Majority Problems in Distributed Systems and Clustering in Structured Graphs

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy by

David Douglas Hamilton

Supervisors:
Prof. Leszek Gąsieniec
Dr. Russell Martin

June 2017

Contents

Illustrations

List of Figures

List of Tables

Notations

The following notations and abbreviations are found throughout this thesis:

MCP	**M**ajority **C**olor **P**roblem
w.h.p	**W**ith **H**igh **P**robability
BASIC	Our proposed algorithm to solve the original MCP.
APC	Our proposed clustering algorithm, **A**gglomerative **P**hylogenetic **C**lustering.
PCG	Constraint generation methodology, **P**airwise **C**onstraint **G**eneration.
TCG	Constraint generation methodology, **T**ernary **C**onstraint **G**eneration.

UNIVERSITY OF LIVERPOOL

Abstract

Faculty of Science
Department of Computer Science

Doctor in Philosophy

by David Douglas Hamilton

This thesis focuses on the study of various algorithms for Distributed Computing and Machine Learning research areas. More precisely, the work within contains research into various communication protocols in different settings of Distributed Computing, accompanied by relevant analysis on protocol performance in time and space. These protocols are designed to operate in analogous environments though using different models for communication, primarily population protocol and random walk variants. In our settings we aim to use as minimal memory as possible, achieving light weight protocols that are powerful in their capabilities and randomized as well as deterministic in nature. We also propose a novel technique of verification which enables multi-step protocols to work in synergy. These protocols generally never terminate, but converge and are difficult to disseminate results throughout the network to be used in dependent processes. With the verification technique proposed, protocols can become adaptive and stacked into a chain of dependent processes.

We also provide experimental analysis of a subarea of Machine Learning, unsupervised clustering algorithms. Gaining inspiration from the agglomerative nature and techniques defined in classical hierarchical clustering as well as the Phylogenetic tree building methods, we provide a comprehensive study and evaluation of new method to agglomeratively combine 'similar' data into clusters based on the general consensus of taxonomy and evaluation of clustering mechanisms.

Acknowledgements

Firstly, I would like to express my gratitude to my supervisors, Dr Russell Martin and Professor Leszek Gąsieniec. It has been a privilege to be imparted with their knowledge and guidance throughout the course of my PhD study. The experiences attained as their student has defined who I am today.
I would also like to thank my academic advisors, Dr. Davide Grossi and Professor Dariusz Kowalski for their time and feedback at critical moments throughout the course of my study.

Secondly, I would also like to acknowledge and thank all of my co-authors for their contributions to the work in our papers and journal submission. The knowledge and experience gained from the collaborations has been invaluable.

Thirdly, I would like to thank my family, Mary Hamilton, Sarah Hamilton and David Evans, partner, Wei Yue and friends for their patience, support and understanding throughout my study.

Lastly, I would like to thank all my colleagues from the Department of Computer Science at The University of Liverpool for the experiences we have shared over the years.

I dedicate this thesis to my father, Douglas Hamilton.

Chapter 1

Introduction

1.1 Problem Scope and Motivation

The work presented in this thesis concentrates on problems under the umbrella of Distributed Computing and the Machine Learning area of Computer Science. More precisely the work focuses on Majority Problems in Random Walk and Population Protocol models in distributed settings as well as Unsupervised Clustering Problems. The distributed models we consider involve agents whose computational power is limited in some fashion, contrary to models allowing more powerful agents.

We live in an information age. Computing in various degrees is ubiquitous, in every walk of life, it is ever present. As a complex society, we are becoming ever more reliant on technology. We have came a long way from the advent of computers, being feasible for solely large corporations, transcending to consumers having their own at home - which was thought never possible. Further along in recent years which have focused on desktop machines but the adoption rate of mobile devices is well past the tipping point. The number of smartphone owners has increased from 400 million to 1.9 billion in the past eight years. These smart devices allow us to always be connected to services and each other via the internet.

As a society and in modern culture, we are wanting to be always connected. We yearn for tasks to be automated, or aid in decision making - so much so that phone applications can control the energy usage in our homes, turn off our lights and appliances from anywhere and now more recently, control our cars ignition, with the cars being able to drive themselves.

Some of these applications can be achieved on a single device efficiently, but other systems may not perform so well. This could be a result of the application being too complex that it takes too much processing power to compute in a feasible amount of time - or it may be simpler, but the quantity of data data is so large that it has an adverse affect on computational performance. After all, by always being connected and using online services we create 2.5 quintillion bytes of data daily, 90% of the worlds data being produced in the last two years alone [80]. This has given rise to the information age. Vast amounts of data, generated from social networks, internet of things, multimedia,

too large and complex for traditional storage and computational mechanisms, referred to as 'Big Data' [74].

In light of the this, it could be argued that the uses and applications of technology in the modern era is outpacing Moore's Law for single machines, reinforced by recent announcements from hardware vendors that Moore's law is slowing [49] - exceeding expectations of the original prediction it would hold for a decade subsequent to the laws inception in 1965 [104]. It is not a sound strategy to rely on the density of transistors on silicon chips to double every two years as a solution to these problems.

The roots of distributed systems have been studied since the 1960s in operating system architectures in singular machines, with processes communicating via message passing models. The field advanced with the principles being applied to local networks, the predecessor of the internet, ARPANET, and ultimately the internet as we know it today. A shift to smarter systems in recent years is becoming standard to remedy the modern limitations on all singular technological devices - utilizing the power of multiple, distributed machines to complete tasks. Intelligently architected topologies that are scalable in terms of power and size, machines connected to each other able to work in synergy. Many businesses that harness 'Big Data' and the opportunities that come with it are opting to transfer their operations from utilizing independent servers to these intelligent systems, or 'clouds' referred to commonly. Most notable examples of these distributed systems are Microsoft's Azure platform [50] and Amazon Web Services platform [19]. There are many advantages of these systems, which include virtualized resources, parallel processing, security, and data service integration with scalable data storage [74]. The increase of organizations in industry shifting to these platforms is telling in isolation, further highlighting the drivers and motivation surrounding the topic that is distributed computing and algorithms being developed for them in the research community.

The same reasons motivate the work in this thesis regarding clustering algorithmics, a branch of unsupervised machine learning - albeit in a more experimental setting. With the rise of big data, as mentioned, concurrently rises problems with dealing with this data. Whether it be storage, manipulation or visualization, there is a need in research and industry to understand the data and derive meaningful information. There have been many advances in architectural solutions to store big data intelligently with the aim of fast data retrieval in mind, some examples being Hadoop [28] an alternative to the lack of scalability from the standard SQL solutions.

A growing problem with big data in research is developing algorithms to aid in discovering salient information from it that isn't easy to obtain heuristically. This is tackled under the umbrella of machine learning, with advances in supervised learning methods such as classification modelling, examples being determining if customers will buy a product or not (propensity to purchase), feeding back a binary result. Another supervised method of regression modelling, following on from the previous example, to determine how much revenue a customer will potentially generate, feeding back a real value. The subtopic of machine learning relevant to this thesis is unsupervised clustering algorithms,

which contrary to supervised, is more exploratory in nature. Clustering is used to group samples together based on some similarity metric, preprocessing data so that they and similar samples can be retrieved quickly or used as a form of meta analysis to derive information. Whether it be for grouping similar web pages together more effectively to provide users with almost instantaneous search results relating to keywords or grouping customers together based on buying habits, demographic or other user specified descriptors or even clustering DNA sequences based on edit distance. A result of the data deluge is large technology corporations such as Microsoft, Google, Amazon, Facebook, IBM are investing heavily in machine learning algorithmic research [108] and a large number of scientific applications for extensive experiments are currently deployed in the cloud and will continue to increase because of the lack of available computing facilities in local servers, reduced capital costs, and increasing volume of data produced and consumed by the experiments [100].

The parallels drawn between these research areas are evident in the models and methodologies we use to solve the problems in the following chapters. All scenarios are embedded in the foundation of graph theory. We consider the local relationships of vertices in graphs and how these can be used in the context of communication, information dissemination and also community derivation. The work in the following chapters is grounded in this premise and propose a variety of protocols to solve majority problems in random walk and population protocol models, as well as algorithms that use only local relationships to determine graph partitions.

1.2 Background

The following subsections discuss the recent work in related fields of research to the work presented in this thesis.

1.3 Algorithms

This section will describe useful algorithmic concepts, from design to analysis of performance. An algorithm is similar to a recipe in the sense of following a routine to solve some problem. More specifically, they are mathematical processes executing sequential instructions, or concurrent instructions to solve some problem that a computer understands.

1.3.1 Design

Designing algorithms is a complex task - they must be correct and also efficient. The concept of algorithm design as stated in [113] relies on two bodies of knowledge, techniques and resources.

Firstly, techniques. The area of algorithms is multidisciplinary in its facet. In the area of Computer Science specifically, algorithm design draws from and capitalizes on

the wealth of knowledge at our fingertips. For instance, there are various data structures that can be incorporated into an algorithm, each with its own insert, update and delete costs. There are other influencing techniques and paradigms such as dynamic programming, divide and conquer, back-tracking and many others. The decision of techniques to combine and use is as a result of each heuristic process.

Secondly, resources. As stated, the general area of computer algorithms is widely researched. There exists abundant research papers on any specific field, with constant advances pushing the boundaries of human knowledge. Understanding this and what algorithms are available already is greatly important in algorithm design. Currently known information can be built upon into and incorporated into newer algorithms.

1.3.2 Analysis

Algorithms can be studied and analyzed in a machine independent way, the take-home lesson in [113]. They can be compared and contrasted against each other in a variety of settings using asymptotic analysis of varying degrees of complexity - generally, without the need of implementation. Another machine independent way of determining an algorithms performance is based on the existence of a Random Access Machine (RAM) whereby each operation in the algorithm accessing memory is classed as one time step - although this method is not used in this thesis.

Asymptotic analysis is denoted using 'Big O Notation', a more abstract level of analysis which does not consider high levels of granularity such as the way an algorithm accesses memory, or how frequently it does so. Using said notation, the best, average and worse-case complexities of algorithms can be studied and compared easily and effectively, independent of machine or implementation.

Definition. Big O: A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items in the dataset.

Generally, algorithms are compared on their order of growth as a function relative to n, $f(n)$. There exist many esoteric functions, but the most commonly seen functions are constant, logarithmic, linear, quadratic, cubic and exponential in nature. The difference in time complexity between varying orders of growth becomes more apparent for larger instances of n.

For the work presented in this thesis, we analyze our proposed solutions to problems in terms of Big O notation. The speed and correctness all available and proven in subsequent chapters.

1.4 Distributed Computing

Two bodies of work constituting two chapters in this thesis are focused in the research area of Distributed Computing. We proposed multiple distributed algorithms in various

distributed settings and models, discussed in detail in later chapters. But firstly, some general definitions to supplement the work in Chapter 2 and Chapter 3.

Definition. Distributed Setting: There are many topologies of machines that make up a distributed setting, but generally these machines are networked in some way, whether it be wireless or physically and have no central or controlling aspect. These machines can then communicate via these physical or wireless links and pass messages through them.

Definition. Distributed Algorithm: Unlike sequential algorithms, these algorithms are designed to work in synergy although running concurrently on separate processors in different machines via communication protocols. Unlike parallel algorithms which have a shared memory resource (and usually multiple processors on one machine), distributed algorithms have their own independent memory and processor.

As defined above, distributed computing is a current research topic driven by the rise in popularity of cloud architectures, being adopted by industry and academia for business and research related purposes. These architectures are a topology of machines connected together that communicate in some fashion to solve a problem that arises from segregated memory on all machines. Although, some systems can be developed to not rely so heavily on inter-process communication by utilizing shared memory resources [87]. Each machine works independently of one another, but work together to solve algorithmic problems and agree consensus on defined issues that arise from such architectures.

Many models can be embedded into a distributed system, with parallel algorithms working concurrently on multiple processes on one machine, using shared or individual memory. The model studied in this thesis are distributed algorithms adopting a message passing model. In this model there exists one algorithm copied on multiple machines that can work independently of the network topology and ultimately determine and disseminate the correct result or consensus to all machines to ensure they are in a consistent state.

Definition. State: Machines in a distributed setting can communicate through physical or theoretical links adhering to some algorithmic process or protocol. These algorithms or protocols have an initial state, before solving or attempting a task has begun. There can be many intermediate states in between initiation and the final state of task being completed and result dissemination.

The evaluation and design of these algorithms adhere to the same paradigms outlined in the previous section with the additional caveat of network stability and failure, which can be included or excluded depending on the model.

1.5 Machine Learning

The final body of work in this thesis is related to the area of Machine Learning - a term to encapsulate techniques and algorithms used to infer information from large datasets.

Described as "the field of study that gives computers the ability to learn without being explicitly programmed"by Samuel [110]. An ever increasingly relevant area for research in a wide array of domains as the amount of data produced is increasing at a rapid rate from a multitude of sources, known as 'Big Data'.

Definition. Big Data: Large and complex data sets, though existing already for decades, is currently being generated at a rapid rate as a result of the inception of smart devices, internet of things, social media, amongst many others. The consensus for general traits of big data are as follows; Volume, data too large to store easily in traditional databases. Variety, data comes in structured format, examples being age, salary, location and also arriving unstructured, in formats such as text, blogs, statuses. Volatility, data can be static, such as birth place, date of birth, but can also be dynamic, examples being current address, employer, current music track being played, with different levels of dynamism.

Machine learning is the umbrella that covers a wide range of computational problems, including a host of feature tuning problems. The work in this thesis is concerned specifically with areas of supervised and unsupervised learning methods, defined as follows.

Definition. Supervised Learning: The task of inferring a function from labelled training data. Supervised learning procedures take two parameters, a set of vectors X and a set of corresponding labels y inferring desired output. Algorithms are typically trained on 70% of the dataset (function generation) and evaluated on the remaining 30% to determine performance of the function. Formally, given a training set (x_i, y_i) for $i = 1..n$, we want to create a supervised model f that will predict label y for a new x.

Definition. Unsupervised Learning: The task of inferring a function to describe hidden structure information from unlabelled data. Unsupervised learning procedures take a set of vectors X and produce a set of labels y. As the data is unlabelled, it is difficult to evaluate the performance of these algorithms as there is no error or residuals to study.

The goal of machine learning is a solution to remedy the problem of designing and implementing explicit algorithms on a per problem basis, which in practice is highly infeasible. Machine learning solutions contain generalized algorithms that learn through éxperienceánd can apply the knowledge that aid in determining outcomes to varying problems. The umbrellas that these algorithms are encapsulated by are defined as follows.

Definition. Classification Problems: the process of determining if an object from a set X belongs to a specific group or not, or belonging to a set of groups or not. This is binary valued and multi-valued classification respectively. As a supervised learning problem, these algorithms learn based on a training subset of data points z from X and corresponding ground truth labels in y to determine the outcome for the remaining $X - z$ samples, which can then be evaluated to determine performance.

Definition. Regression Analysis: the process of determining real-valued outcomes. Another supervised learning problem, these algorithms learn based on a training subset of

data points z from X and corresponding ground truth labels in y (actual real-values) to determine the outcome for the remaining $X - z$ samples.

Definition. Clustering Problems: the process of determining what group, or segmentation of X, an object from a set X belongs to using various similarity metrics. As an unsupervised learning algorithm, the training data X has no ground truth labels y to learn from.

Definition. Recommender Problems: the process of determining suggesting an object to a person based on data derived from their habits. An unsupervised learning algorithm that uses training data X with no ground truth labels y to learn from.

1.6 Research Areas and Recent Work

The following subsections contain recent work in the specific problems we study, including the models and complementary theories.

1.6.1 Distributed Majority Problems

Distributed settings are copious in topologies and communication mechanisms based on models with varying constraining impositions and therefore recent advances are numerous. Recent works on the problem of distributed majority include many papers solving consensus, problems in which all machines must agree on a common value, while being fault tolerant. These problems are important in determining control of distributed systems. An article covering optimal distributed algorithms for minimum weight spanning trees, counting problems and leader elections was published in 1987, new algorithms were presented that improved on many previous results [33]. A recent article studies the complexity of leader election problems in distributed settings, more specifically randomized implicit election [83]. New algorithms for fast Byzantine leader elections in dynamic networks were proposed by Augustine et al. whereby all nodes are aware of all information in the distributed network [31]. Recently, the reduction of multivalued consensus to binary consensus has been shown in [23, 38, 60]. Angluin et al [23] provide an approximate solution to solve this problem which also tolerates Byzantine behavior, the first population protocol of its kind. Ezhilchelvan et al [60] produce the first randomized consensus protocol that doesn't require a priori knowledge of the values that can be proposed by the processes. The notion of using a *charge* was first proposed by Birk et al [38] to solve a majority voting problem, in which they combine efficient spanning forest algorithms with a "charge fusion" algorithm.

1.6.2 Random Walks

Random walks have been extensively studied in many academic disciplines - due to their lightweight and local nature. There is vast literature available covering the topic, its variants and many applications, notably monographed in [78, 79] and more recently

in [114]. Applicable work related to results in this thesis are shown in [32] by Avin et al. Work in this paper proved that random walks can cover all the vertices of dynamic graphs in finite, possibly exponential time when the dynamic graphs either evolve in a Markovian way or they are always connected, i.e. to prevent a random walker becoming isolated at a vertex. They also show that a lazy random walk covers any connected dynamic graph in polynomial time in the size of the graph and that a simple random walk will also cover a dynamic graph itself is obtained by sampling from a certain probability distribution in polynomial time. A rigorous framework for the design and analysis of random walk algorithms in dynamic networks is proposed in [112]. In the same paper a fast distributed algorithm for dissemination was proposed which utilised random walks, employing a fully-distributed token forwarding mechanism. Random walks have also proved useful in developing fast byzantine agreement algorithms.

We also study random walks [78] in the context of community detection as intuitively random walkers remain inside communities longer due to high edge density. Random walks were used to determine distance between pairs of vertices in [119], which was further developed to study biased random walkers in which walkers would orientate towards vertices of high common neighbours in [120].

1.6.3 Population Protocols

Population protocols are a model of distributed computing in which agents cooperate to collectively perform computational tasks. This model limits the agents individual computational resource and they have no control over the interaction schedule of agents. The agents instead interact in pairwise fashion according to a random scheduler governed by some fairness condition. The interaction between agents may update the local state of one or both participants with the goal of having all agents converge to some state that represents the output of some computation.

The first protocols were proposed by Angluin et al. [27] to stably compute any predicate in the class definable by formulas of Presburger arithmetic, which includes Boolean combinations of threshold-k, majority, and equivalence modulo m. The protocols proposed in this paper have been expanded upon as they are useful in representing abstractly various network models such as wireless sensor networks [59, 103], chemical reaction networks and gene regulatory networks [41].

An objective of some population protocols closely related to the contents of this thesis is majority problems or determining consensus in a network. The work in [103] propose models to solve binary consensus in which agents begin in one of two states and ultimately converge with all agents knowing the initial majority. Similar models are proposed in [16, 55] to solve complementary tasks such as *leader election*, a task in which all agents must eventually converge with only one in a *leader* state.

Recent work has been proposed in [16, 46, 55, 56] to understand the complexity of these aforementioned tasks in this distributed model, specifically the time relative to the number of states each node can store. The work by Alistarh et al. [15] considers

the relation between number of allowable states in a protocol and the time complexity and propose improved algorithms to problems in the majority and leader election domain. The book written by Michail et al. [93] further provides a detailed composition of population protocols.

1.6.4 Clustering

The advent of modern clustering arrived subsequent to the proposed algorithm by Girvin and Newman [71], a modularity based method in which edges with the highest *betweeness centrality* were trimmed from the graph until k clusters were derived. *Betweeness* is a metric used to determine the activeness of an edge in the network, i.e. edges of high through-flow. Edges with a high through-flow are considered to be intra-cluster edges connecting communities and therefore should be cut first. This method is computationally expensive, approximation techniques were developed such as *random-walk betweeness* [98] as the recalculation of the betweenness value is essential in determining meaningful communities. Many papers have built upon the foundations of this algorithm in [42, 99] and made augmentations to determine overlapping clusters [73, 76]. We consider algorithms grounded in connectivity centric models, such as hierarchical clustering [66]. The problem of clustering is an inter-disciplinary field and research is segregated. There have been many algorithms to solve the problem, but the solutions are generally data driven. There is currently no universal agreed upon definition of clustering with well defined metrics for comparison, but work to address these deficiencies has been made in [52]. Benchmark graphs were devised in [71] with modifications and adaptions in [29, 43, 47, 84, 117]. Clustering specific evaluation metrics were introduced in [64, 88, 107]. The lack of adherence to a standard evaluation pipeline can lead to an unstructured, chaotic field. The work by Fortunato in [63] serves as a useful guide for any aspiring researcher in this area from all disciplines.

1.7 Summary of Results

1.7.1 Distributed Majority Problems in Random Walk Models

The results we obtained in this area have been published in [67] and are presented in Chapter 2. In the work presented we propose and analyze a distributed protocol that will determine consensus for the Majority Color Problem (MCP).

The work presented here is based upon the work in [89] where a similar protocol was used but in the context of population protocols. Our protocol guarantees that if a majority exists, then eventually each node in the network will learn of the initial majority color.

Originally, if no majority existed then our protocol would leave an arbitrary assignment of colors between nodes in the network. In later studies and documented in the

journal version of this paper, a new state was added such that if there is no majority, each node also eventually learns this fact - now discovering and distributing equality.

Our protocol requires only three bits of memory per node and uses a simple token message, two bits in size. We assume the network is unknown to the nodes, meaning they know nothing of their neighbors or topology. Similarly, each node does not have a unique identifier, meaning the token can not keep track of where it has been. The nodes simulate a random walk as subsequent to every turn, a neighbor is selected at random to receive the token.

We show correctness of our protocol for any connected graph and even for a natural class of dynamic graphs. We show upper and lower bounds on the convergence time of our protocol and discuss termination. We also provide a variant of our protocol, in which the token uses a counter that can count only up to $n \log n$, where n is the number of network nodes.

Subsequent to the publication of this work, we looked at further extending this protocol to solve variations of the MCP. We build on and provide adaptations of the original protocol to solve k-surplus problem.

The original protocol will determine the absolute majority color in the network, whereby more than $n/2$ network nodes are of one color. With the introduction of k colors, there may exist a color that is more abundant than all remaining colors individually, in direct comparison, but there is no color that is believed to be the majority by more than half the network nodes - this is the case of relative majority. We provide a way to use our augmented distributed protocol to determine consensus of equality, absolute and relative majority for k colors.

We analyze different mechanisms for these protocols in terms of the memory required for the nodes or token(s) used to perform the random walk(s). Finally, we also consider random walks that can count the difference of colors and we show upper bounds on the counter value by using coupling arguments.

A simulation of the protocols proposed in this work has been created using Python and is available for use in [8].

1.7.2 Distributed Majority Problems in Population Protocol Models

In this research area we presented efficient population protocols for several variants of the majority problem [68]. Initially we make an important amendment allowing for the discovery and propagation of equality in this setting. We propose memory efficient protocols in populations with an arbitrary number of colors, represented by k-bit labels. Specifically, the protocols are asymptotically optimal for testing absolute majority and relative majority in populations with C colors. The protocols we propose rely on dynamic formulation of the majority problems as opposed to solving static majority which has been widely studied in the past. The case of dynamic majority refers to the case were original colours can change in time due to some external factor (which may alter the

outcome of the majority protocol). Another salient result shown in this section is the inauguration of a framework to allow for multi-stage population protocols to be designed.

1.7.3 Clustering in Structured Graphs

In the context of this research area we present a method of clustering data inspired by methods such as hierarchical clustering and theories from phylogenetics or evolutionary biology in which natural hierarchies are derived from the network. Our method uses the local relationships in the network topology to create constraints that dictate the order in which vertices and clusters should be merged. The objective is to create a hard clustering procedure which retrieves consistent clusters on identical graphs. We propose multiple ways to generate constraints based on the local relationships in the graph, which can affect the generated clustering hierarchy. We study the procedure on simple networks, standard in most clustering papers and discuss the steps which we took to augment the procedure into the final variant. We provide a ranking mechanism for the captured data and order the constraints respectively. The clustering building method borrows concepts from phylogenetics, with the options to include or exclude additional concepts such as fan triplets and forbidden triplets. We show comparisons with other clustering methods on various real-networks and synthetic networks, including popularised benchmark graphs and discuss the performance using clustering specific evaluation metrics. We discuss the advantages and disadvantages of such a method and areas of future research. We provide other prototype functions such as thresholding which allows early termination, which is useful when considering post processing techniques.

1.8 Thesis Structure

1.8.1 Chapter 2

The work in this chapter is focused on the conference paper published in [67] and the adapted journal version. The topic is of distributed majority problems - more specifically, the majority color problem. The chapter explains the problem area, the model and proposes a very efficient solution in time and space in static and dynamic settings.

1.8.2 Chapter 3

The contents of this chapter is focused on the conference paper published in [68] in which we present novel memory-efficient population protocols for several variants of majority problem including dynamic formulation of majority problems.

1.8.3 Chapter 4

The work presented in this chapter proposes a new method to locally sample data points and build clusters agglomeratively using this derived information. We look at various termination criterions, specified k (where k is the number of desired clusters), as well as

thresholding functions and analyze the performance of our method on a set of benchmark graphs and parameters. We assess the performance of our method against the evaluation metrics specified in [63].

1.8.4 Chapter 5

The final chapter concludes the work presented in this thesis and discusses open problems and further channels of related research. We also state the impact of our work in the field in our final remarks.

1.9 Authors Contribution

The work presented in Chapter 2 is based on the paper [67], originally co-authored by Leszek Gąsieniec, David D. Hamilton, Russell Martin and Paul G. Spirakis. Further research was made with the addition of Jurek Czyzowicz and Evangelos Kranakis which resulted in a journal version of this paper. The work in Chapter 3 is based on the paper [68] co-authored by L. Gąsieniec, D.D. Hamilton, R. Martin, P. Spirakis, and G. Stachowiak. Chapter 4 is a study based on unsupervised clustering algorithms, co-authored by L. Gąsieniec, D.D. Hamilton and R. Martin. The remaining work presented here is written by the student for this PhD project and supervised by R. Martin and L. Gąsieniec.

Title	*Authors*	*Appeared*
The Match-Maker: Constant Space Distributed Majority via Random Walks [67]	L. Gąsieniec, D.D. Hamilton R. Martin and P. Spirakis	[1]SSS 2015
The Match-Maker: Journal [51]	J. Czyzowicz, L. Gąsieniec, D.D. Hamilton, E. Kranakis, R. Martin, and P. Spirakis	TBD
Deterministic Population Protocols for Exact Majority and Plurality [68]	L. Gąsieniec, D.D. Hamilton, R. Martin, P. Spirakis, and G. Stachowiak	[2]OPODIS 2016
Agglomerative Phylogenetic Clustering	L. Gąsieniec, D.D. Hamilton, and R. Martin	TBD

TABLE 1.1: A table of the author's publications and co-authors.

[1]SSS 2015: 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems, 2015

[2]OPODIS 2016: The 20th International Conference on Principles of Distributed Systems, 2016

Chapter 2

Majority Problems in Random Walk Models

2.1 Introduction

In this chapter we consider solving majority problems in distributed networks utilizing random walks and limited memory. We propose and analyze here a simple protocol for consensus on the majority color in networks whose nodes are initially one of two colors. Our protocol guarantees that, if a majority exists, then eventually each node learns the majority color. If there is no majority, each node also eventually learns this fact. Our protocol requires only three bits of memory per node and uses a simple token message, two bits in size, that performs a random walk. We show correctness of our protocol for any connected graph (even unknown to the nodes) and also for a natural class of dynamic graphs. We show upper and lower bounds on the convergence time of our protocol. We discuss termination and we also provide a variant of our protocol which the token uses a counter that can count only up to $\sqrt{n}\log n$, where n is the number of network nodes. Our basic (memoryless) protocol takes only $\mathcal{O}(n \log n)$ expected time on the clique which surprisingly does not deviate from the cover time of the random walk, and $\mathcal{O}(n^2 m)$ time on any connected undirected network of m edges. This bound is matched from below by the path. Using this majority protocol, we show extensions so that nodes can conclude if there is a k-surplus (one of the two colors exceeds the other by k or more), and extensions to three or more colors, so that nodes can conclude if there is an absolute or relative majority. We analyze different mechanisms for these protocols in terms of the memory required for the nodes or token(s) used to perform the random walk(s). Finally, we also consider random walks that can count the difference of colors and we show upper bounds on the counter value by using coupling arguments.

2.1.1 The problem, model and motivation

Consider an undirected and connected graph (or network) $G = (V, E)$ with $|V| = n$ vertices (nodes) and $|E| = m$ edges (links). Initially, each node is colored either blue

(BLUE) or red (RED). In the sequel we use $\overline{X}$ for $X = BLUE$ (resp. RED) to denote RED (resp. $BLUE$), i.e. if X is a color then $\overline{X}$ is its "complement". No node can store more than a fixed number of bits, in fact, not more than 3 bits. The main problem is to devise a correct and efficient distributed procedure executed by the network nodes which can communicate with neighbors via constant-size messages. Eventually, all nodes must agree on the initial majority color, if such a majority exists. In the case when no majority exists, the nodes must also eventually ascertain that knowledge. We call this the *Majority Color Problem (MCP)*. The main purpose of the work in this chapter is to propose and analyze a specific algorithmic procedure which solves the MCP, with only $\mathcal{O}(1)$ bits of memory per node and $\mathcal{O}(1)$ bits per message passed between nodes.

We also consider the *k-Surplus Problem*, that is determining whether one of the two colors occurs on k (or more) vertices than the other color. We further consider the case when there are more than two colors, where "majority" can now mean "absolute majority" (where one color is on more vertices than all other colors combined), or a "relative majority" (where one color beats any other single color in a one-on-one comparison, also sometimes called a "plurality"). For more than two colors, we consider how we can adopt our solution to the MCP (for two colors) to solve these other problems and how to implement these mechanisms in terms of their memory usage. First let us describe our model.

We assume that the network is synchronous in that executions of our protocols are coordinated in time and performed in sequence. Specifically, the random walker moves, performs an action, updates itself and the network nodes in one step of the clock. We analyse the runtime of our protocols according to these steps. We consider here networks that are unknown to the nodes, where each node knows only the links to its neighbors. We also consider dynamic networks in which neighbors may change from round to round. Because of the above, we allow a node to select a random link incident to it and send a message via that link. In other words, we allow nodes to initiate and maintain a random walk in the unknown graph.

Random walks have been extensively studied in distributed computing in the context of problems like exploration and information dissemination. In this chapter we show that random walks are suitable, in particular they are very efficient in time and space, to solve the MCP. The random walk acts here like the match-maker person (in olden times) in several countries, going from village to village and trying to match boys (BLUE) with girls (RED). Upon encountering a boy, the match-maker gets his "color" and places him in a "matched" condition and then proceeds to find a corresponding girl via the random walk in the network. Hence, we call our proposed protocol "The Match-Making Algorithm".

Correct majority protocols using messages only a few bits in size, that only perform random walks, are very useful from a security point of view since:

(a) The origin of a random walk cannot be traced back.

(b) An eavesdropper that intercepts a token (of a few bits), doing a random walk, cannot infer anything about the vote of a particular person, nor the result of the voting (assuming that they cannot determine if the process has converged to the final result).

As we shall see, our proposed solution satisfies (a) and (b), and can be used in unknown (and even dynamic) networks, without the need of any centralized controller or vote aggregator.

