ID bit** is 1, r_i visits each of its neighbour and back using *port number* 0 till *port* $\delta(u) - 1$. When r_i meets a new agent r_k, it checks r_k's **current ID bit**. If the **current ID bit** of r_k is 0, it adds r_k and records it to r_i's neighbours list. (This is to ensure that r_k is the original neighbour of r_i and r_k is not an exploratory agent from a different neighbouring node). The agent r_k also simultaneously records r_i as its neighbour as well. The agents ignore any agent that has already been registered.
   (c) After the 2Δ rounds have elapsed, the next left bit in the ID field becomes the **current ID bit**. If no more bits are remaining and $\log \lambda$ rounds have not been completed

(implying that r_i has a smaller ID length than $\log \lambda$ bits), r_i assumes the current bit as 0 and stays back at its own node for the rest of the algorithm.

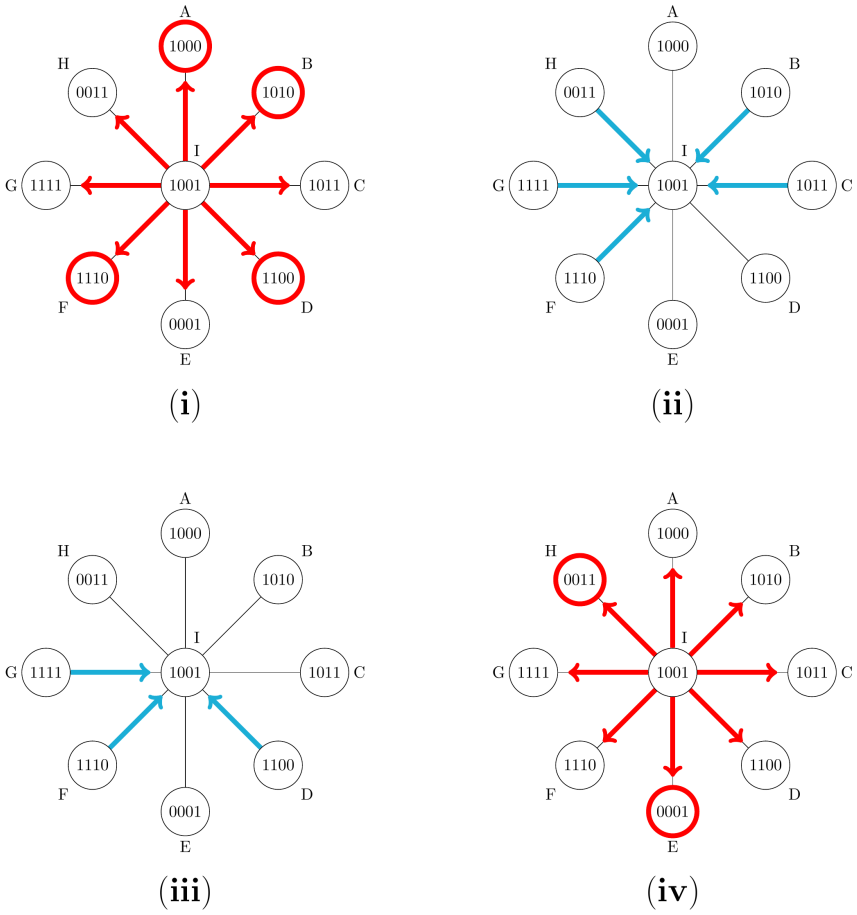


Figure 4-1: Illustration of the neighbourhood meeting protocol used in Phase 1 of the triangle counting algorithm. The figure shows one representative bit-position of the agents' IDs around the central agent I, whose ID is 1001. In each 2Δ-round block, agents whose current ID bit is 1 visit all their neighbouring nodes one by one and return to their original node, while agents whose current ID bit is 0 remain stationary. Since any two distinct IDs differ in at least one bit position, every adjacent pair of agents eventually has a phase in which one agent moves and the other waits. This guarantees that the two agents meet and record each other as neighbours. Red arrows indicate outward visits by active agents, while blue arrows indicate meetings with stationary agents at the central node. The four subfigures illustrate different bit positions of the IDs and show how different subsets of agents become active or passive across the phases.

After the completion of Phase 1 of our algorithm, it is guaranteed that each agent correctly

records all its neighbouring agents in $O(\Delta \cdot \log \lambda)$ rounds, which we prove in the following lemma.

Lemma 4.2. *Each agent r_i correctly records the exhaustive list of its neighbour agents in $O(\Delta \cdot \log \lambda)$ rounds after the completion of Phase 1.*

Proof. Let r_j (placed at node v) be a neighbour of r_i (at node u). Then there exists a port joining u to v having a port number between $[0, \delta(u)-1]$. Since r_i and r_j have distinct IDs, there exists a bit (say the p^{th} bit from the right) in $r_i.ID$ and $r_j.ID$, which are different from one another (one bit being 0, the other being 1)[Lemma 4.1]. Without the loss of generality, let us assume that the p^{th} bit of r_i is 1 and the p^{th} bit of r_j is 0. Therefore, when the p^{th} bit becomes the **current ID bit**, r_i starts exploring its neighbours one by one, during which it finds the agent r_j now stationary at v with **current ID bit** 0. Therefore, r_i records r_j as its neighbour (simultaneously, r_j also registers r_i in its neighbour list). In a similar way, r_i records all of its neighbours exhaustively either by visiting a stationary agent in its neighbourhood or by meeting an exploratory agent from its neighbour. Phase 1 runs for $\log \lambda$ rounds with each round consisting of $O(\Delta)$ sub-rounds to allow the agents to visit each neighbour. Therefore, Phase 1 completes in $O(\Delta \cdot \log \lambda)$ rounds. □

Therefore, with the end of Phase 1, each agent has now enlisted its neighbouring agents in its memory. We call this particular step Protocol MYN, a protocol for *Meeting-Your-Neighbour*. The algorithm now moves to Phase 2, where each agent r_i counts the number of triangles with r_i as one of its nodes.

4.5.2 Phase 2: Local Triangle Counting

In Phase 2, each agent, now equipped with the list of its neighbouring agents, visits each of its neighbours once again, in exactly the same manner as described in Phase 1. Whenever the agent r_i meets its neighbour r_j, they communicate to find out the number of common neighbours they both have. Since the nodes are anonymous, they are identified using the mobile agents that reside on the nodes. This communication is also used to update the following variables of r_i:

1. $r_i.edge(r_j)$: A variable which stores the count of the number of common neighbours of r_i and r_j, which represents the number of triangles based on the edge containing (r_i, r_j). r_i has Δ such variables $r_i.edge(x)$, where x represents another agent that has an edge with r_i. This, therefore, requires an additional $O(\Delta \cdot \log n)$ memory per agent, which forms the dominant factor in the memory requirement. Each of these variables is initially set to 0. A variable $r_i.edge(x)$ remains 0 if there are no triangles with (r_i, x) as an edge.

2. $r_i.local\_sum$: Initially set to 0, adds up the counts of the common neighbours for each distinct neighbour of r_i. As r_i finds its neighbours one by one, it cumulatively adds up the count of the number of common neighbours with each of its neighbours. Mathematically, $r_i.local\_sum = \Sigma_j r_i.edge(r_j)$, where the sum runs over every neighbouring agent r_j of r_i.

In the given window of $\log \lambda$ rounds (each round containing a Δ sub-round), as the agents communicate with their neighbours, the variables $r_i.edge(\cdot)$ are updated. At the end of this phase, r_i also builds on the count of the variable $r_i.local\_sum = \Sigma_j r_i.edge(r_j)$. To get the correct number of triangles with r_i as one of the nodes (local triangle counting), we divide $r_i.local\_sum$ by 2. We state the reason for the same in the following lemma.

Lemma 4.3. *The number of triangles with agent r_i as a node is given by $\frac{1}{2}(r_i.local\_sum)$.*

Proof. First, we observe that if the agents r_i and r_j have a common neighbour r_k, then (r_i, r_j, r_k) form a triangle with r_i as one of the nodes. Now, the triangle (r_i, r_j, r_k) has been tallied once while counting the common neighbour of r_i with r_j and re-counted again as a common neighbour of r_i with r_k. Therefore, each (r_i, r_j, r_k) with r_i as node, is counted twice, once through r_j and the other time through r_k. So, the number of triangles with agent r_i as a node is exactly half of $r_i.local\_sum$. □

Thus, at the end of Phase 2, each agent r_i gets a local count of the number of triangles with r_i as a node and a list of the number of triangles that are based on the edges adjacent to r_i. In the next phase, the algorithm determines the number of triangles of G using the counts generated in this phase.

4.5.3 Phase 3: Counting the Number of Triangles in G

To find the number of triangles in G, we need to take into account the local triangle counts of each agent. First, we need to accumulate the $r_i.local\_sum$ values of each of the n agents and calculate the triangles of G from there. However, due to the absence of a $leader$ agent and the difficulty of synchronising the movement of the agents, gathering the $r_i.local\_sum$ is not straightforward. The high-level idea is to communicate repeatedly with the neighbours and continually gather up the values of $local\_sum$ of each agent, hop by hop, until each agent gets every $local\_sum$ value.

As described previously, the agents can reliably communicate with all their neighbours in $\Delta \cdot \log \lambda$ rounds. During the first $\Delta \cdot \log \lambda$ round of this phase, the agents communicate with their neighbours and exchange their $local\_sum$ values. An agent r_i, along with its own $r_i.local\_sum$,

collects $r_j.local\_sum$ values from each of its neighbours r_j and stores them in memory. During the next $\Delta \cdot \log \lambda$ rounds, r_i meets its neighbour agents r_j again to check if they have collected any new $local\_sum$ values (possibly from their own neighbour). r_i again stores any new $local\_sum$ record (of its neighbour's neighbour) that it receives through its neighbour r_j during this phase. By the end of the second phase, r_i has the information of the $local\_sum$ of all the agents that are at a distance of $2 - hops$ from it. Now, with the completion of D (the diameter of the graph G, which is known to the agents), such $\Delta \cdot \log \lambda$ rounds, each agent r_i now has the $local\_sum$ record of every agent in G. The number of triangles in G is finally calculated by each agent by summing up the n $local\_sum$ values that they have gathered and dividing them by 6.

Lemma 4.4. *Let r_i and r_k be two agents at a distance of d hops from each other. Then, the value of $r_k.local\_sum$ can be communicated to r_j within $d \cdot O(\Delta \cdot \log \lambda)$ rounds.*

Proof. Since r_i and r_k are at a distance of d hops from each other, there exists a sequence of agents (nodes) $(r_i, r_{i_1}, r_{i_2}, \ldots, r_{i_{d-1}}, r_k)$ from r_i to r_k. In the first $O(\Delta \cdot \log \lambda)$ rounds, r_i can communicate with r_{i_1} to get the value of $r_{i_1}.local\_sum$. Meanwhile, r_{i_1} also collects the value of $r_{i_2}.local\_sum$ during the same $O(\Delta \cdot \log \lambda)$ round. Therefore, in the second sub-phase of $O(\Delta \cdot \log \lambda)$ rounds, when r_i communicates with r_{i_1} again, r_i receives the value of $r_{i_2}.local\_sum$ through r_{i_1}. Continuing in a similar way, r_i is guaranteed to receive the value of $r_k.local\_sum$ from r_k at a distance of d hops through the agents $r_{i_1}, r_{i_2}, \ldots, r_{i_{d-1}}$ by the end of $d \cdot O(\Delta \cdot \log \lambda)$ rounds. □

Let D denote the diameter of the graph G. Since any two agents in G are at most D hops apart, the following lemma follows from Lemma 4.4.

Lemma 4.5. *By the end of Phase 3, which takes $O(D \cdot \Delta \cdot \log \lambda)$ rounds, each agent r_i has the complete record of $local\_sum$ values of every agent in G.*

Lemma 4.6. *The number of triangles in graph G is given by $\frac{1}{6} \cdot \Sigma_i(r_i.local\_sum)$, where the sum runs over all the n agents of G.*

Proof. The number of triangles with agent r_i as a node is given by $\frac{1}{2} \cdot r_i.local\_sum$. Now, since each triangle is counted once for every node it has, the total number of triangles in G is given by $\frac{1}{3} \cdot \Sigma_i \frac{1}{2} \cdot (r_i.local\_sum)$. □

Notation: We shall denote the number of triangles containing the node v and the total number of triangles in G with $\mathbf{T}(v)$ and $\mathbf{T}(G)$, respectively. For a node v with an agent r_i on it, we use r_i and v interchangeably to denote the node.

At the end of this phase, each agent has the value of the number of triangles in G, i.e., $\mathbf{T}(G)$. In the following theorem, we assemble the list of results we obtained during the 3 phases.

Theorem 4.1. *Let G be an n-node arbitrary, simple, connected graph with a maximum degree Δ and diameter D. Let n mobile agents with distinct IDs in the range $[0, n^c]$ with the highest agent ID $\lambda \in [0, n^c]$, where c is a constant, be placed at each of the n nodes of G in a dispersed initial configuration. Then,*

1. *Each agent r_i can calculate the number of triangles with r_i as a node i.e., $\mathbf{T}(r_i)$ in $O(\Delta \cdot \log \lambda)$ rounds.*

2. *Each agent r_i can calculate the number of triangles based on each of its adjacent edges in $O(\Delta \cdot \log \lambda)$ rounds.*

3. *Each agent r_i can calculate the number of triangles in G, $\mathbf{T}(G)$, in $O(D \cdot \Delta \cdot \log \lambda)$ rounds,*

using $O(\Delta \log n)$ bits of memory per agent.

4.6 Applications: Truss Decomposition, Triangle Centrality and Local Clustering Coefficient

In this section, we show some applications of our triangle-counting algorithm via mobile agents in an anonymous graph, namely in solving the *truss decomposition* problem and computing the *triangle centrality* metric and *local clustering coefficient* for each node of the graph. Truss decomposition and clustering coefficient calculations using mobile agents offer significant advantages in environments where decentralised and adaptive operations are crucial. In scenarios with limited or unreliable communication infrastructure, mobile agents can perform local computations, reducing the need for extensive data transmission. This capability is especially valuable in temporary or ad hoc networks, such as those formed during disaster response, where agents can quickly analyse the network's structure to identify robust sub-graphs or highly connected areas, ensuring that resources are efficiently allocated. For instance, drones operating in disaster zones can perform truss decomposition to pinpoint areas of strong connectivity, directing rescue efforts where the network of survivors or critical infrastructure is most resilient. In urban environments, mobile agents representing vehicles can compute clustering coefficients to identify highly interconnected regions of a city, enabling more effective traffic management and resource

distribution. In the military context, UAVs can employ triangle counting to reveal strategic communication hubs, improving surveillance and ensuring critical areas are secured against potential disruptions.

4.6.1 Truss Decomposition

We first use the techniques from Section 4.5 to identify a $k-truss$ sub-graph of a given graph G using mobile agents, when it exists. A sub-graph T_k of G is called a $k-truss$ if every edge of T_k is a part of at least $k-2$ triangles, i.e., each edge in $T_k \subset G$ is supported by at least $k-2$ triangles in T_k.

In our algorithm, we use the popular *truss decomposition* method to first calculate what is called the $trussness$ value for each edge. The $trussness$ values are then used to construct the $k-truss$ sub-graph for any value of k.

Before we explain our algorithms in detail, we formally define $support$, k-truss and $trussness$.

Definition 4.1 ($support$). *For a given graph $G(V,E)$, the $support$ of an edge $e \in E$ is the number of $triangles$ in G that contain e.*

Definition 4.2 (k-truss). *$k-truss$ is defined as the largest sub-graph T_k of $G(V,E)$ in which every edge has $support \geq k-2$ with respect to T_k. In case T_k is a $null$ graph, we say k-truss for G does not exist.*

Definition 4.3 ($trussness$). *The $trussness$ of an edge e is defined as the maximum k such that e belongs to T_k but does not belong to T_{k+1}.*

Algorithm for Truss Decomposition: We now propose an algorithm for mobile agents to evaluate $trussness$ for each edge of the graph $G(V,E)$. The algorithm is based on the parallel truss decomposition described in [131]. The $trussness$ values determine a partition (thus an equivalence relation) on E, where each class has the edges of G with the same $trussness$ value. Let $\mathbf{t_k}$ denote the equivalence class of edges having $trussness = t_k$. To find the k-truss, T_k we construct a sub-graph of G with edges from the equivalence classes $\mathbf{t_k}, \mathbf{t_{k+1}}, \ldots, \mathbf{t_{max}}$, where t_{max} is the maximum $trussness$ of any edge in G. Mathematically, $T_k = \cup_{i=k}^{t_{max}} \mathbf{t_i}$. Therefore, by computing the trussness for each edge in $G(V,E)$, we obtain a partition (equivalence classes) of E, thereby obtaining the k-trusses of G for any k by taking the union of the equivalent classes. Therefore, the k-truss decomposition of a graph is equivalent to computing the $trussness$ of each edge in the graph.

$trussness$ values of each edge in G can be computed efficiently using existing serial and parallel algorithms. The serial truss decomposition algorithm first calculates $support$ for each edge and iteratively removes a single edge each time until all the edges of the graph are removed. Each time, the edges are sorted in ascending order, and the edge with the lowest $support$ is removed. The removed edge (say e) keeps its final $support + 2$ as its $trussness$ value. Once e is removed, the $support$ value of the edges that formed the triangle with e is re-evaluated. The remaining edges are once again sorted in order before running another pass. The need to sort the edges in order of $support$ makes the algorithm inherently sequential.

In the parallel version of the algorithm, the sorting condition is relaxed with the use of $h - index$ upgradation. For a set of integers S, the $h - index$ of S is defined as the largest number h such that there are at least h elements in S that are equal to or greater than h. The $trussness$ of an edge is related to the $h - index$, for the $trussness$ of an edge e can be thought of as the largest k such that it is contained in at least k triangles whose edges have a $trussness$ value of at least k. In the algorithm, each edge e in G is initialised to its $support$ as the first approximation to its $trussness$. Now, the $support$ values of all triangles with e as an edge are stored in a set L, and its $h - index$ is computed. At each iteration, the $h - index$ of e is updated to the smallest of its current value and the $h - index$ of L. The algorithm iteratively updates an edge's $h - index$ by computing the $h-index$ of all edges that support it, until achieving convergence when no updates would happen. The final $h - index$ of each edge before no further updates happen, provides the $trussness$ value for each edge. We give below the pseudocode of the parallel version of the truss decomposition algorithm from [131].

Before presenting the algorithm for mobile agents, we first illustrate its execution using the example in Fig. 4-2. For the graph G, the edge set is $E = \{AB, AD, AE, BC, BD, BE, CD, CF, DE\}$. Let $h(e)$ denote the current support of an edge e, and let $scheduled(e)$ indicate whether e is marked for evaluation in the next iteration. The variable $updated$ records whether the $h(\cdot)$ value of any edge changes during an iteration. Two additional sets, N and L, are used as local storage: N stores the edges that, together with e, form a triangle, and for each such triangle (e, e', e''), the value $\rho = \min\{h(e'), h(e'')\}$ is inserted into L. Finally, H denotes the h-index of the multiset L.

From the figure, the initial supports are as follows: $h(BD) = 3$; $h(AB) = h(AD) = h(AE) = h(BE) = h(DE) = 2$; $h(BC) = h(CD) = 1$; $h(CF) = 0$. Every edge is initially scheduled, i.e., $scheduled =$ TRUE. Since $updated =$ TRUE, the algorithm enters the while loop (Lines 6–31). For each edge (Lines 8–30), the algorithm constructs N and L. We illustrate this process for two edges, AE and BD.

Algorithm 8 PARALLEL K-TRUSS DECOMPOSITION (FROM [131])

```
1: function k-Truss-Parallel(G, support)
2:     for each edge e ∈ E do
3:         h[e] ← support[e], scheduled[e] ← TRUE
4:     end for
5:     updated ← TRUE                                  ▷ TRUE if any h[e] is updated
6:     while updated do
7:         updated ← FALSE
8:         for each edge e ∈ E in parallel do
9:             if scheduled[e] = FALSE then
10:                continue
11:            end if
12:            L ← ∅, N ← ∅
13:            for each △ contains e do
14:                e', e'' ← the two edges in △ other than e
15:                N.add(e'), N.add(e'')
16:                ρ ← min{h[e'], h[e'']}
17:                L.add(ρ)
18:            end for
19:            H ← h-index of L
20:            if h[e] ≠ H then
21:                updated ← TRUE
22:                for each edge e_N ∈ N do
23:                    if H < h[e_N] ≤ h[e] then
24:                        scheduled[e_N] ← TRUE
25:                    end if
26:                end for
27:                h[e] ← H
28:            end if
29:            scheduled[e] ← FALSE
30:        end for
31:    end while
32:    for each edge e ∈ E do
33:        trussness[e] ← h[e] + 2
34:    end for
35:    return trussness
36: end function
```

Edge AE. In the first triangle containing AE i.e., (AE, AB, BE), we have $N = \{AB, BE\}$ and $\rho = \min\{h(AB), h(BE)\} = \min\{2, 2\} = 2$, giving $L = \{2\}$. For triangle (AE, AD, DE), we update $N = \{AB, BE, AD, DE\}$ and add $\rho = \min\{h(AD), h(DE)\} = \min\{2, 2\} = 2$,

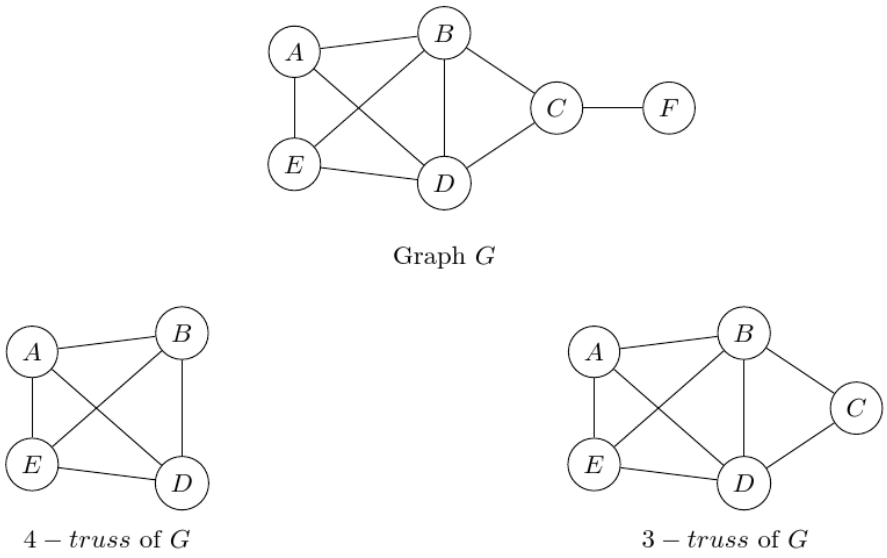


Figure 4-2: The figure illustrates $4-truss$ and $3-truss$ of the graph G, consisting of 6 nodes and 9 edges. By definition, a k-truss is a maximal subgraph of G in which every edge participates in at least $k-2$ triangles (i.e., each edge has $support \geq k-2$).

resulting in $L = \{2, 2\}$. Thus, after all triangles containing AE are processed, we compute $H(AE) = h\text{-index}(L) = 2$. In parallel, the edge BD is also processed.