One of the important measures of performance of a random walk is the cover time:

Definition 2.1. Cover Time: consider a random walk on an undirected, connected graph, starting at node v. Let t_v be the minimum time for the walk to visit all of G's vertices at least once. Let $\mathbb{E}(t_v)$ be the expected value of t_v. The (expected) *Cover Time* of G, denoted $C(G)$, is

$$C(G) = \max_{v \in V(G)} \mathbb{E}(t_v).$$

In this work we also consider dynamic networks i.e. graphs where the neighbors of every node change in an adversarial way in each round of the global clock. However, we assume that our dynamic networks change due to a *benign adversary* that satisfies three properties:

Definition 2.2. Benign adversary: an adversary that changes the graph structure per round is *benign* if and only if

1. The adversary is oblivious to any random choices made by our protocol.
2. For any two nodes u,v and any time t_0, the edge $\{u, v\}$ shall (re)appear in time (round) $t_0 + t_1$, with t_1 bounded above by some finite integer β (which may depend on the number of nodes, n, in the graph).
3. The adversary maintains the nodes of the graph (no node deletions or insertions).

We call β the *tolerance time* of the dynamic graph.

The dynamic graphs we consider are controlled by a benign adversary as we can guarantee that the nodes in the graph are maintained and the edges between a pair of nodes will reappear at some point in the future, defined by β. Combined with the fact a benign adversary is oblivious to the transitions of our protocol, we can guarantee that our protocol will converge on a dynamic graph with these properties. Contrary to this, on a dynamic graph controlled by a malicious adversary, as our protocol in this chapter relies on the graph structure to solve the problem we cannot guarantee the protocol will converge. For example, the random walker may become isolated on a node and never be able to transition to another.

2.1.2 Our Results

We provide here a simple distributed algorithm called BASIC that uses only (1) 3 bits of memory per node, (2) a single token of 2 bits long, and (3) always converts the color

of all n nodes in the graph to the initial majority color, if such a color exists. Our algorithm, in its basic form, does not terminate but converts all colors to the majority color in finite time, even in unknown or dynamic graphs (assuming a benign adversary as in Def. 2.2). One can equip the system with a global clock readable by all nodes allowing termination of our process with high probability (w.h.p.)[1] if the value of n is known by the vertices. We provide a method to compute a lower bound on the convergence time of BASIC for any given initial placement of node colors. If there is no majority, BASIC will also converge, in the sense that every node will become aware of the fact that no majority exists, and this knowledge will persist forever in each node after a finite number of steps of the BASIC protocol.

We show that BASIC converts all nodes to the majority color in expected $\mathcal{O}(n \log n)$ time for the clique graph and in at worst $\mathcal{O}(n^2 m)$ time for any connected graph. Our bound for the clique matches the cover time of the random walk. Our bound for arbitrary graphs is tight on the line. We consider random walks that can count the difference in the number of colors visited, and show non-trivial upper bounds on the counter value in order for such procedures to work correctly. Finally, we consider related problems such as the k-surplus problem (does one color appear on k or more nodes than the other?) and the extension of the majority problem for three or more colors, with different interpretations of the meaning of "majority".

Outline of the Chapter

This chapter is organized as follows:

Section 2.2 considers the MCP, and our proposed (random-walk based) protocol for solving that problem. We show our "BASIC" protocol is correct and give a general upper bound for convergence in any connected graph.

Section 2.4 considers the special case of the clique K_n, showing the $\Theta(n \log n)$ convergence time for BASIC on K_n.

Section 2.5 describes a method for finding a lower bound on the convergence time for BASIC in a general graph by computing matchings in the graph using the Hungarian method.

Section 2.6 investigates terminating the BASIC protocol. The description of BASIC in Section 2.2 indicates that the procedure does not terminate, although it does converge to the correct solution in polynomial-time (in the size of the graph). With additional information, say, the use of a global clock, we show how BASIC can terminate with the correct solution (with high probability).

Section 2.7 considers a special case of graphs with a small cover time, and how BASIC could be made to terminate (with high probability) in that case, by using a counter that only requires a small amount of memory.

[1] "With high probability" means with probability at least $1 - \frac{c}{n}$ for some constant c, where n is the number of nodes in the graph.

All the previous sections consider only static graphs. Section 2.8 considers a class of dynamic graphs. These graphs can vary according to a benign adversary (see Def. 2.2), and BASIC still correctly solves the MCP problem in polynomial-time (with high probability).

Sections 2.10, 2.11, and 2.12 discuss problems related to MCP. In particular, we consider the k-surplus problem (determining if one color occurs on k or more vertices than the other in the graph), and majority in the case of three or more colors. For more than two colors, "majority" can mean "relative majority" or "absolute majority", and we consider both meanings (Sections 2.11 and 2.12, respectively).

Finally, we conclude with some discussion of the use of random walks vs. population protocols in Section 2.14.

2.1.3 Previous Work

Our proposed method is inspired by the work in [89] where a similar protocol was used in the context of population protocols. Here we convert the ideas of [89] into a message-passing protocol that employs random walks and we prove its correctness for unknown static networks and for a certain natural class of dynamic networks. For the clique and for general graphs we show expected convergence time of $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2 m)$ respectively, while the corresponding times in [89] were $\mathcal{O}\left(\frac{n^2 \log n}{|majority|-|minority|}\right)$ and $\mathcal{O}(n^6)$.

Avin et al [32] have proved that random walks can cover all the vertices of dynamic graphs (in finite, possibly exponential, time) when the dynamic graphs either evolve in a Markovian way or they are always connected. Our model of dynamic graphs is not covered by those models because our dynamic graph can evolve in an adversarial way and may also not be connected at any (or all) rounds during the execution of the BASIC algorithm. Because of the finite expected cover time of the model of Avin et al, it can be easily shown that our protocol is also correct for those dynamic graphs.

Other works on distributed majority include [23, 60, 96] which show how to reduce multivalued consensus to binary consensus. However, such protocols assume either a stronger network with broadcast [96] or randomization [60].

The notion of using a *charge* was first proposed by Birk et al [38] to solve a similar problem, in which they combine an efficient spanning forest algorithm with a "charge fusion" algorithm. That paper proposes a stronger model to solve a more general problem, which has more requirements and enables direct access to neighbors. Also, their solution relies on larger memory and additional computation.

In contrast, our method for the MCP (with two colors) requires only a single token of 2 bits able to perform a random walk in the network, and is always correct in the sense that if an initial color majority exists, then eventually all nodes agree on the majority color, or correctly conclude there is no majority color if none was present at the start of the protocol. Our method performs no artihmetic calculations and instead represents a finite state machine. The topology of the graph is unknown to the vertices

and vertices are anonymous. For basic notations on probability, martingales and random walks, see [18, 62, 97].

2.2 The Majority Color Problem

Our proposed method presumes the existence of a single token (message), initially in some arbitrary node which performs a random walk around the graph. The protocol, described in Section 2.2.1, is a slightly modified version of the original match-maker protocol given in [67], one in which a new "color" (namely, one we call $NEUTRAL$) has been introduced that allows for ties in the network to be determined.

Each node stores information as an ordered pair that denotes its state in the form (color, importance). The first component, "color" can have a value in $\{RED, BLUE, NEUTRAL\}$ and begins with a color RED or $BLUE$. The second component denotes "importance" which can have a value in $\{HIGH, LOW\}$ and begins in the state $HIGH$. The token can have a value belonging to $\{RED, BLUE, NEUTRAL\}$ and begins as $NEUTRAL$.

To save space in what follows, we abbreviate RED, $BLUE$, $NEUTRAL$, $HIGH$, and LOW as R, B, N, H, L, respectively.

2.2.1 The Match-Making Algorithm with Equality (BASIC)

Our proposed method to solve the MCP is provided here in the form of a state transition table. Note that in the protocol below, only transitions where the state of the token and/or one of the vertices changes are included. Interactions not listed do not result in any change of state for the token, nor the node with which it is interacting.

We provide the state transitions below for the Match-Maker with Equality protocol that we call BASIC. The protocol will convert all nodes to the majority color if such a color exists, or all nodes will be converted to $NEUTRAL$ if there is no such initial majority.

In the description of the interactions below, we note that $C \in \{R, B\}$, and $X, Y \in \{R, B, N\}$. Further, for $C \in \{R, B\}$, $\overline{C}$ denotes the "opposite" color, e.g., if $C = R$, then $\overline{C} = B$.

The token begins with the state (color) N, and can take on one of the states (colors) R, B, or N during the protocol.

The "Process stage" designations in the table are mnemonics to help us understand the Match-Making Algorithm. Intuitively, BASIC is looking to match pairs of vertices that begin with opposite colors. When the token is in state N, it is looking to begin a match between opposite colored vertices. When the token has color C (in $\{R, B\}$), it is looking for a node with color $\overline{C}$ that has high importance, i.e., to complete a matching between these oppositely colored vertices. The "Inform" transition is to allow the token to pass information to vertices about the majority color (or what the token currently

Initialization: Each node v begins in state (C_v, H) and the token begins in state N at a random node.

Transition table:

Process stage	Token state	Node state	New token state	New node state
Begin Matching	N	(C, H)	C	(N, L)
Complete Matching	C	$(\overline{C}, H)$	N	(N, L)
Inform	X	(Y, L)	X	(X, L)

FIGURE 2.1: The BASIC protocol.

"believes" is the majority color). The correctness of BASIC is proven in Theorem 2.3, which is formalizing the statements above.

Note that the random walks defined here for the token are "extended" or "lazy" in the sense that the token may choose to stay at the same node (of degree d_t at round t) with probability $\frac{1}{d_t+1}$. Also note that, in each round only one node executes the protocol because there is a single token in the network.

2.2.2 Correctness and convergence of BASIC

Theorem 2.3. *(Correctness of BASIC) In any static undirected, connected, finite graph $G = (V, E)$, protocol BASIC eventually turns the color of every node to the initial majority color, even if the graph and its size are unknown to the nodes.*

Proof. The token matches each node of color $C \in \{R, B\}$ and high importance (i.e. as all nodes are initially) to a node of color $\overline{C}$ of high importance, and both vertices turn to low importance. Thus, the initial (high importance) nodes are matched in RED-BLUE pairs. If a majority color C initially existed, then eventually the token will find it (by visiting all nodes), and then it will walk in the graph converting all nodes (of low importance) to the color C, leaving all nodes with the knowledge that C is the majority color. If no majority color existed at the start of the protocol, all nodes will eventually be converted to the state (N, L), and the token will have state N. Specifically, after the last RED-BLUE match occurs and after the token visits every node one additional time, all nodes have the correct conclusion that no majority existed.

For every color matching that needs to be made, the token's random walk needs time at most equal to the cover time of G. Finally, it needs only the cover time of G in order to convert the color of all nodes to the first majority color having no match (if there was a majority), or the cover time to ensure that each node knows that there was no majority. So, the token needs, at worst, $n + 1$ cover times to convert all colors to the initial majority color (if there was an initial majority). We also know that the expected cover time of any finite connected graph G is finite with probability 1. By linearity of expectation, and since the walks are one after the other, the total time to convergence to the initial majority is finite with probability 1. □

Corollary 2.4. *The BASIC protocol needs an expected number of rounds at most equal to* $(n+1) \cdot \mathbb{E}(C(G))$ *until convergence. For any connected graph with* n *nodes and* m *edges, BASIC converges in expected time* $\mathcal{O}(n^2 m)$.

Proof. BASIC needs at most $\frac{n}{2}$ cover times to match appropriate colors of high importance and at most $\frac{n}{2}$ cover times to find a new color of high importance every time. Then BASIC needs a final cover time to convert all node colors to the majority color. Finally, we use the fact that $\mathbb{E}(C(G)) \in \mathcal{O}(nm)$ for any connected graph G. [11] □

Remark 2.5. For any class of graphs with known stronger bounds on the cover time, such bounds can obviously be used to strengthen the expected convergence time for that graph class in Corollary 2.4. For example, the cover time of the clique K_n is $\Theta(n \log n)$. Then Corollary 2.4 gives a convergence time of $\mathcal{O}(n^2 \log n)$ for BASIC, better than the $\mathcal{O}(n^4)$ bound implied the cover time bound for a general graph (as, of course, $m = n(n-1)/2$ for K_n). We show later that BASIC converges even faster than $\mathcal{O}(n^2 \log n)$. See Lemma 2.7 in Section 2.4.

Remark 2.6. BASIC will also converge (i.e. it works correctly) on any directed *strongly connected* graph. Theorem 2.3 only relies on the token being able to match pairs of vertices (of high importance) having opposite colors (and find a suplus color, if one exists). Strong connectivity of a digraph guarantees that such matching of vertices is possible in a digraph. The direction of edges matter in the matching of vertices in regards to the expected cover time as the random walker can only use edges one-way.

We note, however, there are examples of digraphs for which the cover time is exponential, such as the family of digraphs referred to as "combination locks" [36, 95]. Corollary 2.4 could be suitably modified in the case of (strongly connected) digraphs by using an appropriate bound on the cover time for a particular digraph.

Next, we examine other aspects of the BASIC protocol, before considering the additional problems mentioned in the Introduction. First, let us consider the special case of the clique.

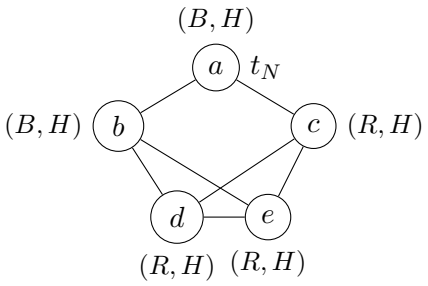
2.3 Execution of BASIC in a simple graph

This section demonstrates an example execution of the *BASIC* protocol, defined in Figure 2.1. The example execution applies *BASIC* on a graph where $n = 5$ and there exists a majority where $|RED| > |BLUE|$.

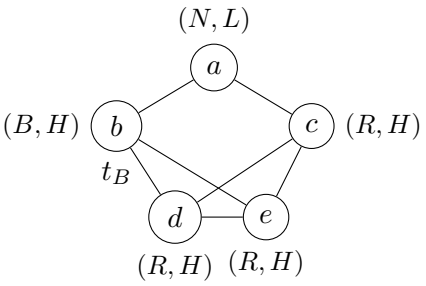
Table 2.1 traces the execution from the initialization step (step 0), when the token t is placed at an arbitrary vertex, to the state when all vertices have been converted to the initial majority color. Each row consists of the step number, the state of t (where $t = \{color, location\}$), the state of all the vertices (uppercase representing high influence and lowercase representing low influence of a color) and finally the transition which should be executed given the current state of the graph. The transitions are Begin Matching, Complete Matching and Inform, abbreviated to BM, CM and I respectively.

Step	t	a	b	c	d	e	Transition
0	Initialization, token placed at a in state N						
1	{N, a}	(B, H)	(B, H)	(R, H)	(R, H)	(R, H)	BM
2	{B, b}	(N, L)	(B, H)	(R, H)	(R, H)	(R, H)	None
3	{B, a}	(N, L)	(B, H)	(R, H)	(R, H)	(R, H)	I
4	{B, c}	(B, L)	(B, H)	(R, H)	(R, H)	(R, H)	CM
5	{N, e}	(B, L)	(B, H)	(N, L)	(R, H)	(R, H)	BM
6	{R, b}	(B, L)	(B, H)	(N, L)	(R, H)	(N, L)	CM
7	{N, d}	(B, L)	(N, L)	(N, L)	(R, H)	(N, L)	BM
8	{R, b}	(B, L)	(N, L)	(N, L)	(N, L)	(N, L)	I
9	{R, a}	(B, L)	(R, L)	(N, L)	(N, L)	(N, L)	I
10	{R, c}	(R, L)	(R, L)	(N, L)	(N, L)	(N, L)	I
11	{R, d}	(R, L)	(R, L)	(R, L)	(N, L)	(N, L)	I
12	{R, e}	(R, L)	(R, L)	(R, L)	(R, L)	(N, L)	I
13	{R, b}	(R, L)	(R, L)	(R, L)	(R, L)	(R, L)	I (no change)
14+	{R,*}	No change in any node's state hereafter					I (no change)

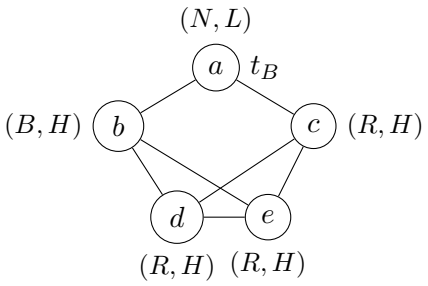
TABLE 2.1: Trace table for an execution of BASIC.



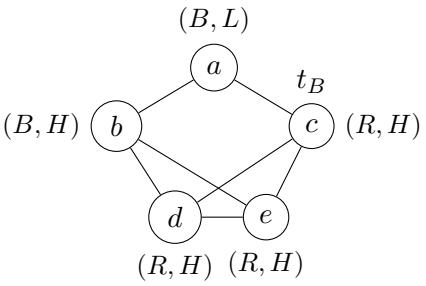
Step 1



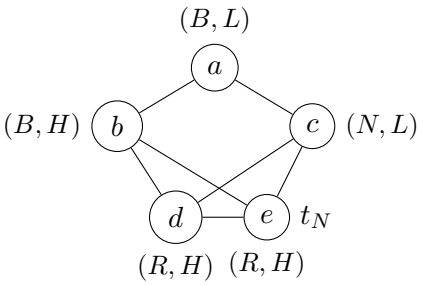
Step 2



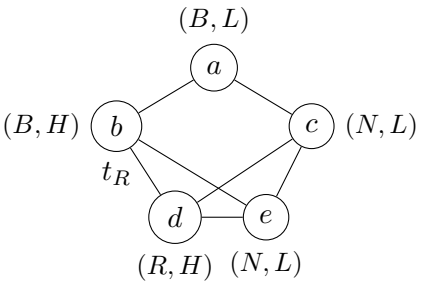
Step 3



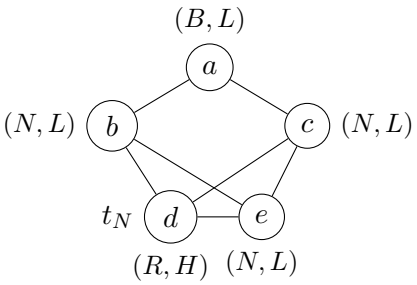
Step 4



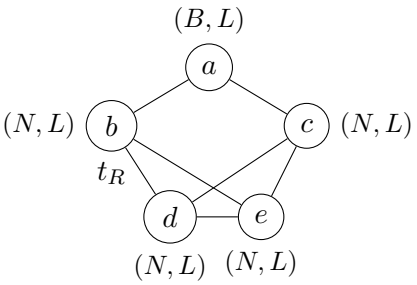
Step 5



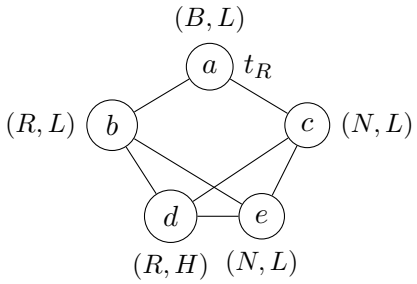
Step 6



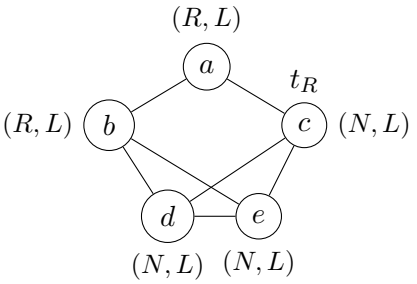
Step 7



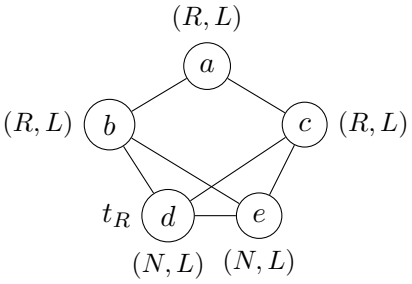
Step 8



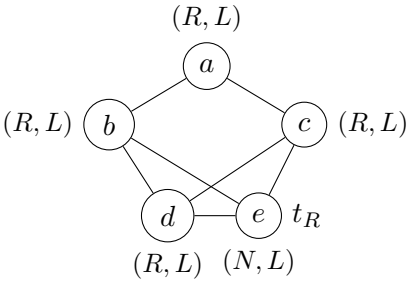
Step 9



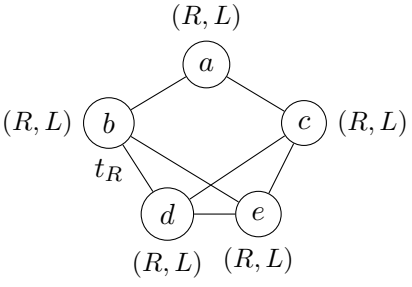
Step 10



Step 11



Step 12



Step 13

2.4 BASIC on the Clique

Let $G = K_n$ = the clique of n nodes, and let us consider the convergence time of BASIC on K_n. Assuming that both BLUE and RED nodes are (still) present in the clique, let us define a "phase" as the time required in the random walk until a BLUE node or a RED node becomes NEUTRAL in color (i.e. a "Begin Matching"' transition takes place or a "Complete Matching" transition takes place). Let b_t, r_t be the number of BLUE and RED nodes, respectively, at phase t. Initially $b_0 + r_0 = n$ at the start of the process, but we immediately have either $b_1 = b_0 - 1$ or $r_1 = r_0 - 1$, as a "Begin Matching" transition will take place in round 1.

The time, T, of BASIC until convergence is $T = T_1 + T_2 + T_3$, where T_1 = the sum of all the times for the token to match two high-importance nodes of the opposite colors, T_2 = the sum of all times for the token to discover the next color of high importance to match, and T_3 = the final cover time to convert all colors to the majority (or, in the case of no majority, the time to convert all nodes to NEUTRAL once the last match has been made).

Since we are considering K_n, during phase t, the probability that the random walk finds a matching node of the opposite color is $\frac{b_t}{n}$ (if it starts from a RED node) or $\frac{r_t}{n}$ (if it starts from a BLUE node). Thus, the expected time until success (i.e. finding a matching node of opposite color) is bounded above by $\frac{n}{b_t}$ (resp. $\frac{n}{r_t}$) depending on the case, as this is the expectation of a geometric random variable.

Since the high importance colors are matched in pairs, we have (in each matching) $r_{t+2} = r_t - 1$ and $b_{t+2} = b_t - 1$. Let a be the phase at which the minority color has only one node with that color. Both expectations of T_1 and T_2 are bounded above by $\sum_{t=1}^{a+2} \frac{n}{b_t} + \sum_{t=1}^{a+2} \frac{n}{r_t}$. This sum is $n(\frac{1}{b_1} + \frac{1}{b_1 - 1} + \cdots + 1) + n(\frac{1}{r_1} + \frac{1}{r_1 - 1} + \cdots + 1)$, i.e., at worse $n \cdot H_n$ (H_n = the n^{th} harmonic number, where the harmonic number is $\sum_{i=1}^{n} \frac{1}{i}$). Also, the expected cover time is $\Theta(n \log n)$ for the clique, so $T_3 \in \Theta(n \log n)$. Thus,

Lemma 2.7. *The expected convergence time of BASIC on K_n is $2nH_n + n \log n \in \Theta(n \log n)$, independently of the placement of the original colors. This matches the expected cover time of the clique, and thus is optimal.*

2.5 A Lower Bound for BASIC in Static Graphs

We now examine what we can say about lower bounds for BASIC in general static graphs.

2.5.1 Match-Making Defines a Weighted Bipartite Graph

Let G be a static graph with some initial (arbitrary) distribution of RED and BLUE colors. Consider $B = \{u_1, \ldots, u_\kappa\}$, the set of all nodes $u_i \in V$ with BLUE color, and $R = \{v_1, \ldots, v_\lambda\}$, the set of all nodes $v_i \in V$ with RED color. (So $\kappa + \lambda = n$). Let w_{ij} = the length (number of edges) of a shortest path between u_i and v_j in G.

Consider now the bipartite graph $U = (B, R)$ with node sets B,R and edges e_{ij} of weights $w(e_{ij}) = w_{ij}$. Consider any particular sequence of random walks of the token in BASIC that matches all the red-blue pairs. Let the token start (say) in u_1 and match it with v_1. Then the token departs uncolored from v_1 until it meets a blue node, say u_2, again. Note that (1) each u_i is matched to a "new" v_i (not in $\{v_1, \ldots, v_{i-1}\}$), and (2) from each v_i the token seeks for a "new" u_i (not matched yet). Thus, the total time until convergence is at least the sum of the weights of two matchings in G, (a) the matching $\{u_i, v_i\}$, call it M_1, and (b) the matching $\{v_i, u_{i+1}\}$, call it M_2 (until all minority color nodes (say B) are matched). Let T be the time until convergence. In time T, the random walk process must hit the edges of the two matchings defined.

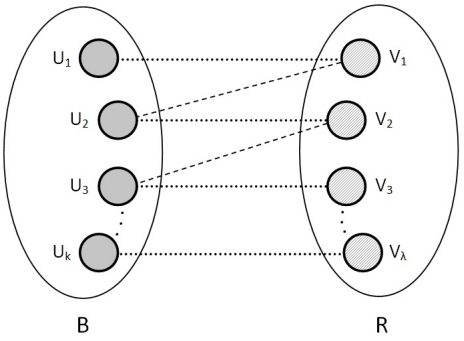


FIGURE 2.2: Bipartite Matching

Thus,

$$T \geq (weight(M_1) + weight(M_2)) \cdot h$$

where h is the minimum time to hit a subsequent node on the other side, which implies

$$\mathbb{E}(T) \geq (weight(M) + weight(M')) \cdot h_{min} \tag{2.1}$$

where M = the minimum weight matching in $U(B, R)$, M' = the minimum weight matching in $U(B, R) - M$, and where $h_{min} \stackrel{def}{=}$ the minimum hitting time of G = the minimum (over all u, v) of the expected time for a random walk starting at u to reach v for the first time. The proof of Eq. (2.1) is done using linearity of expectation. Thus,

$$E(T) \geq 2 \cdot weight(M) \cdot h_{min}. \tag{2.2}$$

If we know the initial placement of colors, then we can compute $weight(M)$ (and $weight(M')$) via a variation of the well-known Hungarian method [72] and the relaxed integer program Π. In the program below, $w_{i,j}$ is the length of the shortest path between vertices v_i and v_j of opposite colors and $x_{i,j}$ is the participation of the edge in the minimum matching between v_i and v_j, i.e. 1 if it's in the shortest path and 0 if not.

$$
\begin{aligned}
\Pi: \quad & \textit{minimize} \sum_{i,j} w_{i,j} x_{i,j} \\
\text{subject to} & \\
\sum_{j} x_{i,j} &= 1 \quad \forall i \in B \\
\sum_{i} x_{i,j} &= 1 \quad \forall j \in R \\
\sum_{i,j} x_{i,j} &= \text{ the number of vertices of the minority color} \\
x_{i,j} &\geq 0, \quad \forall i \in R, j \in B.
\end{aligned}
$$

The Hungarian method (see [72]) shows that this is an integral relaxation in the sense that any extreme point of the polytope of Π's constraints is the incidence vector of a (perfect) matching with respect to the minority's color (see also [101], exercise E). A primal-dual method can compute the weight of M in time $\mathcal{O}(n^3)$ [72]. Thus,

Lemma 2.8. *Given G and the placement of the original color, we can compute a lower bound on the time of BASIC until convergence in $\mathcal{O}(n^3)$.*

Note that the bound of Eq. (2.2) is a very crude one. In fact, even if we know the matchings M_1 (of min weight) and M_2 (of second min weight), the walk requires, for each subsequent pair (u_i, v_i) or (v_i, u_{i+1}) a hitting time on the remaining colors of high importance at that time. This increases by at least the smallest distance between two nodes of the same color and of high importance every time.

Using the idea above, we can show a lower bound for the BASIC protocol on the line with n vertices.

Definition 2.9. A (k, ℓ) RED-BLUE line (on $n = k + \ell$ nodes) is a path of n nodes consisting of a red path of k nodes, joined to a blue path of ℓ nodes.

Lemma 2.10. *The lower bound on the time of BASIC on a $(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil)$ RED-BLUE line (with n odd) is $\Theta(n^3)$.*

Proof. The weight of each edge on the bipartite graph U is the square of the shortest distance between the particular red/blue pair because of the random walk, and more than $n/4$ such edges have weight which is $\Theta(n)$. □

The lower bound above matches the upper bound given by Corollary 2.4 for the line.

2.6 Terminating the BASIC Process in Static Graphs

As defined in Section 2.2.1, BASIC never terminates, i.e. the token continues its random walk indefinitely, depsite the fact that the protocol converges in (expected) polynomial-time on an undirected graph.

We can supply BASIC with a termination criterion assuming:

1. The existence of a global clock, and
2. that each node v knows an upper bound $n' \geq n$ on the size of the graph.

Let T be the time (number of steps of the random walk) required for BASIC to converge. For any connected, undirected graph, we know (from Corollary 2.4) that

$$\mathbb{E}(T) \leq (n+1)\mathbb{E}(C(G)) \leq (n+1)2mn \leq 2n^4. \tag{2.3}$$

By Markov's inequality (see, e.g., [97]) we have

$$Prob(T \geq n\ \mathbb{E}(T)) \leq \frac{1}{n} \tag{2.4}$$

implying that

$$Prob(T \geq 2n^5) \leq \frac{1}{n}. \tag{2.5}$$

Therefore, BASIC can terminate (with probability of correctness at least $1-\frac{1}{n}$) as follows:
Termination Criterion: Each node v reads the global clock. When the global clock shows $2n'^5$ elapsed time steps, then node v reports its current color as the majority color and stops executing BASIC.

Remark 2.11. Alternatively, if the token itself has a clock (or can count the number of time steps it has made in the random walk), the token itself can terminate the random walk after walking for $2n'^5$ time steps. Each node then reports its current color as the majority color. This requires that the token has knowledge of n', and a counter that can record an integer up to the size of $2n'^5$.

2.7 Walks with Limited Counters in Graphs of Small Cover Time

One benefit of BASIC is the circulation of a single token in the graph, having only 2 bits of memory. Suppose that, similar to Remark 2.11, we allow the token to be equipped with a counter, but this time it uses its counter to record the number of RED/BLUE colors it sees. Then a single cover time of the graph clearly suffices for a randomly walking token to count the number of both colors in the graph and thus determine majority. Every time the token first encounters a color, it must mark the node as "visited" to avoid double-counting.

This simple procedure requires a counter that can count up to n (the size of the graph). We describe here a modification of this procedure, with the benefit that the counter of the token needs only be able to count up to $\omega(\sqrt{n}\log n)$.

Basically, we equip the token with a counter (initially zero) and we start its random walk at an arbitrary node. The counter keeps the difference $\delta_t = b_t - r_t$ (b_t, r_t are the

number of BLUE and RED nodes that have been visited by time t) by setting $\delta_t \leftarrow \delta_t + 1$ when the token encounters an unvisited BLUE node, and $\delta_t \leftarrow \delta_t - 1$ when the token encounters an unvisited RED node. Each time the token visits a node, if the status of the node is "unvisited" the token changes it to "visited" to avoid double-counting. After a time at least equal to a cover time, the token checks if the δ_t is positive or negative and then it performs another final walk to convert all nodes to the majority color (BLUE if $\delta_t > 0$, RED if $\delta_t < 0$, otherwise NEUTRAL when $\delta = 0$).

Clearly, if $|\delta_t| \leq g$ for all t until convergence (for some number g), then the counter will report correctly, provided it can count up to some number g' strictly greater than g.

We show here that g is enough to be set to some value $\omega(\sqrt{n}\log n)$ for this procedure to correctly report the majority with high probability. Our argument works under the following assumption.

Assumption A: Let p_t be the probability that the counter visits an unvisited majority color in the round t, and q_t be the probability that the counter visits an unvisited minority color in round t. We assume that $p_t \geq q_t$.

Assumption A is easily shown to hold when the colors are initially placed randomly in the vertices, and when the minimum degree of G is at least $\alpha \log n$ for some $\alpha \geq 2$.