Edge BD. First triangle (BD, BC, CD): $N = \{BC, CD\}$, $\rho = \min\{h(BC), h(CD)\} = \min\{1, 1\} = 1$, $L = \{1\}$. Second triangle (BD, AB, AD): $N = \{BC, CD, AB, AD\}$, $\rho = \min\{h(AB), h(AD)\} = \min\{2, 2\} = 2$, $L = \{1, 2\}$. Third triangle (BD, BE, DE): $N = \{BC, CD, AB, AD, BE, DE\}$, $\rho = \min\{h(BE), h(DE)\} = \min\{2, 2\} = 2$, $L = \{1, 2, 2\}$. Hence, $H(BD) = h\text{-index}(L) = 2$, decreasing from its initial value of 3.

Since at least one edge (here, BD) satisfies the condition in Line 23, it is scheduled for update. Specifically, as $2 < h(BD) = 3 \leq 3$, the h value of BD is updated to 2 (Line 27). In the next iteration of the while loop (Lines 6–31), no further updates occur, and the loop terminates. Finally, the trussness of each edge is set (Line 33), yielding:

$trussness(E) = \{AB : 4,\ AD : 4,\ AE : 4,\ BC : 3,\ BD : 4,\ BE : 4,\ CD : 3,\ CF : 0,\ DE : 4\}$.

Thus, the 3-truss is given by the union $\bigcup\{AB, AD, AE, BC, BD, BE, CD, DE\}$, and the 4-truss is $\bigcup\{AB, AD, AE, BD, BE, DE\}$, as shown in Fig. 4-2.

However, mobile agents bring unique challenges. The agents do not have any global knowledge of the topology of the network and can only communicate with other agents if they are

located in the same node (or they are in sufficient proximity). Further, the movement of the agents, although synchronous, does not have any centralised control. These limitations pose unique challenges in constructing algorithms for truss decomposition. We, therefore, engineer the existing parallel version [131] of the truss decomposition algorithm to adapt to our mobile agent model.

To start with, each of the n agents, r_i has the following variables:

1. $r_i.ID$ - stores the ID string of the agent r_i. As per our assumption, agents have ID in the range $[0, n^c]$, where c is an arbitrary but fixed constant, so $r_i.ID$ requires $O(\log n)$ bits of storage space.

2. $r_i.nbd\_list$: A variable which stores the list of all neighbours of agent r_i. The size of nbd_list is $\Delta \cdot \log n$ bits as it may store up to Δ entries in the worst case.

3. $r_i.edge\_set$ - to store the edges incident on a particular node containing r_i. r_i stores only edges (r_i, r_j), where $r_j > r_i$. $edge\_set$ requires $O(\Delta \log n)$ bit of memory, where Δ is the maximum degree of a node in G. The ordering of edges ensures that each edge becomes associated with exactly one agent. The agent r_i creates the list $r_i.edge\_set$ from the record of $r_i.nbd\_list$.

4. h - a variable associated with each edge in $edge\_set$; initially stores the $support$ of each edge. h is next updated according to the $h - index$ values as the algorithm progresses, as described earlier. The final h values provide the $trussness$ for a particular edge.

5. L - also associated with each edge in $edge\_set$. L is used to store the values $\min\{h(e'), h(e'')\}$ for each triangle (e, e', e'') containing e.

6. N - is used to temporarily store the edges e', e'' for each triangle (e, e', e'') containing e.

7. $scheduled$ - to identify the edges in the $edge\_set$ that need further updating in subsequent rounds of the algorithm. Initially, every edge in $edge\_set$ is scheduled ($true$).

8. $r_i.change$ - initially assigned 0. If no more edges in $r_i.edge\_set$ are scheduled, $r_i.change$ becomes 1.

Our algorithm for mobile agents runs in phases as described below, in detail.

Phase 1 (Protocol MYN)

With each node of G hosting a distinct agent, each agent first visits and registers its list of neighbours. Due to a lack of movement synchronisation, the agents follow the exact protocol as described in Phase 1 of Section 4.6.1 to register the neighbours, the ports that lead to the respective neighbours and the respective edges. To avoid inconsistency, the agents maintain the record of the edges in a specific order. An agent r_i only stores the record of an edge (r_i, r_j) if r_j is a neighbour of r_i and $r_j.ID > r_i.ID$. This gives a total ordering on the edges. This phase takes $O(\Delta \cdot \log \lambda)$ rounds to complete. Now the agents begin the next phase of calculating $support$ for each edge. With the execution of Phase 1, $edge\_set$ gets recorded.

Phase 2 (Initialisation)

The agents now calculate the support for each edge. To do this, the agents use a similar methodology to the one used in Phase 2 of Section 4.6.1. The agents go to each of their neighbours once again. Let us assume, at some point in time during the algorithm, an agent r_i meets its neighbour r_j. r_i then checks the number of common neighbours with r_j and records the count into $r_i.support(r_i, r_j)$ (we have assumed that $r_j.ID > r_i.ID$; otherwise if $r_j.ID < r_i.ID$, then the edge (r_i, r_j) (simultaneously, $r_i.support(r_i, r_j)$) does not register in r_i's records and in such a case, the agent r_i does nothing. However, in such a case, the edge does register in r_j's record, and r_j then does the needful in that respect.). Support calculation takes $O(\Delta \log \lambda)$ rounds, after which, each agent receives its $support$ number. With the execution of this, the h value for each edge in $r_i.edge\_set$ gets initialised to the support of the particular edge.

Phase 3 (Upgradation)

With each edge receiving its initial $support$ value, the algorithm now updates h for each edge through multiple iterations of this phase. The phase iterates till the $change$ value in any of the agents r_i remains 0 and terminates once the value of $change$ in each and every agent becomes 1. In each phase, each agent r_i does the following in parallel. Each step of the iteration requires a specific number of rounds. Since the agents know the parameters Δ, λ, and D, every step is executed within calculated, well-defined communication rounds.

1. Checks if any edge is scheduled for further up-gradation in its $edge\_set$. If no, set $r_i.change \leftarrow 1$ and stop executing the algorithm, but remain active. r_i provides any in-

formation when required to other agents and also takes part in "Checking for Termination (Step 6)". This step of computation takes a single round.

2. Resets the sets L and N. This step of computation also takes a single round.

3. For each edge e in $edge\_set$ which is scheduled for up-gradation, L stores $\min\{h(e'), h(e'')\}$ for each triangle (e, e', e'') in G. To obtain $\min\{h(e'), h(e'')\}$, r_i must communicate with its neighbouring agent. For example, consider the edge (r_i, r_j) with $r_i < r_j$, stored in $r_i.edge\_set$. Also, consider another agent $r_p (r_p < r_i < r_j)$ stationed on another node such that (r_i, r_j, r_p) forms a triangle. Let $e = (r_i, r_j), e' = (r_p, r_i), e'' = (r_p, r_j)$. Now for r_i to calculate $\min\{h(e'), h(e'')\}$, r_i must visit r_p because these edges are stored in the $edge\_set$ of r_p. Therefore, r_i must execute "Protocol MYN" (Phase 1, Sec. 4.5.1) which takes additional $O(\Delta \log \lambda)$ rounds, in order to guarantee that r_i meets r_p so that the required values of $h(e'), h(e'')$ can be obtained. As r_i meets its neighbours, along with the set L, the set N is also updated. Now for $e = (r_i, r_j)$ and for each triangle (e, e', e'') the agents add e', e'' to their respective N sets. As mentioned before, since each edge, e' or e'', is associated with a unique agent, an invocation of "Protocol MYN" (Phase 1, Sec. 4.5.1) is required to inform the correct agent associated with a particular edge so that the agent can schedule the particular edge in the next step if required. So, this step requires $2\Delta \log \lambda$ rounds.

4. r_i now computes the $h - index$ for the set L. If the calculated $h - index$ is less than $h(e)$, then e along with the edges $e' \in N$ with $h - index(L) < h(e') \leq h(e)$ is further scheduled for up-gradation in the next iteration and $h(e)$ is updated to the new $h - index$. Otherwise, e is unscheduled, and its current h value gives the $trussness$ at the end of the algorithm (if not updated further during the algorithm). This step of computation also takes a single round.

5. Based on whether there are scheduled edges in $r_i.edge\_set$, $r_i.change$ updates its value if necessary. This step of computation also takes a single round.

6. Checking for Termination: To determine whether the algorithm is ready for termination, each agent first evaluates the $change$ value for the current phase. Each agent then communicates with all other agents to collect their $change$ values and compute the binary product of these n values. This process requires the application of Protocol MYN repeatedly to ensure that every $change$ value is propagated to all n agents. Given the importance of this step, which may be performed multiple times during iterations, we first establish a communication framework to propagate the binary product across all n agents effectively.

To achieve this, the agents construct a BFS tree of the graph, rooted at the node with the minimum ID agent. This BFS tree construction step is a self-independent one-time process and can be completed either just before the first iteration or at the beginning of the decomposition algorithm. During the iterations, if the binary product of the $change$ values results in 1 (indicating that all agents have the same binary product value), the agents terminate the algorithm. If not, the agents repeat Phase 3 until the product equals 1. Details of the BFS tree construction mechanism are provided in the subsequent section.

Creating a BFS Tree for Communication

As described earlier, to ensure the termination of the *Truss Decomposition* algorithm, the n agents must communicate to verify that no further edges are scheduled for upgrading and that no additional iterations are required. This consensus information must be disseminated across all agents to confirm the completion of the algorithm. However, due to the constraints of local communication, propagating such information globally becomes the most time-intensive part of the algorithm.

To achieve termination through a global consensus, the agents must repeatedly execute Protocol MYN for at least D rounds, where D is the graph's diameter, each time an edge is flagged for further upgrading. The algorithm terminates only after confirming that all agents reach the state $change = 1$, indicating that no further computation is needed. This process intuitively adds a multiplicative factor of at least $O(D\Delta \log \lambda)$ rounds to the time complexity for each iteration (at-most m rounds in the worst case), where m is the number of edges.

The complexity of this process can be significantly reduced if, instead of applying Protocol MYN repeatedly, the agents construct a BFS tree rooted at the minimum ID agent, allowing for more efficient communication across the network.

In this subsection, we present an algorithm that constructs a BFS tree for the graph G, rooted at the agent with the minimum ID, denoted as $r^\star$. The primary purpose of this BFS tree is to compute a global value $x = f(x_1, x_2, x_3, \ldots, x_n)$ from the individual values x_i provided by each agent r_i, and then disseminate the computed value x to all agents in the network in minimum amounts of time. To facilitate the construction of the BFS tree, each agent is equipped with the following variables.

- $r_i.parent$, $r_i.child-$ used for storing respective $child$ and $parent$ pointers for r_i. We recall that each node u with degree δ_u has a local port-numbering $= \{0, 1, \ldots, \delta_u - 1\}$.

The $r_i.parent$ pointer stores the port number that connects u to its parent agent. Similarly, the $child$ pointers store the port numbers that connect the node u to its children. An agent has at most Δ child pointers. Initially, $r_i.parent, r_i.child = \perp$. The child pointers use $O(\Delta \log \Delta)$ bits overall, which is absorbed in the overall memory requirement of $O(\Delta \log n)$ bits for the truss decomposition.

- $r_i.level-$ denotes the level assigned to an agent relative to the tree that r_i is part of. Initially, $r_i.level = -1$
- $r_i.visitor$ - a variable set to 1, if r_i is a visitor to another node from its own node. When r_i is at its home node, the $visitor$ value is 0.

The algorithm starts by electing a leader agent, which then coordinates the construction of a BFS tree of the graph G. After the leader is chosen, the agents proceed to build the BFS tree incrementally, level by level. To ensure proper synchronisation between levels, the agents utilise the knowledge Δ, the maximum degree of the graph. The algorithm operates over $O(D\Delta)$ rounds, with each level of the BFS tree assigned 2Δ rounds for completion.

Step 1: Electing a Leader

To elect a leader, each agent r_i first sends its ID to its neighbours. Each agent now computes the minimum of the IDs received from neighbouring agents and sends this minimum ID to its neighbours. Each round of communication with neighbours is performed using Protocol MYN. This process continues for D rounds, with each communication requiring $O(\Delta \log \lambda)$ rounds. So, to elect a leader, r_i does the following steps for D rounds.

1. r_i calculates the minimum IDs it has received from the neighbours.
2. r_i sends this minimum ID to its neighbours with one execution of Protocol MYN.

Using Lemma 4.4, we can argue that after d steps of communication, every agent r_i, now has the minimum ID of all the agents in its $d - hop$ neighbourhood. Since the diameter of the graph is D, we have the following lemma.

Lemma 4.7. *By the end of $O(D \cdot \Delta \log \lambda)$ rounds, every agent has the knowledge of the least ID agent in the network.*

The agents are now aware of the leader agent. We denote this agent as $r^\star$. $r^\star$ now initiates the construction of the BFS tree rooted at $r^\star$.

Step 2: Construction of BFS Tree

The leader agent $r^\star$ begins constructing a BFS tree, with itself as the root. The process proceeds level by level. First, $r^\star$ marks itself as level 0, initiating the algorithm by visiting all its neighbouring agents one by one. As $r^\star$ visits each neighbour, it marks them as its children, and these newly marked children also mark $r^\star$ as their parent simultaneously.

To explore the graph in levels, agents use the variable $level$. When $r^\star$ identifies its children (and they, in turn, recognise $r^\star$ as their parent), these agents update their level to $r^\star.level+1$. The children of $r^\star$ then begin exploring the next level in search of their own children. However, before they proceed, the children of $r^\star$ at level 1 must ensure that all their siblings have been explored by $r^\star$. Only after this can agents at the next level begin exploring in parallel. This synchronisation is crucial to correctly constructing the BFS tree. If any child prematurely starts extending the tree before all siblings have been explored, it could result in a tree with longer-than-necessary paths, violating the BFS structure. For example, in a complete graph, if the first discovered child immediately continues exploring its own children, the resulting tree could have $n-1$ levels, whereas the correct BFS tree should only have one level. To prevent this, the agents synchronise their actions by waiting for 2Δ rounds.

Below is an outline of the algorithm for an agent r_i during rounds $2k\Delta$ to $2(k+1)\Delta$:

1. **Case 1**: $r_i.level = k$
    - r_i sets $r_i.visitor \leftarrow 1$ moves to its neighbours one by one using increasing port numbers (recall that each node has a local port numbering for every outgoing edge). Let one of its neighbours be v.
    - **Sub-case 1**: If r_i meets a single new agent r_t at v with $level = -1$ (implying the agent has not been visited before), it marks r_t as its child, and simultaneously r_t sets r_i's incoming port as its parent. Additionally, r_t sets $r_t.level \leftarrow r_i.level + 1$.
    - **Sub-case 2**: r_i finds multiple agents at v
        - **1: Original resident present** - If the original resident r_t of v (with $r_t.visitor = 0$) is at v, r_t accepts the request from the visitor agent with the lowest ID. If r_i is that agent, it updates its child pointers accordingly. The newly discovered agent sets its parent accordingly and increments its level to $r_i.level + 1$. If r_i is not selected, it returns to its original node and proceeds to the next neighbour.
        - **2: Original resident absent** - If the original resident is absent, all agents are visitors. In this case, the agent r_i returns to its original home node.

- **Sub-case 3**: If r_i finds v to be empty, r_i returns to explore other neighbourhoods. An empty node indicates that the resident agent of that particular node is at the same level as r_i.
- **Sub-case 4**: If r_i encounters an agent r_t that is already assigned a non-negative level, r_i ignores r_t and returns to its home node for further exploration.
- If all the outgoing ports (neighbours) have been visited, r_i sits at its node till the remaining time $2(k+1)\Delta$ rounds have elapsed.

2. **Case 2**: $r_i.level \neq k$: In this case, r_i does not take any action and remains idle at its current node during the 2Δ rounds. However, if it is visited by a neighbouring agent (which could be a potential parent), r_i behaves as described for r_t in the previous scenario, responding appropriately to establish the parent-child relationship.

To complete the BFS tree construction, this process runs in phases of 2Δ, a D number of times. Therefore, after the completion of $2\Delta D$ rounds, the algorithm terminates, and the BFS construction from $r^\star$ is complete. Since the number of levels in a BFS tree cannot exceed the diameter D of the graph, we have the following lemma.

Lemma 4.8 (BFS Tree Construction)**.** *The n agents construct a BFS tree for G rooted at $r^\star$ in $O(D\Delta \log \lambda)$ rounds.*

We now provide a mechanism that allows the root $r^\star$ to collect n different values x_i from each agent r_i and compute a value $x = f(x_1, x_2, \cdots, x_n)$ and then disseminate this value to all the agents in G.

1. Each agent r_i sends its own value x_i to its parent.
2. Each agent waits till it gets the array of values from all of its children, each appends its own value to the array and continues to forward this updated set of data to its parent.
3. Finally $r^\star$ after receiving all these x_i values from its children and using its own value $x^\star$ computes the function $x = f(x_1, x_2, \cdots, x_n)$
4. To send this value across the BFS tree, the agents use their $level$ number.
5. In the first Δ rounds, the agents with $level = 1$ moves to its parent ($r^\star$) to collect the value of x from $r^\star$.

6. In round t, each agent r_i with $level = t$ similarly moves up to its parent to collect the value of x.

7. Finally, within D rounds, this value of x reaches every agent in the graph.

We therefore have the following lemma.

Lemma 4.9. *Each agent can learn the value of the function $f(x_1, x_2, \ldots, x_n)$, which is computed from n inputs provided by the n agents, in $O(D)$ rounds by utilising the constructed BFS tree as the framework.*

Therefore, using the methodology above, the agents can compute the binary product of the $change$ values, which is the function $f(x_1, x_2, \ldots, x_n)$ here, with x_i representing the $change$ values of an agent r_i. In the following lemma, we now analyse the final time complexity of solving the *Truss Decomposition Problem* using the mobile agents.

Lemma 4.10. *Algorithm 4.6.1 takes $O(m\Delta \log \lambda + mD)$ rounds to terminate.*

Proof. Phase 1 and Phase 2 take $O(\Delta \log \lambda)$ rounds, as shown in Lemma 4.2 and Lemma 4.3. To elect a leader and construct the BFS tree, the agents need additional $O(D\Delta \log \lambda + D\Delta)$ rounds (Lemmas 4.7, 4.8). In each execution of Phase 3, the agent needs to communicate with its neighbour to update its $trussness(h)$ values, taking $O(\Delta \log \lambda)$ rounds. Finally, to check for termination, each agent propagates the $change$ value to every other agent, needing $O(D)$ rounds (Lemma 4.9).

At worst, the algorithm may update the $h - index$ value of one edge during each execution of Phase 3. Also, the up-gradation of the $h - index$ value of an edge is equivalent to the fact that the triangle count ($support$) of that particular edge (and possibly some other edges) in G has been decreased. Therefore, each time an $h - index$ of an edge is updated (it can only decrease), the graph G loses at least one edge. Therefore, the algorithm may need to execute Phase 3, m times before termination. Therefore Algorithm 4.6.1 takes $O(\Delta \log \lambda) + O(D\Delta \log \lambda + D\Delta) + m \cdot (O(\Delta \log \lambda) + O(D)) = O(m\Delta \log \lambda + mD)$ rounds to execute. □

Theorem 4.2. *Let G be an n-node arbitrary, simple, connected graph with a maximum degree Δ and diameter D. Let n mobile agents with distinct IDs in the range $[0, n^c]$ with the highest agent ID $\lambda \in [0, n^c]$, where c is a constant, be placed at each of the n nodes of G in a dispersed initial configuration. Then, the* Truss Decomposition Problem *for G can be solved by the mobile agents in $O(m\Delta \log \lambda + mD)$ rounds with $O(\Delta \log n)$ bits of memory per agent.*

The memory requirement for triangle counting is governed primarily by local neighbourhood information rather than by the size of the entire graph. To correctly identify and count triangles involving a node, an agent must temporarily store information about neighbouring agents and their local connections in order to perform comparisons and avoid redundant counting. Consequently, the memory usage naturally depends on the maximum degree Δ. Although this requirement is higher than the purely logarithmic memory used in some basic coordination and dispersion tasks, the agents still do not maintain the full graph structure or any substantial global representation of the network. Instead, each agent stores only the local information necessary around its own node for the ongoing computation. Furthermore, there exists a natural trade-off between memory and movement complexity: reducing memory would require agents to repeatedly perform additional to-and-fro movements and local interactions to recollect information when needed. Thus, the additional memory used in the triangle-counting framework serves to improve computational efficiency. In particular, for sparse or constant-degree graphs, the memory usage remains modest and scales only with the local neighbourhood size.

4.6.2 Triangle Centrality

The concept of *Triangle Centrality* was introduced in [22] by Paul Burkhardt. The concept may be useful for finding important nodes in a graph based on the concentration of triangles surrounding each node. An important node in triangle centrality is at the centre of many triangles, and therefore, it may be in many triangles or none at all. In this section, we employ n mobile agents, each with a distinct ID starting at each distinct node of an arbitrarily connected anonymous graph, to compute the triangle centrality for each node of the graph.

Mathematically, Triangle Centrality, $TC(v)$ of a node $v \in G$ is formulated in [22] as:

$$TC(v) = \frac{\frac{1}{3}\sum_{u \in N^+_{\mathbf{T}}(v)} \mathbf{T}(u) + \sum_{w \in N(v) \setminus N_{\mathbf{T}}(v)} \mathbf{T}(w)}{T(G)}$$

where, where $N(v)$ is the neighbourhood set of v, $N_{\mathbf{T}}(v)$ is the set of neighbours that are in triangles with v, and $N^+_{\mathbf{T}}(v)$ is the closed set that includes v. $\mathbf{T}(v)$ and $\mathbf{T}(G)$ denote the respective triangle counts based on v and total triangle count in G. Mathematically, $N(v) = \{u : (u, v) \in E(G)\}$, $N_{\mathbf{T}}(v) = \{u \in N(v) : N(u) \cap N(v) \neq \phi\}$ and $N^+_{\mathbf{T}}(v) = \{v\} \cup N_{\mathbf{T}}(v)$. Now, we outline our algorithm that computes $TC(v)$ for a node $v \in G$ using the mobile agents that run in these 3 phases:

1. **Computing $\mathbf{T}(v)$ and $\mathbf{T}(G)$** - We use the triangle counting algorithm for mobile agents

described in Section 4.5 to evaluate the number of triangles involving the node v, $\mathbf{T}(v)$ and the total number of triangles in G, $\mathbf{T}(G)$. The execution of this phase takes $O(\Delta \log \lambda) + O(D\Delta \log \lambda) = O(D\Delta \log \lambda)$ rounds [Lemma 4.3,4.6]. With the completion of this phase, the agents begin the next phase. Note that if the count of the total number of triangles is available a priori, only $T(v)$ needs to be evaluated.

2. **Computing $N(v)$, $N^+_{\mathbf{T}}(v)$ and $N_{\mathbf{T}}(v)$ - $N(v)$,** the neighbours of v can be recorded in $O(\Delta \log \lambda)$ rounds by the agents (Lemma 4.2). After we allow the agents to record their neighbourhood in the first $O(\Delta \log \lambda)$ rounds, in the subsequent $O(\Delta \log \lambda)$ rounds, the agents can now meet their neighbours once again, this time, learning about $N(u)$ for each neighbour agent u of v. Now the agent from v communicates with each $u \in N(v)$ to evaluate the set $N_{\mathbf{T}}(v)$. Similarly, the agents can also build the set $N^+_{\mathbf{T}}(v)$. Now, as the agent meets with each of its neighbours $u \in N(v)$, it cumulatively evaluates the sums $s = \sum_{u \in N^+_{\mathbf{T}}(v)}$ and $x = \sum_{u \in N(v)_{\mathbf{T}}(v)}$ simultaneously. Finally the sum $\sum_{w \in N(v) \setminus N_{\mathbf{T}}(v)} = s - x + \mathbf{T}(v)$ is evaluated. This step takes $O(\Delta \log \lambda)$ rounds.

3. **Computing $TC(v)$** - With the values of $\sum_{u \in N^+_{\mathbf{T}}(v)} \mathbf{T}(u)$, $\sum_{w \in N(v) \setminus N_{\mathbf{T}}(v)} \mathbf{T}(w)$ and $T(G)$ have now been obtained, the agent at v can now evaluate $TC(v) = \frac{\frac{1}{3}\sum_{u \in N^+_{\mathbf{T}}(v)} \mathbf{T}(u) + \sum_{w \in N(v) \setminus N_{\mathbf{T}}(v)} \mathbf{T}(w)}{T(G)}$

To this end, we have the following theorem.

Theorem 4.3. *Let G be an n-node arbitrary, simple connected graph with a maximum degree Δ and diameter D. Let n mobile agents with distinct IDs in the range $[0, n^c]$ with the highest ID λ, where c is an arbitrary constant, be placed at each n node of G in an initial dispersed configuration. Then, the* triangle centrality *of each node $v \in G$ can be calculated in $O(\Delta \log \lambda)$ rounds if $\mathbf{T}(\mathbf{G})$ is known and in $O(D\Delta \log \lambda)$ rounds, if $\mathbf{T}(G)$ is unknown. $\mathbf{T}(G)$ is the total triangle count of the graph G.*

4.6.3 Local Clustering Coefficient

The local clustering coefficient of a node in a graph is used to quantify how close its neighbours are to being a clique (complete graph), i.e., how well connected the network is around a particular node. Mathematically, the local clustering coefficient (LCC) of a node $v \in G$ can be written as $LCC(v) = \frac{T(v)}{\delta(v)(\delta(v)-1)}$,(from [106]). Here $\delta(v)$ denotes the degree of the node v. This metric can be easily calculated using mobile agents. $T(v)$ for a node v can be calculated in $O(\Delta \log \lambda)$

rounds [Lemma 4.3]. The agent that already knows $\delta(v)$ at v can now evaluate $LCC(v)$ using the formula.

Theorem 4.4. *Let G be an n-node arbitrary, simple connected graph with a maximum degree Δ and diameter D. Let n mobile agents with distinct IDs in the range $[0, n^c]$ with the highest ID λ, where c is an arbitrary constant, be placed at each n node of G in an initial dispersed configuration. Then, the* Local Clustering Coefficient *of each node $v \in G$, $LCC(v)$ can be calculated in $O(\Delta \log \lambda)$ rounds.*

We now move to the second part of the chapter, where we use a similar agent-based model for butterfly analysis in bipartite graphs.

Part B: Butterfly Counting in Bipartite Graphs

4.7 Our Contributions

In a bipartite graph, a 2×2 biclique is called a *butterfly* [4, 108]. We begin by providing an algorithm that first finds the bi-partition of the graph and its respective size. We also construct a spanning tree of the graph and elect a leader in the process. This algorithm is then used as a foundation for butterfly counting in the subsequent sections. We list our contributions below (These results are also summarised in Table 4.1):

Let $G((A, B), E)$ be an n node arbitrary, simple, connected bipartite graph with a maximum degree Δ where n mobile agents (with highest ID λ; known to all the agents) are initially placed at each of the n nodes of G in a dispersed initial configuration[1]. The agents solve the following problems:

1. **Leader Election, Spanning Tree and Partition:** The agents (i) elect a leader and construct a rooted spanning tree (ii) determine their bi-partition sets (A or B), and (ii) determine the size of sets A and B in $O(n \log \lambda)$ rounds, using $O(\log \lambda)$ bits of memory per agent.

2. **Counting Node-Based and Total Butterflies:** The agents calculate (i) the number of butterflies at each node in $O(\Delta)$ rounds, and (ii) the total number of butterflies in G in additional $O(\min\{|A|, |B|\})$ rounds using $O(\Delta \log \lambda)$ bits of memory per agent.

[1] In a *dispersed initial configuration*, the agents start with at-most one agent at each node. For other starting configurations of agents, dispersion is first achieved in $O(n)$ rounds using [70].

4.8 Model and Problem Statements

In this part, we adopt the same model used in the first part for triangle counting, except that these agents operate on a bipartite graph $G(V, E)$, where the node set V forms the bipartition $V = (A, B)$. However, for this part, we assume no global graph parameters are known to the agents. The agents are only aware of the value of λ, the highest ID among themselves.

The dispersed or organised placement of agents also plays an important role in the butterfly counting part. Butterfly counting in a bipartite graph depends on local neighbourhood information and on the ability to combine counts across the graph. When agents are distributed over the graph, they can locally represent nodes, exchange information through controlled movement, and later aggregate the computed values using a spanning tree. As in the triangle counting case, this assumption is mainly used as a natural launch pad. Other approaches with fewer agents may be possible, but they would require additional mechanisms for agents to repeatedly visit and serve multiple nodes.

4.8.1 Problem Statement

Let $G((A, B), E)$ be an n node arbitrary, anonymous, simple, connected bipartite graph with a maximum degree Δ. Let n mobile agents with distinct IDs in the range $[0, n^{O(1)}]$ be placed at each of the n nodes of G in a *dispersed* initial configuration. Let λ be the highest ID of the agents, i.e., $\lambda \in [0, n^{O(1)}]$. The agents solve the following problems.

1. *Bi-partition Identification and Partition Size Determination* - The agents determine the bi-partition to which they belong, calculate the size of each bi-partition and construct a spanning tree rooted at the agent with the smallest ID (leader), which is identified in the process.

2. *Butterfly Counting* - The agents compute the number of butterflies associated with each node (local count) and the total number of butterflies (global count) in the graph G.

4.9 Additional Related Works on Butterfly Analysis

Butterfly counting has been vastly studied in both sequential and parallel settings. Sanei-Mehri et al. [107] introduced efficient exact and approximate algorithms, while Wang et al. [125] improved

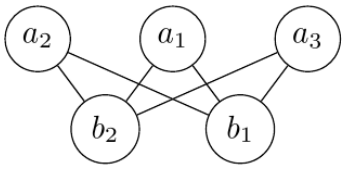


Figure 4-3: A 4-cycle in a bipartite graph represents a butterfly. Here the butterflies can be listed as: $(b_2, a_1, b_1, a_2, b_2), (b_2, a_1, b_1, a_3, b_2)$ and $(b_2, a_2, b_1, a_3, b_2)$. The number of butterflies containing a_1, a_2, a_3, b_1 and b_2 are respectively $2, 2, 2, 3$ and 3. The total number of butterflies in the graph is 3, which can also be calculated by summing the number of butterflies in each partition and dividing by 2 (i.e., $\frac{2+2+2}{2}$ or $\frac{3+3}{2}$). Additionally, the total number of butterflies can be determined by summing the butterfly counts for each node and then dividing by 4, a method we employ in this work.

counting by prioritising nodes. Zhou et al. [136] extended this to uncertain bipartite graphs. Sarıyüce and Pinar [108] introduced tip and wing decomposition for dense subgraph discovery. Parallel algorithms include Shi and Shun's [110] multicore approaches, Xu et al.'s [133] GPU algorithm, and Weng et al.'s [129] distributed methods for dynamic graphs.

4.10 Partition Assignment and Spanning Tree Construction

We assume n agents start in a *dispersed* configuration, each occupying a node in the bipartite graph $G((A, B), E)$. This section describes how they identify their partition and compute its size. A leader agent starts the algorithm by assigning itself a partition, and its children continue to explore and assign partitions to their descendants while constructing a spanning tree. When the leader is known, the tree facilitates information dissemination; otherwise, when a leader is not known a priori, the spanning tree also provides termination, as no global graph parameters are initially known. Our leader election and spanning tree construction extend to general graphs and significantly improve upon prior work [69] in a *dispersed* setting. See Section 4.10.2 for details. We first present the algorithm for the known leader case, which serves as the foundation for the no-leader version in the next section.

4.10.1 A Leader Agent is Known

Each agent maintains the following variables: $partition$ (a single-bit value, 0 or 1, with the leader $\tilde{r}$ set to 0), $parent$ (stores the port number through which the agent was first discovered), $child$ (to record the port number through which the agent has moved to an undiscovered adjacent node to identify new agent), $completion$ (a boolean, initially `false`, set to `true` when all children report completion), and $nextport$ (port number for the next exploration). A variable $sibling$ is employed to reduce memory on the $child$ pointers, which is described in Remark 2.

The algorithm starts with the leader $\tilde{r}$ setting its partition to 0 and moving through the smallest available port. Upon discovering a new agent r, it assigns r a partition of 1. In the second round, both $\tilde{r}$ and r explore their minimum unvisited ports in parallel, continuing the partition assignment and spanning tree construction. In general, when an agent r_i is visited by r_j, it inherits a partition opposite to r_j, records the discovery port as its parent, and starts exploring its neighbours. If the partition is already assigned, the visit is ignored. Meanwhile, r_j updates its child pointers accordingly. The strategy which optimizes the exploration is that all agents with assigned partitions explore their next potential children in parallel. An agent r_i explores its neighbours sequentially via r_i.`nextport`, updating it in each round. When an agent completes exploration (it sets `nextport` $= -1$) and has no children, it sends a `completion` message, node degree value and partition count (or its sum up to these children) to its parent. These messages propagate up the tree until $\tilde{r}$ receives all completions, signalling the completion of the exploration. Now, $\tilde{r}$ transmit these values down the tree in a similar way. In the first round, it sends these values to its first child. In the second round, $\tilde{r}$ and its first discovered children send their second and first children, respectively. In this way, these values reach all agents in the graph within $O(n)$ rounds. Once every agent is aware of these values, they use a more efficient down-casting approach as detailed in Section 4.11.1.

Lemma 4.11. *The algorithm in Section 4.10.1 assigns a partition to each agent in $O(n)$ rounds.*

Proof. An agent receives its partition when it is first visited by a neighbouring agent. Let us partition the set of agents into two disjoint subsets: T_t, the set of agents that have already been visited and assigned a partition after t phases (each phase consists of two rounds—one for exploration and one for return), and S_t, the set of agents yet unvisited after t phases. Without loss of generality, assume that the first k agents were discovered over l distinct phases. i.e., $|T_l| = k$. We aim to show that by the end of the $(k+1)$-th phase, at least one new agent is visited, i.e., $|T_{k+1}| \geq k+1$. For the analysis, we first show how explorations happen till the k^{th} phase, and for that, we divide the k phases into two parts: first, the l phases by which time k agents were already discovered

and next, the remaining $k - l$ phases. In the first l phases, the number of exploration attempts is at least $1 + 2 + \cdots + l = \frac{l(l+1)}{2}$. In the remaining $k - l$ phases, each of the k agents can explore one edge per phase, contributing an additional $(k-l)k$ possible explorations. Therefore, the total number of edge explorations possible within the subgraph induced by the k agents during the first k phases is

$$\frac{l(l+1)}{2} + (k-l)k.$$

Now, the minimum integral value of the expression is $\frac{k(k+1)}{2}$, which is greater than the number of edges a graph on k nodes could possibly have.

Therefore, after k phases, all internal edges among the k agents must have been explored. This shows that even if the subgraph induced by these k agents is dense, all internal edges are eventually examined. Similarly, in a sparse graph, the agents may have already exhausted all internal edges and begun discovering new agents before reaching the k-phase threshold. In either case, the process ensures that at least one new unvisited agent must be discovered during the $(k+1)$-th phase, i.e., $|T_{k+1}| \geq k + 1$. By applying this argument, we can conclude that all n agents will have received their partition assignments within n phases or $2n$ rounds, completing the proof. □

4.10.2 No Leader Agent is Known

When a leader is unknown, we first elect one. A straightforward approach is repeated communication until a leader emerges, but two challenges arise: (i) agents may fail to meet due to simultaneous movement, and (ii) termination is difficult without global parameters like n, preventing agents from knowing when all others have been reached. Previous methods that use Protocol MYN rely on parameters such as Δ (max degree) and λ (max agent ID) for coordination. Since agents meet based on ID bits rather than port order, the algorithm must allow a Δ-round window for interactions, leading to an $O(\Delta)$ round communication window for meeting full neighbours.

We propose a novel approach that enables agents to meet their neighbours one by one in a predefined order. Unlike prior methods, our technique requires no graph parameters except λ, ensuring agents meet efficiently. It helps achieve leader election and spanning tree construction from a *dispersed* configuration using only $O(\log \lambda)$ memory and completes in $O(n \log \lambda)$ rounds, significantly improving upon [69] (which elects a leader among n agents in $O(m)$ rounds using $O(n \log n)$ bits from any initial configuration). Before detailing our leader election algorithm, we introduce this meeting protocol that guarantees two agents meet within $O(\log \lambda)$ rounds.

4.10.2.1 An Improved Meeting Protocol for Two Adjacent Agents - `Meeting with Neighbor()`

In this section, we propose a deterministic protocol that guarantees a meeting between two adjacent agents within $O(\log \lambda)$ rounds. This protocol ensures that a specified agent moves to the other's location and meets its neighbour within the given time window.

Algorithm 9 MEETING WITH NEIGHBOR()

Require: Two agents, r_u and r_v, each with a unique ID, are initially located at adjacent nodes u and v, respectively. Node u is connected to v via outgoing port p_u (and enters v via p_v). Agent r_u has no knowledge of r_v or v, but wants to meet the neighbouring agent via port p_u.

Ensure: Agent r_u meets the agent located at the neighbouring node accessible via port p_u in a span of $4 \log \lambda$ rounds. ▷ A total of $2 \log \lambda$ bits and each bit corresponding to 2 rounds.

1: Pad r_u's ID with leading zeros to obtain a $\log \lambda$-bit binary string $r_u.b$
2: $r_u.new\_ID \leftarrow \overline{r_u.b} \,||\, r_u.b$ ▷ Concatenate bitwise complement of b_u with b_u; length $2 \log \lambda$
3: **for** $i = 0$ to $2 \log \lambda - 1$ **do**
4: $current\_bit \leftarrow i$+1-th least significant bit of $r_u.new\_ID$
5: **if** $current\_bit = 1$ **then**
6: Move to neighbour v via port p_u and meet with r_v (if present) ▷ Round $2i + 1$
7: Return to original node u ▷ Round $2i + 2$
8: **else**
9: Stay at node u during Rounds $2i + 1$ and $2i + 2$
10: **end if**
11: $i \leftarrow i + 1$
12: **end for**

Let agents r_u and r_v initially reside at adjacent nodes u and v, with u joining v via outgoing port p_u at u. r_u wants to meet its neighbour via p_u. Each agent pads its ID to a $\log \lambda$-bit binary string and appends its bitwise complement to the left, forming a $2 \log \lambda$-bit new_ID (Lines 1–2, Algorithm 9). The protocol runs for $4 \log \lambda$ rounds, with two rounds per bit position from LSB to MSB (Lines 3–12). For each bit, if $r_u.new\_ID$ has a 1, it moves to node v in the first round and returns in the second; otherwise, it stays at u. The protocol is local and requires no knowledge of r_v's state. An agent invokes it with any specific port it wants to take.

Since the IDs are distinct, there exists at least one bit position where one visiting agent has a 1 and the respective neighbour has a 0 in their respective new_ID, ensuring that one moves while the other stays—leading to a meeting. The concatenation with the complement is crucial: without it, simultaneous movements might prevent two agents from meeting. Consider an example where agent r_u wants to meet its neighbour r_v via a specific port p_u. Suppose $r_u.ID = 0010$ and $r_v.ID = 0110$. The first, second, and fourth bits of their IDs are identical, meaning they will

both either stay or move simultaneously — making a meeting impossible. While their third bits differ, r_u has a 0, indicating it remains stationary, and r_v has a 1, meaning it moves. However, r_v's movement is not guaranteed to be toward r_u; it could be exploring a different neighbour. Thus, even this differing bit does not ensure a meeting. By concatenating the ID with its complement, r_u's new ID becomes 11010010, and r_v's becomes 10010110, the meeting is now guaranteed. Specifically, at the 7th LSB, r_u moves while r_v remains stationary, ensuring a successful meeting.

Lemma 4.12. *Let r_u be an agent located at node u with degree δ_u. Let v be a neighbouring node of u hosting an agent r_v. Then r_u can communicate with r_v in $\log \lambda$ rounds. Furthermore, r_u can meet all its neighbours in at most $\delta_u \cdot \log \lambda$ rounds.*

Coming back to the main algorithm, the agents first elect a leader and construct a single spanning tree rooted at the leader agent. Each agent begins as a separate single-node spanning tree, considering itself as the leader. The core idea is to implement the same tree construction methodology from the algorithm in Section 4.10.1 with a crucial modification; whenever two agents meet, the one with the smaller ID dominates, causing the larger ID agent to give up on its leader status and join the smaller ID's tree. This process continues until all agents are connected under a single leader with the smallest ID.

Each agent additionally maintains a variable `treelabel`, which initially stores its own ID and helps in tracking the spanning tree to which it belongs. At the start, each agent initialises itself as a leader, assigning its own ID to the `treelabel` variable. Each agent meets its neighbours one by one using Algorithm 9. If two agents belong to different spanning trees (`treelabel` values are different), the one with the smaller `treelabel` absorbs the other. The larger ID agent updates its `treelabel` and reconfigures its `parent` and `child` pointers to join the new tree. After merging, all agents in the absorbed tree eventually update their variables, ensuring the tree structure is maintained correctly. The agent with the smallest ID (i.e., the smallest `treelabel`) continues absorbing other trees until a single spanning tree remains. Once an agent has received `completion` messages from all its children, it propagates the message upward. Eventually, when the agent with the least ID receives completion signals from all its children, the algorithm terminates. Once the leader $\tilde{r}$ is elected, the agents can now execute the algorithm in Section 4.10.1.

Lemma 4.13. *There is a unique agent r satisfying: (i) $r.leader =$ `true`, and (ii) $r.completion =$ `true`.*

Proof. First, we prove there is at least one agent that satisfies these conditions. For this, let us consider the agent with the least ID, say $\tilde{r}$. Since $\tilde{r}$ has the least ID, its tree subsumes all other trees

as it ultimately grows out of reach of all the n nodes. At this point, it starts receiving $completion$ signals from the leaf nodes, until finally, it receives these completion signals from its immediate children and terminates the algorithm by setting its own $completion = \texttt{true}$. Also, since it is never subsumed by any other tree, its $leader$ status ultimately remains to be $\texttt{true}$.

Now, we prove its uniqueness. Assume another agent r sharing the same status of variables $completion$ and $leader$ upon the termination of the algorithm. Now, since the graph is a connected one, there exists a sequence of agents $\tilde{r}, r_{i_1}, r_{i_2}, \ldots, r_{i_j}, r$. As the tree belonging to agent r moves towards $\tilde{r}$ via the sequence of agents, there must be an agent $r_{i_k}; 1 \leq k \leq j$ where the trees from $\tilde{r}$ and r meet. Now, if $\tilde{r}$ has a lower ID, ultimately, it subsumes r's tree or vice versa. Hence, one of them must give up its leadership status. Hence, there cannot be two agents with $leader = \texttt{true}$ at the end of the execution of this algorithm. □

Remark: 2 (Memory Complexity)**.** *The main memory complexity of the algorithm comes from maintaining the child pointers. Since an agent can have up to Δ children, an agent would typically require at least $O(\Delta \log \lambda)$ bits to manage them. However, we can reduce this complexity by introducing a $sibling$ variable. Whenever an agent discovers its first child, it sets its $child$ pointer to the corresponding outgoing port. Now, as it discovers the next child, it does not store it as its children; rather, it moves to its previous (first) child and stores the outgoing port of its new second child into the first $sibling$ variable and so on. This eliminates the need for using Δ child pointers for a single agent, and the $child$ pointers can be efficiently stored using $O(\log \lambda)$ bits. The variables, $parent$, $child$ and $nextport$ requires $O(\log \lambda) + O(\log \Delta) = O(\log \lambda)$ bits. While the other variables require $O(1)$ bits. Therefore, we have a memory complexity of $O(\log \lambda)$ bits.*

Theorem 4.5. *Given an arbitrary simple connected graph G with n nodes and n agents in a dispersed initial configuration. Then, a leader among the n agents can be elected, and a spanning tree rooted at the leader node can simultaneously be constructed in $O(n \log \lambda)$ rounds using $O(\log n)$ bits of memory per agent. The agents do not require any prior knowledge about any global parameters, except for λ, which is an upper bound on the IDs of the agents.*

Proof. The time complexity of the algorithm in Section 4.10.2 is dominated by the leader agent, $\tilde{r}$. Since the algorithm follows the same operations as the algorithm in Section 4.10.1 from the perspective of $\tilde{r}$, it eventually consumes all other trees, making the total time complexity dependent solely on the number of movements performed by $\tilde{r}$.