Without loss of generality, assume that the initial majority color is BLUE. We consider a quite standard coupling process (δ_t, δ'_t) where $\delta_t = b_t - r_t$ and δ'_t is the current location of a simple random walk on (a subset of) the integers with a holding probability, i.e., a random walk on (a subset of) $\mathbb{Z}$ that can either increase or decrease by 1 with equal probablity, or remain stationary with (the remaining) positive probability. We give the details of this coupling below.

Let $\Delta(\delta_t) = \delta_{t+1} - \delta_t$, and $\Delta(\delta'_t) = \delta'_{t+1} - \delta'_t$ the corresponding increase or decrease in the random walk. There are nine cases to consider in the coupling, depending upon the values of $\Delta(\delta_t)$ and $\Delta(\delta'_t)$. The nine cases, together with the coupling probabilities are listed below. We need to define the coupling probabilities x_i for each of the cases.

$(\Delta(\delta_t), \Delta(\delta'_t))$	Coupling probability
$(0,0)$	x_1
$(0,1)$	x_2
$(0,-1)$	x_3
$(1,0)$	x_4
$(1,1)$	x_5
$(1,-1)$	x_6
$(-1,0)$	x_7
$(-1,1)$	x_8
$(-1,-1)$	x_9

First of all, we note that we want to couple the processes so that $\delta_t \geq \delta'_t$ for all t, so that if, for example, $\delta'_t = b_t$ then we guarantee that $\delta_t = b_t$ too. This immediately implies that we have $x_2 = x_7 = x_8 = 0$.

Secondly, to keep the coupling as tight as possible, we set $x_3 = x_4 = 0$.

We also have other conditions on the values x_i as follows:

$$x_1 + x_5 + x_6 + x_9 = 1 \text{ and } x_i \geq 0 \quad \forall i \tag{2.6}$$

$$x_9 = q_t \tag{2.7}$$

$$x_5 + x_6 = p_t \tag{2.8}$$

$$x_5 = x_6 + x_9 \tag{2.9}$$

$$x_5 + x_6 + x_9 = p_t + q_t \tag{2.10}$$

Condition (2.6) come from the fact that the x_i form a probability distribution. (2.7) comes from the definition of the probability q_t, i.e., the chance of the token finding an unvisited minority color, and similarly (2.8) is from the defintion of p_t. Equation (2.9) is from the fact the the process $\Delta(\delta'_t)$ is describing a simple random walk, i.e., $Pr(\Delta(\delta'_t) = 1) = Pr(\Delta(\delta'_t) = -1)$. We note that $p_t + q_t$ is the probability that the value of $\Delta(\delta_t)$ is non-zero.

Thus, solving for the values of x_i, we get the following coupling probabilities below (we show only the non-zero values):

$(\Delta(\delta_t), \Delta(\delta'_t))$	Coupling probability
$(0, 0)$	$1 - p_t - q_t$
$(1, 1)$	$\frac{1}{2}(p_t + q_t)$
$(1, -1)$	$\frac{1}{2}(p_t - q_t)$
$(-1, -1)$	q_t

With these probabilities, we have $\mathbb{E}(\Delta(\delta'_t)) = 0$ and $|\Delta(\delta'_t)| \leq 1$. We can apply the inequality of Azuma (e.g., see [97]) to the martingale $\Delta(\delta'_t)$ with bounded difference. By Azuma's Inequality we then have $|\delta'_t| \in \mathcal{O}(\sqrt{n}\log n)$ through a period of a cover time $\Theta(n \log n)$.

Thus, the difference of colors counted will never exceed $c\sqrt{n}\log n$ in the minority direction (w.h.p.) and will end up with a correct value in the majority direction. Therefore:

Lemma 2.12. *For any static unknown graph G where (a) Assumption A holds and (b) $\mathbb{E}(C(G)) \in \mathcal{O}(n \log n)$, the counter of the token needs to count only up to $\omega(\sqrt{n}\log n)$ in order to report the majority color w.h.p.*

2.8 The BASIC Protocol in Dynamic Graphs

We consider now the execution of the BASIC protocol in dynamic graphs with benign adversaries with tolerance β. Recall Definition 2.2 for the meaning of "benign adversary with tolerance β".

Lemma 2.13. *For any two nodes u,v, for any time t_1, with the token being at node u at time t_1, the probability that the token will visit node v at time at most $t_1 + \beta$ is at least $(\frac{1}{n})^{\beta}$.*

Proof. Suppose the token is at node u at round t_1. Consider that the edge uv appears again in round $t_1 + \beta'$, where $\beta' \leq \beta$. The event $A_{u,v}$ ="the token stays at u for $\beta' - 1$ times and then chooses edge $\{u, v\}$ which then exists" has probability

$$\varphi = \prod_{i=1}^{\beta'} (\frac{1}{d_i + 1})$$

where d_i is the degree of node u at round $t_1 + i$. But then $\varphi \geq (\frac{1}{n})^{\beta}$, since $\beta' \leq \beta$, and $n - 1 \geq d_i \geq 0 \ \forall i$ (so $n \geq d_i + 1 \geq 1$). □

Lemma 2.13 allows us to conclude that BASIC works correctly on dynamic graphs.

Corollary 2.14. *The BASIC protocol converts all node colors to the initial majority color (if any) in any dynamic graph, with a benign adversary, in finite time with probability 1.*

Proof. The events A_{uv} are each a geometric stochastic process of bounded variance. They are also independent of each other. Thus the (total) variance of the cover time of each walk is bounded. □

Then we also have the following result:

Theorem 2.15. *The BASIC protocol converts all node colors to the initial majority color (if any) in expected time at most $n^{\beta+2}$ in any dynamic graph with a benign adversary with tolerance β.*

Proof. The token needs at most n cover times to match all possible color-pairs. Each cover time is at least the cover time due to the repetition of the event $A_{u,v}$ n times. The expected time to visit all nodes is then at most $n^2 \cdot A_{u_i,u_{i+1}}$ where $u_0, \ldots, u_{n-1}$ is any permutation of the vertices, i.e. at most $n^2 \cdot \frac{1}{\varphi} = n^{\beta+2}$. □

2.9 BASIC with multiple tokens

In all sections up to this point, we have considered the use of a single token to solve the MCP.

We can consider speeding up this process by using multiple tokens, each performing a random walk in the graph. This raises issues of concurrency and contention resolution, namely what happens at a node that is visited by multiple tokens during one step of the random walk? To keep things as simple as possible, we will assume that a node visited by several tokens will interact with each token *one at a time* in some arbitrary order, with suitable state transitions occurring with each interaction (and that these interactions all happen within the same time step of the random walk).

Another issue that arises with multiple tokens using the BASIC procedure from Section 2.2.1 is that this may result in the tokens trying to report inconsistent information to the vertices, a state in which BASIC cannot recover from. There are two examples where this may occur, when the number of vertices is odd and there is a majority and when the number of vertices is equal and there is equality. These scenarios require the tokens to interact in some manner to resolve the inconsistency, either to annihilate each other and report equality or to disable tokens reporting neutral when there is in fact a majority being reported by another.

To see this, consider a graph having $2n+1$ vertices, where n are initially colored BLUE and $n+1$ are initially colored RED. Obviously this graph has RED as a majority color, but two (or more) tokens using the BASIC procedure can (eventually) result in one of the tokens being colored NEUTRAL and the other token being colored RED, each token attempting to convert all vertices to report their own color (RED or NEUTRAL) as the majority color, in a never-ending process that does not stabilize to the correct result.

Similarly, in a graph with $2n$ vertices, initially with n RED and n BLUE, a series of interactions could result in one token being colored RED and the other BLUE, with each token attempting to "recolor" (i.e. inform) the vertices in the graph to match their own color, but the truth is that there is a tie for the majority, so, again, the process does not stabilize to the correct solution.

The crux of both of these problems is that, using BASIC as previously defined, each token is unaware of the other tokens (or even if other tokens exist at all) and the color they are (or may be) carrying, and, critically, tokens do not (cannot) interact with one another. Some new method is necessary to avoid these failure issues identified above.

We define the procedure MultiBASIC in Figure 2.3 below (the prefix "Multi" refers to the fact that there can be two or more tokens performing the procedure). There are two essential differences. Intuitively, we can see that a node "remembers" its initial coloring, in that a node of "high" importance remains "high" until a "Complete Matching" stage occurs at that node. Only then does that node understand that it has been matched with a color of the opposite type. The second difference is that tokens can become "disabled". A "disabled" token still continues performing a random walk on the graph, but it ceases to recolor vertices, and simply looks for an unmatched node (i.e. one that has not yet changed from its initial state) to begin a new matching procedure, at which point the token becomes "active" again. These two token states (active or disabled) are denoted by A and D, respectively in Table 2.3. Tokens do not interact with one another.

Each token begins the protocol in state (N, A), and each node begins in state (C, H_C), where $C \in \{R, B\}$ is its initial color.

Note that in the transition table below, it is always the case that $C \in \{R, B\}$ (as before, $\overline{C}$ is the "opposite" of color C), and $X, Y \in \{R, B, N\}$. As before, we only list transitions where the state of a token and/or a node changes, all other interactions leave both token and node states unchanged.

Process stage	Token state	Node state	New token state	New node state
Begin Matching	(N, A)	(C, H_C)	(C, A)	(N, H_C)
Complete Matching	(C, A)	$(X, H_{\overline{C}})$	(N, A)	$(X, L_{\overline{C}})$
Inform	(X, A)	(Y, L_C)	(X, A)	(X, L_C)
Disable Token	(N, A)	(N, H_C)	(N, D)	(N, H_C)
Activate Token (and Begin Matching)	(N, D)	(C, H_C)	(C, A)	(N, H_C)

FIGURE 2.3: The MultiBASIC protocol.

The solution to the MCP, from the point of view of a node, is that the (current belief for the) majority color is $C \in \{R, B\}$ if the state of a node is (C, H_C) or (N, H_C). Alternatively, the (current belief of the) majority color is $X \in \{R, B, N\}$ if the state of the node is (X, L_C) (for some $C \in \{R, B\}$).

In contrast to the BASIC protocol, each node now requires four bits to store its state (using two bits for the current belief of majority color, and two bits for one of the four possible states H_R, L_R, H_B, and L_C). Also, the token requires three bits, two for the current color (R, B, or N) it is carrying, and one bit for denoting wheter it is active/disabled.

Theorem 2.16. *MultiBASIC correctly solves the MCP on any connected graph.*

Proof. This proof is very similar in nature to that of Theorem 2.3, noting that each initial color C on a node will turn the state of another node from $(\cdot, H_{\overline{C}})$ to $(\cdot, L_{\overline{C}})$, and vice-versa.

If there is a surplus of one color $C \in \{R, B\}$, after a finite series of token/node interactions, all vertices that started in state $(\overline{C}, H_{\overline{C}})$ will end up in state $(N, L_{\overline{C}})$. At that point, there still exists (at least) one node in state (C, H_C), and/or there is (at least) one token in state (C, A) (in which case there is at least one node in state (N, H_C)), with all other tokens being colored C or N.

Eventually, every token will end up in one of the two states (C, A) or (N, D), with at least one in the former state, which will correctly inform all vertices of the majority color C.

In the case of a tie for $\#RED$ and $\#BLUE$, the token that completes the final C to $H_{\overline{C}}$ matching at a node (for a $C \in \{R, B\}$, turning the state of that node to $(N, L_{\overline{C}})$), will remain active (i.e. its state is (N, A) and remains so forever more as there are no nodes in state $(\cdot, H_C)$ for $C \in \{R, B\}$) to inform all vertices that the (correct) solution to the MCP is N. □

Remark 2.17. We note, in fact, that the MultiBASIC procedure still remains correct if only one token uses it to solve the MCP (with the vertices following the transitions defined in Figure 2.3).

Remark 2.18. Noting that a node can never reach a state $(C, H_{\overline{C}})$ for some $C \in \{R, B\}$ during execution of MultiBASIC, we can alter the MultiBASIC protocol by exchanging

the "Complete Matching" transition in Figure 2.3 for the pair of state transitions in the table below (keeping all other transitions the same):

Process stage	Token state	Node state	New token state	New node state
Complete Matching	(C, A)	$(N, H_{\overline{C}})$	(N, A)	$(N, L_{\overline{C}})$
	(C, A)	$(\overline{C}, H_{\overline{C}})$	$(\overline{C}, A)$	$(N, L_{\overline{C}})$

FIGURE 2.4: Speeding up the MultiBASIC protocol.

2.10 The k-surplus Problem

We now consider the adaptation of BASIC to a new, yet related problem, the *k-surplus problem*, for $k \geq 1$. As a reminder to the reader, given a connected graph where each node is originally colored BLUE or RED, the k-surplus problem is to determine (and inform all nodes) if one of RED or BLUE occurs k or more times than the other color. (The MCP is therefore the same as the 1-surplus problem.) For simplicity, we will use the phrase "k-surplus" as shorthand for "the k-surplus problem."

By adapting BASIC in a natural manner, we can provide a protocol to solve k-surplus. Specifically, we give an algorithm that converges in finite time, i.e., we can determine if one color outnumbers another and inform all vertices, where by "inform" we mean that each node will eventually converge on a common value of RED (if RED outnumbered BLUE by k or more), BLUE (similarly), or NEUTRAL (if neither color outnumbered the other by k or more). This algorithm is guaranteed to converge to the correct answer, and will find (and inform all nodes) in finite time.

Similar to BASIC, our solution to k-surplus involves a token performing a random walk on the graph. Informally, our protocol is still "matching" opposite colors, but now the token is capable of counting up to the value k. So the token can attempt to match (up to) k vertices of one color to the opposite color. Once all opposite pairs of colors have been matched, the token simply continues to walk in the graph. If there is a k-surplus (i.e. RED outnumbered BLUE by k or more, or vice-versa), the token will (eventually) determine that and inform all nodes by recoloring them to the surplus color. If there is not a k-surplus, the token will inform all nodes of that fact as well, by recoloring them to NEUTRAL.

We provide the state transitions below for the k-surplus protocol. This protocol is a natural extension of BASIC. For this proposed method we assume the existence of a single token which is initially placed in some arbitrary node. The token will store a pair of states in the form $(color, surplus)$. The color which the token can adopt is either R, B, or N (short for RED, BLUE, or NEUTRAL as in BASIC), and will start as N initially. The surplus is an integer i where $i \in \mathbb{Z}$ and $0 \leq i \leq k$. Initially the surplus is 0.

The nodes in the graph store information similar to what they stored in BASIC, namely an ordered pair $(color, importance)$ which are initialised as (C, H), where C is a

color belonging to $\{R, B\}$. Throughout the k-surplus protocol, we always have the first component of each node's state is in $\{R, B, N\}$ and the second component in each node's state is either H or L.

In the interactions in Fig. 2.5, we note that $C \in \{R, B\}$ and $X \in \{R, B, N\}$. As before, for $C \in \{R, B\}$, $\overline{C}$ denotes the "opposite" color, e.g., if $C = R$, then $\overline{C} = B$. Note also that $\alpha \in \mathbb{Z}$ with $1 \leq \alpha < k$.

Process stage	Token state	Node state	New token state	New node state
Begin Matching	$(N, 0)$	(C, H)	$(C, 1)$	(N, L)
	(C, α)	(C, H)	$(C, \alpha + 1)$	(N, L)
Ignore	(C, k)	$(C, \cdot)$	(C, k)	$(C, .)$
Complete Matching	$(C, 1)$	$(\overline{C}, H)$	$(N, 0)$	(N, L)
	(C, α)	$(\overline{C}, H)$	$(C, \alpha - 1)$	(N, L)
	(C, k)	$(\overline{C}, H)$	$(C, k - 1)$	(N, L)
Inform	(C, k)	(X, L)	(C, k)	(C, L)
	(C, α)	(X, L)	(C, α)	(N, L)
	$(N, 0)$	(X, L)	$(N, 0)$	(N, L)

FIGURE 2.5: State transitions for k-surplus.

Theorem 2.19. *(Correctness) In any static undirected, connected, finite $G = (V, E)$, the k-surplus protocol of Fig. 2.5 eventually turns the color of every node to the color having k-surplus (if there was a k-surplus) or to NEUTRAL (if there was no k-surplus), even if G and its size are unknown to the nodes.*

Proof. The token matches each node of color X and high importance (i.e. as initially) to a node of color $\overline{X}$ of high importance, and both nodes turn to low importance. Thus, the initial (high-importance) nodes are paired in RED-BLUE pairs. The only difference between BASIC and the k-surplus protocol is that the token can be attempting to perform up to k matches at the same time (whereas in BASIC the token could only do one match at a time.)

If a k-surplus majority color X initially existed, then eventually the token will find it by visiting all nodes and increasing the surplus count stored in the token to the value k. The token will then walk the graph converting all nodes (of low importance) to the k-surplus color X. Note that for a finite G, the token's random walk needs at most a time equal to the cover time of G every time it needs to match a color and each time it needs to start a new matching. Once all RED-BLUE matches have been made, another cover time will suffice to establish if there is a k-surplus or not, and a final cover time of G in order to convert the color of all nodes to the k-surplus color, or to NEUTRAL if no k-surplus initially existed.

Since G is finite and connected, the cover time of G is finite, and hence the k-surplus protocol will converge in finite time. □

Remark 2.20. The k-surplus protocol on a finite, connected graph G will converge in time at most $(n + 1) \cdot \mathbb{E}(C(G))$. This follows similarly to Corollary 2.4. Indeed, the

convergence time for $k \geq 2$ is no worse than the convergence time for $k = 1$, which is identical to the MCP, as being able to search for more matching pairs simultaneously can only decrease the convergence time.

Remark 2.21. In contrast to the BASIC protocol, for k-surplus the token needs to be able to count up to k, and hence requires $\Theta(\log k)$ bits in order to perform this task.

Remark 2.22. We proposed MultiBASIC to utilize several tokens to speed up solving the MCP. Unlike that problem, it is not possible to utilize more than one token to solve k-surplus without allowing the tokens to interact with one another, since one token could count there are ℓ more RED than BLUE and another could count there are m more RED than BLUE, where $\ell + m \geq k$, but $\ell < k$ and $m < k$.

2.11 Absolute Majority

Let us now consider the case of more than two colors in the graph, and the problem of determining if one color has an *absolute majority*, i.e., is there one color that outnumbers the set, taken collectively, of other colors?

Suppose that k denotes the number of colors that initially exist in the graph, and let us assume that the nodes are aware of k. We denote the set of colors by $\{C_1, \ldots, C_k\}$.

One idea is to use BASIC as a building block to solve this problem. A natural solution for the Absolute Majority Color Problem, henceforth abbreviated as "AbMCP", is to sequentially run the BASIC protocol k times, once for each color in an "One vs. Other" fashion. In other words, we consider the color C_i versus the set of all other colors $C^{-i} \stackrel{def}{=} \{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_k\}$, considered temporarily as a single "color", and determine if the color C_i or the "color" C^{-i} is the solution to the corresponding two-color MCP (or, indeed, if there is a tie).

However, the inherent problem in running BASIC in this sequential manner is that the BASIC protocol in its simplest form does not terminate. Thus, there is no mechanism to stop one iteration for the color C_1 (versus all other colors) and begin the next iteration of BASIC for the color C_2.

Therefore, our proposed method to solve AbMCP is to run k copies of BASIC *in parallel*. What this means is that each node will maintain an ordered k-tuple, where each element of the k-tuple is itself an ordered pair. The ith element of this k-tuple corresponds to an execution of the BASIC protocol where C_i is compared to the set C^{-i} (considered as a single color, as mentioned previously). The token also maintains an ordered k-tuple, where each element in the k-tuple corresponds to the execution of the BASIC protocol of C_i versus C^{-i}.

The transitions of BASIC described in Fig. 2.1 would be modified, operating on each coordinate of the k-tuple to compare C_i with the "color" C^{-i}. We leave it to the reader to work out the details, but the generalization should be natural. We assume that the token interacts with a node on each component of a node's k-tuple in a single step of the random walk.

As an alternative, k tokens could be used instead, where token t_i operates on the ith component of a node's k-tuple. As another alternative, several tokens could operate in parallel with suitable adaptation of the MultiBASIC protocol from Section 2.9, adapted to account for operating on k-tuples of information in the vertices. (In this case, tokens might become "disabled" on specific coordinates of the k-tuples while remaining active on other coordinates.)

In any case, what is the solution to AbMCP? If C_i is the absolute majority color, then each node's k-tuple will converge to $(C^{-1}, C^{-2}, \ldots, C_i, \ldots, C^{-k})$.

If there is no absolute majority color, then the ith component of each node's k-tuple will converge either to C^{-i} or to N, the second case only occurring when $|C_i| = \sum_{j \neq i} |C_j|$, where $|C_i|$ is the number of nodes originally colored C_i. A node simply checks its k-tuple to determine the (current) conclusion on the existence of an absolute majority, keeping in mind that the information present in the k-tuple can possibly be inconsistent until convergence has occurred.

The correctness of BASIC ensures the correctness of this AbMCP protocol. The running time of the AbMCP protocol satisfies the same running time bounds as BASIC (under the assumption that the token/node interactions happen for each component in one time step of the random walk).

Note that each node now requires $\Theta(k)$ bits of memory to correctly perform the AbMCP protocol and record the result (i.e. k copies of BASIC, each using $O(1)$ bits of memory), and, for similar reasons (assuming a single token), the token also needs $\Theta(k)$ bits of memory.

2.12 Relative Majority

In the previous section we studied the case of absolute majority, by which one color has the largest majority over the sum of the other colors combined. In this section we study and propose a solution to the case of relative majority, in which one color beats each other color in the network in a direct comparison (solely between those two colors). We use the abbreviation "RelMCP" to refer to the Relative Majority Color Problem.

As in the last section, our proposed solution to this problem is based on running several executions of the BASIC protocol in parallel. In particular, $\binom{k}{2}$ instances are required to run in parallel, one for each pairwise comparison between the k colors.

Each node in the network will operate with a $\binom{k}{2}$-tuple, where each component of the tuple is itself an ordered pair. A particular component will correspond to the comparison of two colors under the BASIC protocol. Obviously each node must have the same ordering of the components of the their tuples to arrive at a consistent answer (e.g. the color pairs can be arranged in lexicographic order in the tuple).

One token, itself with a $\binom{k}{2}$-tuple, can execute these BASIC protocols, operating in a component-wise fashion interacting with the nodes according to BASIC for each pair of colors. Again we assume that the token can do all $\binom{k}{2}$ updates on each component of

its tuple (and those of the node it is interacting with) in one step of the random walk. Alternatively, several tokens can be used to execute all of the BASIC protocols. In this case either the collection of protocols is partitioned in some fashion amongst the tokens and/or they are utilizing versions of the MultiBASIC protocol defined in Section 2.9 to avoid the potential problems described there with the use of multiple tokens.

Note that the entire collection of $\binom{k}{2}$ pairwise comparisons of colors is necessary to determine an answer to RelMCP. If, say, the comparison of C_1 and C_2 is left out, then it could be the case that C_1 beats each other color C_i for $i \geq 3$, and similarly for C_2. Without the comparison between C_1 and C_2 , we would be unable to conclude if C_1 or C_2 has the relative majority, or if those colors were tied.

A node's $\binom{k}{2}$-tuple is initialized with components equal to (C_i, H_{C_i}) for any comparison that involves its original color C_i, otherwise it is initialized with the component (N, L_{C_j}) (or (N, L_{C_k})) in order to capture the result of the comparison of the colors C_j and C_k for $j \neq i$ and $k \neq i$.

A node examines its $\binom{k}{2}$-tuple to determine the solution to RelMCP. This tuple could be displaying inconsistent information about the solution, but once all $\binom{k}{2}$ processes converge, the set of inequalities can be resolved to determine if one color has the relative majority, or if two or more colors tie for most present in the graph. Note that the $\binom{k}{2}$-tuple also allows determination of all of the order statistics on the colors, i.e. which color(s) is (are) largest, second largest, etc.

Finally, with the suitable use of a single token, or multiple tokens operating on different subsets of the pairwise comparisons, or multiple tokens utilizing variants of the MultiBASIC protocol, this process will converge correctly on all coordinates of the $\binom{k}{2}$-tuple, allowing each node to conclude on the (non)existence of a color having relative majority.

For relative majority finding in the manner described, each node will utilize $\Theta\left(\binom{k}{2}\right)$ bits of memory, as will the token (or set of tokens). With suitable consideration, using only one token, we could instead equip each node with only $O(1)$ bits of memory, and a token with $\Theta\left(\binom{k}{2}\right)$ bits, where the token itself performs the random walk and comparisons to determine the answer to RelMCP. Each node utilizes only $O(1)$ bits for its contribution to a BASIC protocol with the token, and an additional $O(1)$ bits to record the outcome of the protocol *as reported to itself by the token.*

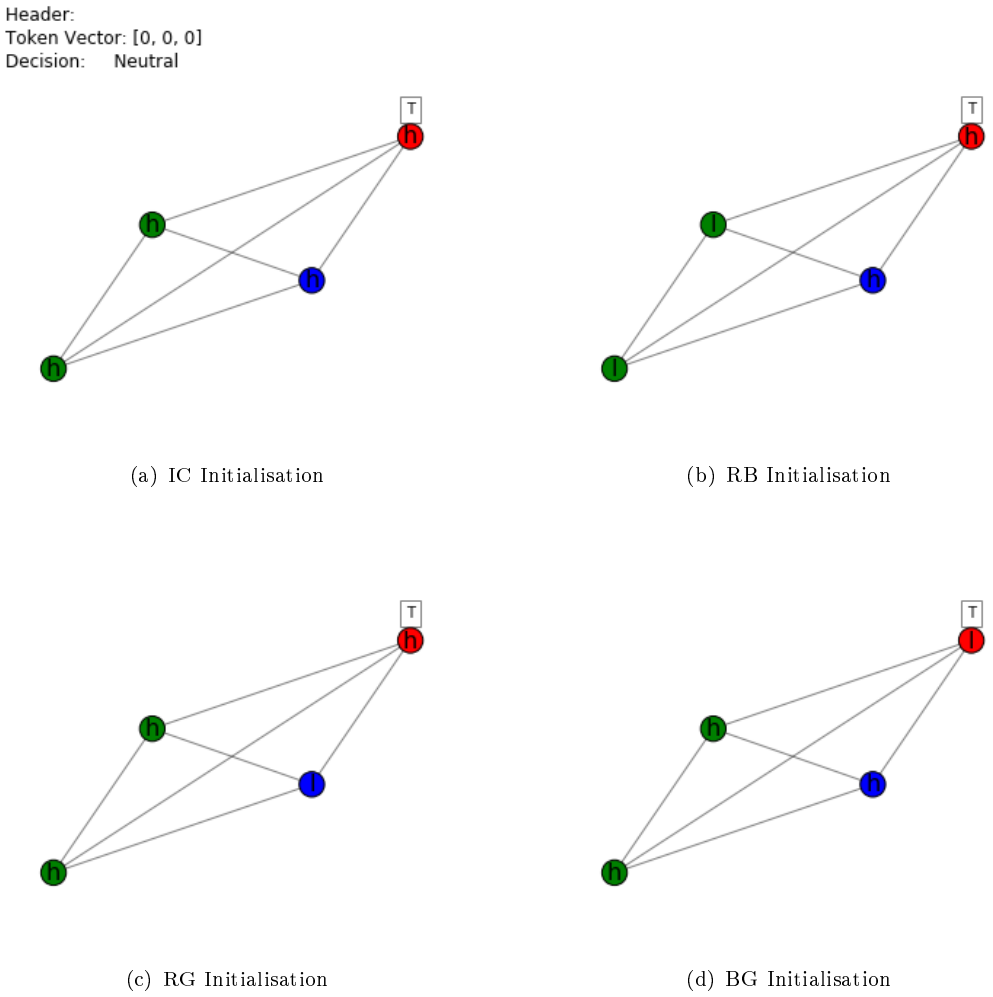
2.13 Relative Majority Simulation Example

In this section we show how BASIC can be used to solve relative majority via our described implementation.

Here we consider a complete graph, where $n = 4$ and $k = 3$. There exists a relative majority of *green* vertices in the graph. The following figures are displayed in order of the execution. The first four figures show the initialisation of the *Instance Controller* (a class which interprets the token vector and the state of all instances upon each step of the

algorithm) and the $\binom{k}{2}$ $BASIC$ instances $[(Red-Blue), (Red-Green), (Blue-Green)]$. In each pairwise $BASIC$ instance, vertices of participating color are initialised high importance (denoted by h on each vertex) and vertices of non-participating colors are initialised low importance (denoted by l on each vertex). For example, consider vertex v_1 with initial color $BLUE$. In this example where $k = 3$, the vertex has 3 instances such that $v_1 = ((B, H), (B, L), (B, H))$. The reasoning for this initialisation is due to blue participating in the first and third *duels*.

At every step the IC instance interprets the token vector based on the state of each $BASIC$ instance, which is displayed at the top of each IC figure. Consider the first instance $(r > b)$, the token at this index will contain 1 if $r > b$, 0 if $r = b$ or -1 if $r < b$. The decision value is the current belief that is being disseminated through the network based on the interpretation of the token vector. The protocol terminates after 11 steps and informs all vertices in all instances the relative majority has been found.