Given that each agent requires $O(\log \lambda)$ rounds to meet a neighbouring agent, and the algorithm in Section 4.10.1 runs in $O(n)$ rounds, the overall complexity is $O(n \cdot \log \lambda)$. Combining this with the memory complexity analysis from Remark 2, we establish the theorem. □

The tree construction and leader election methodology used here can also be extended to general graphs. Also, this process can produce spanning trees of large diameter even in a densely connected graph. For example, in a complete graph of n nodes, a spanning tree could be a path of length $n-1$ (outer ring), although its diameter is 1. Therefore, when two farthest agents need to communicate via the spanning tree, it can take $O(n)$ rounds instead of $O(D)$ rounds. However, in bipartite graphs, we see that the diameter of the spanning tree can only be at most $O(\min\{|A|, |B|\})$, which we note in the following remark.

Remark: 3. *The diameter of a spanning tree constructed on a bipartite graph can be at-most* $2 \cdot \min\{|A|, |B|\}$

Proof. Consider the largest path in the spanning tree and assume that it joins r_i and r_j. Now, the maximum length of this path can be $2 \cdot \min\{|A|, |B|\}$, since it must alternately connect between the agents of two partitions A and B. In addition, since it's a path, it cannot visit any node more than once. □

4.11 Butterfly Counting

In this section, we present an algorithm for counting butterflies in a bipartite graph $G((A, B), E)$. We designate the partition containing the leader (least ID agent) $\tilde{r}$ as A. The algorithm counts butterflies associated with each node in A through Phases 1 and 2. In the final phase, agents compute the total number of butterflies by aggregating individual counts from each $a_i \in A$. Before the algorithm starts, we assume that the n agents are initially dispersed and have executed the algorithm in Section 4.10.2 to identify their partition, determine its size, compute Δ, and construct the spanning tree of G.

We use the following notation: agents in A are denoted as a_i, those in B as b_i; the neighbourhood list of a_i is $BN(a_i)$; the number of common neighbours between a_i and a_j is $BN(a_i, a_j)$; the number of butterflies involving a_i and a_j is $BN(a_i, a_j)$; and the total number of butterflies containing a_i is $BN(a_i)$.

Phase 1: Each agent scans its neighbourhood and records its neighbours in memory. Before counting begins, agents must establish communication, which requires knowledge of their partition, and Δ, which is used to synchronise the movements of the agents. Communication follows alternating phases: in odd rounds, agents in A communicate while those in B remain stationary; in even rounds, vice versa. This ensures that within 2Δ rounds, all agents have exchanged information with their neighbours. Each agent requires $O(\Delta \log \lambda)$ bits of memory for this process,

since each agent a_i needs at most Δ entries for managing $BN(a_i)$ list, with each entry containing an ID of $O(\log \lambda)$ bits. This forms the dominating chunk of the memory for each agent.

Phase 2: In Phase 2, comprising of 2Δ rounds, each agent a_i counts the butterflies it shares with $a_j (\neq a_i)$. It revisits its neighbours and, for each $b_i \in BN(a_i)$, determines the number of common neighbours between itself and a_j, using b_i's recorded neighbour list $BN(b_i)$. After iterating through all b_i, a_i computes the number of butterflies involving (a_i, a_j) as

$$BN(a_i, a_j) = \begin{cases} \binom{BN(a_i,a_j)}{2}, & \text{if } BN(a_i, a_j) > 1 \\ 0, & \text{otherwise} \end{cases}$$

, sums $\sum_j BN(a_i, a_j)$, and stores this value in its memory. This gives the total butterfly count involving a_i, i.e., $BN(a_i)$.

Phase 3: After each a_i has computed its local butterfly count, it sends this count to the leader $\tilde{r}$ via the constructed spanning tree. Although each child of an agent can simultaneously transmit its value to its parent, requiring at most $\min\{|A|, |B|\}$ rounds for all values $BN(a_i)$ to reach $\tilde{r}$, the reverse process—disseminating the total count to all agents—requires a more efficient approach, which we detail below. First, $\tilde{r}$ computes the total butterfly count by summing all $BN(a_i)$ values received and dividing it by 2 using $\min\{|A|, |B|\}$ rounds.

4.11.1 Downward Communication Through the Spanning Tree

To efficiently broadcast a value down a tree, we employ the following strategy: Over the next $2\min\{|A|, |B|\}$ rounds, each child oscillates between its position and its parent, ensuring information propagation. In the first two rounds, $\tilde{r}$'s immediate children receive the value. In the next two rounds, the grandchildren of $\tilde{r}$ obtain the value, and the process continues. By the end of $2\min\{|A|, |B|\}$ rounds, all agents have received this value from $\tilde{r}$.

So, in the next $2\min\{|A|, |B|\}$ rounds, all agents have the total butterfly count. In a similar way, the algorithm can compute butterflies for each $b_i \in B$ by executing Phases 1 and 2 for agents in B.

Remark: 4. *The total number of butterflies in a bipartite graph is half the sum of the individual butterfly counts of all nodes in either A or B [124, 107].*

Lemma 4.14 (Time Complexity)**.** *The agents calculate (i) the number of butterflies based on their*

own node in $O(\Delta)$ rounds, and (ii) the total number of butterflies in G in $O(\Delta + \min\{|A|, |B|\})$ rounds.

Lemma 4.15 (Memory Analysis)**.** *The butterfly counting algorithm requires $O(\Delta \log \lambda)$ bits of memory per agent.*

Theorem 4.6 (Butterfly Counting with Agents)**.** *Let $G((A, B), E)$ be an n node arbitrary, simple, connected bipartite graph with a maximum degree Δ. Let n mobile agents with distinct IDs in the range $[0, n^{O(1)}]$ with the highest agent ID $\lambda \in [0, n^{O(1)}]$, be placed at each of the n nodes of G in a dispersed initial configuration. Then, the agents can*

1. *calculate the number of butterflies based on their own node in $O(\Delta)$ rounds, and*
2. *calculate the total number of butterflies in G in $O(\Delta + \min\{|A|, |B|\})$ rounds.*

using $O(\Delta \log \lambda)$ bits of memory per agent.

The memory used in butterfly counting is mainly governed by local neighbourhood information. Since a butterfly is a 2×2 biclique in a bipartite graph, an agent must keep enough information to compare neighbourhoods and count common-neighbour structures. Thus, the memory requirement depends on the local degree and not on the size of the entire graph. This keeps the algorithm within the low-memory spirit of the thesis, especially for sparse or bounded-degree bipartite graphs, while also reflecting the additional information required by dense subgraph computations.

4.12 Conclusion and Future Directions

In this work, we introduce and analyse novel algorithms for triangle counting within the mobile agent model over an arbitrary anonymous graph. The agents first enumerate triangles based on the nodes and edges, and subsequently calculate the total number of triangles in the graph. We subsequently apply this triangle counting methodology to develop and analyse algorithms for solving the *Truss Decomposition Problem* and for computing important metrics such as *Triangle Centrality* and *Local Clustering Coefficient* for each node.

Future research could focus on several key areas to further enhance our approach. Establishing lower bounds for both time complexity and memory usage per agent would provide valuable insights into the efficiency and limitations of our algorithms. Additionally, exploring the lower

bounds on the number of agents required to execute the algorithms could lead to more optimised and practical implementations. Another important direction is to investigate the performance of our algorithms in the presence of faulty agents, which would help in designing more robust solutions for real-world applications. Another interesting direction for future work could be to investigate scenarios where global parameters or their bounds, like the maximum degree Δ or the diameter D of the graph, are not known in advance.

We aim to extend our work on butterfly counting by reducing the number of agents required. The goal is to analyse the trade-off between agent count, time complexity, and memory usage. As a start, we look to determine whether the same problem can be efficiently solved using only $\min\{|A|, |B|\}$ agents while minimising computational overhead. In addition, we look to use these counting techniques to explore *Tip* and *Wing Decomposition* problems, which are useful in constructing a hierarchy of butterfly dense node and edge induced sub-graphs in bipartite networks.

Chapter 5

Constructing Tree Structures in Anonymous Graphs with Mobile Agents

Breadth-First Search (BFS) and Minimum Spanning Tree (MST) constructions are fundamental primitives in distributed graph algorithms. In this chapter, we study these problems in the same agent-based network, where n autonomous mobile agents operate on an anonymous n-node, m-edge graph G in synchronous rounds. The agents start from a *dispersed* configuration, with exactly one agent located at each node, and communicate only when they meet at the same node. We first present an MST construction algorithm, which improves over previous works [69, 71] and elects a unique leader agent. The algorithm runs in $O(n \log n + \Delta \log^2 n)$ rounds using $O(\log n)$ memory per agent, where Δ is the maximum degree of G. The chapter then turns to BFS construction, for which we present two algorithms. In the first setting, when a root is already known, the agents construct a BFS tree in $O(D\Delta)$ rounds using $O(\log n)$ memory per agent, where D is the diameter of G. In the second, root-free setting, a leader is first obtained through the MST construction and is then used as the root for BFS. In this case, the overall BFS construction takes $O(n \log n + \Delta \log^2 n + \min(D\Delta, m \log n))$ rounds using $O(\log n)$ memory per agent. Thus, MST construction, apart from being an independent and important tree-construction problem, also supplies the coordination needed to construct BFS without prior root knowledge. In this chapter, we assume that each agent knows λ, the maximum agent identifier and no other parameters are needed to be known to the agents a priori.

This chapter is based on the following works:

1. **Prabhat Kumar Chand, Manish Kumar and Anisur Rahaman Molla.** *Agent-Driven BFS Tree in Anonymous Graphs with Applications.* In: Proceedings of the 12th International Conference on Networked Systems (NETYS 2024), Rabat, Morocco, pp. 67–82. https://doi.org/10.1007/978-3-031-67321-4_4

2. **Prabhat Kumar Chand, Manish Kumar, and Anisur Rahaman Molla.** *Computing Tree Structures in Anonymous Graphs via Mobile Agents.* In: Proceedings of the 27th International Symposium *Stabilization, Safety, and Security of Distributed Systems (SSS 2025)*, Kathmandu, Nepal, pp. 111–127. DOI: 10.1007/978-3-032-11127-2_11.

5.1 Introduction

Tree structures are among the most fundamental primitives in graph algorithms and distributed computing. In particular, a *Breadth-First Search (BFS)* tree provides a layered view of the network that supports shortest-path discovery in unweighted graphs and enables efficient level-by-level traversal and parallelisation. A *Minimum Spanning Tree (MST)* provides a sparse backbone that is widely used for routing, aggregation, synchronisation, and as a core subroutine in numerous distributed tasks. BFS-based techniques have been used, for example, to design parallel graph procedures on large platforms [113], to support memory-efficient graph exploration [105], and to perform subgraph enumeration in directed graphs [83]. In modern autonomous systems, these structures are also valuable because they provide a structured way to coordinate decentralised entities and to reduce repeated exploration.

We study BFS and MST construction in an *agent-based* distributed computing model on an anonymous, simple, connected graph $G = (V, E)$ with $|V| = n$, $|E| = m$, maximum degree Δ, and diameter D. The nodes of G are indistinguishable and have no identifiers, no memory, and no computational ability; they act only as containers. Computation is performed by n mobile agents, each having a unique identifier (ID), local computation ability, and limited memory. The computation proceeds in synchronous rounds: in each round, an agent may perform arbitrary local computation and may traverse one incident edge to move to a neighbouring node. Communication is *purely by meeting*: two agents can exchange information if and only if they are co-located at the same node in the same round. This model captures settings where reliable long-range communication is unavailable or undesirable, and where mobility itself is the primary means of interaction. It also introduces a fundamental algorithmic difficulty: when an agent moves to a neighbour to communicate, the intended neighbour may simultaneously move away, and the node retains no state that could help coordinate or buffer information. A central point of this chapter is that MST and BFS are treated as related but distinct tree-construction problems. The MST construction is studied first as an independent problem and also because it elects a unique leader agent. This leader is useful later when no BFS root is specified in advance. However, BFS construction itself does not inherently require an MST: when a root is already known, the BFS tree can be constructed directly from that root. Thus, the MST phase is used only to remove the prior-root assumption in the root-free BFS setting.

Recent work demonstrates that mobile agents can solve a variety of classical graph problems in anonymous networks, including dispersion [68, 75], maximal independent set constructions [103, 98], dominating sets [26], and subgraph analytics such as triangle counting [24].

Kshemkalyani *et al.* [69] initiated the study of deterministic leader election and MST construction without assuming any knowledge of global graph parameters; however, their algorithms use $O(n \log n)$ bits per agent and require $O(m)$ rounds for leader election and $O(m+n \log n)$ rounds for MST construction. A recurring bottleneck behind many agent-based graph algorithms is *local coordination*: to guarantee that adjacent agents meet and exchange information, prior approaches often rely on knowing Δ to allocate sufficiently long scheduling windows (typically proportional to Δ) [26, 98, 24]. Removing such parameter knowledge while keeping both time and memory small is non-trivial, and is central to building fast and robust higher-level constructions such as MST and BFS.

5.1.1 Our Contributions

We consider an arbitrary, connected, undirected, simple, anonymous graph G with n agents initially in a dispersed configuration (one agent per node). Our main contributions are as follows:

- We design an efficient neighbour-meeting protocol that guarantees a meeting with any chosen neighbour within $O(\log n)$ rounds per neighbour, without requiring any prior knowledge of graph parameters [**Algorithm 10**].
- We present an MST construction algorithm for the mobile-agent model. The algorithm also elects a unique leader agent and runs in $O(n \log n + \Delta \log^2 n)$ rounds using $O(\log n)$ memory per agent [**Algorithm 15, Theorems 5.2 and 5.3**].
- We present two BFS constructions. If a root is already known, the agents construct a BFS tree directly from that root in $O(D\Delta)$ rounds. If no root is known initially, the leader obtained from the MST construction is used as the root. In the root-free setting, the overall BFS construction takes $O\big(n \log n + \Delta \log^2 n + \min(m \log n, D\Delta)\big)$ rounds using $O(\log n)$ memory per agent [**Algorithm 18, Theorem 5.5**].

All algorithms assume that the maximum agent ID λ is known a priori, and require $O(\log n)$ bits of memory per agent.

5.1.2 Challenges and Techniques

As the nodes are memoryless and agents can communicate only upon meeting, even coordinating two adjacent agents is delicate: an initiator may arrive exactly as its neighbour departs. Many

earlier agent-based algorithms, therefore, assume that agents know Δ to provision sufficiently long degree-dependent schedules that force neighbour meetings [26, 98, 24]. Our first objective is to remove this assumption while keeping per-agent memory small. We achieve this via a deterministic neighbour-meeting primitive (Section 5.4.1) that depends only on agent IDs. Each agent uses a padded binary string formed by concatenating its ID with its bitwise complement, ensuring that any two agents differ at some position where one has 1 and the other has 0; this asymmetry forces a meeting with any chosen neighbour within $O(\log n)$ rounds (per neighbour).

Since communication in our model is strictly local and possible only when agents become co-located at the same node, the spanning structures constructed later are more than just direct simulations of classical message-passing algorithms. Instead, they serve as coordination frameworks that enable controlled movement, synchronisation, and gradual information propagation through repeated local interactions between agents. The MST construction is not only a preprocessing step for BFS; it is a fundamental tree-construction problem in its own right. At the same time, its leader-election property is useful for the root-free BFS setting, because the elected leader provides a natural root from which BFS can be initiated. When a root is already known, this MST phase is unnecessary, and BFS can be constructed directly. Moreover, all of these constructions are non-trivial adaptations of classical message-passing algorithms, carefully redesigned to operate under the constraints of the mobile-agent model. Since nodes are memoryless and agents can communicate only when they become co-located, every exchange of information must be achieved through controlled movement, local interactions, and explicit synchronisation among agents, while operating with minimal assumptions regarding global parameter knowledge.

For MST, we impose a deterministic order on edges in an anonymous, unweighted graph by assigning distinct virtual weights computed on demand from endpoint IDs and port numbers, and emulated the fragment-growth structure of GHS [45] under meet-only communication: fragments repeatedly identify their MWOE, aggregate candidate minima to the fragment leader, and broadcast the selected edge back along the fragment tree using periodic traversals, all with $O(\log n)$ memory per agent, until a single fragment remains whose leader becomes the global leader. For BFS, we addressed the uncertainty of frontier coordination (and the potential blow-up of repeated to and fro traversal-based dispersal, e.g., $O(\Delta^D)$ in [68]) using two routines: one with a known root in a dispersed start, agents first compute Δ and then expand level-by-level in $O(D\Delta)$ rounds; second, when the root is unknown, we first run the MST phase to elect a leader and obtain a stable tree backbone for parameter gathering and synchronisation, and then construct BFS by taking the better of the two routines.

5.2 Related Works

5.2.1 Classical Distributed BFS, MST, and Leader Election

Breadth-First Search (BFS), Minimum Spanning Tree (MST), and leader election are among the most fundamental primitives in distributed computing and have been extensively studied in the classical message-passing model. Early BFS constructions include the layer-by-layer algorithm of Cheung [28] and the asynchronous BFS frameworks of Gallager *et al.* [46, 47], later improved by Awerbuch [11]. Subsequent works refined synchronisation and communication complexity in specialised settings such as planar networks and asynchronous BFS variants [44, 89]. In parallel and large-scale systems, BFS has also been studied through scalable partitioning strategies and optimised implementations for massive graphs [30, 134, 21, 18, 121, 86].

For MST construction, early distributed approaches date back to Spira [114] and culminated in the classical GHS algorithm [45], which remains one of the foundational distributed MST algorithms. Later works improved round complexity and established lower bounds for spanning-tree and MST-type constructions [10, 48, 101]. Leader election, introduced by Le Lann [80], has likewise been extensively investigated, leading to deterministic algorithms and tight trade-offs between time and message complexity [11, 102, 79]. These works collectively motivate the study of how such primitives can be adapted to mobile-agent settings where communication occurs only through local meetings rather than direct message passing.

5.2.2 BFS and Exploration with Mobile Agents

In mobile-agent and agent-based models, BFS often appears as both an exploration primitive and a coordination framework. Awerbuch *et al.* [12, 13] studied BFS-style exploration by a single agent under periodic return constraints, resulting in the *STRIP-EXPLORE* framework. In multi-agent settings, Palanisamy *et al.* [96] proposed a cluster-based approach that performs BFS sequentially across clusters, achieving a running time of $O(k(|V|+|E|))$ with k agents. BFS has also been used extensively in dispersion algorithms, where agents spread so that at most one agent occupies each node. Since the literature on dispersion has already been discussed in Chapter 1 and Chapter 2, we avoid repeating those results here. In particular, Kshemkalyani *et al.* [75] developed BFS-based dispersion algorithms under both local and global communication assumptions, improving over earlier exponential-time behaviour such as the $O(\Delta^D)$ phenomenon analysed in [68].

5.2.3 MST and Other Graph Structures in Agent-Based Models

Deterministic MST construction and leader election in the mobile-agent setting were initiated more recently. Kshemkalyani *et al.* [69] proposed the first deterministic MST and leader-election algorithms in the agent-based model without assuming prior knowledge of graph parameters. Their work also introduced a broader perspective on adapting message-passing algorithms to mobile-agent systems. Subsequent works studied improved trade-offs under different assumptions and parameter knowledge [71]. Beyond BFS and MST, there has been growing interest in computing classical graph structures using mobile agents. Recent works include maximal independent set constructions [103, 98], dominating set computation [26], and triangle counting and related graph analytics [24]. These works collectively demonstrate that mobile agents can perform increasingly sophisticated distributed graph computations despite severe communication constraints.

5.2.4 Deterministic Coordination and Applications

Deterministic label-based coordination has a rich history in rendezvous and exploration problems. In particular, strongly universal exploration sequences (UXS) and related techniques have been used to guarantee rendezvous and treasure-hunt objectives under limited information [119]. Although these works primarily focus on global exploration, they illustrate the broader principle that agent identifiers can be exploited to enforce deterministic coordination without relying on node identities. Our work follows this philosophy in the local neighbourhood setting by developing efficient deterministic neighbour-meeting techniques that support faster BFS and MST constructions in anonymous graphs. Such agent-based coordination mechanisms are also motivated by practical applications in domains including underwater navigation [34], network-centric warfare [82], social-network analysis [137, 91], search and rescue [135, 104, 132, 130], and drone-assisted wireless sensor networks [5, 39, 58, 55], where mobility and intermittent local communication are natural operational constraints.

5.3 Model and Problem Definition

Graph: The underlying graph $G(V, E)$ that is connected, undirected, and anonymous with $|V| = n$ nodes and $|E| = m$ edges. Nodes of G do not have any distinguishing identifiers or labels. These nodes do not possess any memory and hence cannot store any information. The

Algorithm	Knowledge	Time Complexity (Rounds)	Memory/Agent	Initial Config.
Leader Election				
Section 5.4	λ	$O(n \log n + \Delta \log^2 n)$	$O(\log n)$	Dispersed
Kshemkalyani *et al.* [69]	$-$	$O(m)$	$O(\log^2 n)$	Arbitrary
Kshemkalyani *et al.* [71]	n, Δ	$O(n \log^2 n + D\Delta \log n)$	$O(\log n)$	Arbitrary
MST Construction				
Section 5.4	λ	$O(n \log n + \Delta \log^2 n)$	$O(\log n)$	Dispersed
Kshemkalyani *et al.* [69]	$-$	$O(m + n \log n)$	$O(\max(\Delta, \log n) \log n)$	Arbitrary
Kshemkalyani *et al.* [71]	n, Δ	$O(m + n \log^2 n + D\Delta \log n)$	$O(\Delta \log n)$	Arbitrary
BFS Construction				
Section 5.5	λ	$O(n \log n + \Delta \log^2 n + \min(m \log n, D\Delta))$	$O(\log n)$	Dispersed
Section 5.5	root	$O(\min(m \log n, D\Delta))$	$O(\log n)$	Dispersed

Table 5.1: Comparison of previous and recent results for leader election, MST construction, and BFS construction in the mobile-agent model. Here, λ denotes the maximum agent ID among the n agents in an n-node graph with m edges, diameter D, and maximum degree Δ. The symbol '$-$' denotes that no prior knowledge is assumed.

degree of a node $v \in V$ is denoted by $\delta(v)$, and the maximum degree of G is Δ. Edges incident on v are locally labelled using port numbers in the range $[0, \delta(v) - 1]$. We denote a port number that connects the nodes u to v by p_{uv}. In general, $p_{uv} \neq p_{vu}$. The edges of the graph serve as *routes* through which the agents can commute. Any number of agents can travel through an edge at any given time. For BFS, the graph is considered unweighted. For MST construction, we additionally allow positive edge weights. When the graph is unweighted, distinct virtual weights are assigned to simulate the GHS framework. For a weighted graph, the edge weights are positive numbers bounded by $O(n^c)$ for some constant $c \geq 1$. The edge weights are associated with the port numbers of the respective edges. An agent at a node v can see the weights of the edges adjacent to v.

Mobile Agents: We have a collection of n agents enumerated as $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ residing on the nodes of the graph with each having a unique ID $\in [0, n^c]$ for some constant $c \geq 1$, and has $O(\log n)$ bits of memory to store information. We assume that the maximum ID among

the n agents is known to all agents, denoted by λ with ($\lambda \leq n^c$)[1]. Two or more agents can be present (*co-located*) at a node in G; however, an agent is not allowed to stay on an edge. An agent can recognise the port number through which it has entered and exited a node. The agents do not have any visibility beyond their (current) location at a node. An agent at a node v can only realise its adjacent ports (connecting to edges) at v. Only the collocated agents at a node can sense each other and exchange information. An agent can exchange all the information stored in its memory instantaneously. We consider that n agents start from a complete *dispersed initial configuration*[2] such that each node of the graph G has exactly one agent.

Communication Model: We consider a synchronous system where the agents are synchronised to a common clock and the *local communication* model, where only co-located agents (i.e., agents at the same node) can communicate among themselves. In each round, an agent r_i performs the $Communicate - Compute - Move$ (CCM) task-cycle as follows: (i) *Communicate:* r_i may communicate with other agents at the same node, (ii) *Compute:* Based on the gathered information and subsequent computations, r_i may perform all manner of computations within the bounds of its memory, and (iii) *Move:* r_i may move to a neighbouring node using the computed exit port. We measure the complexity in two metrics, namely, time/round and memory. The *time complexity* of an algorithm is the number of rounds required to execute the algorithm. The *memory complexity* is measured w.r.t. the amount of memory (in bits) required by each agent for computation.

For the tree construction problems studied in this chapter, dispersed configurations help the agents maintain local information about the graph. For example, an agent can store information such as parent, children, level, fragment label, or edge-related values on behalf of the node it represents. This is useful for constructing BFS and MST structures in anonymous graphs, where the nodes themselves cannot store any information. The agents must move, meet, exchange information locally, and coordinate their actions under the restrictions of the mobile agent model. Thus, the organised placement of agents is used as a useful starting point for building global tree structures.

5.3.1 Problem Statements

We consider an n-node simple, connected, anonymous graph $G(V, E)$ with n autonomous agents initially placed in a *dispersed* configuration (one agent per node). The goal is to solve and analyse

[1]We use $O(\log n)$ in place of $O(\log \lambda)$ throughout to express bounds in standard graph parameters and enable direct comparison with existing results.

[2]For other starting configurations of agents, dispersion is first achieved in $O(n)$ rounds using [70].

the complexity of the following problems:

1. *Minimum Spanning Tree (MST) Construction:* Given a weighted graph G, where each agent can access the weights of the edges incident to its current node, construct an MST of G such that each MST edge is known to at least one of its endpoint agents.

2. *Leader Election:* Elect a unique leader among the n agents.

3. *Breadth-First Search (BFS) Tree Construction:* Construct a BFS tree of G such that each tree edge is known to at least one of its endpoint agents.

Before proceeding with our main algorithms, we provide the list of variables used in this chapter in Table 5.2.

Agent Variables Used in Chapter 5		
Variable	**Values / Type**	**Description**
$r_i.ID$	Unique ID	Agent ID.
$r_i.b$	Binary string	Padded binary representation of the agent ID.
$r_i.new\_ID$	Binary string	Concatenation of the complemented ID and original ID used in neighbour meetings.
b_j	Binary bit	Current bit examined during `Meeting with Neighbor()`.
$r_i.parent$	Port / Agent ID	Parent reference maintained in the MST/BFS tree.
$r_i.child$	Port / Agent ID	Stores the first child pointer in the constructed tree.
$r_i.sibling$	Port / Agent ID	Links sibling children to reduce memory usage.
$r_i.treelabel$	Agent ID	Identifies the fragment/tree to which the agent belongs.
$r_i.treelevel$	Integer	Stores the level of the fragment during MST merging.
$r_i.min\_weight$	Weight value	Minimum outgoing edge weight discovered locally by the agent.
$r_i.min\_local$	Weight value	Aggregated local minimum received from children and self.
$r_i.leader\_min$	Weight value	Global MWOE value maintained by the fragment leader.
$w_{u,v}$	Edge weight	Dynamically computed virtual edge weight of (u, v).
$r_i.level$	Integer	BFS level of the agent in the constructed BFS tree.
$r_i.completed$	Boolean	Initially set to `false`; becomes `true` once all neighboring nodes of r_i (except its parent) have been visited.
$r_i.visitor$	$0/1$	Set to 1 when agent r_i is visiting another node from its home node; otherwise set to 0.

Table 5.2: List of agent variables used in this chapter. List of notations on Table 1.5 (Introduction).

5.4 Agent-Based MST Construction

In this section, we describe the construction of a Minimum Spanning Tree (MST) that simultaneously elects a leader agent. Besides its independent importance, the MST construction also enables the collection of useful graph parameters and provides a leader that is later used in the root-free BFS setting, where no designated root is available in advance. The MST is constructed by adapting the classical distributed GHS algorithm [45]; however, this adaptation is non-trivial in the mobile-agent model. In the classical message-passing setting, neighbouring nodes can directly exchange messages across edges, and nodes can store local information. In contrast, in our model, the nodes are anonymous and memoryless, and communication is possible only when two agents physically meet at the same node. Thus, every exchange of information required by the GHS framework must be implemented through carefully synchronised agent movements and local meetings. Although the graph G is unweighted, we adopt the GHS framework due to its parallel nature. Consequently, our approach improves upon the existing agent-based MST result in [69], which also implements a GHS-based strategy but in a sequential manner. To apply the MST framework to our unweighted graph, we first assign distinct virtual weights to the edges, computed using agent IDs and port numbers rather than stored at the nodes (details in Section 5.4.2). A key requirement in this process is that adjacent agents must be able to meet reliably. Since agents may move simultaneously and nodes cannot buffer information, synchronising such meetings is one of the main challenges. To address this, we develop a neighbour-meeting procedure that allows two adjacent agents to meet across a given edge within $O(\log n)$ rounds, without requiring prior knowledge of Δ (Section 5.4.1).

5.4.1 Meeting with Neighbouring Agents

The agents operate autonomously without any centralised control, so ensuring that an agent successfully meets its intended neighbour is challenging. This difficulty arises because the targeted neighbour might simultaneously move to meet another of its own neighbours in the same round. Therefore, a mechanism is needed to guarantee that two adjacent agents, intending to meet, can do so within a specified time frame. One such mechanism, known as `Protocol MYN`, has been employed in several studies [26, 24], where agents use the knowledge of Δ (the maximum degree) along with their ID bits to ensure that each agent meets all its neighbours at least once within $O(\Delta \log \lambda)$ rounds. However, this approach has two key drawbacks. First, agents must have prior knowledge of Δ, the maximum degree in the graph. Secondly, the order in which neighbours are

Algorithm 10 IMPROVED MEETING WITH NEIGHBOUR()

Require: Two agents r_u and r_v with unique IDs, located at adjacent nodes u and v.
Ensure: The agent r_u meets r_v at v within $4 \log \lambda$ rounds.

1: Pad each agent's ID to $\log \lambda$ bits by adding leading zeros and call it b.
2: $r_u.new\_ID \leftarrow (\sim r_u.b)||(r_u.b)$.
3: Let us suppose $round\_no$ represents the round number initialized to 0.
4: **for** $4 \log \lambda$ rounds **do**
5: **if** $round\_no \ mod \ 2 = 0$ **then**
6: r_u scans $r_u.new\_ID$ from LSB to MSB.
7: **if** $(round\_no/2 + 1)^{th}$ bit of $r_u.new\_ID$ is 1 **then**
8: r_u moves to node v via port p_u and return to node u in the next round.
9: **else**
10: r_u remains at node u for 2 consecutive rounds.
11: **end if**
12: **end if**
13: **end for**

met is dictated by the ID bits rather than the node's port numbers. In some algorithms, agents may be required to meet only a specific neighbour or visit the neighbours in a sequence of port numbers. Our proposed approach overcomes both of these limitations.

Our proposed algorithm `Meeting with Neighbour()` (Algorithm 10) is as follows. Consider two agents, r_u and r_v, initially located at adjacent nodes u and v, respectively. Let p_u and p_v denote the ports connecting these nodes. Each agent r_u first pads its ID $r_u.ID$ with leading 0s to ensure a binary length of exactly $\log \lambda$ bits. Let us call this padded ID $r_u.b$ (Line 1). It then computes a new identifier, new_ID, by appending the bitwise complement of its ID to the left (Line 2), resulting in a $2 \log \lambda$-bit string. The protocol runs for $4 \log \lambda$ rounds, with two rounds corresponding to a bit position (Lines 4-13). Agents scan their new_ID from the least significant bit (LSB) to the most significant bit (MSB). In each of these rounds, if the current bit of $r_u.new\_ID$ (new_ID of agent r_u) is 1, r_u moves to neighbouring node v; otherwise, it stays at node u. Since the agents have distinct IDs, there must exist a bit position in $r_u.new\_ID$ and $r_v.new\_ID$ where the former has a 1 and the latter has a 0. This guarantees that r_u moves to v and encounters r_v.

For example, let r_u have ID 100 and $\lambda = 1100$. Padding gives 0100, and computing new_ID results in 10110100. If r_v has ID 110, then $r_v.new\_ID$ is 10010110. In this case, r_u meets r_v in the $2 \cdot 6 - 1 = 11$th round.

Observation 5.1. *Let r_u be an agent located at node u with degree $\delta(u)$. Let v be a neighbouring node of u hosting an agent r_v. Then r_u can meet r_v within $4 \log \lambda$ rounds. Furthermore, r_u can meet*

all its neighbours in at most $\delta(u) \cdot 4 \log \lambda$ *rounds.*

Observe that concatenating the IDs with their bitwise complement is crucial; otherwise, the two agents may not meet. For instance, consider agents r_u and r_v with IDs 0010 and 0110, respectively. Suppose r_u aims to meet r_v, but r_v is simultaneously engaged in another meeting with a neighbour $r_w \neq r_u$. In the second round, both r_u and r_v move, preventing their meeting. In the subsequent rounds, r_u remains stationary, failing to meet r_v. However, using new_ID, this issue is avoided. The new IDs for r_u and r_v are 11010010 and 10010110, respectively. In this case, r_u is guaranteed to meet r_v in the 7th LSB, where r_u has a 1 and r_v has a 0.

Having established the meeting with neighbouring agents, we now move to the next section, where we construct a minimum spanning tree of a graph using agents.

5.4.2 Constructing a Minimum Spanning Tree

Although the high-level fragment-growth strategy follows the classical GHS framework, adapting it to the mobile-agent model introduces several challenges. Unlike the message-passing setting, agents can communicate only upon meeting, nodes are memoryless, and storing explicit edge information may violate the desired low-memory bound. Therefore, the main technical aspects of our approach are: (i) guaranteeing deterministic neighbour meetings without prior knowledge of Δ, (ii) dynamically computing edge weights instead of storing them, and (iii) maintaining fragment communication and parent-child relationships using only $O(\log n)$ bits of memory per agent.

Now, we present an algorithm that enables agents to construct an MST (which becomes a spanning tree for an unweighted graph) for a given graph $G(V, E)$. Our approach is an adaptation of the GHS algorithm [45], a classical algorithm for constructing a Minimum Spanning Tree (MST) in the message-passing model of distributed computing. To adapt the GHS algorithm to our agent-based model, we first address the requirement of unique edge weights, as the algorithm assumes distinct weights for all m edges. Since we consider a graph with unweighted edges, we must assign unique weights to meet this requirement. While this could be achieved by assigning random weights to the edges, our algorithm is deterministic, necessitating a deterministic method for weight assignment. To this end, we leverage the agents' unique identifiers (IDs) and the port numbering at each node to systematically assign distinct edge weights. Furthermore, this weight assignment approach introduces additional memory requirements for storing and managing the assigned weights. For an agent r_u, the memory required would be $O(\delta(u) \log \lambda)$ bits, where $\delta(u)$ denotes the degree of node u and λ represents the maximum ID value of the agents. To

eliminate this overhead, we do not store edge weights in the agents' memory. Instead, these weights are computed dynamically when needed. Below, we outline the method used by the agents to dynamically assign distinct weights to each edge in the graph as needed.

5.4.2.1 Generating Distinct Edge Weights:

To obtain distinct edge weights, an agent r_u does the following:

- r_u meets its neighbours one by one.
- Once r_u meets a neighbour r_v, let r denote the endpoint agent with smaller ID, i.e., $r.ID = \min\{r_u.ID, r_v.ID\}$. The weight of the edge (r_u, r_v) is defined as $r.ID + \frac{1}{p_{uv}+2}$, where p_{uv} is the port number at agent r leading to the other endpoint agent.
- Both r_u and r_v assign the same weight $r.ID + \frac{1}{p_{uv}+2}$ to the edge (r_u, r_v).

Lemma 5.1 (Distinct Edge Weights)**.** *The edge weights assigned to each edge are distinct.*

Proof. To prove this, we consider two arbitrary edges of G, say e_1 and e_2. First, we assume that e_1 and e_2 are adjacent edges having the common node u containing the agent r_u. Let e_1 connect r_u (at node u) to r_v (at node v) and e_2 connect r_u to r_w (node w) via port number p_{uv} and p_{uw} respectively. Now, if $r_u.ID = \min\{r_u.ID, r_v.ID, r_w.ID\}$, then, the weights of e_1 and e_2 namely w_{e_1} and w_{e_2} are given by $r_u.ID + 1/(p_{uv} + 2)$ and $r_u.ID + 1/(p_{uw} + 2)$ which are, clearly, distinct as $p_{uv} \neq p_{uw}$. If $r_u.ID$ is not the minimum then the weights w_{e_1} and w_{e_2} are assigned by two different agents from the set $\{r_u, r_v, r_w\}$. Since the IDs of these agents are integers and the addition fraction in the weight calculation due to port number, i.e., $1/(port + 2)$ is less than 1, so w_{e_1} and w_{e_2} must be distinct. On the other hand, if e_1 and e_2 are not adjacent, then w_{e_1} and w_{e_2} are assigned weights based on different agents' IDs, which, similarly, must be distinct. □

To compute the weight of an edge, an agent must interact with its neighbouring agent at least once, which can be achieved using the `Meeting with Neighbour ()` protocol. Through this process, each agent sequentially meets all its neighbours and determines the weights of its outgoing edges. Once distinct edge weights have been assigned, the agents proceed to simulate the GHS algorithm to construct a spanning tree. Now, we provide the details of the MST construction (Algorithm 15). We divide the algorithm into the following three main stages:

1. **Minimum Weight Outgoing Edge (MWOE) Calculation:** Each fragment's leader determines the minimum weight outgoing edge (MWOE) among all its outgoing edges using Algorithm 11.

2. **MWOE Identification and Selection:** Using Algorithm 12, an agent gets to know whether its outgoing edge is MWOE. For this, the fragment's leader broadcasts the minimum weight value to all the agents in its fragment. Once an agent realises that one of its incident edges is the MWOE, it proceeds to the next stage.

3. **Fragment Merging:** In this stage, as the agent holding the MWOE (say, r_u) meets with its adjacent agent (say, r_v), r_u shares its fragment level ($treelevel$) and its intention to merge with or absorb the fragment of r_v. At this point, agent r_u executes either `Merge()` (Algorithm 14) or `Absorb()` (Algorithm 13), depending on the respective $treelevel$ values of agents r_u and r_v and whether the edge (r_u, r_v) is the MWOE for both fragments.

The process continues until all the agents become part of a single fragment with no more outgoing edges. At this point, the algorithm terminates. Below, we explain the three stages in detail.

5.4.2.2 Stage 1 - Minimum Weight Outgoing Edge (MWOE) Calculation:

Each edge is assigned a unique virtual weight, as derived from Section 5.4.2.1. Initially, each of the n agents individually represents a distinct tree fragment. As the algorithm progresses, these fragments merge via the Minimum Weight Outgoing Edge (MWOE). In Algorithm 11, the agents within each fragment collaboratively compute the MWOE of the fragment, while the fragment leader[3] finally obtains the global minimum through aggregation. The algorithm begins with each agent exploring its neighbours in parallel. For each neighbour, the agent checks whether the edge connecting to it is internal or outgoing by comparing their respective $treelabel$ values. If the $treelabel$ values match, the edge is internal; otherwise, it is considered outgoing (Algorithm 11, Line 3). Then each agent sequentially examines its neighbours to identify the minimum-weight outgoing edge. During this process, it maintains a variable r_u.min_weight to store the smallest weight encountered among all outgoing edges. Although each agent searches its local neighbourhood sequentially, these searches occur in parallel across the agents. The agent then aggregates the minimum weights received from its children, storing the local minimum in the variable min_local (Lines 2–11, Algorithm 11). This value is subsequently propagated up in the fragment's hierarchy with the parent node, ensuring that the fragment leader obtains the global minimum (Algorithm 11, Lines 12-15). The algorithm terminates when only a single fragment remains,

[3]During MST construction, the leader is assumed to be the fragment leader unless specified otherwise. Once all fragments merge into a single one, the fragment's leader becomes the leader of all agents.

which occurs when there are no outgoing edges. This termination condition is detected in Algorithm 11, Line 16. Each agent is also equipped with the variable $treelevel$, which tracks the level of a given fragment. Specifically, when two fragments with the same $treelevel$ merge, the newly formed fragment's $treelevel$ is incremented. The variables $treelabel$ and $treelevel$ serve distinct roles: $treelabel$ identifies agents within the same fragment (which serves like a tree ID), while $treelevel$ facilitates the merging of different fragments. Initially, $treelevel = 0$, and $treelabel$ is set to the fragment leader's ID. Since each agent starts as an independent fragment, its $treelabel$ is initially assigned as its own ID. As the fragments merge, the $treelabel$ of the agents in the merged fragment is determined by the leader of the new fragment.

Algorithm 11 MWOE Calculation within a Fragment

1: r_u.min_weight $\leftarrow \phi$
2: **for** each agent r_u (in parallel), and for each neighbour r of r_u **do**
3: **if** r.treelabel $\neq$ r_u.treelabel **then** ▷ Outgoing edge check
4: r_u computes the weight $w_{r_u,r}$ of edge (r_u, r)
5: **else**
6: $w_{r_u,r} \leftarrow \perp$
7: **end if**
8: **if** r_u.min_weight $> w_{r_u,r}$ **then**
9: r_u.min_weight $\leftarrow w_{r_u,r}$
10: **end if**
11: **end for**
12: r_u waits until it receives r_v.min_weight from each child r_v (if any)
13: r_u calculates r_u.min_local $= \min(\{r_u.\text{min\_weight}\}, \{r_v.\text{min\_weight} \mid r_v \text{ is a child of } r_u\})$ and sends it to its parent
14: **if** r_u is the leader agent **then**
15: r_u.leader_min $\leftarrow$ r_u.min_local
16: **if** r_u.leader_min $=\perp$ **then**
17: **terminate algorithm.**
18: **end if**
19: **end if**

Before proceeding to the selection of MWOE, we highlight two important remarks. The first deals with memory complexity, which enables the agents to construct the MST using $O(\log \lambda)$ bits. The second introduces a simple synchronisation technique that allows the leader of a fragment to efficiently communicate with all agents in its subtree.

Remark: 5 (Memory Complexity)**.** *The primary memory overhead of the algorithm arises from maintaining child pointers. An agent with up to Δ children would typically require $O(\Delta \log \lambda)$ bits for storage. However, this complexity can be reduced using a* `sibling` *variable. When an*

agent discovers its first child, it sets its `child` *pointer to the corresponding outgoing port. For each subsequent child, instead of storing multiple child pointers, the agent updates the previously added child's* `sibling` *variable with the new child's outgoing port. This cascading approach eliminates the need for Δ child pointers, allowing the* `child` *pointers to be stored efficiently in $O(\log \lambda)$ bits. The other variables* `parent`*,* `child`*, and* `treelabel` *require $O(\log \lambda) + O(\log \Delta) = O(\log \lambda)$ bits, while other variables require only $O(1)$ bits. Thus, the overall memory complexity is $O(\log \lambda)$ or $O(\log n)$ bits, since $\lambda \leq n^{O(1)}$.*

Remark: 6 (`parent-child` communication)**.** *To efficiently exchange information between a parent and its children in a single round, we introduce the following strategy: After every $4 \log \lambda$ rounds of neighbour interactions, an additional round is reserved in which each agent (except the leader) moves to its parent for updates. Thus, after every $4 \log \lambda$ round, each child oscillates once between its node and its parent node. When the leader initiates a broadcast, its immediate children receive the value in the first oscillation, followed by the grandchildren in the next, continuing until all agents are updated after D oscillations, where D is the tree diameter. To ensure updates propagate correctly, once all children at a given level have received the update, they stay at their node for $4 \log \lambda$ rounds, allowing deeper levels to update sequentially. This ensures a smooth transfer of information. Conversely, during the same reserved round, children can send information to their parents simultaneously. Thus, transmitting any value requires an additional $O(n)$ rounds in the worst case.*

5.4.2.3 Stage 2 - MWOE Identification and Selection:

To determine the MWOE of a fragment, an agent must verify whether any of its adjacent edges has been selected by the leader as the MWOE for connecting to another fragment. This requires the leader to broadcast the minimum computed weight value to all agents, a process carried out using Algorithm 12. Each agent checks if any of its adjacent edges is the MWOE by comparing its variable $r_u.min\_weight$ with the value sent by the leader; if they match, one of its adjacent edges is identified as the MWOE (Line 4). Since agents do not store edge weights in memory, they must reevaluate the weights (Line 5) to identify the edge corresponding to the MWOE weight received from the leader, introducing an additional computational factor of $\Delta \log n$ into Algorithm 11. With the MWOE of the fragment determined, we now proceed to describe the merging of fragments.