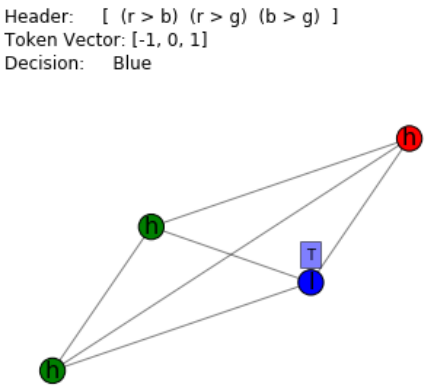


(a) IC Initialisation

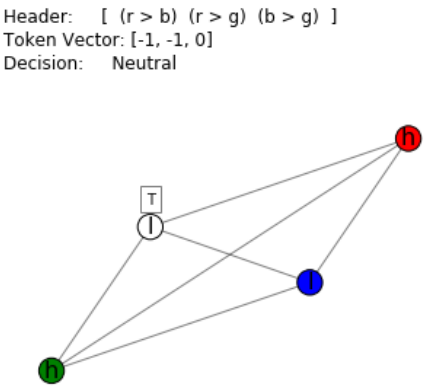
(b) RB Initialisation

(c) RG Initialisation

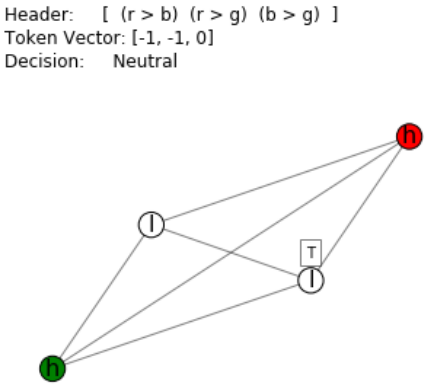
(d) BG Initialisation



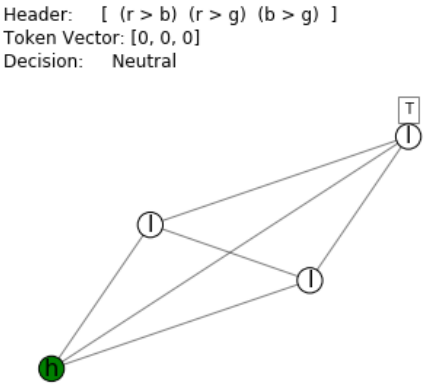
(e) IC Step 1



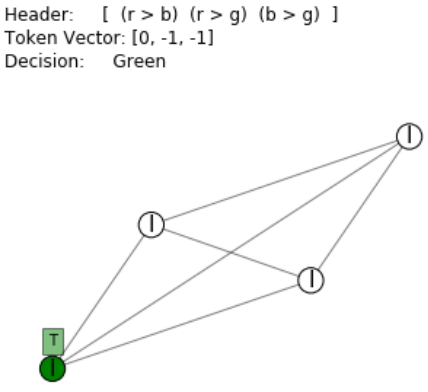
(f) IC Step 2



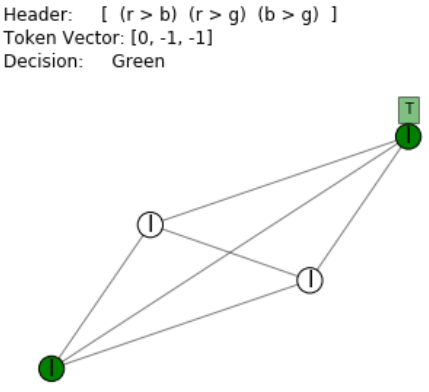
(g) IC Step 3



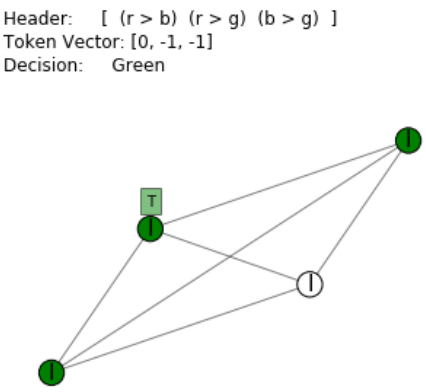
(h) IC Step 4



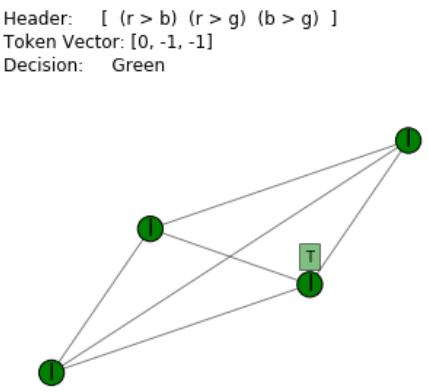
(i) IC Step 5



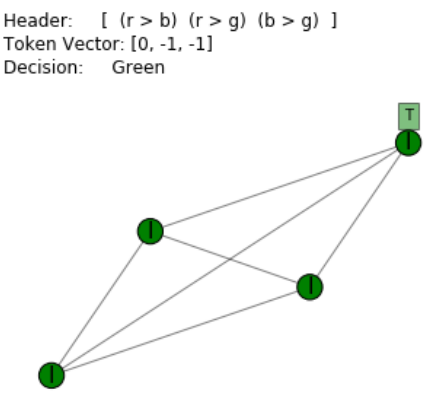
(j) IC Step 6



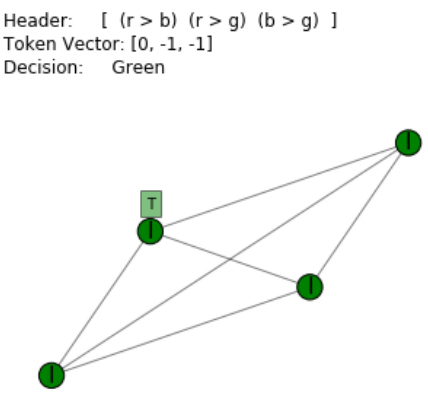
(k) IC Step 7



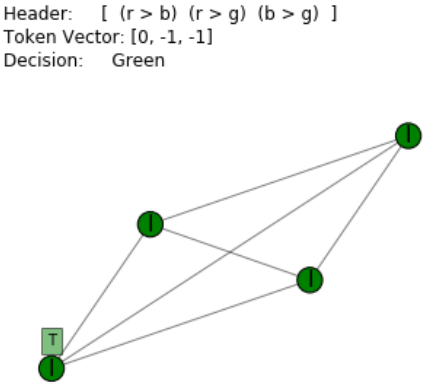
(l) IC Step 8



(m) IC Step 9



(n) IC Step 10



(o) IC Step 11

2.14 Future Work

Several problems remain open.

The use of a finite-state machine in each node in our protocols bear a strong resemblance to the Population Protocol model of Angluin, et al [27]. It is well-known [25] that the computational power of the basic Population Protocol model coincides with the class of semi-linear predicates, consisting of all predicates definable by the first-order logical formulas of Presburger arithmetic [70].

However, simple extensions such as the Mediated Graph Protocol (MGP) model [44] in which each network link is characterized by a state drawn from a finite set allow more complex computations, and, in particular, testing all graph properties decidable by a non-deterministic Turing machine with linear space that takes as input the adjacency matrix of the input graph.

Analogously, one could ask about the computational power of systems based on the random walk (combined, say, with finite-state machines as we are using here), a biased random walk [81], multiple random walks [17], and other extensions.

Furthermore, one can look at deterministic counterparts of the random walk. A good example is the rotor-router mechanism [105] also known as the Propp Machine [86]. On first look, the rotor-router would guarantee $\mathcal{O}(n^2)$ MCP computation on the path, comparing to the cubic performance of the (usual) random walk. However, it appears that the rotor-router's performance on cliques would likely be less efficient than the $\mathcal{O}(n \log n)$ bound from Section 2.4 in the worst case.

Finally, it could be interesting to study the Majority Color Problem on non-trivial special classes of graphs as complete graphs can be solved in $\mathcal{O}(n \log n)$ expected time and $\mathcal{O}(n^2 m)$ time on any connected undirected graph. Using Corollary 2.4, any upper bound on the expected cover time for a class of graphs immediately translates into an upper bound on the convergence time of BASIC. For example, it is known that the cover time for any regular graph on n vertices is at most $2n^2$, giving an upper bound of $\mathcal{O}(n^3)$ for convergence of BASIC on such graphs [61].

Chapter 3

Deterministic Population Protocols for Absolute Majority and Plurality

3.1 Introduction

The model of *population protocols* adopted in the work of this chapter was proposed first in the seminal paper by Angluin *et al.* [21] and popularised later in [24]. Their model provides a suitable theoretical framework for studying pairwise interactions within a large collection of anonymous (indistinguishable) *entities*, also referred to as *agents*, equipped with little computational power. The entities are modelled as *finite state machines*. When two entities engage in interaction they mutually access their local states and, on the conclusion of the encounter, their states get updated according to the global (shared) *transition function*. In the *asynchronous model*, also adopted in the work of this chapter, the order of interactions in consecutive rounds is unpredictable but fair, i.e., none of the pairs of entities can be starved from interaction. In this model, the main emphasis is on feasibility of the solution, subject to the limit on the number of states available to the entities. In the *probabilistic model*, in each round the *random scheduler* picks a pair of entities uniformly at random. In the presence of the *random scheduler*, on the top of space restrictions, one is also interested in the time complexity of a specific distributed task. A population protocol terminates if all participating entities eventually agree on some value represented by dedicated states, independently of the order of interactions. This value can reflect the colour or the size of selected majority [22, 24, 67, 90], the identity of the leader [12, 13, 55], but also completion of more complex tasks such as network formation [92], counting [94], and others.

The adopted computation model of the work in this chapter, encompasses a population A of n entities, each equipped with a $O(k)$-bit memory, where 2^k is the bound on the number C of colours present in the population. This is in contrast to the majority settings considered earlier in [22, 24, 67, 90] where only two original colours were permitted. Here each entity is coloured with exactly one of C available colours and a k-bit label representing this colour is kept in the entity's memory.

As indicated before, the entities communicate in pairs in an *asynchronous manner*. The main task in the *majority problem* is to identify the most frequent colour in the population. Due to presence of more than two colours in the population, we distinguish between the *absolute majority*, i.e., where one colour dominates all others taken together, and the *relative majority*, also known in the literature as *plurality consensus*, where the population is expected to agree on (one of) the most frequent colour(s). We also distinguish between the *static majority* in which the original colours of entities cannot be altered in time - the assumption used in the past work on majority protocols [22, 24, 67, 90], and the *dynamic majority* in which the original colours of entities can be changed in due course by an *external force*, and by doing so may alter the outcome of the majority protocol. This is the main reason why in our model the entities must store their original colour, which could be altered at any time but only by the external force, in addition to $O(k)$ memory bits required during interactions and to report the majority on the conclusion of the computation process.

The model with the external force adopted by the work in this chapter was considered earlier in [91] under the name *computing with stabilizing inputs*. Note that the dynamic protocol described in Section 3.3 is a special variant of self-stabilization, as state alterations done by the external force are permitted only between certain (colour indicating) states. We would also like to emphasise that protocols for absolute majority presented in Section 3.4 and the relative majority in Section 3.6 refer to earlier work on *composition of population protocols* from [20].

In our model, entities interact using a classical population protocol, i.e., via global grammars mapping pairs of states to pairs of states. In particular, no exchange of local memories happens during pairwise interactions. The entities use their local memory in order to organise the sub-protocols executed and in order to draw local conclusions. Thus, if we count the states needed for entities' interactions, we require only $\Theta(k)$ states in our algorithms for absolute and relative majority, and only a constant number of states for protocols computing static and dynamic majority of two colours. In addition, we need only $O(k)$ bits of local memory per entity in our absolute and relative majority protocols in order to handle up to 2^k colours, which is optimal in terms of space requirements.

3.1.1 Related Work

The population protocol model was initially introduced to simulate behaviour of animal populations [21, 22]. In [21] we can find a formal definition of computations in populations where pairwise interactions of finite-state agents advance the computation. The authors showed a fundamental result that any predicate which is semi-linear can be stably computed by such protocols. In the introduction of their paper, they present a protocol for majority which is exactly the same as the protocol in Section 3.2 of this chapter. In [91] the authors present several models of population protocols including protocols in which each entity of the population is allowed to have some memory, and they discuss several classes of computable predicates in those models. In the first pages,

they present a protocol for majority as in [21], which is almost the first protocol presented here. We have included this first protocol in this chapter because we add a detailed explanation about reporting ties. Self-stabilizing population protocols were defined in [26] and properties of such protocols were demonstrated. Stabilizing population protocols in the presence of faults were considered in [53].

In due course, population protocols proved to be a useful abstraction in diverse environments including, e.g., wireless sensor networks [20, 58, 102], chemical reaction networks [46], and gene regulatory networks [41]. A large portion of work devoted to population protocols refers to the majority problem. In particular, in [24] the authors study populations with entities governed by 3 states and propose a probabilistic population protocol for *approximate majority*, i.e., where the initial difference between the volumes of the two colours does not fall below $\omega(\sqrt{n}\log n)$. The algorithm stabilises in $O(n\log n)$ rounds with high probability. It also tolerates groups of $o(\sqrt{n})$ entities expressing Byzantine behaviour. Further analysis of this protocol and its 4-state amendment leading to the first efficient exact majority protocol can be found in [90]. Another aspect referring to the parallelism of majority population protocols in the presence of a random scheduler has been studied by Alistarh et al. in [14]. They proposed a poly-logarithmic time majority protocol for entities equipped with memories of size $O(1/\varepsilon+\log n\log 1/\varepsilon)$, for any $\varepsilon>0$. They also study the respective lower bounds. In a very recent work [12] Alistarh et al. consider a wide spectrum of time and space trade-offs for population protocols and they propose a fast Split-Join majority algorithm stabilising in $O(\log^3 n)$ parallel rounds with high probability. An interesting extension of population protocols to the *random walk* model can be found in [67]. Please note that neither of the majority algorithms discussed above is able to report the tie.

The relative majority variant considered in this chapter is well known in the literature under the name of *plurality consensus*. In contrast to the deterministic sequential model adopted in the work presented in this chapter, so far plurality consensus was considered solely in the *gossiping model*. In this model, in a sequence of synchronous rounds each entity contacts a random neighbour simultaneously. Moreover, the protocols converge under the assumption that the number of entities supporting the winning colour must exceed those supporting any other colour by a sufficiently large bias. In this model one explores parallelism of connections aiming at protocols stabilising rapidly with high probability. Doerr et al. [54] explored the *power of two choices* in complete graphs, proposing a stabilisation protocol in the binary case requiring constant memory and message size. Their protocol converges in $O(\log n)$ rounds assuming a bias of size $\Omega(\sqrt{n\log n})$. A more rigorous analysis of this protocol can be found in [48], also in networks modeled by regular graphs, for which the authors provide tight bounds on convergence time as a function of the second-largest eigenvalue of the graph. In [34] Bechetti et al. consider a plurality consensus protocol based on a sequence of local majority agreements with three randomly chosen neighbours during each round requiring bias $\Omega(\sqrt{Cn\cdot\log n})$. The protocol converges in $\Theta(\min\{C,n^{1/3}\}\cdot\log n)$ rounds using $\Theta(\log C)$ memory and message

size, where C refers to the number of original opinions. In later work [35] the authors solve general plurality consensus in complete graphs via *undecided-state dynamics* using an extra state to accommodate intermediate disagreements. They propose the notion of *monochromatic distance* which reflects on the difference between the initial colour configuration from the closest monochromatic solution. Their plurality protocol converges with a logarithmic overhead on the top of the monochromatic distance. A more recent study on plurality consensus in noisy communication channels can be found in [65].

There is also growing interest in exact-space complexity in probabilistic plurality consensus. In particular, in [37] Berenbrink et al. proposed a plurality consensus protocol converging in $O(\log C \cdot \log\log n)$ synchronous rounds using only $\log C + (\log\log C)$ bits of local memory. They also show a slightly slower solution converging in $O(\log n \cdot \log\log n)$ rounds using only $\log C + 4$ bits of local memory. This disproves a conjecture by Becchetti et al. [35] implying that any protocol with local memory $\log C + O(1)$ has the worst-case running time $\Omega(k)$. In [69] Ghaffari and Parter propose an alternative algorithm converging in $O(\log C \log n)$ rounds while having message and local memory sizes based on $\log C + O(1)$ bits. In addition to the above, some work on the application of the random walk in plurality consensus protocols can be found in [35, 67].

3.1.2 Our results and organisation of the chapter

The work in this chapter documents the study space-optimal population protocols for several variants of the majority problem. The work presents space-efficient algorithms for majority with many colours, and these algorithms are obtained by using a combination of known protocols for simple majority. In Section 3.2 we discus an amendment allowing majority protocols to report a tie (equality) if neither of the two original colours dominates the other. In Section 3.3 we discuss a solution to the dynamic version of the majority problem in which the original colours assigned to the entities can be changed by an external force. Such a solution is a special case of self-stabilizing population protocols which were considered in [26]. We discuss it here to prepare the ground for our space-optimal protocols for many colours.

We consider space-efficient majority protocols in populations with an arbitrary number C of colours represented by k-bit labels, where $k = \lceil \log C \rceil$. In Section 3.4 we present an asymptotically space-optimal $O(k)$-bit protocol for the *absolute majority*, i.e., a protocol which answers the question whether one colour dominates all others taken together. In Section 3.6 we propose a multistage $O(k)$-bit protocol for *relative majority*, where all most frequent colours eventually become aware of their dominance, and all nodes learn about the most frequent colour with the largest label. In Section 3.7 the chapter concludes with final comments and a list of open problems.

3.2 Population protocol for static majority with equality

This section reformulates the algorithm for majority presented in [21].

Initially each entity $a \in A$ obtains its original colour c_a, being one of the three available denoted by integers $-1, 0$, and 1. Thus, the main goal in our reformulation of majority protocols is to determine whether there are more 1's than (-1)'s (green domination), more (-1)'s than 1's (red domination), or whether there is a tie between the two. In other words, our majority protocols aim at determining the sign of the expression:

$$\sum_{a \in A} c_a.$$

If this sign is positive, there are more 1's, if negative, there are more (-1)'s, and if the sum is 0, we report equivalence between the two competing colours. During the communication process each entity $a \in A$ has an attributed *state* s_a. In due course we will also use the notion of *knowledge* of entities, which includes information about the state and the original colour of the entity.

Throughout the computation process the entities can be in one of the three *strong states* $[-1], [0]$, and $[1]$ or the three weak states $\langle -1\rangle, \langle 0\rangle, \langle 1\rangle$. In the begining, each entity $a \in A$ with attributed colour $c_a = x$ is in state $[x]$. With each state s we associate a weight $w(s)$ such that $w([x]) = x$ and $w(\langle x\rangle) = 0$. This association is illustrated by the table in Fig. 3.1.

state s	weight $w(s)$
$[-1]$	-1
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	0
$[1]$	1

FIGURE 3.1: The states and their weights.

In due course, when two entities $a, b \in A$ interact the shared *transition function* determines their resulting states. And, in particular, if an entity in a strong state $[x]$ meets another in a weak state $\langle y\rangle$, the weak state becomes $\langle x\rangle$ and the strong state remains unchanged. If during a meeting a strong state $[x]$, for $x \neq 0$, meets $[0]$ then only state $[0]$ is changed to $\langle x\rangle$. Finally, if $[1]$ interacts with $[-1]$ both states are changed to $[0]$. Other type of encounters does not change the states of entities. The respective shared transition function is illustrated by the table in Fig. 3.2.

$s_a \backslash s_b$	$[-1]$	$[0]$	$[1]$	$\langle -1\rangle$	$\langle 0\rangle$	$\langle 1\rangle$
$[-1]$	$([-1],[-1])$	$([-1],\langle -1\rangle)$	$([0],[0])$	$([-1],\langle -1\rangle)$	$([-1],\langle -1\rangle)$	$([-1],\langle -1\rangle)$
$[0]$	$(\langle -1\rangle,[-1])$	$([0],[0])$	$(\langle 1\rangle,[1])$	$([0],\langle 0\rangle)$	$([0],\langle 0\rangle)$	$([0],\langle 0\rangle)$
$[1]$	$([0],[0])$	$([1],\langle 1\rangle)$	$([1],[1])$	$([1],\langle 1\rangle)$	$([1],\langle 1\rangle)$	$([1],\langle 1\rangle)$
$\langle -1\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle -1\rangle,\langle -1\rangle)$	$(\langle -1\rangle,\langle 0\rangle)$	$(\langle -1\rangle,\langle 1\rangle)$
$\langle 0\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle 0\rangle,\langle -1\rangle)$	$(\langle 0\rangle,\langle 0\rangle)$	$(\langle 0\rangle,\langle 1\rangle)$
$\langle 1\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle 1\rangle,\langle -1\rangle)$	$(\langle 1\rangle,\langle 0\rangle)$	$(\langle 1\rangle,\langle 1\rangle)$

FIGURE 3.2: The transition table for static majority protocol with ties.

Lemma 3.1 (Invariant 1). *Initially, the sum* $S = \sum_{a \in A} w(s_a)$ *equals to* $\sum_{a \in A} c_a$*, and its value remains unchanged during the computation process.*

Proof. Follows directly from the definition of the transition function. □

Observation If the sum S is negative, it declares majority of reds (denoted by -1), positive S indicates majority of greens (denoted by 1), otherwise S refers to the tie.

Lemma 3.2 (Invariant 2). *The value of the sum* $R = \sum_{a \in A} |w(s_a)|$ *decreases monotonically throughout the communication process and it stabilises eventually on the value* $R_{\text{fin}} = |S|$.

Proof. At any stage of the algorithm R represents the number of strong states $[-1]$ and $[1]$ still present in the population. According to the transition function the number of such states can only decrease when two states $[1]$ and $[-1]$ annihilate one another during a direct interaction. Thus, eventually the sum R stabilises on the original difference between the number of strong states $|S|$. □

We conclude this section with a theorem.

Theorem 3.3. *The population protocol presented in this section computes majority and returns equality if neither of the colours dominates the other.*

Proof. According to the observation and the two lemmas, if a majority exists, the remaining entities in strong states of the dominating colour will recolour all entities accordingly. Otherwise, the annihilation of the last pair of states $([1], [-1])$ results in obtaining two entities with states $[0]$ which in due course will change states in all other entities to $\langle 0 \rangle$. Finally, if neither of the states $[1]$ or $[-1]$ is initially present in the population all entities remain in the neutral state $[0]$. □

3.3 Population protocol for dynamic majority with equality

In this section we consider a variant of population protocols in which the original colours (attributes) of entities could be altered by an *external force* for some unspecified, however limited, period of time. After this initial period, the relevant population protocol is expected to eventually stabilize. The model of changing inputs from [20] and the concept of composing several population protocols as described in [91] are the inspirations for our approach here. In essence, we reformulate the majority algorithm from [20] and show how to modify this reformulation so that it can be used as a subprotocol for our next section.

We assume that an entity is aware when its original colour changes, and is able to modify its current state as a result, but such a change is again governed by common state transition rules for all entities. We also assume that it is not possible for the external force to alter the original colour of an entity while it is simultaneously interacting with another entity.

We use the protocol we propose here as a subroutine in more structurally complex population protocols for the absolute majority in Section 3.4, and for the relative majority in Section 3.6.

The population protocol presented below determines whether there are more original 1's, more (-1)'s, or there is a tie after the last intervention of the external force. For the purpose of our protocol each entity $a \in A$ must store its original colour $c_a \in \{-1, 0, 1\}$, and this stored colour can be altered only by the external force at any time. Besides the colour, the entity maintains a state s_a governed by the shared transition function. More formally, an entity's knowledge refers to the pair (c_a, s_a). We define five strong states:$[-2], [-1], [0], [1], [2]$, and three weak states $\langle -1\rangle, \langle 0\rangle, \langle 1\rangle$. Before the protocol is initiated, if $c_a = x$ we set $s_a = [x]$. On the conclusion all entities are in state

- $[1], [2]$ or $\langle 1\rangle$ if there are more 1's than (-1)'s,
- $[-1], [-2]$ or $\langle -1\rangle$ if there are less 1's than (-1)'s, and
- $[0]$ or $\langle 0\rangle$ when there is a tie.

We define the weight function, $w(s)$, on a state s as $w([x]) = x$ and $w(\langle x\rangle) = 0$, see the table in Fig. 3.3.

s	$w(s)$
$[-2]$	-2
$[-1]$	-1
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	0
$[1]$	1
$[2]$	2

s_a, $c_a = 1$ changes to $c'_a = -1$	s'_a
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	$[-2]$
$[1]$	$[-1]$
$[2]$	$[0]$
s_a, $c_a = -1$ changes to $c'_a = 1$	s'_a
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	$[2]$
$[-1]$	$[1]$
$[-2]$	$[0]$

FIGURE 3.3: The weight function $w(s)$ and the state transition rules when recolouring occurs by an external force.

During execution of the majority protocol we maintain two invariants:

1. $\sum_{a\in A} c_a = \sum_{a\in A} w(s_a)$, and
2. for each $a \in A$, $|w(s_a) - c_a| \leq 1$.

The two invariants are preserved thanks to carefully crafted state transition rules and counterparting alterations of an entity's state caused by changes of the original colour c_a imposed by the external force. When the colour c_a is changed to $c'_a = c_a + \delta$, the state is changed from s_a to $s'_a = [w(s_a) + \delta]$. Note that this rule preserves both invariants 1 and 2. This is illustrated by the table to the right in Fig. 3.3 describing how states are changed when $c_a = 1$ is changed to $c'_a = -1$, or vice-versa. In this table we do not consider, for example, combinations of states $s_a = [-1], [-2]$ with colour $c_a = 1$ because of the invariant 2.

In what follows we describe what happens to the states when two entities $a, b \in A$ interact. If a strong state $[1]$ or $[2]$ meets a weak state $\langle y\rangle$ or $[0]$, then this second state becomes $\langle 1\rangle$. If a strong state $[-1]$ or $[-2]$ meets a weak state $\langle y\rangle$ or $[0]$, then the latter

state becomes $\langle -1 \rangle$. If a strong state [0] meets a weak state, the weak state is changed to $\langle 0 \rangle$. If [1] meets [−1] or [2] meets [−2], they are both changed to [0]. If [2] meets [−1], they are changed to [1] and [0] respectively. If [−2] meets [1], they are changed to [−1] and [0] respectively. Other encounters do not result in state alteration. This is illustrated by the table in Fig. 3.4 which does not take into account encounters between entities where both are in weak states, because they do not result in state alteration.

$s_a \backslash s_b$	$[-2]$	$[-1]$	$[0]$	$[1]$	$[2]$
$[-2]$	$([-2],[-2])$	$([-2],[-1])$	$([-2],\langle -1 \rangle)$	$([-1],\langle -1 \rangle)$	$([0],[0])$
$[-1]$	$([-1],[-2])$	$([-1],[-1])$	$([-1],\langle -1 \rangle)$	$([0],[0])$	$(\langle 1 \rangle,[1])$
$[0]$	$(\langle -1 \rangle,[-2])$	$(\langle -1 \rangle,[-1])$	$([0],[0])$	$(\langle 1 \rangle,[1])$	$(\langle 1 \rangle,[2])$
$[1]$	$(\langle -1 \rangle,[-1])$	$([0],[0])$	$(\langle 1 \rangle,[1])$	$([1],[1])$	$([1],[2])$
$[2]$	$([0],[0])$	$([1],\langle 1 \rangle)$	$([2],\langle 1 \rangle)$	$([2],[1])$	$([2],[2])$
weak	$(\langle -1 \rangle,[-2])$	$(\langle -1 \rangle,[-1])$	$(\langle 0 \rangle,[0])$	$(\langle 1 \rangle,[1])$	$(\langle 1 \rangle,[2])$

FIGURE 3.4: The state transition table for interacting entities for dynamic majority.

Lemma 3.4. *The invariants 1 and 2 are preserved during execution of the majority protocol.*

Proof. First, we consider interactions between pairs of entities.

Invariant 1 is preserved, because for any state transition, if the weight of one entity is reduced, then the weight of the other is increased by the same (absolute) value. Also, if colour c_a is changed, then the weight $w(s_a)$ is changed too by the same value.

Invariant 2 is preserved because during every interaction of entities $|w(s_a)|$ can only decrease and $w(s_a)$ does not change its sign. So if $c_a = 1$, then s_a is initially in the interval $[0, 2]$ and it remains in this interval. The reasoning in the remaining cases when $c_a = 0$ or -1 is analogous.

Now we consider the invariants when the external force changes the colour of an entity. Suppose that an entity is coloured $c_a = 1$ and its colour is changed to $c'_a = -1$ (the other case will be similar).

Invariant 1 is preserved by the choice of the transitions shown in the table in the right of Fig. 3.3. The left hand side of the equation in invariant 1 decreases by 2 (since the colour changes from 1 to -1). If the state of the entity was $s_a \in \{[0], \langle -1 \rangle, \langle 0 \rangle, \langle 1 \rangle\}$, the new state is $s'_a = [w(s_a) - 2] = [-2]$. Hence the corresponding weight changes from $w(s_a) = 0$ to $w(s'_a) = -2$, so the right hand side of invariant 1 also decreases by 2 (i.e., preserving the invariant). Similarly, if $s_a = [1]$, then the new state is $s'_a = [-1]$, hence the contribution to the right hand side of invariant 1 from the entity changes from $w(s_a) = 1$ to $w(s'_a) = -1$, again a decrease by 2. We can check the remaining case, where $s_a = [2]$, in an analogous manner.

Invariant 2 is also maintained by the rules that govern how the entity's state is updated when its colour is changed by an external force. E.g., $c_a = 1$ changing to $c'_a = -1$ means that the new weight $w(s'_a) \in \{0, -1, -2\}$ from the rules in Fig. 3.3. □

Lemma 3.5. *The value of* $R = \sum_a |w(s_a)|$ *does not increase after the last intervention of the external force. Moreover the value of* R *stabilises when eventually there are no two entities* $a, b \in A$ *such that* $w(s_a) > 0$ *and* $w(s_b) < 0$.

Due to Lemma 3.5 the majority process stabilises in three possibile configurations with respect to $C_{\text{fin}} \stackrel{def}{=} \sum_{a \in A} c_a$ (where c_a is referring to the *final* colour of the entity a, after any external forces have stopped changing the colours of entities). If on the conclusion $C_{\text{fin}} > 0$, there must be some entities in states [1] or [2] which would earlier ensure that all weak states and the state [0] are switched to $\langle 1 \rangle$. If $C_{\text{fin}} < 0$, there must be some entities in states $[-1]$ or $[-2]$ which would earlier ensure that all weak states and the state [0] are switched to $\langle -1 \rangle$. However, if on the conclusion $C_{\text{fin}} = 0$, there are no entities in states $[x]$ with $x \neq 0$ and the last entity that reached state [0] will have a chance to alter all weak states to $\langle 0 \rangle$.

3.4 Absolute majority

The work in the remaining part of the chapter works under the assumption that the population is coloured with an arbitrary number C of colours, where $2^{k-1} < C \leq 2^k$, for some integer $k \geq 1$ that is known to all entities. Each colour is denoted by a k-bit label $l[0..k-1]$, and single labels are attributed to entities with the relevant colours. As in previous sections, we interpret the individual bits $l[i]$ in this label as -1 or 1, rather than more standard 0 or 1. Each entity is assumed to own an extra $O(k)$ bits used to support the computation process, including interaction with other entities in the population.

In this section we present an asymptotically optimal $O(k)$-bit population protocol computing *absolute majority*, i.e., answering whether there exists a colour which dominates all the remaining colours in the population taken together. The absolute majority algorithm presented here is a combination of the static majority protocol introduced in Section 3.2, and later referred to as P_1, as well as the dynamic majority protocol from Section 3.3, from now on referred to as P_2. We recall that protocol P_2 assumes full knowledge of entities and it is using two types of state transitions: (1) imposed by the *external force* and altering original colours associated with entities, and (2) caused by the interaction with other entities in the population.

Memory organisation Each entity uses $O(k)$ bits of memory to accommodate:

1. The k-bit label $l[0..k-1]$ representing the original colour of the entity,
2. An array $s[0..k-1]$ representing k independent instances of protocol P_1, and
3. An instance of protocol P_2 with the *external force* based on k instances of P_1.

For the purpose of our algorithm we define k independent instances of static majority protocols $P_1(i)$, for $i = 0, \ldots, k-1$, such that colours competing in $P_1(i)$ refer to the bits $l[i]$ drawn from each entity in the population. Assume $l^*[0..k-1]$ is a k-bit label of

the colour of the absolute majority in the population. One can observe that when the majority protocols stabilise, for all $i = 0, \ldots, k-1$, each bit $l^*[i]$ must be in majority reported by $P_1(i)$ via entry $s[i]$. Thus, if the absolute majority exists, one can run k static majority protocols $P_1(i)$ to determine the majority colour. However, if there is no absolute majority the protocol proposed above may still return a false positive "winner". This can happen, e.g., if no entity has a colour with the label in which all bits are set to 1s but the majority of bits $l[i]$, for all $i = 0, .., k-1$ for all entities are 1's. In such case, the non-existing colour with the label filled with 1s would be wrongly recognised by the entities as the absolute majority. In order to overcome this clear deficiency of the protocol, an extra (final) test is performed with the help of protocol P_2 to decide whether the returned colour is in the absolute majority.

3.4.1 Algorithm Absolute-Majority

Initialisation Stage

1. Before execution of the algorithm, each entity $a \in A$ sets for itself $s[i] = [1]$ if $l[i] = 1$ and $[-1]$ otherwise, for all $i = 0, .., k-1$. This choice refers to the belief that its original colour c_a is in majority. And, indeed, each entity initially adopts an extra colour 1 (denoting membership in the majority) for the purpose of protocol P_2.

2. Later, during pairwise interactions between entities, the current states in $s[i]$ get updated by the relevant majority protocols $P_1(i)$, for each $i = 0, \ldots, k-1$ independently. And if at any time the contents of $s[i]$ and $l[i]$ do not reflect its initial setting, the belief of the entity changes to -1. However, this belief becomes 1 again as soon as the consistency between bits in $s[0..k-1]$ and $l[0..k-1]$ is restored. This consistency measure determines actions of the *external force* in protocol P_2.