Algorithm 12 MWOE Identification and Selection

1: r_u checks r_t.leader_min. ▷ Communication within the constructed tree is achieved using techniques established in Remark 6
2: **if** r_u.leader_min $\neq r_t$.leader_min **then** ▷ Fragment leader has calculated a new weight
3: r_u.leader_min $\leftarrow r_t$.leader_min
4: **if** r_u.min_weight $= r_u$.leader_min **then** ▷ One of r_u's adjacent edges is the MWOE
5: r_u visits each neighbour to re-calculate edge weights and find the edge matching r_u.leader_min
6: The matched edge is computed as the MWOE, connecting r_u to some neighbour r_k (say)
7: r_u informs r_k that its incident edge is the MWOE and schedules it for merging.
8: **end if**
9: **else** ▷ No new edge weight calculated
10: r_u does nothing
11: **end if**

5.4.2.4 Stage 3 - Fragment Merging:

Let F_1 and F_2 be two fragments connected via a MWOE e. Then the fragments F_1 and F_2 merge based on the following rule:

1. **Rule 1** - If $F_1.treelevel < F_2.treelevel$, then F_1 joins the fragment F_2 and the $treelabel$ of the new fragment will be $F_2.treelabel$. Moreover, the leader of the fragment will be the leader of fragment F_2.
2. **Rule 2** - If $F_1.treelevel = F_2.treelevel$, then F_1 and F_2 will merge if and only if, e is the common MWOE of F_1 and F_2. In such a case, the leader of the fragment will be the leader of F_1 and F_2 that has the minimum ID, and the fragment level increases by 1 after merging.
3. **Rule 3** - In other cases, fragments wait till one of the rules 1 or 2 applies to them.

Now, let us consider two adjacent agents r_u and r_k connected via the MWOE e of r_u. Now, r_u does the following

1. If $r_u.treelevel > r_k.treelevel$, r_u does nothing.
2. If $r_u.treelevel = r_k.treelevel$ and e is the MWOE of the fragment belonging to r_k as well, r_u executes `Merge()`.
3. If $r_u.treelevel = r_k.treelevel$ and e is **not** the MWOE of the fragment belonging to r_k, r_u does nothing.

4. If $r_u.treelevel < r_k.treelevel$, r_u executes `Absorb()`.

Specifically, `Merge()` is used when two fragments have the same level, requiring an increment in their $treelevel$ value. In contrast, `Absorb()` is typically applied when two fragments can be ordered based on their $treelevel$ and ID values. Through `Absorb()`, the fragment of r_u merges into the fragment of r_k. During the merging process, the merging fragment terminates all ongoing operations and updates its child and parent pointers, as well as $treelabel$ and $treelevel$ accordingly.

Algorithm 13 ABSORB()

1: r_u immediately aborts any operation it may be doing.
2: r_u marks r_k as parent and collects $r_k.treelabel, r_k.treelevel$ and leader information of r_k's fragment.
3: r_u moves to its parent agent, and instructs it to immediately abort any ongoing operation.
4: r_u informs its parent about the new leader, changes its parent's $treelabel, treelevel$ to $r_k.treelabel, r_k.treelevel$, and changes the parent pointer to child pointer.
5: Once the parent pointer has been changed, the old parent moves to its own parent, requests abortion of any ongoing operation, marks its parent pointer as a child pointer, and similarly transmits and modifies the new information. Continuing similarly, when this message reaches the root, the root sends out completion information to r_k via the newly changed parent chain.
6: Once r_u, receives the completion information from its old root, it sends this new leader and $treelabel, treelevel$ information across its other (old) children in a similar fashion. The leaf children, after modifying their own information, again send out completion information to r_u. After r_u has received the completion information from all of its children, it now informs r_k about the completion of the `Absorb()`. As r_u visits r_k with the completion information, r_k sets the port leading to r_u as its child pointer.
7: The fragment of r_u has now merged with r_k's fragment with a new leader of r_k's fragment.

Algorithm 14 MERGE()

1: r_u and r_k compare their leader IDs
2: **if** r_u has a higher leader ID **then**
3: r_k increments $r_k.treelevel$ by 1.
4: r_k executes `Absorb()`
5: **else**
6: r_u increments $r_k.treelevel$ by 1.
7: r_u executes `Absorb()`
8: **end if**

Algorithm 15 MST Construction Using Mobile Agents

1: **while** $r_u.leader\_min \neq \perp$ **do**
2: Calculate the MWOE using Algorithm 11.
3: Identify and select the MWOE using Algorithm 12. ▷ Let e be the MWOE from the agent r_u which connects to the agent r_k.
4: **if** $r_u.treelevel = r_k.treelevel$ and r_k also scheduled for merging via e **then**
5: Execute `Merge()`.
6: **else if** $r_u.treelevel = r_k.treelevel$ and r_k not scheduled for merging via e **then**
7: r_u does nothing and waits.
8: **else if** $r_u.treelevel > r_k.treelevel$ **then**
9: r_u does nothing and waits.
10: **else if** $r_u.treelevel < r_k.treelevel$ **then**
11: r_u executes `Absorb()`.
12: **end if**
13: **end while**

5.4.2.5 Complexity Analysis of Algorithm 15 (MST Construction)

Lemma 5.2. *The respective time complexities of the 3 stages of the MST construction are respectively* $O(\Delta \log \lambda + n)$, $(\Delta \log \lambda + n)$ *and* $O(n)$ *rounds.*

Proof. For *Stage 1*, each agent takes $O(\Delta \log \lambda)$ rounds in the worst case to search its neighbours, which is done in parallel by the agents of a fragment. To report the minimum weight to the leader, the agents must send this value via their parent, and the leader computes the minimum of all the received weights. Since this transmission occurs via the tree, in the worst case, it can take $O(n)$ rounds. So, calculating the MWOE by the leader takes $O(\Delta \log \lambda + n)$ rounds. To distribute the MWOE to the whole fragment, the agents can take a maximum of $O(n)$ rounds, as the agents that are at the same distance from the root collect this MWOE from their parents, in parallel. Now the agents once again evaluate the edge weights to compute the matching edge with the selected edge weight. So, *Stage 2* also takes $O(n + \Delta \log \lambda)$ rounds. Finally, for *Stage 3*, the agent that connects its fragment via the MWOE needs to visit the chain of parents to inform them about the merging and changing of their child-parent pointers, respectively. This modification takes $O(n)$ rounds in the worst case. □

According to the GHS algorithm, the number of fragments while constructing the spanning tree reduces by at least half in each phase, so it takes $O(\log n)$ phases for the fragments to merge into a single fragment, giving out the final spanning tree. Now since, during each phase, weight is evaluated (at most twice), Algorithm 12 and Algorithm 11 is invoked once followed by merging of fragments, the time complexity of constructing the spanning tree is given by $O(\log n) \cdot$

$(O(\Delta \log \lambda) + O(\Delta \log \lambda + n) + O(n + \Delta \log \lambda) + O(n)) = O(\Delta \log n \log \lambda + n \log n)$ rounds. Considering that $O(\log \lambda) = O(\log n)$, we can rewrite the complexity as $O(\Delta \log^2 n + n \log n)$. The maximum memory overhead for the agents is used to store the IDs and pointers. Since the IDs are bound by n^c for some constant $c \geq 1$, the memory requirement for each agent is $O(\log n)$ bits. Combining other details from the discussion in Remark 5, the correctness of the original GHS algorithm, and the preceding lemma, we have the following theorem.

Theorem 5.2 (MST)**.** *Given an arbitrary simple connected anonymous graph G with n nodes, m edges, maximum degree Δ and diameter D. Then, an MST can be constructed in $O(\Delta \log^2 n + n \log n)$ rounds, when the n agents begin in a* dispersed *initial configuration with $O(\log n)$ bits of memory per agent. The agents do not require any prior knowledge about any graph parameters, but need to know an upper bound of the ID of the agents, λ.*

5.4.3 Leader Election via MST Construction

Following the construction of the spanning tree, all fragments eventually merge into a single connected component containing all n agents. As a consequence of this merging process, the fragment leader of the final component naturally serves as a unique leader for the entire network. Thus, the unweighted MST construction implicitly yields a leader election procedure as a by-product of fragment merging. This observation leads to the following result.

Theorem 5.3 (Leader Election via MST)**.** *Given a dispersed configuration of n agents with unique identifiers, placed one agent per node of an anonymous n-node, m-edge graph G with maximum degree Δ, there exists a deterministic algorithm that elects a unique leader in $O(\Delta \log^2 n + n \log n)$ rounds using $O(\log n)$ bits of memory per agent. The algorithm requires no prior knowledge of any graph parameters and assumes that each agent knows only λ, the maximum among all agent identifiers.*

We emphasise that this leader election is not an independently designed primitive; rather, it emerges implicitly from the fragment-based merging process of the MST algorithm. While this approach suffices for establishing a unique leader, it inherits the time complexity overhead of MST construction, particularly the dependence on Δ arising from repeated neighbourhood probing.

Weighted MST. In the case of a weighted graph with distinct edge weights, the algorithm proceeds analogously to Algorithm 15. The only difference is that edge weights are predefined and

do not need to be computed explicitly, as discussed in Section 5.4.2.1. Consequently, all analytical results, including time and memory complexity bounds, remain unchanged. This yields Theorem 5.2 for weighted graphs.

5.5 Agent-Based BFS Tree Construction

In this section, we present two algorithms for BFS construction. The first is a basic approach that assumes the system starts from a dispersed configuration and that a designated root agent is known. It proceeds in two phases: first, the agents compute Δ, the maximum degree of the graph; next, starting from the root, they build the BFS tree level by level, waiting exactly Δ rounds per level, which yields an overall running time of $O(D\Delta)$ rounds. This routine is useful whenever a root is already known or pre-selected: even from an arbitrary initial configuration, the agents can first disperse in $O(n)$ rounds [70] and then construct the BFS tree in an additional $O(D\Delta)$ rounds.

When no leader or root is known in advance, the above BFS construction cannot be applied directly, since there is no distinguished agent from which the BFS exploration can be initiated. Our second algorithm, therefore, first uses the MST construction developed earlier to elect a unique leader and to gather the graph-parameter information needed for synchronisation. The elected leader then serves as the root for the subsequent BFS construction. The agents construct the BFS tree using a different technique based on repeated applications of the `Meeting with Neighbour()` protocol (Algorithm 10) to assign correct BFS levels. This construction completes in $O(m \log n)$ rounds. Since the agents can compute the expected running time of the available approaches, they can choose the faster option between this $O(m \log n)$ method and the $O(D\Delta)$ level-expansion phase of the basic known-root algorithm, thereby selecting the BFS construction strategy that best matches the input graph. We now describe both algorithms in detail.

5.5.1 Basic Algorithm for BFS Construction

In this section, we describe a BFS tree construction algorithm for n agents, initially placed in a *dispersed* configuration, and the $root$ of the BFS tree is pre-defined. The algorithm executes in two phases. In the first phase, the agents compute Δ, the maximum degree of a node in the graph G. In the second phase, we construct the BFS tree of G from the given $root$ using the acquired value of Δ.

We first provide a list of variables that are used by each agent r_i during the execution of the algorithm.

1. ID - stores the ID bits of an agent. An $O(\log n)$ bit of memory space is used for the purpose.

2. $parent$ - stores the $port$ number from which an agent r_j finds r_i and r_i becomes the $child$ of r_j. Initially $parent = \perp$.

3. $child$ - stores the outgoing $port$ number through which r_i exited its own node to move to a neighbouring node to find a previously unvisited agent. Initially $child = \perp$

4. $completed$ - a boolean variable, initially set to $false$, which takes a value $true$ if and only if each of its neighbouring nodes (except $parent$) has been visited and no new unvisited neighbouring node is left.

5. $level$ - initially -1, it represents the $level$ of r_i in a tree with respect to the $root$ agent which has $level = 0$

6. $visitor$ - a variable set to 1, if r_i is a visitor to another node from its own node. When r_i is at its home node, the $visitor$ value is 0.

We denote the $root$ agent as $r^\star$ and use the notation r^j to denote an agent at a $level$ j with respect to the tree from the $root$. We first provide a high-level pseudocode for the BFS construction algorithm in Algorithm 16. The detailed execution of the two phases is described subsequently.

Algorithm 16 BASIC BFS CONSTRUCTION

```
Root agent r* initiates exploration.
Construct a spanning tree while propagating the maximum degree Δ upward to the root.
Root broadcasts the value of Δ to all agents.
Root sets level = 0.
for each BFS level in increasing order do
    Agents at the current level explore neighbouring nodes.
    Unvisited agents set parent pointers and assign their BFS levels.
    Wait 2Δ rounds before expanding the next level.
end for
Propagate completion information back to the root.
```

5.5.1.1 Phase 1: Knowing the Value of Δ

The agent at the $root$ node $r^\star$ begins the algorithm by visiting each of its neighbouring nodes. As $r^\star$ visits each child one by one, it immediately instructs each child r_i^1, upon receiving this instruction for the first time, to begin visiting its own children. Consider an agent r_i^j at the j^{th} level who has received instruction from its parent to visit its children. Now the following cases may arise:

1. r_i^j finds a new unvisited agent (with $parent = \perp$) in its adjacent node - r_i^j sets the new agent as its $child$ and similarly, the new agent stores the appropriate $parent$ pointer to r_i^j.
2. r_i^j finds that some of its adjacent nodes are empty - In such a case, r_i^j returns to its original node, marks the port as visited, and continues exploring the graph through the other available ports.
3. r_i^j finds multiple agents on one of its adjacent nodes.
   (a) The original resident of the node is present at the node (whose $visitor = 0$) - In such a case, the agent that has the least level (tie is broken using lower ID) becomes the parent. If r_i^j is not the selected agent, it returns to its original node and marks the node as visited.
   (b) The original resident of the node is absent at the node - In such a case, all agents are visitors and all of them return to their original home node.
4. r_i^j finds an agent r that has already been visited (i.e., $r.parent \neq \perp$) - In such a case, r_i^j ignores r, returns to its original node, and continues exploring the remaining ports.
5. If r_i^j receives an instruction from an agent to visit its neighbour for the second or subsequent times, it ignores any such request.

If some agent r_i^j has no children or all of its children have already been visited (except the parent), it sets its $completed$ value to $true$ and sends the degree value of its node to its parent. Now, the parent of r_i^j waits till it gets a $completed$ message from each of its children and simultaneously collects the maximum degree information from all of its children (whose $completed = true$), compares it with its own degree, and sends the maximum to its parent along with the message $completed = true$. Once an agent has received the maximum degree information from all of its children, it continues to move and inform its parent. When the $root$ receives the maximum degree

information from all of its children, it compares it with its own degree and thereby establishes the value of the maximum degree of the graph G, Δ. Now each agent has established the child-parent relationship with its appropriate neighbours. Thereafter, the $root$ propagates (downcast) the value of Δ throughout the graph using the constructed tree. Every agent in the graph is now acquainted with the maximum degree of the graph Δ.

Now all the agents have the value of Δ, and we execute the next phase of constructing the BFS tree from the $root$. One may notice that the tree formed during the computation of the maximum degree (Δ) may not form a BFS tree. The gradual construction of the BFS tree during the first exploration phase is illustrated in Figure 5-1.

5.5.1.2 Phase 2: Constructing a BFS Tree

Now, the agents have the prior knowledge of Δ, and they create a BFS tree from the $root$. The $root$ starts the algorithm by visiting each of its neighbours. The $root$ marks the appropriate ports leading to each of its neighbours as its child ports. Simultaneously, when a new agent (a probable child) gets a visit from its parent for the first time, the child sets its parent port accordingly. To maintain synchronicity between the agents at each level, each agent is equipped with a variable $r_i.level$, which is initially set to -1. The $root$ sets its level to 0. During the first visit of $root$ to its children, each of its children r_i^1 set their $level$ as $r_i^1.level \leftarrow root.level + 1$. Now, each of the agents at level 1 waits for 2Δ rounds before proceeding further. By that time, it is guaranteed that $root$ has visited each of its children. Now, with the end of the first 2Δ rounds, the agents at $level = 1$ begin to explore their neighbours in search of potential children. Consider the situation where, after $2\Delta j$ rounds, all agents at level j have received the instruction from their parents to start exploring their respective children. Let an agent r_i^j at $level\ j$ start exploring its neighbours and move to a node v. Then, the following cases may arise:

1. r_i^j finds out an agent r at v with $r.level = -1$ - It marks r as its child, and simultaneously r marks r_i^j's incoming port as its parent port. r sets $r.level \leftarrow r_i^j.level + 1$

2. r_i^j finds multiple agents at v

   (a) The original resident of v, r is present at the node (whose $visitor = 0$) - r accepts the request for becoming a child of the visitor agent which has the least ID. In case r_i^j is that particular agent, it updates its child pointers accordingly, and the newly discovered agent sets its parent port and $level \leftarrow r_i^j.level + 1$. Notice that all visiting agents reaching the node at this stage belong to the same BFS level, since the exploration

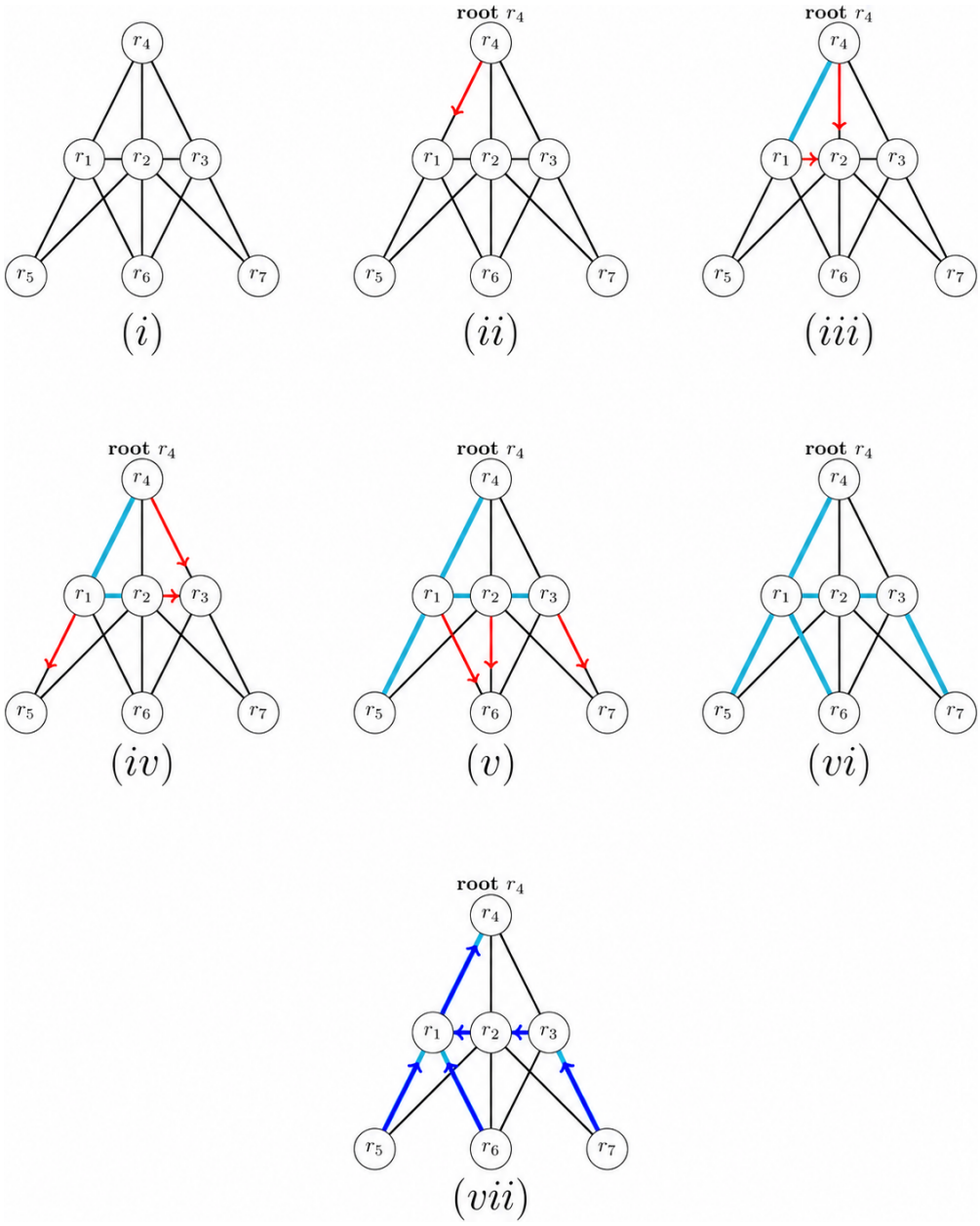


Figure 5-1: Phase 1 initiated by the root agent r_4. In subfigure (i), the initial graph configuration is shown. In subfigures (ii)–(v), the tree exploration gradually expands outward from the root, where red arrows indicate the current exploration step performed by the exploring agents. The blue edges denote the edges that have already been incorporated into the spanning tree. During the process, each newly discovered node selects the first incoming exploration edge as its parent edge in the tree. As the exploration progresses, additional nodes become attached to the current tree structure while avoiding redundant parent assignments. Subfigure (vi) shows the completed spanning tree after all reachable nodes have been discovered. Finally, subfigure (vii) illustrates the upward communication process through the constructed spanning tree, where information propagates from the leaf agents toward the root agent r_4.

proceeds level-by-level. If r_i^j is not the selected agent, it returns to its original node and marks the node as visited.

(b) The original resident of the node is absent at the node - In such a case, all agents are visitors and all of them return to their original home node.

3. r_i^j finds v to be empty - r_i^j returns to explore the other neighbourhoods. An empty node implies that the resident agent of that particular node is at the same level as r_i^1.

4. r_i^j finds an agent r that is already assigned a non-negative level - r_i^j ignores r and returns back to its home node for further exploration.