Stabilisation Stage

1. At first, the majority algorithm stabilises on all protocols $P_1(i)$, for $i = 0, \ldots, k-1$, which allows each entity to establish the final relationship between the corresponding bits in $s[0..k-1]$ and $l[0..k-1]$. This, in turn, determines the extra colour (1 or -1) of the entity adopted for the purpose of protocol P_2.

2. When eventually protocol P_2 also terminates and concludes with colour 1 in majority, all entities receive confirmation that the final states in $s[0..k-1]$ refer to the absolute majority colour $l^*[0..k-1]$. Otherwise, the entities learn that none of the colours is in the absolute majority.

Note that all protocols described above run simultaneously right from the beginning, and, in particular, protocol P_2 works at least for some time on unstable data. Nevertheless, as the bits generated by protocols P_1 eventually stabilise, thanks to protocol P_2's tolerance of dynamic changes, the absolute majority (if such exists) is confirmed. We conclude with this theorem.

Theorem 3.6. *Algorithm Absolute-Majority computes absolute majority on populations with at most 2^k colours with the help of $O(k)$ memory bits in each entity.*

Proof. If an absolute majority colour exists (represented as a k-bit label $l[0..k-1]$) then, when the k independent instances of P stabilize, each $P_1(i)$ stabilizes in the bit $l(i)$. In fact, each bit of the label of the colour of the absolute majority is then reported by $P_1(i)$ via its entry $s[i]$. However, the population still needs to verify this since, in case of no absolute majority colour, the above protocol may return a false positive "winner" . This can happen if for each i there is an absolute majority bit but the whole tuple of these bits does not correspond to a colour in the population. In order for this case not to be wrongly understood as the absolute majority , we need a verifying step. This is exactly what protocol P_2 does. In fact, P_2 always runs a test to decide whether the returned supposed absolute majority colour is indeed the absolute majority. Protocol P_2 works for some time on unstable data. However, after a time t by which all $P_1(i)$ have stabiized, protocol P_2 shall also stabilize either by concluding that the assumed majority colour (indicated by colour 1 in the algorithm) is indeed an absolute majority, or it shall stabilize reporting nonexistence of the absolute majority colour to all entities. Note that each time P_2 has to check only one supposed majority colour against all others, treated as a single colour -1 in the algorithm. The above proof works due to the established fact that P_2 tolerates dynamic changes in the input colours. □

3.5 Absolute Majority Example

Here we provide an example of the absolute majority protocol, referencing the memory allocation stated in Section 3.4. Recall we defined the memory allocation for each entity A in the graph, a k-bit label l, an array s storing each instance of protocol P_1 and a bit which stores the instance of P_2 that reports if the initial value of l in A is in the majority color. We also use one final instance of P_2 to verify whether the initial majority color has been found.

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} l \\ \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} s \\ \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} P_2 \\ \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \end{array} \begin{array}{c} P_2 \\ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{array}$$

FIGURE 3.5: Absolute Majority Example

The example in Figure 3.5 illustrates this memory allocation where there exists an initial majority color in entities A_1, A_3 and A_4 of 1011. Each index in array s stores the protocol P_1 for each column in l. For example, consider the column l_1 and s_1, there exists two colors 1 and 0, the only 0 appearing from entity A_2. As a result of this, the

value of each entity for element $s_1 = 1$ as 1 is the majority color. For the column at index l_2, s_2 stores the initial majority color in this instance which is 0. This is repeated for all corresponding indexes in l and s until the correct result has been found. Each entity stores an instance of P_2 which reports if the entities majority color l is equal to the determined majority color s. As $s_{A_1} = l_{A_1}$, $P_{2_{A_1}} = 1$. Conversely, as $s_{A_2} \neq l_{A_2}$, $P_{2_{A_2}} = 0$. The final bit, is one final instance of P_2 on the previous instance of P_2 which determines the majority color in this instance. As 1 is the majority color and is reported in all rows, the protocol has verified the initial majority color has been found.

3.6 Relative majority

As in Section 3.4, in this section we assume that the population is attributed with an arbitrary number C of colours, where $2^{k-1} < C \leq 2^k$, for some integer $k \geq 1$ that is known to all entities. Each colour is denoted by a k-bit label $l[0..k-1]$, where $l[i] \in \{-1, 1\}$. Each entity is assumed to have extra $O(k)$ bits used to support the computation process, including communication with other entities in the population. The *relative majority* problem refers to the task of finding the most frequent colour in the population. Note that there can be more than one colour that is the most frequent. In such case the colour with the latest in the lexicographical order label $l^*[0..k-1]$ is declared as the *winner*.

Computing relative majority is a more complex task, comparing to the absolute majority, as here one needs to collect evidence confirming that the winning colour beats any other colour in the population. At first we describe a protocol for the relative majority which only finds the winner $l^*[0..k-1]$. This is done by marking all entities possessing this colour with the winning label. In this setting, the colour in the relative majority always exists. The case in which the uniqueness of the majority colour is required is commented later in Section 3.6.2.

In the relative majority protocol, instead of engaging in the total comparison (via majority computation) in pairs formed of any two colours, which would require $O(k^2)$-bit memories, we propose a solution similar to finding maximal elements in parallel stages based on duels. In each stage the winning colours perform pairwise duels via majority protocols to reduce the number of winners by half.

This multi-stage computation is made feasible thanks to pipelining of dynamic majority protocols P_2 which gradually stabilise starting from the lowest stage and finishing at the highest stage of the dueling process. The diagram in Figure 3.6 demonstrates the dueling process between C colors.

Stages are enumerated by descending numbers from the lowest stage $k-1$ to the highest 0. In stage i, for all $i = k-1, .., 0$, two colours are in the same group if their k-bit labels $l[0..k-1]$ share i-bit prefix $l[0..i-1]$ (in stage 0 all labels form one group). In this stage agents in one group aim at finding the majority colour label in each group.

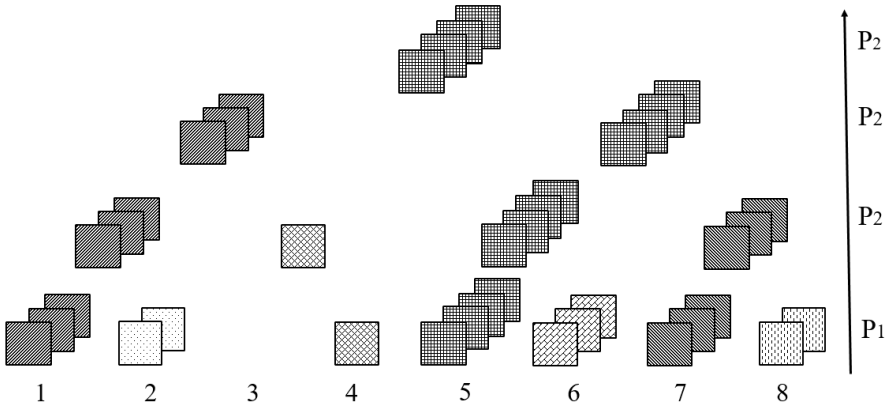


FIGURE 3.6: Relative Majority Duels

Memory organisation. Each entity $a \in A$ uses $O(k)$ bits of memory to accommodate:

1. The k-bit label $l[0..k-1]$ representing the original colour of the entity. This colour is fixed (never changed) throughout the computation process.

2. The k-bit label $c[0..k-1]$ represents current colours $c[i]$ of the entity in each consecutive stage i, with the decreasing index $i = k-1, .., 0$. On the conclusion of stage i, if label $l[0..k-1]$ is declared as the winner in the group of labels with prefix $l[0..i]$, the value $c[i]$ equals to ± 1, otherwise $c[i] = 0$. All entities with the winning colour $l[0..k-1]$ in its group in higher stage $i-1$ have the value $c[i]$ set to $l[i]$. Before the stabilisation of $P_2(i-1)$ the value $c[i]$ reflects the current belief of the entity about this value.

3. An array $s[0..k-1]$ representing states $s[i]$ in k independent instances of protocol $P_2(i)$ associated with colours $c[i]$. The computations with respect to $P_2(i)$ are performed only if the two interacting entities have the same label prefix $l[0..i-1]$. Otherwise protocol $P_2(i)$ is not executed. We emphasise here that computations in $P_2(i)$ can change values $c[i-1]$ whose change in turn cause alteration of states $s[i-1]$. Also, changes in $c[i]$ can change $c[i-1]$.

3.6.1 Algorithm Relative-Majority

Initialisation Stage Before execution of the algorithm, each entity sets $c[i] = l[i]$ and $s[i] = [1]$ if $c[i] = 1$ and $[-1]$ otherwise, for all $i = 0, .., k-1$.

Stabilisation Stage

1. The algorithm stabilises first on protocol $P_1(k-1)$, as at the beginning of the pipeline there is no external force, and then subsequently on protocols $P_2(k-2)$, $P_2(k-3)$, ..., $P_2(0)$.

2. An entity believes that its colour wins on stage i if, either $c[i] = -1$ and $s[i] \in \{[-1], [-2], \langle -1 \rangle\}$, or $c[i] = 1$ and $s[i] \in \{[0], [1], [2], \langle 0 \rangle, \langle 1 \rangle\}$. The states $[0], \langle 0 \rangle$

correspond to a tie and in this case the lexicographically larger label becomes the winner. If the entity believes its label $l[0..k-1]$ is the winner in stage i, it sets $c[i-1] = l[i-1]$ and adjusts $s[i-1]$ as specified in protocol $P_2(i-1)$ if $c[i-1]$ gets changed. If, to the contrary, the entity believes it did not win, it sets $c[i-1] = 0$ and also adjusts $s[i-1]$ should change occur in $c[i-1]$. Note, that in both cases changes in $c[i-1]$ are propagated to $c[i-2]$ and further on.

3. Eventually protocol $P_2(0)$ stabilizes. At that time entities that win in stage 0 hold the winning majority colour.

Theorem 3.7. *Algorithm Relative-Majority computes relative majority on population with at most 2^k colours with the help of $O(k)$ memory bits in each entity.*

Proof. The memory requirement follows directly from the formulation of the protocol. In order to prove correctness, we proceed by induction on stage numbers i taken in reverse order. The colours $c[k-1]$ do not change during the protocol so in some moment t_{k-1} protocols $P_2(k-1)$ stabilize and states $s[k-1]$ stop being changed. These states determine unique winning k-bit colours in groups corresponding to all possible prefixes $l[0..k-2]$.

Now let $i > k-1$ be a stage number. By inductive hypothesis in some time t_{i+1}, protocols $P_2(i+1)$ stabilize and states $s[i+1]$ stop being changed. They indicate unique winning k-bit colours in groups corresponding to each prefix $l[0..i]$. So, since t_{i+1} colours $c[i]$ are ± 1 for these winners, 0 for others and do not change anymore. Thus, in some later time t_i, protocols $P_2(i)$ stabilize and states $s[i]$ cease being changed. From the formulation of the protocol these final states $s[i]$ determine the winning k-bit colours in groups corresponding to prefixes $l[0..i-1]$.

Finally ,at some time t_0, protocols $P_2(0)$ stabilize and all entities compute states $s[0]$ corresponding to the unique winning k-bit colour amongst all of them. □

3.6.2 Uniqueness in relative majority

As indicated at the beginning of Section 3.6, one may want to report only unique relative majority colours, i.e., when there is exactly one, the most frequent colour. And indeed if the winning colour l^* is not unique, there must exist some other colours which lost to l^* in a tie at some stage. The purpose of the mechanism presented below is to encounter such ties (if they exist) and to distribute this information to all entities in the population. This can be done by performing an additional *dissemination protocol* with the help of an extra bit c' drawn from the set $\{0, 1\}$. This dissemination protocol is run by each entity in conjunction with the relative majority protocol described above, and its actions are governed by the current belief of the entity whether it is a winner or not and by encountered or not ties in duels. The following four rules govern values of the extra bit c'.

Initially, (1) in each entity the extra bit c' is set to 0 to denote that the entity does not carry any information about ties between the winners. This value can be changed

to 1 if (2) the colour of the entity is still a potential winner (did not lose any duel yet in the most recent climb through the stages) and at some stage its duel ends up in a tie; or if (3) the colour of the entity is already deemed as the loser and it meets another entity with the colour still being a potential winner and its extra bit $c' = 1$. And (4) the extra bit c' can be changed back to 0 if and only if the colour of its owner is deemed as loser and it meets another entity with the colour still being a potential winner and its extra bit $c' = 0$.

In due course the values of each extra bits c' can be altered several times according to the rules 1, 2 or 3. However, when eventually the relative majority protocol determines the winning colour l^* in stage 0, only entities coloured with l^* are able to change values of extra bits in other entities. Now, if the extra bit associated with entities coloured by l^* is 0, i.e., the winning colour has never experienced a tie, all other entities are eventually informed accordingly by rule 4. And, if the extra bit associated with entities coloured by l^* is 1, i.e., the winning colour has encountered a tie in the past, all other entities are eventually informed accordingly by rule 3.

3.7 Conclusion

The work in this chapter presented memory-efficient population protocols for several variants of the majority problem.

In Section 3.2 we show how to amend majority protocols to report ties. The proposed protocol relies on a relatively large number of states used by entities. One can show a more space-efficient solution limited to six states. Also in a wider context, in our solutions the emphasis was on asymptotic space optimality. One open problem, however, is to determine more exact bounds on the number of states required to compute the considered types of majorities for a given number of colours C. Another interesting problem refers to the time complexity and parallelism of considered majority problems in the presence of a random scheduler. Finally, one can ask what other computations are possible through a composition of several "partially self-stabilizing" (sub)protools.

Chapter 4

Agglomerative Phylogenetic Clustering in Structured Graphs

4.1 Overview

Heuristically nonuniform data appears structureless on initial inspection. The process of identifying the heterogeneity in the data and grouping elements is the process of clustering. Clustering has been a focal point in many multidisciplinary academic fields for decades, a problem which still hasn't been satisfactorily solved. The salient contributing factors are that there are a wide variety of applications for clustering across these disciplines and the data has a wide variety of formats, such as text, multimedia, web pages, biological, sociological amongst many others. The applications, domains and data types are diverse and the solutions are difficult to generalize. This has resulted in the creation of many constrained and problem dependent algorithms. Another contributing factor is that the research is often fragmented, studied in various academic disciplines and further subdivided into subdivisions of each discipline. For example, in computer science, clustering is studied in machine learning, databases and data-mining. There is little effort to address this research area in a interdisciplinary and unified way [9].

Clustering in the context of this thesis is grounded in machine learning, as unsupervised learning algorithms. Contrary to supervised learning, datasets generally are not accompanied by truth labels. This creates obstacles when applying evaluation methods to clustered data and adds a layer of subjectivity in regards to the 'best' clustering for any given scenario.

Given the broad nature and setting of clustering, many models exist to solve the problem. The area we will focus on in this chapter is distance-based algorithms. These methods are very popular due to their generality and applicability to data types of varying degrees - providing a correct distance metric is used [9].

In this section, we propose an algorithm inspired by hierarchical clustering, more specifically the agglomerative variant in which each data-point is a cluster initially -

these clusters are successively merged using one of three common strategies, ward, complete or average linkage schemes (the similarity values calculated post merging of two clusters). The method is also inspired by phylogenetics, a branch of biology that studies evolutionary relationships between biological entities - a notable example being the phylogenetic tree of life. In recent years there has been an exponential increase of sequence data being produced from various sources and the study of phylogenetics and the building of phylogenetic trees has become an integral part in biological studies. The work in [85] discusses and provides new tools to represent and manipulate phylogenetic trees visually. Hierarchical clustering and phylogenetics are are inherently similar but separate - we have synergised key concepts from these areas with the aim to produce a new algorithm to aid in identifying clusters in various datasets.

4.2 Our Results

The solution we propose is based on agglomerative hierarchical clustering and phylogenetic or evolutionary trees. A salient feature of the algorithm is that it focuses on local sampling to assist in determining overall network structure. As a result of such, many of the initial redundant values from all pairwise computations in a network are dropped, which is a fundamental step in classic hierarchical clustering mechanisms. In our algorithm we consider pairwise and ternary relationships in the data, up to one step away from the root node. From the information inherent to the local environments we create a set of triplets, which are used as constraints in our method. The algorithm will then catalogue this information and interpret the triplets with the goal to derive good clustering. We present in this chapter multiple variations of the algorithm with accompanying analysis, beginning at the most incubative form in terms of strictly adhering to the rules of phylogenetics to more complex forms that allow for a more flexible rule set. We introduce new parameters to alleviate the problems inherent in a simpler approach and ultimately provide an augmentable algorithm that is robust and scalable across various datasets.

More specifically, considering an undirected graph G, we want to generate a set T of constraints from the local structure of the vertices in G. Using the constraints in set T we construct a set of clusters C in a similar fashion to how evolutionary trees are built in phylogenetics. Note we use a set initially to adhere to phylogenetic building rules until we introduce the notion of ranking. Henceforth we adapt to using lists to storing constraints. We consider a model of building phylogenetic data structures in [82] as well as models with restrictions, such as the inclusion of resolved, forbidden and fan triplets. The order of the constraints in T is initially unimportant in early experiments when considering strict phylogenetic building rules. Our final algorithm terminates when a user specified k is given (where k is the number of desired clusters), but we also consider autonomous termination when $T = \emptyset$ and thresholding mechanisms. We show the algorithm in its

basic form along with with successive augmentations to improve robustness in a wider array of graph types. Each iteration of the algorithm is analyzed on various graph types against previous iterations of the Agglomerative Phylogenetic Clustering (APC) algorithm to showcase the significance of the design decisions. The final iteration of the algorithm is analyzed on various graph types against similar algorithms in its domain, specifically Girvan-Newman [71] and Label Propagation [106]. The algorithm is evaluated and discussed in terms of suitable applications of such a mechanism, where it performs strongly and cases in which it may encounter difficulty.

4.3 Test Data

The beginning sections of this chapter (Section 4.10, 4.13, 4.16 and 4.18) show progressive iterations of our algorithm, we generate small examples to demonstrate the fundamental concepts of the solution and also identify problem cases that initiate adaptations of such. These small datasets are fundamentally small barbell graphs and will be defined and discussed individually in each section of the relevant experiment.

To test the quality of clusters, we must apply the algorithm to a problem in which a well defined solution is known, a graph where well connected communities are clear - typically called partitions. All clustering procedures share similar notions of what constitutes as a cluster, though each algorithm tackles the problem differently. Therefore, we use a set of synthetic graphs or computer-generated benchmark graphs that have become popular in the field over the last couple of years [63]. The class of graphs are generated using the planted l-partition model, which partitions a graph with $n = g \cdot l$ vertices in l groups with g vertices each. Vertices of the same group are linked with a probability p_{in} and conversely, vertices in a different group with probability p_{out}. Each subgroup in these graphs are then random graphs, specifically mutually interconnected random graphs as in Erdos-Renyi, where the connection probability is $p = p_{in}$. These graphs contain clusters of the exact same size by design, which is rarely seen in real systems. A modified model, Gaussian random partition generator accounts for the heterogeneity of degrees and community sizes of the planted l-partition model, which will generate clusters of different sizes. We finally consider Random Partition Model generators that are flexible in allowing the ability to define specific cluster sizes.

We use the library NetworkX [2] which implements these various network topologies, including random graphs using planted l-partition models [5], Gaussian random partitions [3] and Random Partition Graphs [6]. We use multiple synthetic graphs in this form by manipulating the parameters used to create it, including a special case of the planted l-partition model defined by Girvin and Newman and referenced in [63].

We adopted the scikit-learn machine learning library [7] for testing the algorithms. The scikit-learn library contains datasets for various machine learning problems, including clustering. The library also contains various evaluation metrics discussed in the Section 4.4. These synthetic datasets are created by various functions plotting samples

in two-dimensional space and are initially plotted without edges - therefore they are not naturally suitable for our graph based method. All the experiments are using the same random seed for these synthetic datasets to ensure consistency and continuity between experiments and between methods. Further motivation for selecting graph generators are specified when used in later experiments.

4.4 Evaluation Metrics

It is necessary when an algorithm is designed, to test the performance, assess the quality and compare it with that of other methods. To determine the performance of our algorithm we must first use a set of standard benchmark graphs and relevant criterions to evaluate how similar the returned cluster labellings are with that of the desired cluster labelling. Melia provides a comprehensive introduction to graph partition similarity measures in [88], a paper which also defines the advantages and disadvantages of common evaluation metrics. The metrics discussed are used to determine the quality of partitions and the quality of the overall clustering and can be divided into three categories: pair counting, cluster matching and information theory.

In this chapter we document experiments on various iterations of the APC algorithm, using various benchmark graphs discussed in Section 4.3. The motivation for using such graphs is that the desired partitions or solution, is already known and the truth labels or ground truths have been derived *a priori*.

The set of evaluation metrics we consider are adjusted rand index, completeness score, homogeneity score and V-measure score.

Definition 4.1. Adjusted Rand Index [77]: a pair counting mechanism that determines the similarity between two clusterings - the use of this metric in the context of classification is discussed in [111]. Specifically and firstly it is an augmentation to the Rand Index [107] which is the ratio of the number of pairs correctly classified in both partitions by the total number of pairs - the correctly classified pairs can either be in the same or different clusters. The resulting value is returned in the range $[0, 1]$, although in practice the range is more often $[0.5, 1]$ and therefore the adjusted variant is preferably used [116]. It is adjusted to allow for the introduction of a null model, and compensating for the shortcomings of RI by introducing a contingency table. ARI yields the value 1 when partitions are identical and 0 when partitions are independent.

Definition 4.2. Completeness score, a complementary concept to V-measure that captures desirable properties in clustering tasks [109]. Completeness is satisfied if all data points that are members of a given class are elements of the same cluster. An algorithm which merges all partitions into one cluster can satisfy this property, but used alone does not indicate good clustering.

Definition 4.3. Homogeneity score, a complementary concept to V-measure that captures desirable properties in clustering tasks [109]. Homogeneity is satisfied if all clusters

contain only data points which are members of a single class. An adaptation to completeness, all partitions must be separate and must contain all their elements to achieve the maximum value.

Definition 4.4. V-measure score, an entropy-based measure which explicitly measures how successfully the criteria of homogeneity and completeness have been satisfied. [109]. This method is considered widely over other methods such as *Purity* and *Entrophy* proposed in [118] as they only consider the concept of homogeneity and not completeness. The computation of completeness, homogeneity and V-measure are completely independent of the number of classes, the size of the data set and the clustering algorithm used. A characteristic not shared by other existing evaluation metrics.

4.5 Our Proposed Method: Agglomerative Phylogenetic Clustering and Parameters

In this section we explain our proposed algorithm in its basic form along with augmentations and variants that resemble the final result. Each iteration of the algorithm is grounded by a common concept in that they all consist of three unique steps, defined in *Algorithm 1.*

Data: Graph G, Integer k
Result: Output k clusters
initialization;
Function *calculate_ similarities*
Function *generate_ constraints()*
Function *build_ clusters()*

Algorithm 1: APC Algorithm Framework

The APC algorithm will compute all nested pairwise similarities between $\forall v \in V$ and neighbors of depth 1 from v. The similarity metric we consider is cosine similarity, defined in Equation 4.1, which is commonly used in hierarchical clustering and other connectivity based clustering methods.

$$cos(a,b) = \frac{common\_neighbours(a,b)}{\sqrt{deg(a) \times deg(b)}} \tag{4.1}$$

Conversely hierarchical clustering computes all pairwise similarities between all vertices in G. In our method, calculating the similarities considers each node in direct comparison with their neighbors, utilizing the set of common neighbors between them and ultimately labelling each pairwise relationship $0 \leq sim \leq 1$. Therefore our method only considers local sampling in the graph as the *generate_ constraints()* phase in *Algorithm 1* will only generate constraints according to the functions defined in Section 4.6 and later

in Section 4.15. These functions only require similarity values for relationships of at most depth 2 from the root vertex. The similarities are used to generate ternary triplets or constraints in the form $((ab), c)$. The constraints are used in the *build_ clusters()* phase in Algorithm 1 and is formatted such that it implies a and b are more similar than c - a notion explained in more detail in Section 4.7.

Parameter	Experiments	Extended Experiments
Termination	All Constraints, k	Threshold
Distance Measure	Cosine	
Constraint Function	PCG, TCG	
Allow Forbidden	True, False	
Allow Fans	True, False	
Rank Constraints	True, False	

TABLE 4.1: APC Parameters

Our proposed method can be instantiated in multiple settings using the complete list of parameters in Table 4.1. The table shows which parameters are discussed in the experiments present in this chapter and also features that are considered in extended experiments and future work. We consider two termination criterions, all constraints and k, where k is the number of desired clusters. We also consider two methods of generating constraints based on local relationships, these are *pairwise constraint generation (PCG)* and *ternary constraint generation (TCG)*. The parameters are discussed and introduced in detail when necessary to experiments. Firstly we introduce the parameters necessary for the basic version of APC, requiring only a constraint generation function, PCG, and termination criterion, all constraints.

4.6 Pairwise Constraint Generation

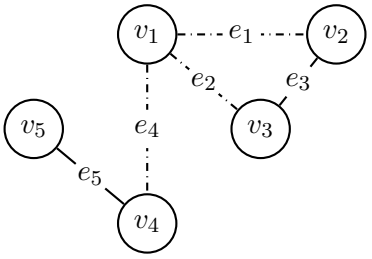


FIGURE 4.1: PCG Local Relationships

Our first constraint generation function is *Pairwise Constraint Generation*, *PCG* henceforth. This is the concept of considering root vertices directly with their neighbors and creating constraints accordingly. This was implemented by a dynamic programming mechanism which enables the computation of constraints with the previously generated similarities. The following example is in reference to Figure 4.1. We firstly compute all pairwise similarities in the local neighbourhood and set the edges labels to the corresponding similarity between the source and target vertex. Firstly, we select v_1 as the root node and consider each relationship iteratively. The first relationship we consider is (v_1, v_2) and as there is no other data to compare to, no constraint is generated. Although, we store the pair (v_1, v_2) as the highest similarity pair in the neigbourhood of v_1. Secondly, we consider the pair (v_1, v_3) which can then be compared to the similarity of the previously stored pair. If the similarity $(v_1, v_3) > (v_1, v_2)$, then the constraint $((v_1, v_3), v_2)$ is generated and added to the list of constraints T, else $((v_1, v_2), v_3)$ is added to T. The last vertex in the neighbourhood of v_1 is then considered by comparing the similarity (v_1, v_4) with all other considered pairs. If $(v_1, v_4) > (v_1, v_3)$ then the constraint $((v_1, v_4), v_3)$ is generated and added to the list of constraints T, else $((v_1, v_3), v_4)$ is added. Finally, if $(v_1, v_4) > (v_1, v_2)$ then the constraint $((v_1, v_4), v_2)$ is generated and added to the list of constraints T, else $((v_1, v_2), v_4)$ is added. This method only considers vertices of depth 1 away from the root vertex, as annotated by the dashed edges. PCG allows for

the generation of forbidden triplets in the list T, explained in more detail in Section 4.11 and includes a different interpretation of fan triplets explained in Section 4.12.

4.7 Building the Phylogenetic Tree

Subsequent to generating a set of constraints, it is then possible to build clustering from these constraints. The rules we apply are based on the building method defined in [10], which determines if a phylogenetic tree can be built from the constraints - defined as the *tree discovery problem*. The paper defines the problem as follows: in a rooted tree, the lowest common ancestor of two vertices x and y, denoted by (x, y), is the vertex a that is an ancestor of both x and y such that no proper descendant of a is also an ancestor of both x and y. The constraints in [10] are defined in the form $(i, j) < (k, l)$ where $i \neq j$ and $k \neq l$, which implies the lowest common ancestor of (i, j) is a proper descendant of the lowest common ancestor of (k, l). Note that a u is a proper descendant of v if u is a descendant and $u \neq v$.

There are two building rules that pertain to the generated constraints in this form to generate a phylogenetic tree. These are as follows:

1. i and j must be in the same set.

2. Either k and l are in different sets or i, j, k and l are in the same set.

The sets which are referred to here are groupings of vertices based on the topology of the graph and the result of executing the constraints. If the constraints are satisfied and the sets are consistent, there exists a phylogenetic tree.

A paper built upon the work in [10] authored by Henzinger et al [75] regards the *subtree consistency problem*, the form of the constraints and building rules were simplified - the form used in this chapter. The new form still finds the lowest common ancestor between two vertices and is now represented as a triple, $((a, b), c)$, indicating the lowest common ancestor of a, b is below that of a, c.

Initially each vertex in the graph is assigned to a cluster, therefore $C = n$ (reduced to k). For each constraint in T, vertices are merged adhering to the above rules. For example, given $T_i = ((a, b), c)$, clusters containing a and b will be merged, if they do not already exist in the same set. The process is repeated until $C = k$, which is a user defined value.

This process will create a phylogenetic or evolutionary tree that is consistent, in which leaves belonging together given the constraints are merged into a subset. This process is similar to the merging process in hierarchical clustering, though pairs are considered in this context, a dendogram is created through the merging of similar vertices. The denodogram is a natural representation of agglomerative and divisive methods of building sets. Along the x axis, each individual vertex is represented as a singular set. The y axis represents the distance between clusters and can be empirically studied to determine what could be considered as natural clusters. As the distance between clusters reduces

as does the quality of the clustering. By analysing the structure of the dendogram, a user defined k value can be empirically discovered.

4.8 Termination - All Constraints

The following experiments in this chapter will use this method, the first termination parameter in Table 4.1. Subsequent to the list of constraints T being generated, the APC algorithm will recursively construct clusters until $T = \emptyset$. Once a constraint $t \in T$ has been used, t is removed from T. This termination method is useful when there are few constraints or when no difficult constraints have been generated, discussed more in succeeding sections. Some post processing may need to be performed then to achieve k if desirable clusters have not been produced or to merge vertices which belong to no clusters. This is also useful experimentally when aiming to prune problematic constraints from T.

4.9 Our Proposed Method: Pseudocode

As the key concepts have been explained, we expand on the framework in Algorithm 1 and provide the pseudocode for the *generate_constraints*() and *build_clustering*() procedures. The pseudocode for these procedures is based on the termination criterion *all constraints* and constraint generation function *PCG* documented in the table of parameters 4.1 and previous sections. The pseudocode for this variant is defined in Algorithm 2. Recall a triplet is synonymous with constraint in this context and stored in the form $((a, b), c)$. We also define here an object of type *triplet*, a triplet contains elements a, b and c, all of which are vertex label. Therefore we check set membership based on the vertex label.

```
Input: Graph G
Output: Set of labels for vertices in G
cosine_matrix ← cosine_similarity(G)
generate_contraints()
generate_clusters()
Function generate_ constraints()
    T = empty
    for root in G.vertices() do
        neighbours ← G.neighbours(root)
        for neighbour in neighbours do
            if root = neighbour then continue
            root_neighbour_sim ← cosine_matrix[root, neighbor]
            for n in neighbours do
                if root = neighbour then continue
                if n = neighbour then continue
                root_n_sim ← cosine_matrix[root, n]
                if root_n_sim > root_neighbour_sim then T.add(new
                  Triplet(max(root, n), min(root, n), neighbour))

                else if root_neighbour_sim > root_n_sim then T.add(new
                  Triplet(max(root, neighbour), min(root, neighbour), n))

            end
        end
    end
    return T
Function generate_ clusters(G, T)
    clusters ← ∅
    for v in G.vertices() do
        //each vertex begins in its own set.
        clusters.add(v)
    end
    for triplet in T do
        for c in clusters do
            if triplet.a in c and triplet.b in c then break

            if triplet.a in c then merge

            if triplet.b in c then merge

        end
    end
```

Algorithm 2: APC(PCG)

4.10 Experiment 1

In this section we evaluate the APC algorithm in its most basic setting, using only a small subset of parameters from Table 4.1. The purpose of this experiment is to understand the core process and determine the viability of such a mechanism independently of more complex parameters. To illustrate this, the algorithm needs only a graph G as input and APC is configured such that constraints are generated using PCG and the algorithm terminates when $T = \emptyset$ (recall termination explanation in Section 4.8). The parameters for this experiment are shown in Table 4.2.