During the latter stage of the execution of the algorithm, if

1. r_i^j receives an instruction from an agent to visit its neighbours for the second or subsequent times, it ignores any such request or, if it

2. receives a visit from an agent with the same or lower $level$ as itself, r_i^j ignores such an agent and continues exploration.

Now, as the algorithm continues to execute, after each 2Δ round, we have a new level of the BFS being created.

When an agent uses up all the available ports for exploration, it sets its $completed = true$ and informs its $parent$ agent about the completion. As and when an agent receives the completion information from all its children, it marks its own $completed = true$ and similarly informs its parent agent about the same. As soon as the $root$ receives the completion information from all its children, it terminates the algorithm and sends the termination information across all agents via the tree.

The continuation of the BFS expansion process after the initial discovery phase is illustrated in Figure 5-2.

Lemma 5.3. *With the execution of Phase 2, the agents correctly create a BFS tree for the graph G.*

Proof. Since an agent always ignores any repeat request for becoming a $child$ node, algorithms in both Phase 1 and Phase 2 always create a tree substructure of the graph G. Let T be the tree generated by the agents after Phase 2. Let u be any node of the graph and $P(root = v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, u = v_h)$ be the path from $root$ to u in T. Since the algorithm explores the graph level-by-level, it explores all the nodes at level k before moving on to

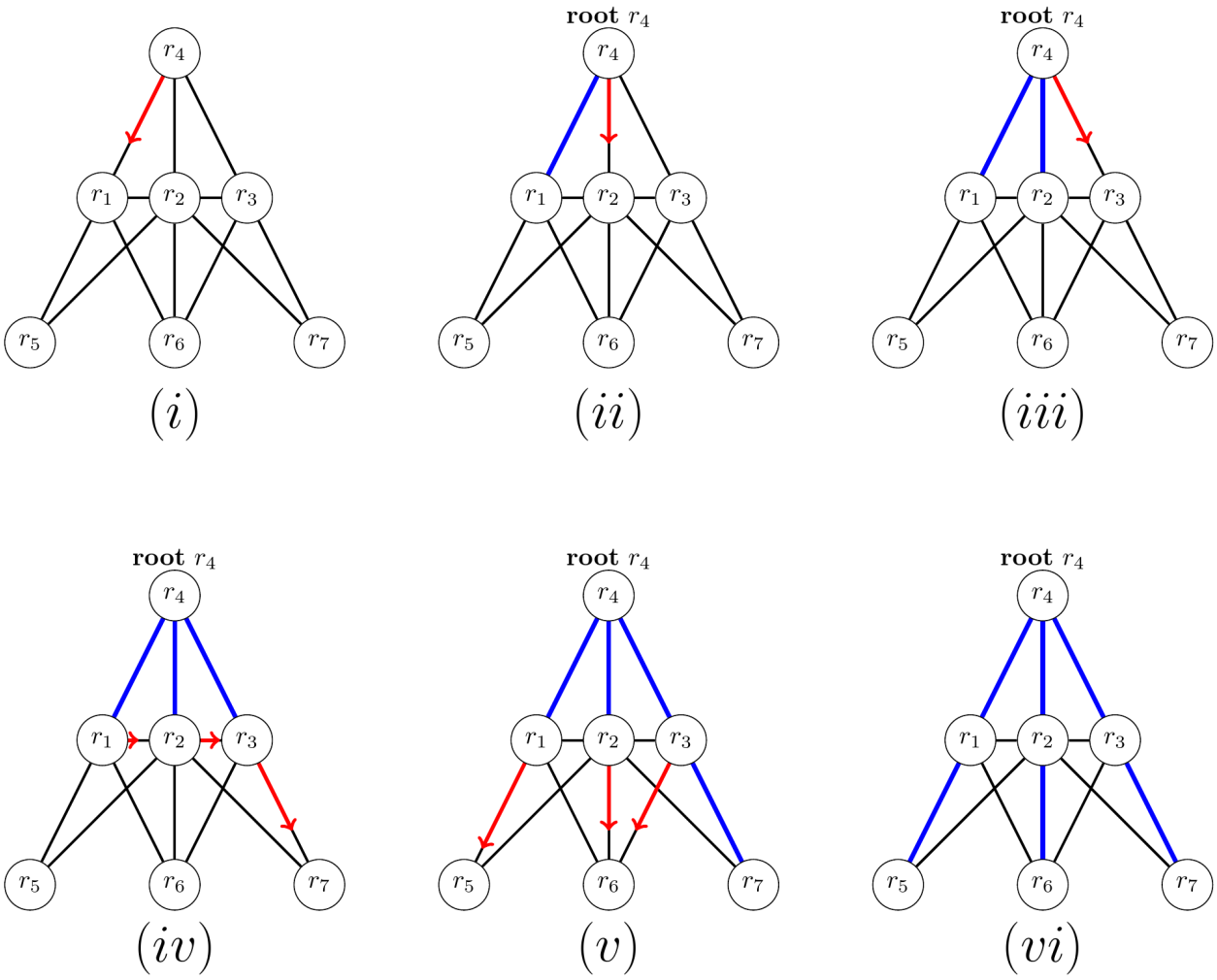


Figure 5-2: Illustration of Phase 2 of the BFS tree construction process. The BFS tree construction continues to expand level by level from the root agent r_4. Red arrows represent the current exploration step performed by the active agents, while the blue edges denote the portion of the BFS tree that has already been established. In subfigures (i)–(v), the exploration frontier gradually advances through undiscovered nodes, and newly visited nodes attach themselves to the BFS tree. As the exploration proceeds, the BFS layers are completed incrementally, with each level taking 2Δ rounds, without modifying previously assigned parent relationships. Subfigure (vi) shows the final BFS tree after all nodes have been explored and incorporated into the structure.

nodes at level $k+1$, ensuring that it finds the shortest path to each node from the $root$. Therefore, the path P represents the shortest path from $root$ to u. Hence, T is a BFS tree. □

Lemma 5.4. *Phase 2 takes $O(D\Delta)$ rounds to construct the BFS tree.*

Proof. In Phase 2, each level is completed in $O(\Delta)$ rounds during the construction of the BFS tree. Furthermore, the height of the BFS tree is $O(D)$. Therefore, BFS tree construction takes $O(D\Delta)$ rounds. As the agents continue to construct the BFS tree, some agents might have already completed visiting their children. These agents keep moving to their parents to inform them about their completion. The time for transmitting this completion information by all agents to the root can take at most $O(D\Delta)$ rounds, although most of this time overlaps with the BFS construction phase. Hence, the algorithm achieves completion within $O(D\Delta)$ rounds. □

Lemma 5.5. *Phase 1 requires $O(D\Delta)$ rounds to compute the Δ.*

Proof. We first establish that any agent at a distance d from the $root$ gets visited within $O(d\Delta)$ rounds, using induction on d.

- $d = 1$: The $root$ informs all its adjacent nodes in 2Δ rounds to explore their respective children.
- $d = i$: We assume that all agents at a distance i have been visited in $2i\Delta$ rounds.
- $d = i + 1$: With the beginning of the $2i\Delta + 1$ round, all agents at a distance i that have already been visited from the $root$, so the agents at distance i can begin instructing their children for exploration (some may have already initiated this process). Therefore, by the end of $2i\Delta + 2\Delta$ rounds, it is guaranteed that all agents at a distance $i + 1$ from the $root$ have been visited.

Since the distance between $root$ and any arbitrary agent is $\leq D$, it can be guaranteed that all the n agents must have received the exploration instruction from their parents by the end of $2D\Delta$ rounds and hence the lemma.

□

Remark: 7 ($child$ pointers)**.** *To efficiently manage the storage of child node pointers, the agent uses the same sibling pointer strategy as the MST construction (described in 5).*

Lemma 5.6 (Memory Complexity)**.** *Each agent requires an $O(\log n)$ bits of memory to execute the algorithm.*

Proof. To execute the algorithm, the agents need to store the $port$ numbers for $child$, $parent$ and $sibling$ pointers, needing an $O(\log \Delta)$ memory space. For the IDs, the agent requires $O(\log n)$ bits. Other variables require a constant number of bits. Therefore, each agent would require a $O(\log n)$ bits of memory to execute the algorithm. □

Combining lemmas from 5.3 to 5.6, we have the following result.

Theorem 5.4. *Given an arbitrary simple connected graph G with n nodes, m edges, maximum degree Δ and diameter D. Then, a BFS tree can be constructed in $O(D\Delta)$ rounds from the $root$ node when such a node is known a priori and the n agents begin in a* dispersed *initial configuration with $O(\log n)$ bits of memory per agent. The agents do not require any prior knowledge about any global parameters.*

5.5.2 Second algorithm for BFS Construction

With the basic BFS construction algorithm in place, we now introduce a new BFS construction algorithm that completes in $O(m \log n)$ rounds with m denoting the number of edges in G. By combining these two algorithms, we achieve a round complexity of $O(\min(D\Delta, m \log n))$ for the BFS construction. As mentioned earlier, the assumptions of a known root and m can be removed using the leader election protocol in Section 5.4. In fact, the values of m, and any other necessary graph parameters can be computed by Algorithm 15. The BFS algorithm heavily uses the `Meeting with Neighbour()` (Algorithm 10) as a subroutine, which requires λ to be known to the agents. Below, we discuss an overview of the algorithm.

High-Level Idea: The algorithm proceeds by propagating levels and marking tree edges in the BFS tree, similar to the flooding-based algorithm in the message-passing model. We use the `Meeting with Neighbour()` (Algorithm 10) to send level information from one agent to its neighbouring agents, one by one. Suppose an agent $r^\star$ is located at the specified root node, which is at level 0 in the BFS tree. We denote an agent's level as the level of its corresponding node in the BFS tree. Thus, the root agent $r^\star$ sets its level as $r^\star.level = 0$, and every other agent r_u sets its level as $r_u.level = \perp$ initially. Whenever an agent r_u updates its level, it informs its neighbouring agents of its new level. A neighbouring agent r_v updates its level such that $r_v.level = r_u.level + 1$ if, $r_v.level > r_u.level + 1$ or $r_v.level = \perp$. r_v considers r_u as its parent, and the corresponding edge is considered as the edge of the BFS. In case more than one neighbour sends their level simultaneously (in the same round), such that they can be considered as the parent, then the minimum level neighbour is considered as the parent. In case there is more than one

such minimum considerable level sent, the one sent by the lower ID agent is considered as the parent. The agent informs its neighbours one by one based on port numbering (in increasing order) using the `Meeting with Neighbour()` that takes $O(4 \log \lambda)$ rounds per neighbour. Notice that to meet any neighbour `Meeting with Neighbour()` start the procedure in the round number that is divisible by $4 \log \lambda$ and this procedure runs for $2m$ times, after which each agent learns its actual level in the BFS tree.

Now, considering the basic algorithm (particularly *Phase 2*) described earlier in Section 5.5.1, we have two algorithms with round complexity $O(m \log n)$ (from Algorithm 17) and $O(D\Delta)$ (from 5.5.1). We consider the best of these two algorithms for BFS construction. For that, first, we run Algorithm 16; either Algorithm 16 terminates within $O(D\Delta)$ rounds, if so, then we have the BFS tree. Otherwise, we run the Algorithm 17 that terminates in $O(m \log n)$ rounds. Thus, we get the best round complexity of these two algorithms. The pseudocode for that is given in Algorithm 18.

Algorithm 17 $O(m \log n)$-Round BFS Construction.

Require: Graph G with a specified root and m.
Ensure: a BFS tree where each tree edge is identified by an agent.

1: Let us consider the agent r_u whose level is denoted by $r_u.level$. Initially, every r_u agent has $r_u.level = \perp$ and $r_u.parent \leftarrow \perp$.
2: Consider root agent $r^\star$ such that $r^\star.level = 0$.
3: $r^\star$ sends $r^\star.level$ to its neighbours one-by-one using the meeting neighbours algorithm `Meeting with Neighbors()` (Algorithm 10). ▷ Takes $4 \log \lambda$ rounds
4: **for** $2m$ time **do**
5: **if** agent r_u updates its level **then**
6: r_u sends $r_u.level$ to neighbours one-by-one using the `Meeting with Neighbour()`. ▷ Takes $4 \log \lambda$ rounds
7: **end if**
Let $\ell = \min\{L(r_v) \mid r_v \in H\}$ where $L(r_v)$ denotes the level of a neighbour r_v in the set H.
8: **if** r_u receives ℓ such that $(\ell + 1 < r_u.level)$ or $(r_u.level = \perp)$ **then**
9: $r_u.level \leftarrow \ell + 1$
10: Choose an arbitrary $r_v \in H$ such that $L(r_v) = \ell$
11: $r_u.parent \leftarrow r_v$
12: **end if**
13: **end for**

Lemma 5.7. *Algorithm 17 correctly constructs a BFS tree.*

Proof. We prove two cases to show that Algorithm 17 constructs a BFS. i) Each agent r_v is assigned a minimum level (shortest distance from the root). ii) There does not exist a cycle. We prove these

Algorithm 18 IMPROVED BFS CONSTRUCTION.

Require: Graph G having n nodes and n agents in a dispersed configuration.
Ensure: BFS construction.

1: Elect the leader/root by running MST Algorithm 15; during this phase, the root learns the graph parameters m, λ, and Δ in $O(\Delta \log^2 n + n \log n)$ rounds. See Section 5.4.
2: Root $r^\star$ initiates the BFS construction from Algorithm 16, but only up to $8m \log \lambda$ rounds.
3: **if** the BFS construction has completed within $8m \log \lambda$ rounds **then**
4: Terminate.
5: **else**
6: Root $r^\star$ runs Algorithm 17.
7: **end if**

cases by contradiction. Let us assume that the level assigned to an agent r_v is not the minimum, then in the Algorithm 17, if the level assigned to an agent r_v is not minimum, then there exists a neighbour of r_v, say r_u, such that $r_u.level + 1 < r_v.level$ or $r_v.level = \perp$. In both conditions, the neighbour r_u did not pass its level to r_v, which is contradictory to the Line 5. If r_u also does not have the minimum level, then one of the neighbours did not pass its level to r_u and so on. Eventually, by induction, we have that $r^\star$ does not pass its level to one of its neighbours, which is a contradiction to the Line 3.

For the second case, let us consider agents $r_1, r_2, r_3, \ldots, r_w$ forms a cycle such that (w.l.g.) $r_2.parent \leftarrow r_1, r_3.parent \leftarrow r_2$, ..., $r_1.parent \leftarrow r_w$. This implies $r_1.level > r_2.level > r_3.level > \cdots > r_w.level > r_1.level$. This implies that there exists more than one parent to an agent, which is contradictory. □

Lemma 5.8. *Algorithm 17 constructs a BFS tree in $O(m \log \lambda)$ rounds and requires $O(\log n)$ bits of memory at each agent.*

Proof. Let us assume there exists an agent r_v that requires $8m \log \lambda + t$ rounds to get the minimum level, for any $t > 0$. $r_v.parent$'s, say r_u, took at most $4 \log \lambda \cdot \delta(u)$ rounds to send the minimum level to r_v. Similarly, $r_u.parent$'s, say r_w, took at most $4 \log \lambda \cdot \delta(w)$ rounds to send the minimum level to r_u and so on. Therefore, the total number of edges in the shortest path, from $r^\star$ to r_v is not more than the sum of the degrees of $r_v.parent$'s and their parent, recursively, up to $r^\star$. As a result, this degree sum is not more than $2m$. Recall that communication between any two neighbouring agents takes not more than $4 \log \lambda$ rounds (Algorithm 10). Consequently, the total number of rounds required to send the minimum level to any of the agents is not more than $8m \log \lambda$, which is a contradiction. In the case of memory per agent, the Algorithm 17 stores only its parent's port number and level, which does not require more than $O(\log n)$ bits at each agent.

Hence, the lemma. □

Remark: 8. *Starting from an arbitrary initial configuration, agents can first achieve dispersion in $O(n)$ rounds [70], provided that n is known. Once dispersion is reached, the BFS construction completes in $O(\min(D\Delta, m \log n) + n \log n + \Delta \log^2 n)$ rounds (Theorem 5.5). Consequently, the overall round and memory complexity for BFS from an arbitrary initial configuration remains unchanged.*

From the above discussion, we have the following theorem.

Theorem 5.5 (BFS Tree). *Given a dispersed configuration of n agents with unique identifiers positioned one agent at every node of n nodes m edges, maximum degree Δ, and diameter D, graph G with no node identifier. Then, there is a deterministic algorithm that constructs a BFS tree in $O(\min(D\Delta, m \log n) + n \log n + \Delta \log^2 n)$ rounds using $O(\log n)$ bits per agent. The algorithm requires no prior knowledge of any graph parameters and assumes that each agent knows only λ, the maximum among all agent identifiers.*

Remark: 9. *The term $O(n \log n + \Delta \log^2 n)$ in the root-free BFS construction comes from obtaining a leader through the MST construction. If a root is already known, this preliminary phase is not needed, and the BFS tree can be constructed directly from the given root.*

5.6 Conclusion and Future Work

In this chapter, we studied the complexity of constructing Minimum Spanning Tree (MST) and Breadth-First Search (BFS) trees using mobile agents in arbitrary anonymous graphs. We presented an efficient MST construction algorithm that, beyond its independent significance, also enables leader election and the collection of useful graph parameters, improving upon prior agent-based constructions. We then presented BFS constructions for both known-root and root-free settings. In particular, the known-root BFS construction is independent of the MST construction, while the root-free BFS construction uses the leader obtained from the MST phase to avoid assuming a designated root in advance. Both the MST and BFS algorithms rely on a new neighbour-meeting primitive that guarantees information exchange between adjacent agents within $O(\log n)$ rounds, without assuming prior knowledge of global graph parameters. In particular, this primitive removes the standard dependence on knowing Δ in advance and may help accelerate several existing agent-based procedures that rely on Δ-dependent waiting schedules. As future work, it would be interesting to establish matching lower bounds on the round complexity of agent-based MST and BFS constructions, and to extend the proposed framework to crash- and Byzantine-fault settings motivated by real-world deployments.

Chapter 6

Conclusion and Future Works

This thesis presented a comprehensive exploration of distributed graph computations using autonomous mobile agents. We investigated a wide range of classical graph problems, including dispersion, dominating set, triangle and butterfly counting, BFS and MST construction, through the algorithmic lens of mobile agents operating on anonymous, port-labelled graphs with limited memory and local communication capabilities.

Our first contribution addressed the dispersion problem in the presence of crash faults. We designed fault-tolerant algorithms that operate under both rooted and arbitrary initial configurations, with provable round and memory complexity bounds. These results not only extend the theoretical foundations of agent dispersion but also provide theoretical foundations that may support practical deployment; in dynamic environments where faults are inevitable. In this thesis, dispersion also serves as an important primitive for several later computations. Once the agents are dispersed, each agent can act as a local representative of a node, which helps in designing algorithms that resemble distributed graph computations while still respecting the restrictions of the mobile agent model.

We developed algorithms for computing minimal and approximate dominating sets, together with related independent-set based techniques. Our methods accommodated both single-source and multi-source initialisations, and included an approximation algorithm that produces near-optimal dominating sets within logarithmic approximation factors. These results are relevant for applications like network backbone formation and distributed monitoring.

In the context of subgraph enumeration, we introduced agent-based algorithms for triangle counting and used these techniques to solve the truss decomposition problem and to compute triangle centrality and local clustering coefficient. For bipartite graphs, we proposed butterfly counting algorithms, which form a critical building block for analysing dense substructures in two-mode networks. These algorithms are not only efficient in terms of memory and communication but are also scalable to large graphs.

The thesis further examined BFS tree construction under different agent configurations. Starting with the case where the BFS root is known, we provided time and memory-efficient algorithms that also support practical applications such as bipartiteness testing and agent gathering. We then

improved this methodology by removing the root assumption, reducing the round complexity, and introducing a novel neighbour-meeting protocol. This protocol enabled agents to coordinate without knowledge of global parameters such as the maximum degree, and played a central role in enabling decentralised MST construction and leader election.

Looking ahead, several compelling directions arise from our work. A deeper understanding of lower bounds for the problems studied here — in terms of time, space, and number of agents — could lead to more optimal algorithms and tighter analyses. Fault tolerance remains a rich area for further exploration, particularly under more severe adversarial models such as the Byzantine setting, where faulty agents may behave arbitrarily or maliciously. Another natural progression is to consider algorithms that make fewer assumptions about global knowledge; for instance, the highest agent ID λ. This is especially important in settings where agents must self-organise in entirely unknown environments.

Another interesting direction is to study these problems with fewer agents. In several parts of this thesis, a dispersed configuration is used as a natural starting point. This gives a clean setting where each node has an agent that can store local information and participate in the computation. However, it is natural to ask how far these ideas can be extended when the number of agents is less than the number of nodes. For example, in triangle counting, truss decomposition, and related problems, one extreme case is to solve the problem using a single agent by first constructing a complete map of the graph. However, in anonymous graphs where nodes do not have identifiers, efficient exploration is highly challenging. As shown in [23], if the agent has no prior knowledge of n, the exploration requires $O(n^6\Delta^2 \log n)$ rounds and $O(n^3\Delta^2 \log n \log \Delta)$ bits of memory. By contrast, if a single node is marked, or equivalently if two agents are used and one agent remains fixed as a marker, the exploration can be completed in $O(n^3\Delta)$ rounds using only $O(n\Delta \log n)$ bits of memory. Other results in the same direction also show a severe trade-off: optimising memory complexity may lead to very high, even exponential, time complexity. This highlights the difficulty of solving such problems with only a constant number of agents.

Between the two extremes of one agent and one agent per node, there is a large and interesting middle ground. If a larger number of agents is available, then they may communicate among themselves and reduce the amount of exploration needed. However, the extent of this reduction depends heavily on the network topology, the number of agents, and their initial placement. For instance, if we consider a graph composed of a series of triangles, each connected to the next by a single edge. Suppose one agent is positioned at a degree-two node in each triangle. In this setup, the graph has n nodes and $n/3$ agents. Each agent can locally cover its own triangle, since all three nodes of the triangle are adjacent to it. Yet, despite this coverage, no agent may encounter

another agent, even if it waits indefinitely at each of its neighbouring nodes. Consequently, even with $O(n)$ agents, locating another agent for communication and making progress may still require searches of order $O(\Delta^2)$ under a natural port-based search strategy. In addition, when the number of agents is smaller than the number of nodes, the agents may need to oscillate between several nodes in order to maintain coverage over time. This suggests a rich trade-off among the number of agents, memory per agent, graph topology, and total time complexity. A particularly interesting question is the following: if the agents are deployed so that they are roughly t hops apart, can communication be achieved with optimal memory in $O(\Delta^t \log \lambda)$ rounds?