Parameter	Experiments
Termination	All Constraints
Constraint Generation	PCG
Similarity Measure	Cosine

TABLE 4.2: Experiment 1: APC Parameters

Given the current configuration, the APC algorithm functions as specified in Algorithm 2 perform two key stages, generating constraints and building clusters. Initially, all vertices in G are considered to be in their own cluster. Then, $\forall t \in T$ where t is a triplet of the form $t = ((a, b), c)$ and T is the constraint list, we merge clusters containing a and b if they do not belong to the same cluster already. This is repeated for all elements in T.

4.10.1 Data

To demonstrate the core concept we will run the algorithm on small networks with specific topologies. The motivation for this is to ensure the number of constraints generated in T is small enough to generate a manageable $|T|$. This will allow us to empirically analyse the elements in T and highlight the benefits and drawbacks of this APC configuration.

Dataset 1. The first small network we consider is a simple barbell graph consisting of eight vertices, two groups connected by one edge - as shown in Figure 4.2.

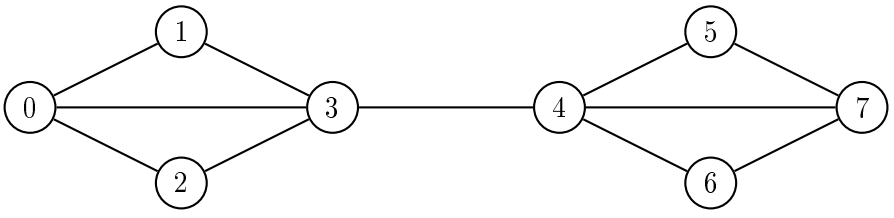


FIGURE 4.2: Barbell

Dataset 2. We also consider a variant on the simple barbell graph which connects the two communities by an additional edge - as shown in Figure 4.3.

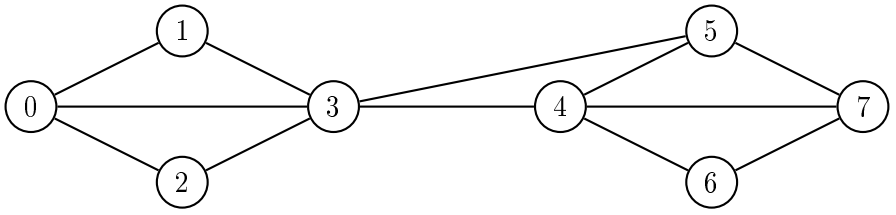


FIGURE 4.3: Barbell with additional edge

4.10.2 Experiment 1a

The first experiment sets the parameter G as the input graph of Dataset 1.

Constraint Generation

Firstly, the algorithm computes the pairwise similarity values $\forall v \in V$ with the neighbours of depth 1 from v, where the similarity measure is cosine. We store the computed similarity values in Table 4.3. Note that the similarities between pairs $(2, 3)$ and $(5, 6)$ are 1 as they share identical neighbours. We do not consider these values when generating constraints as they do not share an edge. Not all values are utilised in the computation of the constraints, but they are shown for completeness.

v_i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	1	0.408	0.408	0.577	0.288	0	0	0
1	0.408	1	1	0.353	0.353	0	0	0
2	0.408	1	1	0.353	0.353	0	0	0
3	0.577	0.353	0.353	1	0	0.353	0.353	0.288
4	0.288	0.353	0.353	0	1	0.353	0.353	0.577
5	0	0	0	0.353	0.353	1	1	0.408
6	0	0	0	0.353	0.353	1	1	0.408
7	0	0	0	0.288	0.577	0.408	0.408	1

TABLE 4.3: Cosine similarity matrix

Given APC Algorithm 2 and the configuration of this experiment, subsequent to calculating the similarity values, we have the information necessary to generate constraints. The constraint list T stores triplets using the PCG function, resulting in T containing the elements shown in Table 4.4.

i	T_i	$Sim(T_i)$
1	((5, 7), 4)	0.408
2	((0, 1), 3)	0.408
3	((0, 3), 2)	0.577
4	((0, 3), 4)	0.577
5	((1, 3), 4)	0.353
6	((2, 3), 4)	0.353
7	((4, 5), 3)	0.353
8	((4, 6), 3)	0.353
9	((4, 7), 3)	0.577
10	((4, 7), 5)	0.577
11	((4, 7), 6)	0.577
12	((6, 7), 4)	0.408
13	((0, 3), 1)	0.577
14	((0, 2), 3)	0.408

TABLE 4.4: Experiment 1a T

The constraints are generated using the similarity values in Table 4.3 and stored in *cosine_matrix* in Algorithm 2.

APC parses all the vertices in the graph in an arbitrary order - therefore, the resulting elements in T are ordered relative to the sequence in which the vertices were parsed during the execution of the algorithm, as well as the underlying implementation of the programming languages data structure.

The first triplet in Table 4.4 $T_0 = ((5, 7), 4)$ was generated by using the *cosine_matrix* and Table 4.3 as $sim(5, 7) > sim(5, 4) = 0.408 > 0.353$. This table contains all elements in T that can be generated using PCG on the input graph G.

Cluster Building

The clusters are built according to the rules of phylogenetics, discussed in Section 4.7. The constraints were created and stored in T which can now be resolved to create clusters. Consider the following example respective to Table 4.4. The elements of T are parsed sequentially starting with T_0.Therefore, adhering to this rule, the cluster containing v_5 is merged with the cluster containing v_7. The complete example is demonstrated below in trace Table 4.5. The APC algorithm merges all clusters containing elements of the pair $(a, b) \in ((a, b), c)$, sequentially, as annotated by the steps column. The APC algorithm terminates when $T = \emptyset$, resulting in the final clustering being identified at step 13, although no further merges were made after step 7.

Step	T	C
-	-	{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}
0	((5, 7), 4)	{0}, {1}, {2}, {3}, {4}, {5, 7}, {6}
1	((0, 1), 3)	{0, 1}, {2}, {3}, {4}, {5, 7}, {6}
2	((0, 3), 2)	{0, 1, 3}, {2}, {4}, {5, 7}, {6}
3	((0, 3), 4)	{0, 1, 3}, {2}, {4}, {5, 7}, {6}
4	((1, 3), 4)	{0, 1, 3}, {2}, {4}, {5, 7}, {6}
5	((2, 3), 4)	{0, 1, 2, 3}, {4}, {5, 7}, {6}
6	((4, 5), 3)	{0, 1, 2, 3}, {4, 5, 7}, {6}
7	((4, 6), 3)	{0, 1, 2, 3}, {4, 5, 6, 7}
8	((4, 7), 3)	{0, 1, 2, 3}, {4, 5, 6, 7}
9	((4, 7), 5)	{0, 1, 2, 3}, {4, 5, 6, 7}
10	((4, 7), 6)	{0, 1, 2, 3}, {4, 5, 6, 7}
11	((6, 7), 4)	{0, 1, 2, 3}, {4, 5, 6, 7}
12	((0, 3), 1)	{0, 1, 2, 3}, {4, 5, 6, 7}
13	((0, 2), 3)	{0, 1, 2, 3}, {4, 5, 6, 7}

TABLE 4.5: APC(PCG) Trace Table

4.10.3 Experiment 1b

This experiment sets the parameter G as the input graph of *Dataset 2*.

Constraint Generation

The input parameter G contains an additional edge between partitions, which creates new cases to consider. The constraint list T stores triplets again using the PCG function, resulting in T containing the elements shown in Table 4.7. Comparatively, the vertex with the additional edge will now be present in more comparisons when generating triplets - resulting in $|T|$ increasing relative to the number of edges in E.

v_i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	1	0.408	0.408	0.516	0.288	0.333	0	0
1	0.408	1	1	0.316	0.353	0.408	0	0
2	0.408	1	1	0.316	0.353	0.408	0	0
3	0.516	0.316	0.316	1	0.223	0.258	0.316	0.516
4	0.288	0.353	0.353	0.223	1	0.577	0.353	0.577
5	0.333	0.408	0.408	0.258	0.577	1	0.816	0.333
6	0	0	0	0.316	0.353	0.816	1	0.408
7	0	0	0	0.516	0.577	0.333	0.408	1

TABLE 4.6: Cosine similarity matrix

T	$Sim(T_i)$
((6, 7), 4)	0.408
((4, 7), 5)	0.577
((0, 1), 3)	0.408
((0, 3), 2)	0.516
((0, 3), 4)	0.516
((0, 3), 5)	0.516
((1, 3), 5)	0.316
((2, 3), 5)	0.316
((3, 5), 4)	0.258
((4, 6), 3)	0.353
((4, 5), 3)	0.577
((4, 5), 6)	0.577
((4, 5), 7)	0.577
((1, 3), 4)	0.316
((6, 7), 5)	0.408
((4, 7), 6)	0.577
((4, 7), 3)	0.577
((2, 3), 4)	0.316
((0, 3), 1)	0.516
((5, 7), 3)	0.333
((0, 2), 3)	0.408

TABLE 4.7: APC(PCG) Constraints list

The constraints are generated using the similarity values between all vertices in the graph, which are shown in Table 4.6 and stored in *cosine_matrix* in Algorithm 2. Not all values are necessary in the computation of the constraints, but they are shown for completeness. As a result of the additional edge, the similarity between v_5 and the vertices in its own cluster has been reduced, which can be problematic when generating reliable constraints.

Moreover, the APC algorithm on this graph parses all the vertices in the graph in an arbitrary order, defined by the order in which vertices were reached and the underlying implementation of the languages data structure. Consequently, the order in which clusters are merged are subject to the ordering of T - which can result in different and therefore fuzzy, clustering.

The process for constraint generation remains the same with the first triplet in Table 4.7, $T_0 = ((6,7),4)$, being generated using Table 4.6 as $sim(6,7) > sim(6,4) = 0.408 > 0.353$. This table contains all elements in T that can be generated using PCG in G.S

Cluster Building

The function in which builds the clustering, remains unchanged in that the clusters are built according to the rules of phylogenetics. The constraints were created and stored in T which can now be resolved to create the communities. Consider the following example respective to Table 4.7. The elements of T are parsed sequentially starting with $T_0 = ((6, 7), 4)$. T_0 states that v_6 and v_7 should be clustered together before v_6 and v_4. Therefore, clusters containing v_6 should be merged with v_7. The complete example is demonstrated below in trace Table 4.8. The APC algorithm merges all clusters containing elements of the pair $(a, b) \in ((a, b), c)$, sequentially, as annotated by the steps column. The APC algorithm terminates when $T = \emptyset$, resulting in the final clustering being identified at step 20, although no merges occurred after step 10.

Step	T	C
-	-	{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}
0	((6, 7), 4)	{0}, {1}, {2}, {3}, {4}, {5}, {6, 7}
1	((4, 7), 5)	{0}, {1}, {2}, {3}, {4, 6, 7}, {5}
2	((0, 1), 3)	{0, 1}, {2}, {3}, {4, 6, 7}, {5}
3	((0, 3), 2)	{0, 1, 3}, {2}, {4, 6, 7}, {5}
4	((0, 3), 4)	{0, 1, 3}, {2}, {4, 6, 7}, {5}
5	((0, 3), 5)	{0, 1, 3}, {2}, {4, 6, 7}, {5}
6	((1, 3), 5)	{0, 1, 3}, {2}, {4, 6, 7}, {5}
7	((2, 3), 5)	{0, 1, 2, 3}, {4, 6, 7}, {5}
8	((3, 5), 4)	{0, 1, 2, 3, 5}, {4, 6, 7}
9	((4, 6), 3)	{0, 1, 2, 3, 5}, {4, 6, 7}
10	((4, 5), 3)	{0, 1, 2, 3, 4, 5, 6, 7}
11	((4, 5), 6)	{0, 1, 2, 3, 4, 5, 6, 7}
12	((4, 5), 7)	{0, 1, 2, 3, 4, 5, 6, 7}
13	((1, 3), 4)	{0, 1, 2, 3, 4, 5, 6, 7}
14	((6, 7), 5)	{0, 1, 2, 3, 4, 5, 6, 7}
15	((4, 7), 6)	{0, 1, 2, 3, 4, 5, 6, 7}
16	((4, 7), 3)	{0, 1, 2, 3, 4, 5, 6, 7}
17	((2, 3), 4)	{0, 1, 2, 3, 4, 5, 6, 7}
18	((0, 3), 1)	{0, 1, 2, 3, 4, 5, 6, 7}
19	((5, 7), 3)	{0, 1, 2, 3, 4, 5, 6, 7}
20	((0, 2), 3)	{0, 1, 2, 3, 4, 5, 6, 7}

TABLE 4.8: APC(PCG) Trace Table

4.10.4 Discussion

The main strategy in this experiment was to identify whether the fundamental concepts of phylogenetics could work at an abstract level using only the topology of a simple network.

The results, when only one edge connected the small communities, the procedure is sufficient in identifying the correct constraints. By adding a new edge to the network in the case of Dataset 4.3, the number of constraints in $|T|$ increases. This would be a non-issue under the assumption all constraints are reliable in finding good clustering - this is not the case, at least naively, as shown in Section 4.10.3.

The constraints generated in experiment 1b, Table 4.7, includes 7 additional constraints than in Table 4.4. The result of APC when terminating under the condition $T = \emptyset$ produces undesirable results - as shown in trace Table 4.8. Here there are two issues; 1. APC determines all vertices belong in the same cluster in this setting and 2. there exist constraints in T that should not be used considering the topology of the graph.

Firstly, empirically studying the contents of T in Table 4.7, there are no constraints involving all vertices in the graph as we consider only local samples in G. Therefore, we have no data which states vertices from opposite sides of the graph should belong together in the same cluster. We can only make inferences given the information we have derived, which can be useful, but also proven to be potentially volatile unless managed.

Secondly, by introducing more intra-edges connecting separate clusters it also becomes difficult to differentiate between clusters and ultimately reduce the variance between intra-cluster similarity and inter-cluster similarity. This increases the likelihood of difficult triplets being generated as shown in step 8 in transition Table 4.8. The constraint $t_7 = ((3, 5), 4)$ is the first occurrence of an erroneous triplet (a triplet which incorrectly merges partitions and reduces the quality of the overall clustering), inferring the set containing v_3 should be merged with the set containing v_5. The resulting constraints merge other clusters with the set containing v_5 correctly, but ultimately grouping all vertices together into one undesirable cluster.

We considered potential solutions to solve this particular case, introducing concepts grounded in phylogenetics that classify triplets. These concepts are forbidden triplets and fan triplets.

4.11 Forbidden Triplets

During the process of calculating the initial similarities, it was noticed that the constraint list T contained forbidden triplets. Generally when building phylogenetic trees, forbidden triplets are disallowed and the building process is prematurely terminated.

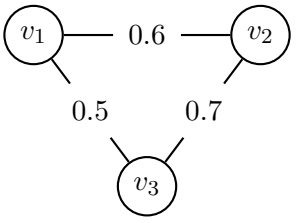


FIGURE 4.4: Forbidden Triplets

For example, consider the graph in Figure 4.4, consisting of the vertices (v_1, v_2, v_3). The edge weights in this instance are the similarity values between the vertices they connect. During the process of generating constraints using the PCG function and given the similarly values, the constraint $((v_1, v_2), v_3)$ will be added when v_1 is considered the root vertex as v_1 and v_2 are more similar than v_1 and v_3 in direct comparison, $sim(v_1, v_2) > sim(v_1, v_3) = 0.6 > 0.5$. Although when we consider v_2 as the root vertex, the constraint $((v_2, v_3), v_1)$ is added, as v_2 and v_3 are more similar than v_2 and v_3 in direct comparison, $sim(v_2, v_3) > sim(v_2, v_1) = 0.7 > 0.6$. The information in the constraints $((v_1, v_3), v_2)$ and $((v_2, v_3), v_1)$ is forbidden as they are a contradiction. They are forbidden in the context of phylogenetics as if T contains a triple which is forbidden, this means there exists no phylogenetic tree in G. In our method we still use forbidden triplets as they can be helpful in determining partitions.

But also, by disallowing forbidden triplets, another avenue of experimentation is opened and can reduce the likelihood of troublesome triplets being generated. The exclusion of said triplets decreases the granularity of the constraints and therefore there are less building rules in T. We resolve the case of forbidden triplets by keeping the constraint that pertains to the relationship of highest similarity. Therefore, in the same example above, only constraint $((v_2, v_3), v_1)$ would be stored in T as $((v_1, v_2), v_3)$ is forbidden and $0.7 > 0.6$.

There are a few challenges presented by reducing the granularity of T and disallowing forbidden triplets. Most notably, in some topologies there may not be enough constraints generated to find the user specified k-clustering.

4.12 Fan Triplets

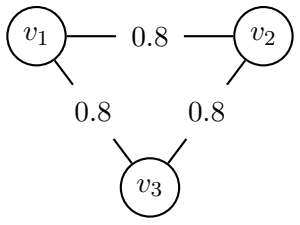


FIGURE 4.5: Fan Triplets

Previously the basic process of generating constraints did not consider the case of fan triplets. A fan triplet occurs when each sample in the comparison shares the same similarity value in direct comparison. For example, consider the graph in Figure 4.5, consisting of the vertices (v_1, v_2, v_3). The edge weights represent the similarity values between the vertices they connect. During the process of generating constraints, given the logic of Algorithm 2 and the PCG function, no constraint for this local relationship would be generated even though the vertices are highly similar. This information is important in understanding the topology of the graph and building accurate partitions, which until now has been omitted.

When allowing for fan triplets to be considered, the constraint (v_1, v_2, v_3) is added to T and will occur during the PCG function when any of vertices are the root node. When parsing such a constraint, the sets containing v_1, v_2 and v_3 are merged together. By allowing fan triplets, another avenue of experimentation is opened. Conversely to forbidden triplets, the inclusion of fan triplets increases the granularity of the constraints list T.

4.13 Experiment 2

In this section we evaluate the APC algorithm in its most basic format, building on the scenario in experiment 1. We now include the additional parameters in that we are disallowing forbidden triplets from being used in process of building clusters and allowing fan triplets to be generated as shown in parameter Table 4.9. The purpose of this experiment is to understand the effect of disallowing forbidden triplets and allowing fan triplets in association with the core process and again to determine the viability of such a mechanism independently of other parameters.

Parameter	Experiments
Termination	All Constraints
Constraint Generation	PCG
Similarity Measure	Cosine
Allow Forbidden	False
Allow Fans	True

TABLE 4.9: Experiment 2: APC Parameters

The APC Algorithm 2 remains fundamentally similar in that the PCG function is used to generate constraints although with the additional caveat shown in Algorithm 3 of enabling fan triplets when similarities are equal. The termination criterion is enabled when reaching the state $T = \emptyset$.

The other notable change in this setting is that post-processing is used to filter and resolve forbidden triplets in T. We achieve this by firstly finding all contradictory triplets and secondly, keeping the triplet of highest similarity and discarding the remaining as discussed in Section 4.11.

```
Function generate_constraints()
    T = empty
    for root in G.vertices() do
        neighbours ← G.neighbours(root)
        for neighbour in neighbours do
            if root = neighbour then continue
            root_neighbour_sim ← cosine_matrix[root, neighbor]
            for n in neighbours do
                if root = neighbour then continue
                if n = neighbour then continue
                root_n_sim ← cosine_matrix[root, n]
                if root_n_sim > root_neighbour_sim then T.add(new
                  Triplet(max(root, n), min(root, n), neighbour))

                else if root_neighbour_sim > root_n_sim then T.add(new
                  Triplet(max(root, neighbour), min(root, neighbour), n))

                else if root_neighbour_sim == root_n_sim then T.add(new
                  Fan_Triplet(max(root, neighbour), min(root, neighbour), n))

            end
        end
    end
    return T
```

Algorithm 3: APC(PCG) Fan Amendment

4.13.1 Datasets

The datasets we used for this experiment are consistent with Experiment 1a, to promote discussion between the previous configuration and this experiments. We therefore use Dataset 2 which resulted in undesirable partitioning.

Dataset 3. We also consider another small graph with an additional natural partitioning. More interconnections between clusters are present and there exists a triangle between partitions. This graph is shown in Figure 4.6.

Dataset 4. The final dataset for this experiment is a similar addition to that of Dataset 2 to Dataset 1. An additional edge is inserted to Dataset 3 to further connect two partitions. This graph is shown in Figure 4.7.

4.13.2 Experiment 2a

This experiment sets the parameter G as the input graph of Dataset 2.

Replicating the settings of Experiment 1b, with the additional parameters configured, the constraint list T stores triplets again using the PCG function, results in T containing the elements shown in Table 4.10. We also omit the similarity matrix and include the

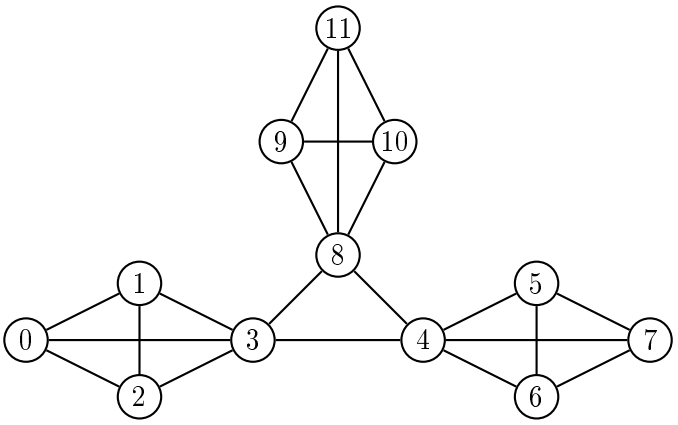


FIGURE 4.6: 3-Community Barbell

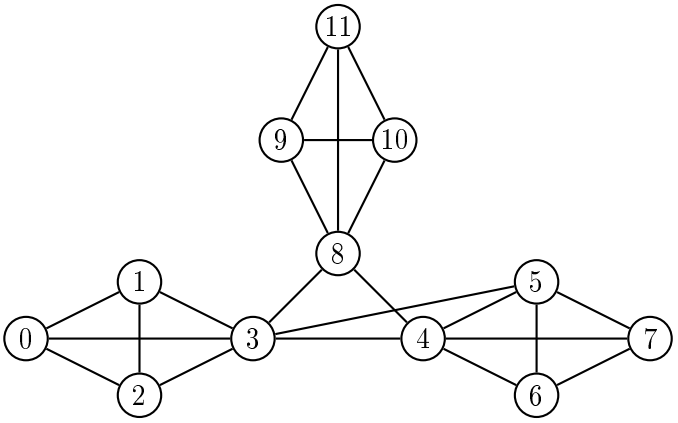


FIGURE 4.7: 3-Community Barbell with additional edge

similarity values in the constraint tables. By resolving forbidden triplets and disallowing their use, $|T|$ has decreased by 4 triplets. The set of purged triplets T_{forb} are also listed in Table 4.10. Also by introducing fan triplets, T has increased by 3 fan triplets $(0,1,2),(4,5,7)$ and $(4,5,7)$, consolidating forbidden triplets in the process.

The first forbidden element, $t_{forb} = ((6,7),4)$ is discarded because there exists a resolved triplet that contradicts t_{forb} which suggests v_6 should be merged with v_7 before v_4. The contradictory, resolved triplet $t_{res} = ((4,7),6)$ states v_4 should be merged with v_7 before v_6. We therefore check the similarity value that created each triplet and retain the maximum and discard the minimum, $max(sim(t_{forb}), sim(t_{res}) = max(0.408, 0.577)$. In the case of ties, we keep the first triplet we encounter in the resolving process.

The contradictory triplet of the second forbidden triplet $t_{forb} = ((0,1),3)$ is $t_{res} = ((4,7),6)$, resulting in $max(sim(t_{forb}), sim(t_{res}) = max(0.408, 0.577)$. The contradictory triplet of the third forbidden triplet $t_{forb} = ((0,1),3)$ is $t_{res} = ((4,5),3)$, resulting in $max(sim(t_{forb}), sim(t_{res}) = max(0.258, 0.577)$. The final contradictory triplet of the fourth forbidden triplet $t_{forb} = ((0,2),3)$ is $t_{res} = ((0,3),2)$, resulting in $max(sim(t_{forb}), sim(t_{res}) = max(0.408, 0.516)$.

T_{res}	$Sim(T_i)$
((0, 3), 1)	0.516
((0, 3), 2)	0.516
((0, 3), 4)	0.516
((0, 3), 5)	0.516
((1, 3), 5)	0.316
((2, 3), 5)	0.316
((4, 6), 3)	0.354
((4, 7), 3)	0.577
((4, 5), 6)	0.577
((4, 7), 6)	0.577
((6, 7), 5)	0.408
(0, 1, 2)	0.408
((5, 7), 3)	0.333
(4, 5, 7)	0.577
(1, 2, 3)	0.316
((4, 5), 3)	0.577
((1, 3), 4)	0.316
((2, 3), 4)	0.316

T_{forb}	$Sim(T_i)$
((6, 7), 4)	0.408
((0, 1), 3)	0.408
((3, 5), 4)	0.258
((0, 2), 3)	0.408

TABLE 4.10: Experiment 2a Resolved and Forbidden Constraints

Cluster Building

The function which builds the clustering remains unchanged, but the function for generating constraints is more restricted and T now contains the reduced and consolidated constraints. The trace table showing the execution of APC(PCG, FORB=F, FAN=T) is demonstrated below in trace Table 4.11. The APC algorithm merges all clusters containing elements of the pair $(a, b) \in ((a, b), c)$ and all clusters containing elements of the triplet (a, b, c) when it is a fan. This is performed sequentially, as annotated by the steps column. The APC algorithm terminates when $T = \emptyset$, resulting in the final clustering being identified at step 18, although no additional merges were made after step 8.

Step	T_{res}	C
-	-	{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}
0	((0, 3), 1)	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}
1	((0, 3), 2)	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}
2	((0, 3), 4)	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}
3	((0, 3), 5)	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}
4	((1, 3), 5)	{0, 1, 3}, {2}, {4}, {5}, {6}, {7}
5	((2, 3), 5)	{0, 1, 2, 3}, {4}, {5}, {6}, {7}
6	((4, 6), 3)	{0, 1, 2, 3}, {4, 6}, {5}, {7}
7	((4, 7), 3)	{0, 1, 2, 3}, {4, 6, 7}, {5}
8	((4, 5), 6)	{0, 1, 2, 3}, {4, 5, 6, 7}
...	...	...
17	((2, 3), 4)	{0, 1, 2, 3}, {4, 5, 6, 7}

TABLE 4.11: APC(PCG, FORB, FAN) Trace Table

Considering the purged set of forbidden constraints in Table 4.10. The constraint $((3,5),4)$ would merge the graph into undesirable partitions and has been successfully removed. The final clustering is accurate and an improvement over the results of Experiment 1b.

4.13.3 Experiment 2b

This experiment sets the parameter G as the input graph of Dataset 3. The new dataset contains a triangle between three communities which creates a new case for consideration and discussions. The constraint list T stores triplets again using the PCG function, purging forbidden triplets and including fan triples. The execution of APC(PCG, FORB=F, FAN=T) and the resulting list T is shown in the trace Table C.2. The tables have been merged, but contain the original triplets, the similarities used to create them and the impact on the clusters in C.

Step	T_{res}	Sim(T)	C
-	-	-	{0}, {1}, {2}, {3} {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
1	((0, 3), 1)	0.516	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
3	((1, 3), 4)	0.316	{0, 1, 3}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
4	((2, 3), 4)	0.316	{0, 1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
...	...	...	...
8	((4, 7), 3)	0.516	{0, 1, 2, 3}, {4, 7}, {5}, {6}, {8}, {9}, {10}, {11}
9	(4, 5, 6)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8}, {9}, {10}, {11}
...	...	...	...
13	((8, 9), 3)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9}, {10}, {11}
14	((4, 5), 8)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9}, {10}, {11}
15	((8, 11), 3)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 11}, {10}
...	...	...	...
18	((8, 10), 4)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
...	...	...	...
27	(3, 4, 8)	0.200	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
...	...	...	...
30	((8, 9), 4)	0.316	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

TABLE 4.12: Experiment 2b Trace Table

Table 4.13 contains the list of forbidden triplets and fan triplets generated for this input graph G. There are no elements in T_{forb} that would cause the the incorrect clusters to merge, but there exists an element in T_{fans} that does. Given the topology of G there exists a local triangle connecting all three partitions through vertices (v_3, v_4, v_8). As there are no other interconnecting edges, these vertices have equal similarity and will generate a fan triplet. The triplet that is ultimately used to merge all sets into the same cluster. This erroneous triplet can simply be avoided by disallowing fan triplets in the configuration, which would result in the desirable clusters being form and $|C| = 3$. The trace table for this configration is shown in Appendix C.1.

T_{forb}	Sim(T)
((0, 1), 3)	0.408
((0, 2), 3)	0.408
((5, 7), 4)	0.408
((6, 7), 4)	0.408
((9, 11), 8)	0.408
((10, 11), 8)	0.408

T_{fans}	$Sim(T_i)$
(4, 5, 6)	0.316
(0, 1, 2)	0.408
(1, 2, 3)	0.316
(3, 4, 8)	0.200
(5, 6, 7)	0.408
(8, 9, 10)	0.316
(9, 10, 11)	0.408

TABLE 4.13: Experiment 2b Forbidden and Fan Constraints

4.13.4 Experiment 2c

In this final experiment of the section we set the parameter G as the input graph of Dataset 4. Here we configure APC differently as a result of the fix for experiment 2b. The configuration is shown in Table 4.14. The only parameter that has changed is the ability to allow fan triplets from being included in T. Therefore, this experiment disables forbidden and fan triplets and only allows resolved triplets to be used in the cluster building process.

Parameter	Experiments
Termination	All Constraints
Constraint Generation	PCG
Similarity Measure	Cosine
Allow Forbidden	False
Allow Fans	False

TABLE 4.14: Experiment 2: APC Parameters

Step	T_{res}	Sim(T)	C
-	-	-	{0}, {1}, {2}, {3} {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
0	((0, 1), 3)	0.667	{0, 1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}
1	((0, 2), 3)	0.667	{0, 1, 2}{3}{4}{5}{6}{7}{8}{9}{10}{11}
2	((1, 2), 3)	0.667	{0, 1, 2}{3}{4}{5}{6}{7}{8}{9}{10}{11}
3	((0, 3), 4)	0.471	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
4	((0, 3), 5)	0.471	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
...	...	...	...
10	((4, 7), 3)	0.516	{0, 1, 2, 3}{4, 7}{5}{6}{8}{9}{10}{11}
11	((3, 4), 8)	0.365	{0, 1, 2, 3, 4, 7}{5}{6}{8}{9}{10}{11}
12	((4, 5), 7)	0.671	{0, 1, 2, 3, 4, 5, 7}{6}{8}{9}{10}{11}
13	((4, 5), 8)	0.671	{0, 1, 2, 3, 4, 5, 7}{6}{8}{9}{10}{11}
14	((4, 6), 8)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8}{9}{10}{11}
15	((4, 7), 8)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8}{9}{10}{11}
16	((8, 9), 3)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9}{10}{11}
17	((8, 10), 3)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10}{11}
18	((8, 11), 3)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10, 11}
...	...	...	...
34	((8, 9), 4)	0.516	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10, 11}

TABLE 4.15: Experiment 2c Trace Table

Trace Table 4.15 shows the execution of APC given the specified configuration. The additional edge has created more local triangles between the natural partitions in G - a scenario in which the likelihood of erroneous triplets being generated is significantly

increased. The contents of T is parsed and the clusters in C are merged until $T = \emptyset$. At the final step $|C| = 2$, and has reasonably clustered the graph, although incorrect merges were made.

T_{forb}	Sim(T)
((5, 7), 4)	0.577
((4, 8), 3)	0.200
((3, 4), 5)	0.365
((5, 6), 4)	0.577

T_{fans}	$Sim(T_i)$
(0, 1, 2)	0.667
(0, 1, 3)	0.471
(0, 2, 3)	0.471
(1, 2, 3)	0.471
(4, 6, 7)	0.516
(5, 6, 7)	0.577
(8, 9, 10)	0.516
(8, 9, 11)	0.516
(8, 10, 11)	0.516
(9, 10, 11)	0.667

TABLE 4.16: Experiment 2c Forbidden and Fan Constraints

The number of forbidden and fan triplets in Table 4.16 is considerable. All elements in T_{fans} would have a positive effect when building the final clustering, but have been disabled for this experiment. Conversely, there exists constraints in T_{forb}, specifically $((4, 8), 3)$ and $((3, 4), 5)$ that would have a negative effect when building the final clustering and are also excluded. Ultimately, the configuration of the allowable parameters in this experiment is unhelpful in finding the correct partitions as there exists constraints in trace Table 4.15 that merge the vertices into incorrect clusters.

4.13.5 Discussion

The main motivation for this experiment was to identify whether the fundamental concepts of phylogenetic trees, including now the addition of forbidden and fan triplets as parameters could be used to cluster vertices in relatively simple networks with well defined partitions. The results from experiment 2a, Section 4.13.2, show that the removal of forbidden triplets from T decrease the likelihood of triplets creating erroneous clusters, which is a desirable improvement. In this scenario terminating when $|T| = \emptyset$ produced the correct clusters.

The inclusion of the Dataset 3 in experiment 2b, Section 4.13.3, provided a new scenario in which difficult triplets would be generated that would ultimately merge all vertices into the same cluster. A solution, was to impose further restrictions on T and disallow the generation of fan triplets.

The final experiment, Section 4.13.4, introduced a more complex graph in Dataset 4 and modified the configuration to disallow the generation of fan triplets as a result of the previous experiment. The results showed that regardless of tuning the current

parameters, erroneous triplets would still be generated and parsed creating incorrect clustering.

We propose two solutions to this problem, 1. a new triplet generation function *Ternary Constraint Generation* and 2. a mandatory termination criterion parameter, used in clustering mechanisms such as k-means and hierarchical clustering. A user specified k value where k represents the number of desired clusters.

4.14 Termination - User Defined k

A user defined k value is often selected based on the distribution, density, shape and scale of the data points. The method of selecting k is a separate problem from clustering and beyond the scope of this chapter. This form of termination will be the primary method henceforth in which we consider when stopping our clustering process. Similarly to hierarchical clustering, the method uses k as a defined resolution of the clustering, before all vertices are merged into increasingly less granular clusters. Hierarchical clustering does this based on the similarity between clusters being the largest, whereas our method merges based on the next constraint in T. The user can heuristically specify a k or an estimator can be used in the context of machine learning. In our experiments, we know k as it is also a parameter in the generation of the datasets.

Given k, each vertex in the graph will be initialised into their own cluster. Upon parsing the constraint list T and merging these clusters, the algorithm will terminate when $|C| = k$. Terminating the algorithm upon reaching k has proven to be more beneficial than terminating when $T = \emptyset$, primarily because, as shown in the experiments, not all constraints are deemed useful to the clustering procedure - this is discussed in more detail in a later section. Additionally, using this termination criterion saves time parsing unnecessary constraints if $|C| = k$ early in the execution. This can be seen in the previous experiments, most notably in trace Table 4.11 of experiment 2a, where the algorithm could of terminated at step 8 as opposed to step 17.

4.15 Ternary Constraint Generation

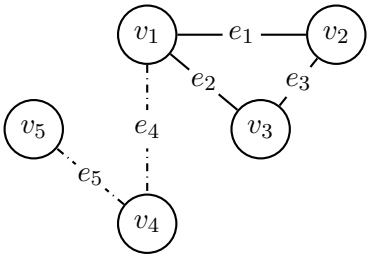


FIGURE 4.8: TCG Local Relationships

Here we consider another mechanism for generating constraints, *ternary constraint generation*, *TCG* henceforth. This is motivated by the intent of limiting the number

of erroneous or troublesome triplets generated in T. TCG is a process in which we consider stricter constraint generation by using local triangles in the graphs structure as annotated by the black edges in Figure 4.8. Specifically, it is the notion of considering root vertices directly with their neighbours and only creating constraints if there exists shared common neighbours between them. By introducing this concept, we will be reducing the granularity of T by only considering relationships in the graph where local triangle exist between vertices. The mechanism does not remove the potential for forbidden triplets as they can occur if there exists overlapping local triangles in the graph. Ultimately this adheres to stricter phylogenetic tree building rules, but less constraints are generated which can introduce a different set of difficulties in the cluster building phase.

An example of ternary constraint generation is as follows. Firstly, we determine the set of neighbours of N_r where r is the root vertex. Relative to the example above, $r = v_1$, therefore N_{v_1} will be generated. For each direct neighbour of v_1, i.e. $n \in N_{v_1}$ we perform the following procedure. We obtain the set of neighbours N_n and determine the common neighbours with set N_r. Relative to the example, $N_{common_{(v_1,v_2)}} = N_{v_1} \cap N_{v_2}$, which excludes v_1 and v_2. All the information necessary to create constraints based on the local triangles in the graph has been generated. $\forall c \in N_{common_{(r,n)}}$ we compute the similarities between (r, n), (r, c) and (n, c) and ultimately only one constraint for each local triangle is produced. In the example, only $v_3 \in N_{common_{(v_1,v_2)}}$, therefore the similarities (v_1, v_2), (v_1, v_3) and (v_2, v_3) denoted by e_1, e_2 and e_3 respectively are considered. The vertices with the highest similarity in the local triangle are added to the constraint list T, the other two constraints are therefore omitted.

As aforementioned, this will remove the possibility of generated forbidden constraints and still allow the possibility of fan triplets, which is mandatory to enable in this case otherwise important topological information will be lost. By generating only one constraint per triangle, there is a potential for losing important information in the graph as well as removing troublesome triplets. Therefore we propose an alternate version of TCG in which we allow the top two constraints in the comparison to be added to T. This ensures there are enough triplets to segment G but still also decrease the likelihood of undesirable triplets being generated.

The pseudocode for this method is provided in Algorithm 4 is an alternative to the generate constraints function of the previous algorithm.

```
Function generate_constraints_ternary()
    limit = {1, 2, 3}
    T = ∅
    ternary_triplets = ∅
    for root in G.vertices() do                    // Iterate over all vertices
        simAB, simAC, simBC = 0;
        neighbours ← G.neighbours(root)
        for neighbour in neighbours do              // Iterate over neighbours
            if root = neighbour then continue

            common_neighbours = G.common_neighbours(root, neighbour)
            for cn in common_neighbours do
                simAB ← cosine_matrix[root, neighbour]
                simAC ← cosine_matrix[root, cn]
                simBC ← cosine_matrix[neighbour, cn]
                ternary_triplets.add(new Triplet(min(root, neighbour), max(root,
                  neighbour), cn), simAB)
                ternary_triplets.add(new Triplet(min(root, cn), max(root, cn),
                  neighbour), simAC)
                ternary_triplets.add(new Triplet(min(neighbour, cn),
                  max(neighbour, cn), root), simBC)
                sort(ternary_triplets)      // Sort by largest similarity
                while i = 0 to limit do
                    T.add(ternary_triplets[i])
                end
            end
        end
    end
    return T
```

Algorithm 4: Ternary constraint generation

4.16 Experiment 3

This experiment revisits all previous datasets and demonstrates the performance differences of PCG compared with new executions using TCG. We have created a few parameters for the APC algorithm to allow for various sets of constraints to be generated, TCG, PCG and by allowing or disallowing forbidden and fan triplets. In this section we also introduce the evaluation metrics discussed in Section 4.4 to demonstrate the quality of the resulting clusters generated by the triplets generated in T for each triplet generation function.

We compute all constraint generation functions with forbidden triplets and fan triplets allowed. This configuration in Table 4.17 allows for the maximum number of constraints to be generated to be used for comparison.

Parameter	Experiments
Termination	All Constraints
Constraint Generation	PCG, TCG(1), TCG(2)
Similarity Measure	Cosine
Allow Forbidden	True
Allow Fans	True

TABLE 4.17: Experiment 3a: APC Parameters

Constraint Generation Function Discussion

The results in Table 4.18 lists the contents of T for each constraint generation procedure PCG, TCG_1 and TCG_2 for Dataset 1. The triplets have been ordered by the similarity values used to create them. We can see that PCG generates far more constraints than any of the TCG methods, which is a consistent feature present in testing on all datasets. The reason for this is that all relationships between the root and all its neighbours are directly compared and used to generate constraints, whereas TCG only generates constraints if there exists a local triangle between the root and its neighbour, i.e. when the root and neighbour share a common neighbour. Therefore, when graphs are more sparse and are not well connected, to obtain more information about the graph in the form of triplets, PCG as a constraint generator would be more suited whereas on the contrary for graphs that are more dense, TCG would be preferable. The functions PCG and TCG_2 create enough triplets for correct clustering to be performed on Dataset 1, with PCG generating more than is necessary and TCG_2 generate exactly the required amount. As Dataset 1 is quite sparse and the clusters are not complete graphs and are missing edges $e_1(v_1, v_2)$ and $e_2(v_5, v_6)$, there are not enough local triangles to generate enough constraints to describe the relationship of vertices in the clusters. Therefore, TCG_1 does not find all elements of the clusters but no incorrect merges are made either. We could expand on the solution through post-processing techniques if k clusters were not achieved as a result of the constraints generated by this function. This is discussed in a later section.

i	$PCG(T)$	$TCG_1(T)$	$TCG_2(T)$	Sim(T_i)
0	((4, 7), 5)	((4, 7), 5)	((4, 7), 5)	0.577
1	((0, 3), 1)	((0, 3), 1)	((0, 3), 1)	0.577
2	((0, 3), 2)	((0, 3), 2)	((0, 3), 2)	0.577
3	((4, 7), 6)	((4, 7), 6)	((4, 7), 6)	0.577
4	((0, 3), 4)	-	-	0.577
5	((4, 7), 3)	-	-	0.577
6	((0, 1), 3)	-	((0, 1), 3)	0.408
7	((0, 2), 3)	-	((0, 2), 3)	0.408
8	((6, 7), 4)	-	((6, 7), 4)	0.408
9	((5, 7), 4)	-	((5, 7), 4)	0.408
10	(5, 6, 7)	-	-	0.408
11	(0, 1, 2)	-	-	0.408
12	((1, 3), 4)	-	-	0.354
13	((2, 3), 4)	-	-	0.354
14	(1, 2, 3)	-	-	0.354
15	((4, 5), 3)	-	-	0.354
16	((4, 6), 3)	-	-	0.354
17	(4, 5, 6)	-	-	0.354

TABLE 4.18: Contents of T for all constraint functions on Dataset 1.

Constraint Generation Functions on All Datasets

The results of each constraint generation function is showed in Table 4.19. The columns represent the method that was used and the dataset D in which the method was applied to. The following columns describe the types of constraints found in T, annotated by $|T|$ showing the total elements in T, $|T_{forb}|$ the number of which are forbidden and finally, $|T_{fan}|$ the number of which are fan triplets. The remaining columns are the results of the evaluation metrics given the final clustering labels, described in more detail in Section 4.4. The metrics present are adjusted rand index, completeness, homogeneity and v-measure respectively.

Firstly, we can see the total number of constraints that are generated by each function in the relative configuration. The constraint generation function PCG operates as expected in that it will create more constraints as E increases. TCG_1 increases as more local triangles are introduced into the dataset, most notably in $D = B4$ when each cluster is a complete graph - highlighted by the sharp increase in $|T|$ from $B3$ to $B4$. TCG_2 was designed to compensate for the lack of local triangles in a sparse graph, which works well, but as expected creates more forbidden triplets - each comparison will add two constraints, which contradict each other in a standard phylogenetic tree building sense.

Secondly, PCG and TCG_2 generate more forbidden triplets, which is an inherent trait in their mechanism. TCG_2 as mentioned, is more pronounced given that a forbidden

triplet is created every comparison. PCG discounts local triangles and only compares the edges incident to the root vertex in the comparison. The forbidden triplets occur when a neighbour of the previous root contains edges of higher similarity than between root and neighbour. TCG_1 alleviates this issue by considering local triangles and adding only constraints that are consistent - but this mechanism can still generate forbidden triplets when local triangles overlap in the graph, as shown in datasets $B2$ and $B4$.

Thirdly, naturally PCG will generate more fan triplets than the TCG methods, again as an inherent trait of the mechanism as two incident edges from a root node are more likely to be equal – thus creating a fan – than three edges of a local triangle being equal. If there exists a local triangle in the graph, then PCG will also generate at least one unique fan triplet. This trend in relation to $|T_{fan}|$ is noticeable across all datasets in Table 4.19.

Finally, the results of the evaluation metrics provide consistent results with the complexity of the dataset. For example, PCG and TCG_2 create the correct clustering based on the truth labels in $B1$, as shown by the highest score of 1 across all metrics. Conversely, TCG_2 does not generate enough constraints to generate the correct complete clustering, although a partial clustering is achieved. R demonstrating a low cluster inter-similarity due to the disparity between $|C_{actual}|$ and $|C_{expected}|$. Also, the resulting labelling of vertices are partially split across different clusters, therefore the assignment is partially complete and is represented in completeness. However, as only partial clustering was performed correctly, the non-perfect labellings further split classes into more clusters - this can be perfectly homogeneous and therefore still obtains the highest homogeneity score. V-measure, being the harmonic mean between homogeneity and completeness, shows that the samples are homogeneous but contain unnecessary splits due to the lack of constraints - this harms completeness and thus penalises the V-measure.

As the datasets become more complex, the evaluation metrics noticeably diminish. This is as a result of PCG and TCG_2 ultimately merging all vertices into the same cluster. Conversely, TCG_1 performs increasingly better in comparison as the graph becomes more complex, more local triangles exist and ultimately more constraints are generated. Although, TCG_1 does not generate enough constraints to merge the vertices into one cluster. As a result of this method only generating partial clustering, maintains relatively good scores in the evaluation metrics.

Method	D	$\|T_{res}\|$	$\|T_{forb}\|$	$\|T_{fan}\|$	R	$Comp$	H	V
PCG	B1	18	4	4	1.000	1.000	1.000	1.000
TCG(1)	B1	4	0	0	0.186	0.400	1.000	0.571
TCG(2)	B1	8	4	0	1.000	1.000	1.000	1.000
PCG	B2	24	6	3	0.000	1.000	0.000	0.000
TCG(1)	B2	7	2	1	0.364	0.464	1.000	0.634
TCG(2)	B2	11	6	1	0.000	1.000	0.000	0.000
PCG	B3	37	6	7	0.522	1.000	0.579	0.734
TCG(1)	B3	7	0	1	0.103	0.455	0.790	0.577
TCG(2)	B3	13	6	1	0.522	1.000	0.579	0.734
PCG	B4	49	12	10	0.000	1.000	0.000	0.000
TCG(1)	B4	22	3	10	0.468	0.608	0.855	0.711
TCG(2)	B4	34	15	10	0.000	1.000	0.000	0.000

TABLE 4.19: Constraint generation functions on all datasets.

4.16.1 Discussion

From a computation and complexity perspective, the number of constraints is important - especially if the same information can be described using less triplets. But given a topology complex enough or a graph containing vertices that could be classified as multiple clusters, then it is unavoidable in that triplets will be generated that would compromise the tree building process in its current form - ultimately merging all vertices into one cluster. Consequently, even by introducing various mechanisms to generate a more restricted constraint list, the building process of using just the constraints to create clusters is unreliable.

The triplets, regardless of the generation method used, are not all created equal, even though they are treated as such in the preceding experiments. This can be seen by considering the similarity value used to create the triplets. Consider the example from Table 4.18, the triplet $t_3 = ((5,7),4) = 0.408$ is parsed before $t_4 = ((0,3),4) = 0.577$, which even though in this instance do not interfere with one another, should not be the case. More information from the triplets can be deduced and not just used in the traditional sense for what they syntactically represent. By considering the triplets and the relationship of the elements comprising them, we can create additional succinct features and dimensions in order to enhance the cluster building process. Accordingly, we further analyse the triplets in T and introduce the notion of ranking constraints.

4.17 Ranking Constraints

4.17.1 Motivation

Building clustering from the constraint list T is not sufficient independently. Thus far, the building process is dependent on the order in which constraints are added to the list, therefore the clusters derived from T are partially fuzzy in nature - the resulting clusters will be different upon every execution of the algorithm as the method relies on not only the internal data structures of the implemented language, but also the order in which the vertices are accessed. Previous experiments are influenced purely on phylogenetic building rules and constraint definitions - but the decision that samples should be clustered because there exists a constraint that implies so, is not strong enough to determine good clustering. In phylogenetics, the building process terminates if there exists a forbidden triplet, as this implies there is no phylogenetic tree.

The aim is to achieve a *hard* algorithm that achieves consistent clusters irrespective of the number of times executed, independent of implementation language and the order in which vertices are accessed - the ordering of the generated constraints is necessary to satisfy this objective and has been a crucial parameter which has been so far emitted. Here we derive and propose new knowledge from triplets to introduce new features to consistently order constraints to achieve good clustering. Fundamentally, the goal is to order constraints such that well connected, highly similar, important vertices are ranked highly and outliers or overlapping vertices are ranked sufficiently low that the algorithm will terminate before they are used. This mechanism enables the constraint generation methodologies that create more troublesome triplets increasingly useful as more topological information of G is known and can be used in the ranking procedure. Conversely, limiting the number of constraints is still paramount as there are less constraints that would compromise the quality of the resulting clusters.

4.17.2 Mechanism

Interpreting triplets based on what they semantically represent is not sufficient to derive good clustering as this implies all constraints are created equally. In this context of clustering, this should not be the case. There is more information about the topology of the graph to be interpreted from the syntactical representation of the triplets, the components of a triplet and their respective position.

As we have defined, a resolved triplet is of the form $t = ((a, b), c)$, implying that clusters containing a and b should be merged before either of those with c. Currently in the cluster building process we use only the left hand side of the triplet definition, (a, b) and ignore the right hand side c. We therefore define the triplet components as LHS and RHS respectively. The LHS is important considering clusters are merged based on the interpretation of such and consider this component as positive impact. Conversely, a sample in RHS of a triplet is considered negative impact as this states there exists a

better pairing of vertices in the graph. Effectively ordering triplets is paramount to our clustering objective and is based on this relationship of all samples and the involvement in each triplet in the contents of T.

We therefore create two registers of size n, recording the appearance of samples in LHS and RHS. Therefore, $\forall t_{res} \in T$ $LHS_a+=1$, $LHS_b+=1$ and $RHS_c+=1$ and $\forall t_{fan} \in T$, $LHS_a+=1$, $LHS_b+=1$ and $LHS_c+=1$. The contents of LHS and RHS registers is vital as we understand that high scoring samples in LHS imply they are prominent or influential in their local neighbourhood and high scoring samples in RHS imply the inverse.

More information from the syntax is to be derived and that is the concept of co-occurrence of elements in LHS. Therefore we create another $n \times n$ register CO to store the co-occurrence of samples in LHS. Therefore, $\forall t \in T$ we increment $CO_{a,b}+=1$. The contents of CO is used to better understand the relationship between a and b in LHS and not just the independent occurrence of each element in LHS or RHS. Elements scoring highly in CO state that a and b are not just present in LHS frequently, but they are frequently seen together, further reinforcing the decision that they should be merged with high priority.

Using the derived information from syntactical representation of triplets in T, we proposed the following formulas to determine a triplets rank R.

$$R(t) = RCO(a,b) \times \frac{F(a)+F(b)+F(c)}{3} \tag{4.2}$$

$$R(t) = \frac{RCO(a,b)+RCO(a,c)+RCO(b,c)}{3} \times \frac{F(a)+F(b)+F(c)}{3} \tag{4.3}$$

Equation 4.2 is calculated for resolved triplets and the largest value of R signifies a triplet of high priority. The LHS of the equation RCO is the percentage of common occurrences out of total triplets involving the contributing elements. More specifically, $RCO_{ab} = ((CO_{ab}/Total_a)+(CO_{ab}/Total_b))/2$. This value will determine how frequently elements a and b are together relative to the number of constraints they are involved in. This calculation is used to avoid bias in vertices of high degree. The value calculated is a modifier applied to the overall *fitness* of a triplet, represented as F in the equation. The calculation to determine the fitness of a vertex is the difference between the percentage of total constraints where the vertex is involved in LHS compared to the percentage of total constraints where the vertex is involved in RHS. More specifically, $F(a) = ((LHS_a/Total_a) - (RHS_a/Total_a))$. The *fitness* of each vertex involved in LHS of a constraint is summed along with the fitness of the RHS vertex and the average is taken to determine the fitness of the constraint.

The rationale for this equation is vertices which are fit, determined by the number of *positive* relationships they are involved in, and vertices which are fit occur together frequently will score highly. As opposed to constraints involving unfit vertices being clustered together, such as outliers, which would contain similar or even negative F

score. The constraints which involve a vertex of high F value paired with a vertex of low F score should still score higher than the two outlier example.

Alternatively for the occurrence of fan triplets, equation 4.3 is applied which shares the same notion as equation 4.2. Although in a fan triplet of the form (a, b, c), c is considered LHS and positive element as opposed to RHS and negative. We therefore need to consider the average of common occurrences between all pairs in (a, b, c) over the total constraints they are involved in. The fitness of c is now also included in the final part of the equation. The denominators has increased to represent the addition of c being a positive influencer.

4.18 Experiment 4

The following experiment demonstrates the APC algorithm with a new parameter, a boolean to enable or disable the ability to rank constraints using equations 4.2 and 4.3. Here we test the performance of the ranking procedure in different settings, combining various aforementioned parameters, including all three of the constraint generation functions and enabling and disabling forbidden and fan triplets. The algorithm with different combination of parameters are all executing on the datasets from previous experiments discussed in Section 4.10.1. The used configurations for this experiment is shown in Table 4.20.

Parameter	Configuration
Termination	k
Constraint Generation	PCG, TCG(1), TCG(2)
Similarity Measure	Cosine
Rank Constraints	True
Allow Forbidden	True, False
Allow Fans	True, False

TABLE 4.20: Experiment 4: APC Parameters

4.18.1 Ranking Constraints on All Datasets

The first set of results for this experiment are obtained by configuring APC to enable ranking, fan and forbidden triplets. We show all constraint generation functions on datasets, the results from which are shown in Table 4.21. We show the total constraints $|T|$, how many of which are forbidden $|T_{forb}|$, fans $|T_{fan}|$ and how many are used to reach k clustering $|T_{used}|$. The constraint generation functions remained unchanged from previous experiments and therefore $|T|$, $|T_{forb}|$ and T_{fans} are equivalent. The salient result is $|T_{used}|$ to achieve k clustering as a result of the ranking procedure scoring triplets based on their fitness. The same issues arrive from previous experiments when assessing different constraint generation mechanisms. PCG generates many triplets as there are more comparisons, whereas TCG creates triplets only when there are triangles present

in the topology. Therefore, there some functions not generating enough constraints to create k clusters in certain datasets and more importantly there are less constraints to rank. The quantity of triplets determines the clarity of the information we understand about the graph using the ranking mechanism we have defined. Therefore, the ranks for triplets created using certain constraint generation functions will produce small variance in ranks due to the fewer constraints. This is an artefact also present when forbidden triplets and fan triplets are disabled.

Overall, under this configuration the constraints were effectively ranked using the PCG in all datasets. All clusters were correctly identified using substantially less constraints than generated, which is even more pronounced in Dataset 3 and Dataset 4. The ternary function $TCG(2)$ performed comparatively well, identifying the correct clusters using a similar number of constraints to PCG although considerably less constraints were generated overall in all datasets a trade-off worth considering on larger graphs. The final function $TCG(1)$ demonstrates similar performance to previous experiments as the topologies in these small datasets do not contain enough local triangles to create meaningful triplets, a problem further exacerbated in subsequent configurations with forbidden and fan triplets disabled. $TCG(1)$ shows notable improvement as the datasets become more complex, still out performing previous experiments but not generating enough constraints to retrieve the final clusterings exact ground truth.

Method	D	$\|T\|$	$\|T_{forb}\|$	$\|T_{fan}\|$	$\|T_{used}\|$	ARI	$Comp$	H	V
PCG	1	18	4	4	5	1.000	1.000	1.000	1.000
TCG(1)	1	4	0	0	4	0.533	0.552	1.000	0.711
TCG(2)	1	8	4	0	8	1.000	1.000	1.000	1.000
PCG	2	24	6	3	11	1.000	1.000	1.000	1.000
TCG(1)	2	7	2	1	7	0.000	0.254	0.138	0.179
TCG(2)	2	11	6	1	7	0.000	0.254	0.138	0.179
PCG	3	37	6	7	7	1.000	1.000	1.000	1.000
TCG(1)	3	7	0	1	7	-0.048	0.328	0.250	0.284
TCG(2)	3	13	6	1	12	1.000	1.000	1.000	1.000
PCG	4	49	12	10	21	1.000	1.000	1.000	1.000
TCG(1)	4	22	3	10	21	0.737	0.826	0.810	0.818
TCG(2)	4	34	15	10	22	1.000	1.000	1.000	1.000

TABLE 4.21: APC(PCG, FORB=T, FAN=T, RANK=T).

Trace Table 4.22 below demonstrates the ranking values that were calculated using $APC(PCG, FORB = T, FAN = T)$ on Dataset 4. Steps where no update to the clusters occurred have been omitted but can be seen in Appendix C.1. The highest rank given our mechanism for this configuration was applied to constraint $t_0 = (9, 10, 11)$ where $R(t) = 0.429$. Through empirical analysis of the topology of Dataset 4 we can see there are three partitions defined by their truth labels consisting of clusters $c_1 =$

$\{0, 1, 2, 3\}, c_2 = \{4, 5, 6, 7\}$ and $c_3 = \{8, 9, 10, 11\}$. There exists a triangle between the three vertices $\{3, 4, 8\}$ creating opportunities for erroneous triplets to be generated in the context of clustering. There exists an additional edge between v_3 in cluster c_1 and v_5 in cluster c_2. This additional edge further reduces the inter-cluster similarity between elements in c_2 more so than other clusters as there exists more vertices which have external edges which further increases inter-cluster similarity. Therefore, c_2 is the weakest connected community, followed by c_1 as v_3 has two external edges, more than the one external edge from v_8 in c_3. This affects the cosine similarity values of internal nodes in the clusters $c_1 = \{0, 1, 2\}$, $c_2 = \{6, 7\}$ and $c_3 = \{9, 10, 11\}$ with their outlier neighbours $c_1 = \{3\}, c_2 = \{4, 5\}$ and $c_3 = \{8\}$.

As a result of this observation the ranking algorithm has effectively ordered triplets in this experiment for the following reasons. Fan triplet t_0 ranks slightly above fan triplet t_1 as although empirically similar, the fitness score of each individual vertex is taken into consideration. The vertices in c_1 are not considered as fit due to the relationship with v_3, as they are involved in more triplets overall, diluting the fitness value. The last remaining set of internal nodes is then paired in t_2, considered less fit due to the relationship with v_4 and v_5.

Step	T	R(t)	C
-	-	-	{0}, {1}, {2}, {3} {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
0	(9, 10, 11)	0.429	{0}{1}{2}{3}{4}{5}{6}{7}{8}{9, 10, 11}
1	(0, 1, 2)	0.375	{0, 1, 2}{3}{4}{5}{6}{7}{8}{9, 10, 11}
2	((6, 7), 5)	0.280	{0, 1, 2}{3}{4}{5}{6, 7}{8}{9, 10, 11}
3	((9, 10), 8)	0.279	{0, 1, 2}{3}{4}{5}{6, 7}{8, 9, 10, 11}
...	...	...	...
15	((6, 7), 4)	0.252	{0, 1, 2}{3}{4}{5}{6, 7}{8, 9, 10, 11}
16	(5, 6, 7)	0.205	{0, 1, 2}{3}{4}{5, 6, 7}{8, 9, 10, 11}
17	((0, 3), 5)	0.195	{0, 1, 2, 3}{4}{5, 6, 7}{8, 9, 10, 11}
...	...	...	...
20	(4, 6, 7)	0.174	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}
...	...	...	...
44	((4, 5), 8)	0.033	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}
45	((3, 4), 5)	0.017	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10, 11}
46	((3, 5), 8)	0.007	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10, 11}
47	((3, 4), 8)	0.006	{0, 1, 2, 3, 4, 5, 6, 7}{8, 9, 10, 11}
48	((4, 8), 3)	0.003	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

TABLE 4.22: Experiment 4 Trace Table

Moreover, the last elements in Table 4.22 illustrate a similar point. There exists four erroneous triplets in $|T|$ when $k = 3$, all of which are ranked in the last four positions of the ordered list. Element t_{45} is the first triplet which causes incorrect merges in C

if not ranked sufficiently as c_1 and c_2 are the most similar as opposed to cluster c_3. Vertices v_3 and v_4 should be merged before v_5 considering they share an additional common neighbour in v_8. These vertices are all ranked sufficiently low as they do not co-occur frequently and they are unfit in that they appear on the LHS and RHS almost indistinguishably or occur more so in the latter. Similarly, the difference between v_{47} and v_{48} is noteworthy as the same concept is held as discussed for the higher ranking triplets, v_{47} shares more common neighbours between elements in LHS than in v_{48} and the overall fitness of the contributing vertices in these triplets is sufficiently low, combined with a low co-occurrence score in LHS that effectively ranks them desirably last in T.

4.18.2 Additional Configurations

The following tables contain the results from the effects of enabling and disabling forbidden and fan triplets in the APC algorithm and illustrate how the resulting clustering performs relative to finding the natural partitions defined in the ground truth labels.

Firstly, Table 4.23 shows the results for the configuration APC(PCG, FORB=F, FAN=T, RANK=T) on all datasets. As designed $|T|$ is decreased in all cases. PCG generates forbidden triplets inherently in the design of the comparison mechanism and is impacted substantially. $TCG(1)$ is inherently designed to minimised the occurrence of forbidden triplets but they can occur when triangles overlap and edges are involved in multiple comparisons such as in Dataset 2 and 4. $TCG(2)$ is inherently designed to include forbidden triplets to generate more information to aid in clustering. Therefore, when disabled, the performance of $TCG(2)$ is identical to $TCG(1)$.

Method	D	$\|T\|$	$\|T_{forb}\|$	$\|T_{fan}\|$	$\|T_{used}\|$	ARI	$Comp$	H	V
PCG	B1	14	4	4	8	1.000	1.000	1.000	1.000
TCG(1)	B1	4	0	0	4	0.533	0.552	1.000	0.711
TCG(2)	B1	4	4	0	4	0.533	0.552	1.000	0.711
PCG	B2	18	6	3	11	1.000	1.000	1.000	1.000
TCG(1)	B2	5	2	1	5	0.000	0.254	0.138	0.179
TCG(2)	B2	5	6	1	5	0.000	0.254	0.138	0.179
PCG	B3	31	6	7	15	1.000	1.000	1.000	1.000
TCG(1)	B3	7	0	1	7	-0.048	0.328	0.250	0.284
TCG(2)	B3	7	6	1	7	-0.048	0.328	0.250	0.284
PCG	B4	37	12	10	25	1.000	1.000	1.000	1.000
TCG(1)	B4	19	3	10	19	0.737	0.826	0.810	0.818
TCG(2)	B4	19	15	10	19	0.737	0.826	0.810	0.818

TABLE 4.23: APC(PCG, FORB=F, FAN=T, RANK=T).