The scope for incorporating heterogeneous agents—differing in mobility, sensing, or communication capabilities—opens up a new dimension for both theoretical modelling and practical deployment. Furthermore, many classical graph problems, such as connected dominating set, graph colouring, and ruling sets, remain largely unexplored in the mobile agent setting, and they represent valuable future directions. Our work on triangle and butterfly structures also lays the groundwork for studying higher-order structures like tip and wing decomposition, which are useful for understanding the hierarchical organisation of dense subgraphs in bipartite networks.

Another important future direction is to study these algorithms under more realistic execution models. The algorithms developed in this thesis are primarily analysed in a synchronous round-based setting, where movement, communication, and coordination occur in a structured manner. However, in practical mobile-agent or multi-agent systems, several additional factors may influence the performance of the algorithms. For example, communication between nearby agents may suffer from delays, movement along edges may incur non-uniform costs, and agents may have only restricted opportunities to coordinate with one another. These issues become even more significant in partially asynchronous settings, where agents may not move or communicate in perfectly synchronised rounds. Similarly, limited communication environments, where agents can exchange only small messages or communicate intermittently, may require new techniques for synchronisation, information propagation, and termination detection. Extending the algorithms of this thesis to such settings would help bridge the gap between the theoretical mobile-agent model and practical multi-agent systems.

Finally, while our contributions have been theoretical, implementing these algorithms in realistic settings, either through simulations or on multi-agent platforms, would provide important empirical validation. This would allow us to understand the robustness, scalability, and practical trade-offs of our techniques under various forms of uncertainty.

In conclusion, this thesis offers a unified framework for solving a broad class of graph problems using mobile agents, balancing algorithmic rigour with practical considerations. Dispersion

provides one important starting point for this framework, while the future directions above suggest how the model can be extended beyond fully dispersed configurations and toward settings with fewer agents, weaker assumptions, and richer forms of coordination. We believe that the insights and methods developed here will serve as a foundation for advancing distributed coordination and computation in multi-agent systems across both theory and application.

Bibliography

[1] Wali Mohammad Abdullah, David Awosoga, and Shahadat Hossain. Efficient calculation of triangle centrality in big data networks. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2022.

[2] Ankush Agarwalla, John Augustine, William K. Moses Jr., Sankar Madhav K., and Arvind Krishna Sridhar. Deterministic dispersion of mobile robots in dynamic rings. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 19:1–19:4. ACM, 2018.

[3] Esra Akbas and Peixiang Zhao. Truss-based community search: a truss-equivalence based indexing approach. *Proc. VLDB Endow.*, 10(11):1298–1309, August 2017.

[4] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks*, 5(4):581–603, 2017.

[5] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[6] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: a parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, page 529–538, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Trans. Knowl. Discov. Data*, 14(1), December 2019.

[8] Algorithmic aspects of triangle-based network analysis. *Schank, Thomas.* PhD thesis, Institut für Theoretische Informatik (ITI), 2007.

[9] John Augustine and William K. Moses Jr. Dispersion of mobile robots: A study of memory-time trade-offs. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 1:1–1:10. ACM, 2018.

[10] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 230–240, New York, NY, USA, 1987. Association for Computing Machinery.

[11] B. Awerbuch and R. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33(3):315–322, 1987.

[12] Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot (extended abstract). In *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, COLT '95, page 321–328, New York, NY, USA, 1995. Association for Computing Machinery.

[13] Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.

[14] Evangelos Bampas, Leszek Gasieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, and Adrian Kosowski. Euler tour lock-in problem in the rotor-router model. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, volume 5805 of *Lecture Notes in Computer Science*, pages 423–435. Springer, 2009.

[15] Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. Optimal fault-tolerant dispersion on oriented grids. In *Proceedings of the 26th International Conference on Distributed Computing and Networking*, ICDCN '25, page 254–258, New York, NY, USA, 2025. Association for Computing Machinery.

[16] Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. Optimizing robot dispersion on unoriented grids: With and without fault tolerance. In Quentin Bramas, Arnaud Casteigts, and Kitty Meeks, editors, *Algorithmics of Wireless Networks*, pages 31–45, Cham, 2025. Springer Nature Switzerland.

[17] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, page 623–632, USA, 2002. Society for Industrial and Applied Mathematics.

[18] Scott Beamer, Aydin Buluç, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1618–1627, 2013.

[19] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '08, page 16–24, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Ulrik Brandes. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[21] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Paul Burkhardt. Triangle centrality. *ACM Trans. Knowl. Discov. Data*, 18(9), October 2024.

[23] Jérémie Chalopin, Shantanu Das, and Adrian Kosowski. Constructing a map of an anonymous graph: Applications of universal sequences. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, Lecture Notes in Computer Science, pages 119–134. Springer, 2010.

[24] Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla. Agent-based triangle counting and its applications in anonymous graphs. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '24, page 2186–2188, Richland, SC, 2024. International Foundation for Autonomous Agents and Multiagent Systems.

[25] Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Fault-tolerant dispersion of mobile robots. In *Algorithms and Discrete Applied Mathematics - 9th International Conference, CALDAM 2023, Gandhinagar, India, February 9-11, 2023, Proceedings*, volume 13947 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2023.

[26] Prabhat Kumar Chand, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Run for cover: Dominating set via mobile agents. In Konstantinos Georgiou and Evangelos Kranakis, editors, *Algorithmics of Wireless Networks*, pages 133–150, Cham, 2023. Springer Nature Switzerland.

[27] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. Distributed algorithms for k-truss decomposition. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 471–480, 2014.

[28] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, SE-9(4):504–512, 1983.

[29] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[30] Edmond Chow, Keith W. Henderson, and A Yoo. Distributed breadth-first search with 2-d partitioning. In *LLNL Technical Report*, 2005.

[31] Hsin Chuang, Kuan-Lin Hou, Seungmin Rho, and Bo-Wei Chen. Cooperative comodule discovery for swarm-intelligent drone arrays. *Computer Communications*, 154:528–533, 2020.

[32] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 2008.

[33] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, 2008.

[34] Yang Cong, Changjun Gu, Tao Zhang, and Yajun Gao. Underwater robot sensing technology: A survey. *Fundamental Research*, 1(3):337–345, 2021.

[35] Archak Das, Kaustav Bose, and Buddhadeb Sau. Memory optimal dispersion by anonymous mobile robots. In *Algorithms and Discrete Applied Mathematics - 7th International Conference, CALDAM 2021, Rupnagar, India, February 11-13, 2021, Proceedings*, volume 12601 of *Lecture Notes in Computer Science*, pages 426–439. Springer, 2021.

[36] Raja Das, Avisek Sharma, and Buddhadeb Sau. Maximum independent set formation on a finite grid by myopic robots. *arXiv preprint arXiv:2207.13403*, 2022.

[37] Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemysław Uznanski. Fast collaborative graph exploration. *Information and Computation*, 243:37–49, 2015. 40th International Colloquium on Automata, Languages and Programming (ICALP 2013).

[38] Janosch Deurer, Fabian Kuhn, and Yannic Maus. Deterministic distributed dominating set approximation in the congest model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 94–103, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Matthew Dunbabin and Lino Marques. Robots for environmental monitoring: Significant advancements and applications. *IEEE Robotics & Automation Magazine*, 19(1):24–39, 2012.

[40] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences*, 99(9):5825–5829, 2002.

[41] Abdulrahman M. El-Sayed, Peter Scarborough, Laura Seemann, and Sandro Galea. Social network analysis and agent-based modeling in social epidemiology. *Epidemiologic Perspectives & Innovations*, 9(1):1, February 2012.

[42] Fatemeh Esfahani, Mahsa Daneshmand, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Truss decomposition on large probabilistic networks using h-index. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, SSDBM '21, page 145–156, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, 2005.

[44] Greg N. Frederickson. A single source shortest path algorithm for a planar distributed network. In *Proceedings of the 2nd Symposium of Theoretical Aspects of Computer Science*, STACS '85, page 143–150, Berlin, Heidelberg, 1985. Springer-Verlag.

[45] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[46] R.G. Gallager. Distributed minimum hop algorithms. *M.I.T. Technical Report*, 1982.

[47] Robert G. Gallager and Baruch Awerbuch. Distributed BFS algorithms . In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 250–256, Los Alamitos, CA, USA, October 1985. IEEE Computer Society.

[48] Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

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

[50] Sayan Ghosh. Improved distributed-memory triangle counting by exploiting the graph structure. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2022.

[51] Sayan Ghosh and Mahantesh Halappanavar. Tric: Distributed-memory triangle counting by exploiting the graph structure. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2020.

[52] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[53] Tien-Ruey Hsiang, Esther M. Arkin, Michael A. Bender, Sandor Fekete, and Joseph S. B. Mitchell. Online dispersion algorithms for swarms of robots. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, SCG '03, page 382–383, New York, NY, USA, 2003. Association for Computing Machinery.

[54] Tien-Ruey Hsiang, Esther M. Arkin, Michael A. Bender, Sándor P. Fekete, and Joseph S. B. Mitchell. Algorithms for rapidly dispersing robot swarms in unknown environments. In *Algorithmic Foundations of Robotics V, Selected Contributions of the Fifth International Workshop on the Algorithmic Foundations of Robotics, WAFR 2002, Nice, France, December 15-17, 2002*, volume 7 of *Springer Tracts in Advanced Robotics*, pages 77–94. Springer, 2002.

[55] Hailong Huang, Andrey V. Savkin, Ming Ding, and Chao Huang. Mobile robots in wireless sensor networks: A survey on tasks. *Computer Networks*, 148:1–19, 2019.

[56] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1311–1322, New York, NY, USA, 2014. Association for Computing Machinery.

[57] Xin Huang, Wei Lu, and Laks V.S. Lakshmanan. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 77–90, New York, NY, USA, 2016. Association for Computing Machinery.

[58] Volkan Isler. Robotic sensor networks for environmental monitoring. In *AAAI Spring Symposium: AI, The Fundamental Social Aggregation Challenge*, 2012.

[59] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 1–10, New York, NY, USA, 1977. Association for Computing Machinery.

[60] Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distrib. Comput.*, 15(4):193–205, 2002.

[61] Songwei Jia, Lin Gao, Yong Gao, and Haiyang Wang. Anti-triangle centrality-based community detection in complex networks. *IET Systems Biology*, 8(3):116–125, 2014.

[62] Humayun Kabir and Kamesh Madduri. Shared-memory graph truss decomposition. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 13–22, 2017.

[63] Sayaka Kamei and Sébastien Tixeuil. An asynchronous maximum independent set algorithm by myopic luminous robots on grids. *arXiv preprint arXiv:2012.03399*, 2020.

[64] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[65] Anzuruni Maulidi Katunka, Cairong Yan, Kossonon Brandou Serge, and Zhaohui Zhang. K-truss based top-communities search in large graphs. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 244–249, 2017.

[66] Tanvir Kaur and Kaushik Mondal. Distance-2-dispersion: Dispersion with further constraints. In *Networked Systems: 11th International Conference, NETYS 2023, Benguerir, Morocco, May 22–24, 2023, Proceedings*, page 157–173, Berlin, Heidelberg, 2023. Springer-Verlag.

[67] Ajay D. Kshemkalyani and Faizan Ali. Fast graph exploration by a mobile robot. In *First IEEE International Conference on Artificial Intelligence and Knowledge Engineering, AIKE 2018, Laguna Hills, CA, USA, September 26-28, 2018*, pages 115–118. IEEE Computer Society, 2018.

[68] Ajay D. Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 218–227. ACM, 2019.

[69] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. Brief announcement: Agent-based leader election, mst, and beyond. In *38th International Symposium on Distributed Computing, DISC 2024, October 28 to November 1, 2024, Madrid, Spain*, volume 319 of *LIPIcs*, pages 50:1–50:7, 2024.

[70] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion is (almost) optimal under (a)synchrony, 2025. Preprint. arXiv. 2503.16216.

[71] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. Faster leader election and its applications for mobile agents with parameter advice. In *Distributed Computing and Intelligent Technology: 21st International Conference, ICDCIT 2025, Bhubaneswar, India, January 8–11, 2025, Proceedings*, page 108–123, Berlin, Heidelberg, 2025. Springer-Verlag.

[72] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *Algorithms for Sensor Systems - 15th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2019, Munich, Germany, September 12-13, 2019, Revised Selected Papers*, volume 11931 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2019.

[73] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings*, volume 12049 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2020.

[74] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Efficient dispersion of mobile robots on dynamic graphs. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 732–742. IEEE, 2020.

[75] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots using global communication. *J. Parallel Distributed Comput.*, 161:100–117, 2022.

[76] Ajay D. Kshemkalyani and Gokarna Sharma. Near-optimal dispersion on arbitrary anonymous graphs. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[77] Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. What cannot be computed locally! In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, page 300–309, New York, NY, USA, 2004. Association for Computing Machinery.

[78] Fabian Kuhn and Rogert Wattenhofer. Constant-time distributed dominating set approximation. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, page 25–32, New York, NY, USA, 2003. Association for Computing Machinery.

[79] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1):7:1–7:27, 2015.

[80] Gérard Le Lann. Distributed systems - towards a formal approach. In Bruce Gilchrist, editor, *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, pages 155–160. North-Holland, 1977.

[81] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458–473, 2008.

[82] Jaeyeong Lee, Sunwoo Shin, Moonsung Park, and Chongman Kim. Agent-based simulation and its application to analyze combat effectiveness in network-centric warfare considering communication failure environments. *Mathematical Problems in Engineering*, 2018(1):2730671, 2018.

[83] Itay Levinas, Roy Scherz, and Yoram Louzoun. Bfs-based distributed algorithm for parallel local-directed subgraph enumeration. *Journal of Complex Networks*, 10(6):cnac051, 12 2022.

[84] Fuhuan Li and David A. Bader. A graphblas implementation of triangle centrality. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2021.

[85] B. Liang and Z.J. Haas. Virtual backbone generation and maintenance in ad hoc network mobility management. In *IEEE INFOCOM*, pages 1293–1302 Vol.3, 2000.

[86] Chengyao Liu. Optimization communication for BFS based on 1d-partition. In *Journal of Physics: Conference Series*, volume 1684, page 012125. IOP Publishing, 2020. The 2020 International Seminar on Artificial Intelligence, Networking and Information Technology, 18–20 September 2020, Shanghai, China.

[87] Giuseppe Antonio Di Luna, Paola Flocchini, Linda Pagli, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Gathering in dynamic rings. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, volume 10641 of *Lecture Notes in Computer Science*, pages 339–355. Springer, 2017.

[88] Linyuan Lü, Tao Zhou, Qiming Zhang, Hai-Feng Zhang, and Yi-Cheng Zhang. The h-index of a network node and its relation to degree and coreness. *Nature Communications*, 7:10168, 2016.

[89] S. A. M. Makki. Efficient distributed breadth-first search algorithm. *Comput. Commun.*, 19(8):628–636, July 1996.

[90] Subhrangsu Mandal, Anisur Rahaman Molla, and William K. Moses Jr. Live exploration with mobile robots in a dynamic ring, revisited. In *Algorithms for Sensor Systems - 16th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2020, Pisa, Italy, September 9-10, 2020, Revised Selected Papers*, volume 12503 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2020.

[91] S. Meivel, Nidhi Sindhwani, Rohit Anand, Digvijay Pandey, Abeer Ali Alnuaim, Alaa S. Altheneyan, Mohamed Yaseen Jabarulla, and Mesfin Esayas Lelisho. Mask detection and social distance identification using internet of things and faster r-cnn algorithm. *Computational Intelligence and Neuroscience*, 2022(1):2103975, 2022.

[92] Anisur Rahaman Molla and William K. Moses Jr. Dispersion of mobile robots: The power of randomness. In *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019, Kitakyushu, Japan, April 13-16, 2019, Proceedings*, volume 11436 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2019.

[93] Anisur Rahaman Molla and William K. Moses Jr. Dispersion of mobile robots. In *ICDCN '22: 23rd International Conference on Distributed Computing and Networking, Delhi, India, January 4 - 7, 2022*, pages 217–220. ACM, 2022.

[94] Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Efficient dispersion on an anonymous ring in the presence of weak byzantine robots. In *Algorithms for Sensor Systems - 16th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2020, Pisa, Italy, September 9-10, 2020, Revised Selected Papers*, volume 12503 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2020.

[95] Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Byzantine dispersion on graphs. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 942–951. IEEE, 2021.

[96] Vigneshwaran Palanisamy and Senthooran Vijayanathan. Cluster based multi agent system for breadth first search. In *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 54–58, 2020.

[97] Ha-Myung Park, Francesco Silvestri, U. Kang, and Rasmus Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, page 1739–1748, New York, NY, USA, 2014. Association for Computing Machinery.

[98] Debasish Pattanayak, Subhash Bhagat, Sruti Gan Chaudhuri, and Anisur Rahaman Molla. Maximal independent set via mobile agents. In *Proceedings of the 25th International Conference on Distributed Computing and Networking*, ICDCN '24, page 74–83, New York, NY, USA, 2024. Association for Computing Machinery.

[99] Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. Dispersion of mobile robots tolerating faults. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021, Adjunct Volume*, pages 133–138. ACM, 2021.

[100] Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. Dispersion of mobile robots in spite of faults. In *Stabilization, Safety, and Security of Distributed Systems: 25th International Symposium, SSS 2023, Jersey City, NJ, USA, October 2–4, 2023, Proceedings*, page 414–429, Berlin, Heidelberg, 2023. Springer-Verlag.

[101] D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999.

[102] David Peleg. Time-optimal leader election in general networks. *J. Parallel Distrib. Comput.*, 8(1):96–99, jan 1990.

[103] Subhajit Pramanick, Sai Vamshi Samala, Debasish Pattanayak, and Partha Sarathi Mandal. Filling mis vertices of a graph by myopic luminous robots. In Anisur Rahaman Molla, Gokarna Sharma, Pradeep Kumar, and Sanjay Rawat, editors, *Distributed Computing and Intelligent Technology*, pages 3–19, Cham, 2023. Springer Nature Switzerland.

[104] Mohamed Rihan, Mahmoud M. Selim, Chen Xu, and Lei Huang. D2d communication underlaying uav on multiple bands in disaster area: Stochastic geometry analysis. *IEEE Access*, 7:156646–156658, 2019.

[105] H. Ryu and W. K. Chung. Local map-based exploration using a breadth-first search algorithm for mobile robots. *International Journal of Precision Engineering and Manufacturing*, 16(10):2073–2080, 2015.

[106] Peter Sanders and Tim Niklas Uhl. Engineering a distributed-memory triangle counting algorithm. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 702–712, 2023.

[107] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2150–2159, 2018.

[108] Ahmet Erdem Sarıyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 504–512, 2018.

[109] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. Parallel local algorithms for core, truss, and nucleus decompositions. *CoRR*, abs/1704.00386, 2017.

[110] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In *Massive Graph Analytics*, pages 287–330. Chapman and Hall/CRC, 2022.

[111] Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Efficient dispersion of mobile agents without global knowledge. In *Stabilization, Safety, and Security of Distributed Systems - 22nd International Symposium, SSS 2020, Austin, TX, USA, November 18-21, 2020, Proceedings*, volume 12514 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2020.

[112] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, 2015.

[113] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, 2014.

[114] Philip Spira. Communication complexity of distributed minimum spanning tree algorithms. In *Proceedings of the second Berkeley conference on distributed data management and computer networks*, pages 236–244, 1977.

[115] Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. An agent exploration in unknown undirected graphs with whiteboards. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security*, WRAS '10, pages 1–6, New York, NY, USA, 2010. Association for Computing Machinery.

[116] Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Near-Linear Time Dispersion of Mobile Agents. In Dan Alistarh, editor, *38th International Symposium on Distributed Computing (DISC 2024)*, volume 319 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[117] Evan A. Sultanik, Ali Shokoufandeh, and William C. Regli. Dominating sets of agents in visibility graphs: distributed algorithms for art gallery problems. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, page 797–804, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[118] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 607–614, New York, NY, USA, 2011. Association for Computing Machinery.

[119] Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 2014.

[120] Charalampos E. Tsourakakis, Petros Drineas, Elias Michelakis, Michalis Faloutsos, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1:75–81, 2011.

[121] Kota Ueno, Toyotaro Suzumura, Noriyuki Maruyama, and Satoshi Matsuoka. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering*, 2(1):22–35, 2017.

[122] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[123] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.

[124] Jia Wang, Ada Wai-Chee Fu, and James Cheng. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*, pages 17–24, 2014.

[125] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proc. VLDB Endow.*, 12(10):1139–1152, June 2019.

[126] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 661–672. IEEE, 2020.

[127] Roger Wattenhofer. Chapter 12, lecture notes: Principles of distributed computing, 2004.

[128] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[129] Tongfeng Weng, Xu Zhou, Kenli Li, Kian-Lee Tan, and Keqin Li. Distributed approaches to butterfly analysis on large dynamic bipartite graphs. *IEEE Transactions on Parallel and Distributed Systems*, 34(2):431–445, 2022.

[130] Grzegorz Wilk-Jakubowski, Radoslaw Harabin, and Stanislav Ivanov. Robotics in crisis management: A review. *Technology in Society*, 68:101935, 2022.

[131] Jian Wu, Alison Goshulak, Venkatesh Srinivasan, and Alex Thomo. K-truss decomposition of large networks on a single consumer-grade machine. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 873–880, 2018.

[132] M. Wu. Robotics applications in natural hazards. *Highlights in Science, Engineering and Technology*, 43:273–279, April 2023.

[133] Qingyu Xu, Feng Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, and Bingsheng He. Efficient load-balanced butterfly counting on gpu. *Proceedings of the VLDB Endowment*, 15(11):2450–2462, 2022.

[134] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 25–25, 2005.

[135] Nan Zhao, Weidang Lu, Min Sheng, Yunfei Chen, Jie Tang, F. Richard Yu, and Kai-Kit Wong. Uav-assisted emergency networks in disasters. *IEEE Wireless Communications*, 26(1):45–51, 2019.

[136] Alexander Zhou, Yue Wang, and Lei Chen. Butterfly counting on uncertain bipartite graphs. *Proceedings of the VLDB Endowment*, 15(2):211–223, 2021.

[137] Chengxiang Zhuge, Chunfu Shao, and Binru Wei. An agent-based spatial urban social network generator: A case study of beijing, china. *Journal of computational science*, 2018.