Secondly, Table 4.24 shows the results for the configuration APC(PCG, FORB=T, FAN=F, RANK=T) on all datasets. Again, $|T|$ is decreased in all cases due to the

topology of the test graphs including vertices of identical common neighbours and therefore similarity. PCG was modified to create fan triplets inherently in an unconventional way and $|T|$ is impacted slightly. $TCG(1)$ is inherently designed to discover the ternary relationship between vertices and disabling fan triplets for this method diminishes one of the key strengths of the mechanism. $TCG(2)$ will discover the same number of fan triplets as $TCG(1)$, but with the inclusion of forbidden triplets produces different results. The inclusion of fan triplets in ternary comparisons are arguably more legitimate in this mechanism based on the definition of fan triplet being true to phylogenetics.

Method	D	$\|T\|$	$\|T_{forb}\|$	$\|T_{fan}\|$	$\|T_{used}\|$	ARI	$Comp$	H	V
PCG	B1	14	4	0	10	1.000	1.000	1.000	1.000
TCG(1)	B1	4	0	0	4	0.533	0.552	1.000	0.711
TCG(2)	B1	8	4	0	8	1.000	1.000	1.000	1.000
PCG	B2	21	5	0	10	1.000	1.000	1.000	1.000
TCG(1)	B2	6	1	0	6	0.000	0.254	0.138	0.179
TCG(2)	B2	10	5	0	7	1.000	1.000	1.000	1.000
PCG	B3	30	6	0	17	1.000	1.000	1.000	1.000
TCG(1)	B3	9	2	0	9	-0.048	0.328	0.250	0.284
TCG(2)	B3	15	8	0	12	1.000	1.000	1.000	1.000
PCG	B4	39	4	0	26	1.000	1.000	1.000	1.000
TCG(1)	B4	18	4	0	15	0.737	0.826	0.810	0.818
TCG(2)	B4	30	16	0	15	0.737	0.826	0.810	0.818

TABLE 4.24: APC(PCG, FORB=T, FAN=F, RANK=T).

Finally, Table 4.25 shows the results for the configuration APC(PCG, FORB=F, FAN=F, RANK=T) on all datasets. By allowing only resolved triplets, $|T|$ is further decreased to a more refined and restrictive set of constraints. PCG under this configuration generates enough constraints in all datasets to compute the correct clustering relative to the ground truth partitions, even in sparse graphs such as Dataset 1. $TCG(1)$ and $TCG(2)$ under this configuration is again identical in performance and therefore the results have been collapsed into one record. The ternary constraint generation functions already produce minimal constraints and have been restricted even further. Although not generating enough constraints in Dataset 1, 2 and 3, the algorithm terminates before all constraints have been generated in Dataset 4. The clustering returned is almost consistent with the ground truth partitions, incorrectly merging v_3 from c_1 into c_2 which could arguably be correct.

Method	D	$\|T\|$	$\|T_{forb}\|$	$\|T_{fan}\|$	$\|T_{used}\|$	ARI	$Comp$	H	V
PCG	B1	10	4	0	6	1.000	1.000	1.000	1.000
TCG(1,2)	B1	4	0	0	4	0.533	0.552	1.000	0.711
PCG	B2	16	5	0	9	1.000	1.000	1.000	1.000
TCG(1,2)	B2	5	1	0	5	0.774	0.711	1.000	0.831
PCG	B3	24	6	0	18	1.000	1.000	1.000	1.000
TCG(1,2)	B3	7	2	0	7	0.312	0.572	0.685	0.623
PCG	B4	35	4	0	26	1.000	1.000	1.000	1.000
TCG(1,2)	B4	14	4	0	12	0.737	0.826	0.810	0.818

TABLE 4.25: APC(PCG, FORB=F, FAN=F, RANK=T).

4.18.3 Discussion

We have shown all combination of parameters for the APC algorithm that have been defined thus far on small datasets with varying topologies. The various tuning of the parameters can impact the ranking mechanisms we propose significantly depending on the topology of the graph. We introduced different constraint generation functions with the intention of creating mechanisms that work well in sparse and dense graphs. A by-product of these procedures, combined with the ability to enable or disable fan or forbidden triplets allows quite a different set of constraints to be generated. Although, regardless of the parameter combinations the objective of the algorithm and the constraints generated are inherently similar. The constraint generation functions create constraints based on the notion of vertex similarity in the network and thus regardless of the method, they will share a similar subset of constraints. The additional rules built into the function constrain the interactions between vertices resulting in fewer constraints being generated. As shown in the results from this experiment, for each dataset the size of the constraint list $|T|$ is reduced as parameters are disabled. Consider the constraint generation function PCG on D_4, $|T|$ ranges from $|T| = 49$ when all parameters are enabled to $|T| = 37, 39$ with either forbidden or fan triplets disabled respectively, to ultimately $|T| = 35$ with all parameters disabled. Computational time is saved when creating the constraints list with features disabled and there is less constraints to parse during the building phase. Conversely, based on the results in the tables using the same example, although $|T|$ decreases the required number of constraints to parse resulting in k clusters is slightly increased. This is an artefact of the topology of the graph as this is a trend consistent with all constraint generation functions. Given all the datasets we consider are based on the barbell graph or a small variation, many of vertices will share identical similarity to other vertices in the graph. Therefore, it would be unwise to disable fan triplets as much of the information about internal clusters is lost, but in the case of Dataset 3 the information regarding the inter-cluster triangle is also removed. The parameters regarding the enabling or disabling of fan triplets and forbidden

triplets is to produce a method that is consistent with phylogenetics, but also to compromise between a trade-off of increasing the number of constraints to understands more information about the topology of the graph and reducing the number of constraints to encounter less erroneous triplets. We are concerned not only with pairs of vertices that are of high similarity but pairs of low similarity. Additionally, we are concerned with vertices individually and the *fitness* of each element. The ranking mechanism we have defined considers these attributes and an artefact of such benefits functions as PCG, as well as TCG(2) with fan and forbidden triplets enabled, because we consider erroneous information equally important as resolved information.

4.19 APC Experiments and Resultss

In this section we configure our clustering mechanism and parameters on various benchmark graphs discussed in Section 4.3 and evaluating the performance using clustering specific metrics discussed in Section 4.4. This is a response to the call for standardisation of evaluation metrics in [63]. We therefore show our mechanism on various benchmark graphs, including caveman graphs, relaxed caveman graphs, l-planted partition graphs and gaussian random partitions with different settings. These graphs are discussed in detail in the following subsections. We also compare the performance of our algorithm to influential mechanisms, specifically Girvan-Newman (denoted as GN) proposed in [71] and implemented in [4] as well as *Label Propagation* (denoted as LP) proposed in [106] and implemented in [1]. We analyse and discuss the overall performance and the resulting clustering of our and comparator methods.

4.19.1 Caveman Graphs

The graphs discussed in this subsection are popularised in *Small Words* [117]. Caveman graphs, CM, are a type of graph which became popular in social network theory, the notion of a simple social network with unique properties. A social network in which everyone has one completely connected group of acquaintances. A k-caveman graph is essentially a graph of k independent cliques. We show only one example of the clustering algorithm working on a k-caveman graph, which detects independent cliques as individual clusters in all configurations and graph sizes.

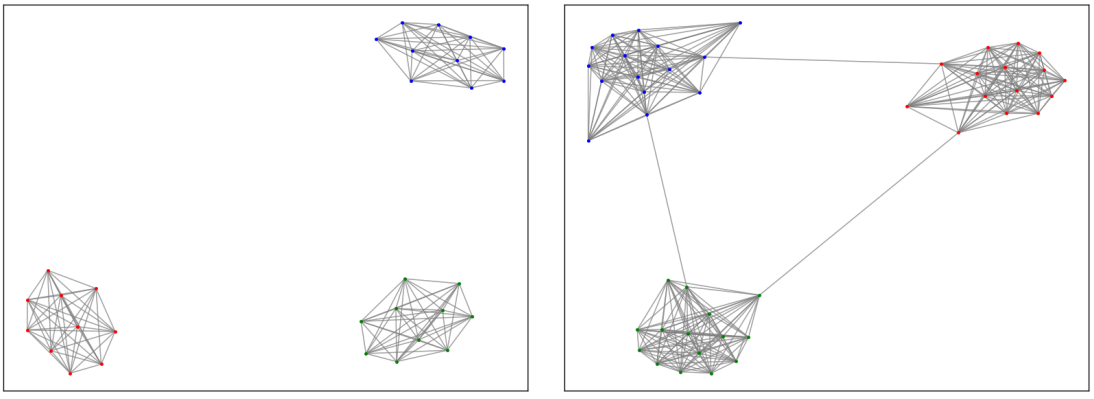
We also tested the algorithm on the concept of *connected caveman graphs*, CCM, a minor extension of standard caveman graphs, a less extreme variant of a caveman world in which each world contains well connected groups of people. In this setting, each caveman group communicates through one person. Given the inherent properties of the graph, our APC algorithm will always find these communities.

The final caveman topology we consider is a *relaxed caveman graph*, RCM, adapted to more realistically explain the clustering properties of social networks. In this variant edges are rewired with a specified probability to link different cliques. These graphs are interesting to study since they represent ideal graphs with "perfect" communities.

Method	D	n	k	p	ARI	$Comp$	H	V
APC	CM	30	3	-	1.000	1.000	1.000	1.000
GN	CM	30	3	-	1.000	1.000	1.000	1.000
LP	CM	30	3	-	1.000	1.000	1.000	1.000
APC	CCM	45	3	-	1.000	1.000	1.000	1000
GN	CCM	45	3	-	1.000	1.000	1.000	1.000
LP	CCM	45	3	-	1.000	1.000	1.000	1.000

TABLE 4.26: Caveman Graph Experiments

The results in Table 4.26 show the clustering quality returned from our APC method on the caveman graphs created with the specified parameters, as well as the performance of HC. We use the standardised clustering metrics discussed in previous sections to evaluate the clustering quality. As expected on the caveman topology and its variants, our method and HC determine the correct partitions. The first two topologies, CM and CCM, are identical in structure when the parameters to generate the graphs, n and k are increased. Therefore, further experiments on these graphs are unnecessary. The results for these topologies are shown in Figure 4.9. Conversely, the final topology RCM is generated using extra parameters other than n and k - specifically we use p, the parameter which defines the probability of rewiring each edge in G. The example in Figure 4.9 is consistent with the parameters in Table 4.26

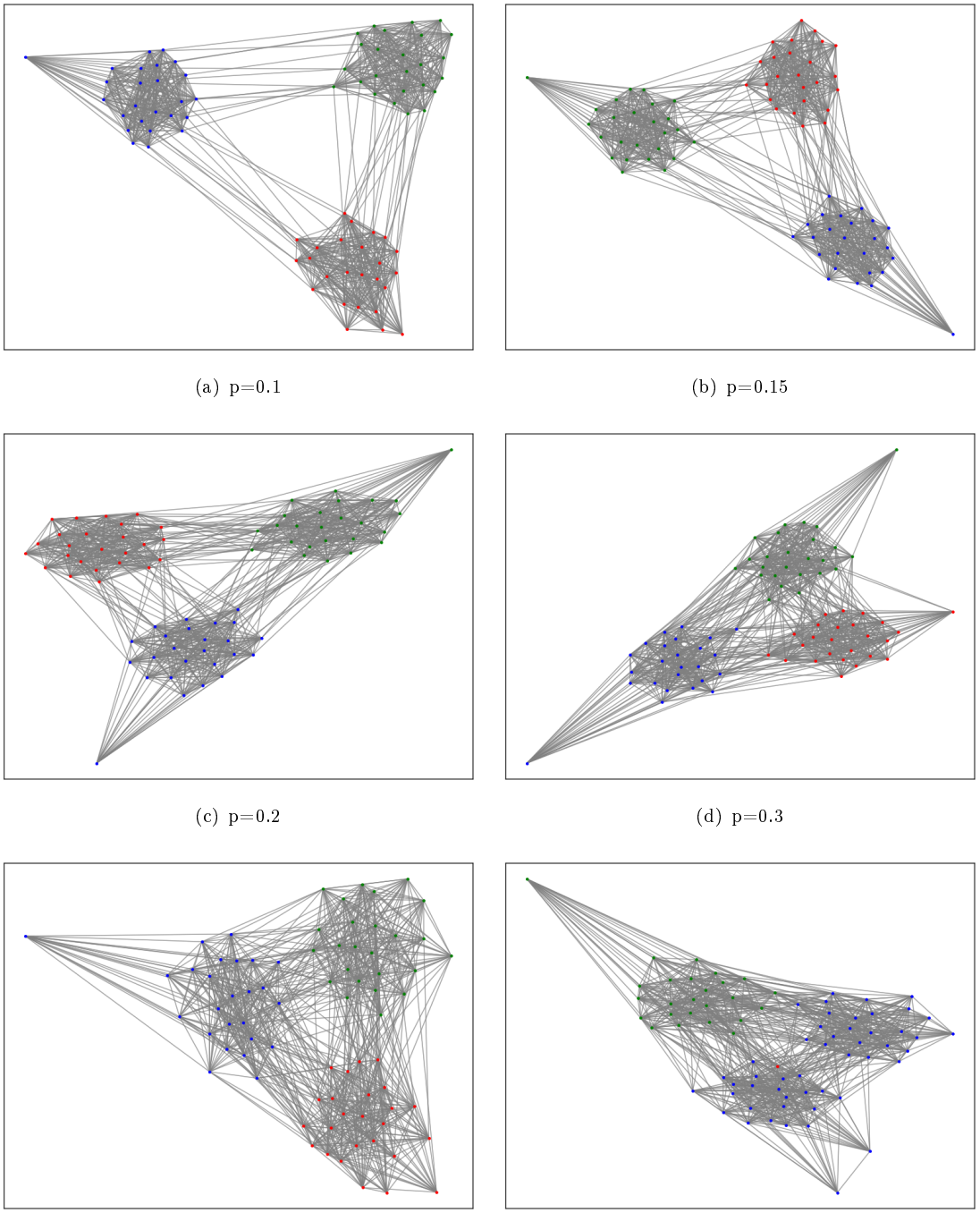


(a) Caveman Graph (b) Connected Caveman Graph

FIGURE 4.9: APC Results on Caveman Graphs

The variant RCM is a standard benchmark example used to simulate real-world social networks. We have therefore provided further experiments using different configurations of RCM to determine performance on increasingly complex graphs created by this model. We consider various increments of p, increasing the value until our method encounters difficulty in identifying the correct clustering.

The resulting clustering from running our method on different RCM graphs with varying p values is visualised in Figure 4.10. The details of the experiment and cluster



(a) p=0.1 (b) p=0.15

(c) p=0.2 (d) p=0.3

(e) p=0.35 (f) p=0.4

FIGURE 4.10: APC on Relaxed Caveman Graphs

quality are shown in Table 4.27. The value of p is incremented each experiment, increasing the likelihood of an edge being rewired to connect an external cluster. This can be seen visually as the graph becomes more complex and clusters become increasingly difficult to distinguish.

Method	D	n	k	p	ARI	$Comp$	H	V
APC	RCM	25	3	0.3	1.000	1.000	1.000	1000
GN	RCM	25	3	0.3	1.000	1.000	1.000	1.000
LP	RCM	25	3	0.3	1.000	1.000	1.000	1.000
APC	RCM	25	3	0.35	1.000	1.000	1.000	1000
GN	RCM	25	3	0.35	1.000	1.000	1.000	1.000
LP	RCM	25	3	0.35	1.000	1.000	1.000	1.000
APC	RCM	25	3	0.4	0.560	0.920	0.588	0.717
GN	RCM	25	3	0.4	1.000	1.000	1.000	1.000
LP	RCM	25	3	0.4	0.565	1.000	0.579	0.734

TABLE 4.27: APC on Relaxed Caveman Graphs

Our method detects the original partitions generated using RCM models well until $p = 0.4$, encountering issues detecting the original partitions at a similar point to comparator algorithm LP, but before GN. The ARI is the measure of similarity between two data clusterings, specifically in this setting between our returned clustering and the original partitions or truth labels. For lower probabilities of p the $ARI = 1$, confirming our returned clustering exactly matches the truth labels of G. At higher probabilities of edges being rewired, specifically when $p = 0.4$ in Table 4.27 the ARI value for our method and LP decreases accordingly. Specifically for APC, $ARI = 0.560$ states the sets of labels agree on the clustering over half the time, but the returned clustering is not perfect. This is consistent with Subfigure(f) in Figure 4.10, APC discovers two of the partitions but two clusters are merged before an individual vertex in the graph (highlighted as the red vertex) - an APC artefact inherent in our method at the point of cluster degradation and is discussed later in detail.

The remaining three evaluation metrics are all related but highlight different properties of clustering. The metric *Completeness* ($Comp$) is symmetrical to homogeneity. In order to satisfy the completeness criteria, a clustering must assign all of those vertices that are members of a single class to a single cluster [109]. Therefore, considering the example executions of LP, $Comp = 1$ in all instances even when cluster quality degrades and two clusters are collapsed due to a larger edge rewiring probability p. $Comp$ is still satisfied, all vertices in a truth label belong to the same resulting cluster which is the fact that is being measured. The reason APC achieves an imperfect $Comp = 0.920$ is because of the same artefact in that one vertex is considered a cluster and the condition *all* is not met. Homogeneity H is similar but with another restriction, *all* and *only* vertices in a partition should be clustered together. Similarly, $H = 1$ for perfect clusterings and APC returns a homogeneity score $H = 0.588$ when $p = 0.4$ as *all* and *only* vertices in one partition satisfies this strict property.

The final metric we consider is the amalgamation of H and $Comp$ in the form of *V-measure*, an entropy-based measure which explicitly measures how successfully the criteria of homogeneity and completeness have been satisfied. Vmeasure is computed as

the harmonic mean of distinct homogeneity and completeness scores, just as precision and recall are commonly combined into F-measure [109, 115].

4.19.2 Planted l-Partition Graphs

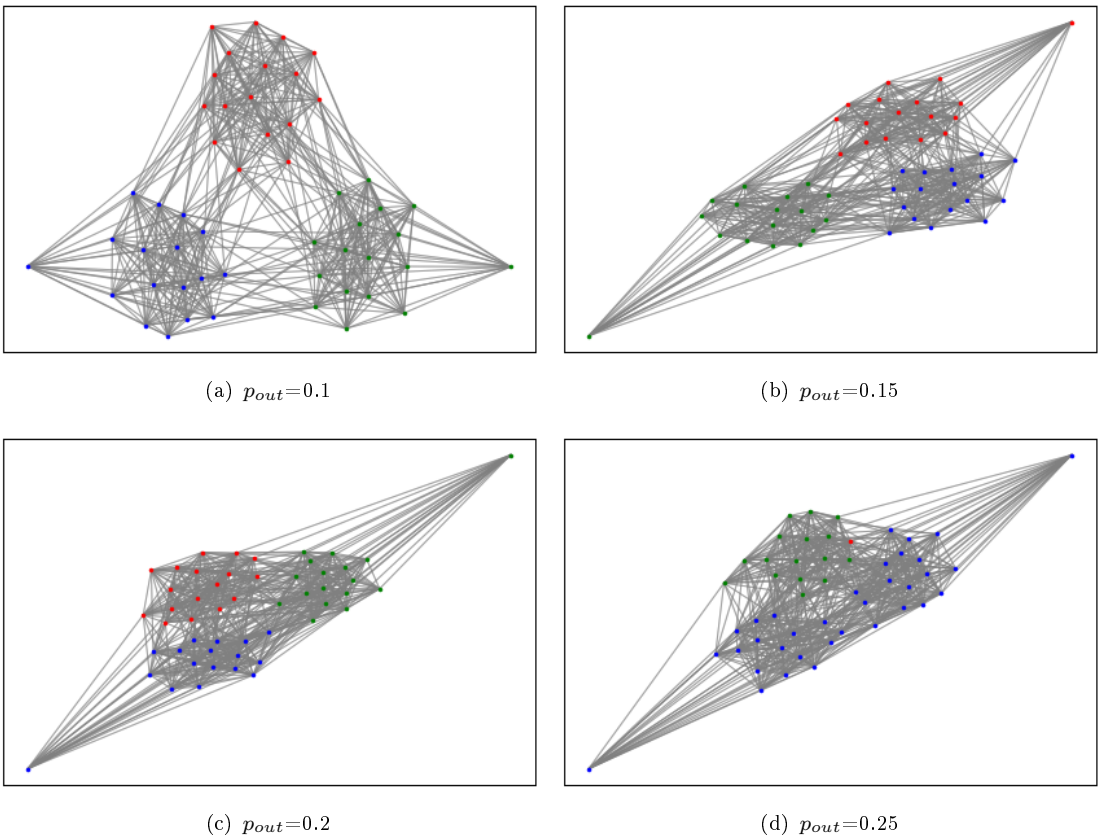
This type of graph is a benchmark graph used to evaluate clustering mechanisms, described in [63]. The implementation of the graphs discussed in this subsection were proposed by Condon et al [47]. The implementation in [47] is a linear-time algorithm to generate planted l-partition graphs (denoted as LPP). The fundamental attributes of such graphs is the partitioning of n vertices into l groups, each of size n/l with each n connected to vertices in the same partition with some probability p_{in}. Vertices are also connected to elements of other partitions with some probability $p_{out} < p_{in}$. The intra-cluster edge density should therefore exceed the inter-cluster edge density as this ensures a community structure in the graph exists. Each partition is a random sub-graph, generated using the Erdős-Rényi model with probability p_{in}. We show the performance of our algorithm on this type of graphs with different generation values for n, l, p_{in} and p_{out}. The graphs generated by this model are more complex than RCM as each vertex is considered in a pairwise fashion and a new edge is created relative to p_{out} as opposed to the probability of the edge being rewired.

The first configuration of this graph type we consider is $l = 3$ and $k = 17$, this is as not to overcomplicate visualisations. We start with l complete graphs as $p_{in} = 1$ and increase parameter p_{out} until truth labels are not correctly found.

Method	D	k	l	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	LPP	17	3	1	0.2	1.000	1.000	1.000	1000
GN	LPP	17	3	1	0.2	1.000	1.000	1.000	1.000
LP	LPP	17	3	1	0.2	1.000	1.000	1.000	1.000
APC	LPP	17	3	1	0.25	0.533	0.895	0.579	0.703
GN	LPP	17	3	1	0.25	1.000	1.000	1.000	1.000
LP	LPP	17	3	1	0.25	0.561	1.000	0.579	0.734
APC	LPP	17	3	1	0.35	0.002	0.232	0.041	0.069
GN	LPP	17	3	1	0.35	0.002	0.232	0.041	0.069
LP	LPP	17	3	1	0.35	0.000	1.000	0.000	0.000

TABLE 4.28: Planted l-Partition Graph Experiments

This model of graphs becomes complex and partitions less distinguishable quicker than RCM graphs due to new edges being added. Therefore, the graphs above in Figure 4.11 become increasingly well connected and more difficult to separate using force-directed graph drawing algorithms. As a result of this, as shown in Table 4.28, our method and LP begin to produce different clusterings than the defined truth labels when $p_{out} = 0.25$. Conversely, GN is still able to maintain accurate reproduction of clusters until $p_{out} = 0.35$ due to the benefits of recalculation of vertex betweenness. Our method



(a) p_{out}=0.1 (b) p_{out}=0.15

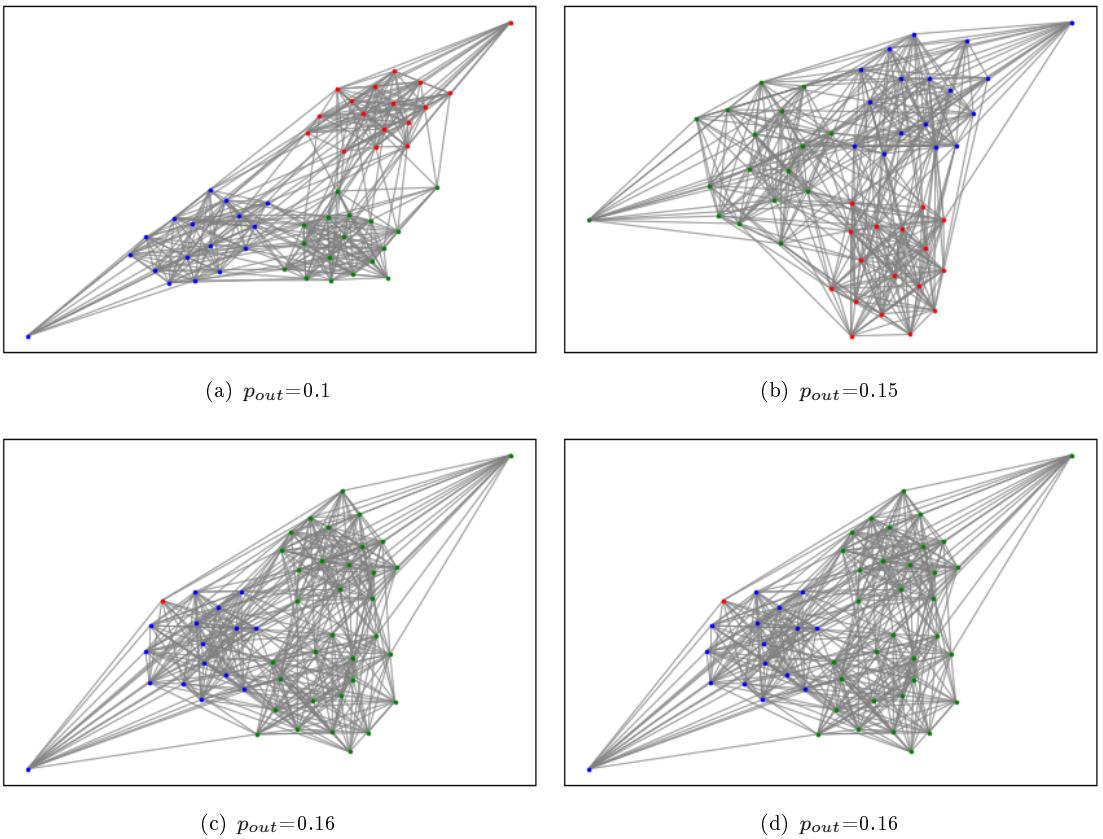
(c) p_{out}=0.2 (d) p_{out}=0.25

FIGURE 4.11: APC on Planted l-Partition Graph $p_{in} = 1$

and LP begin to encounter difficulty at $p_{out} = 0.25$, although partial clusterings are found as shown by ARI and V values. $Comp$ values for APC is again less than perfect due to the 1-vertex cluster artefact, whereas LP maintains $Comp = 1$. Conversely, H is identical between the two methods.

GN maintains cluster quality until $p_{out} = 0.35$, but the method did not find intermediary clusters, showing perfect evaluation metrics until a sudden drop. At the level of p_{out} all cluster methods perform similarly in evaluation metrics as mostly all vertices are in the same cluster. It is worth noting APC and GN have a similar 1-vertex artefact which results in a lower H score, unlike LP in which all vertices belong to the same cluster.

We now consider a different configuration of LPP where the l-partitions are not complete but still contain $l = 3$ groups and $k = 17$ vertices. We begin these experiments setting $p_{in} = 0.8$ and increase p_{out} incrementally.



(a) p_{out}=0.1 (b) p_{out}=0.15

(c) p_{out}=0.16 (d) p_{out}=0.16

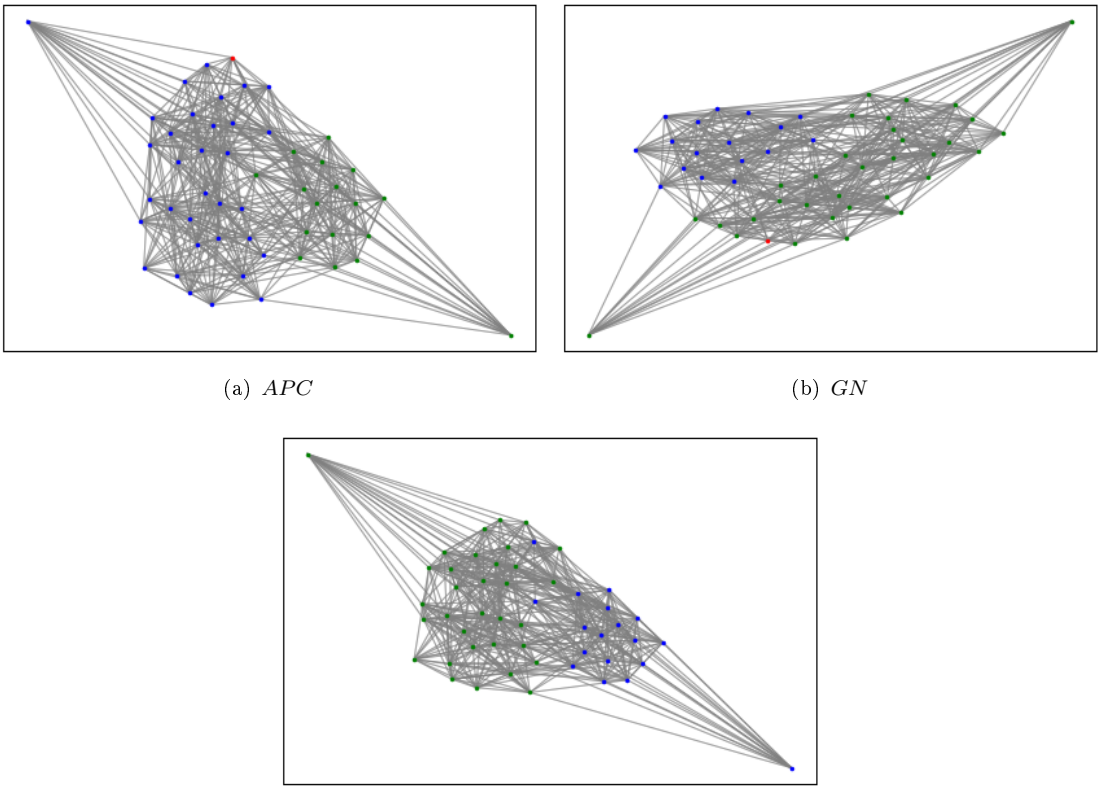
FIGURE 4.12: APC on Planted l-Partition Graph $p_{in} = 0.8$

Method	D	k	l	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	LPP	17	3	0.8	0.15	1.000	1.000	1.000	1.000
GN	LPP	17	3	0.8	0.15	1.000	1.000	1.000	1.000
LP	LPP	17	3	0.8	0.15	1.000	1.000	1.000	1.000
APC	LPP	17	3	0.8	0.16	0.533	0.895	0.579	0.703
GN	LPP	17	3	0.8	0.16	1.000	1.000	1.000	1.000
LP	LPP	17	3	0.8	0.16	0.561	1.000	0.579	0.734
APC	LPP	17	3	0.8	0.19	0.555	0.897	0.592	0.713
GN	LPP	17	3	0.8	0.19	0.555	0.897	0.592	0.713
LP	LPP	17	3	0.8	0.19	0.533	0.885	0.523	0.658

TABLE 4.29: Planted l-Partition Graph Experiments

As expected, shown in Figure 4.12 and Table 4.29 the accuracy of the resulting clusters relative to the truth labels depreciate at lower p_{out} values than the previous graph configuration due to the increased sparseness of interconnectivity. Our method and LP are unable to find the exact clustering when $p_{out} = 0.16$ with GN performing effectively until $p_{out} = 0.19$. An interesting observation at this point $p_{out} = 0.19$, our method returns the same clustering as GN, maintaining clustering consistency from

$p_{out} = 0.16$. The returned clusters for all three methods when $p_{out} = 0.19$ is shown in Figure 4.13.



(a) *APC* (b) *GN*

(c) *LP*

FIGURE 4.13: Planted l-Partition Graph $p_{in} = 0.8$ and $p_{out} = 1.9$

Our method maintains the same clustering from parameters in Subfigure (d) in Figure 4.12 to Figure 4.13 where all methods encounter difficulty.

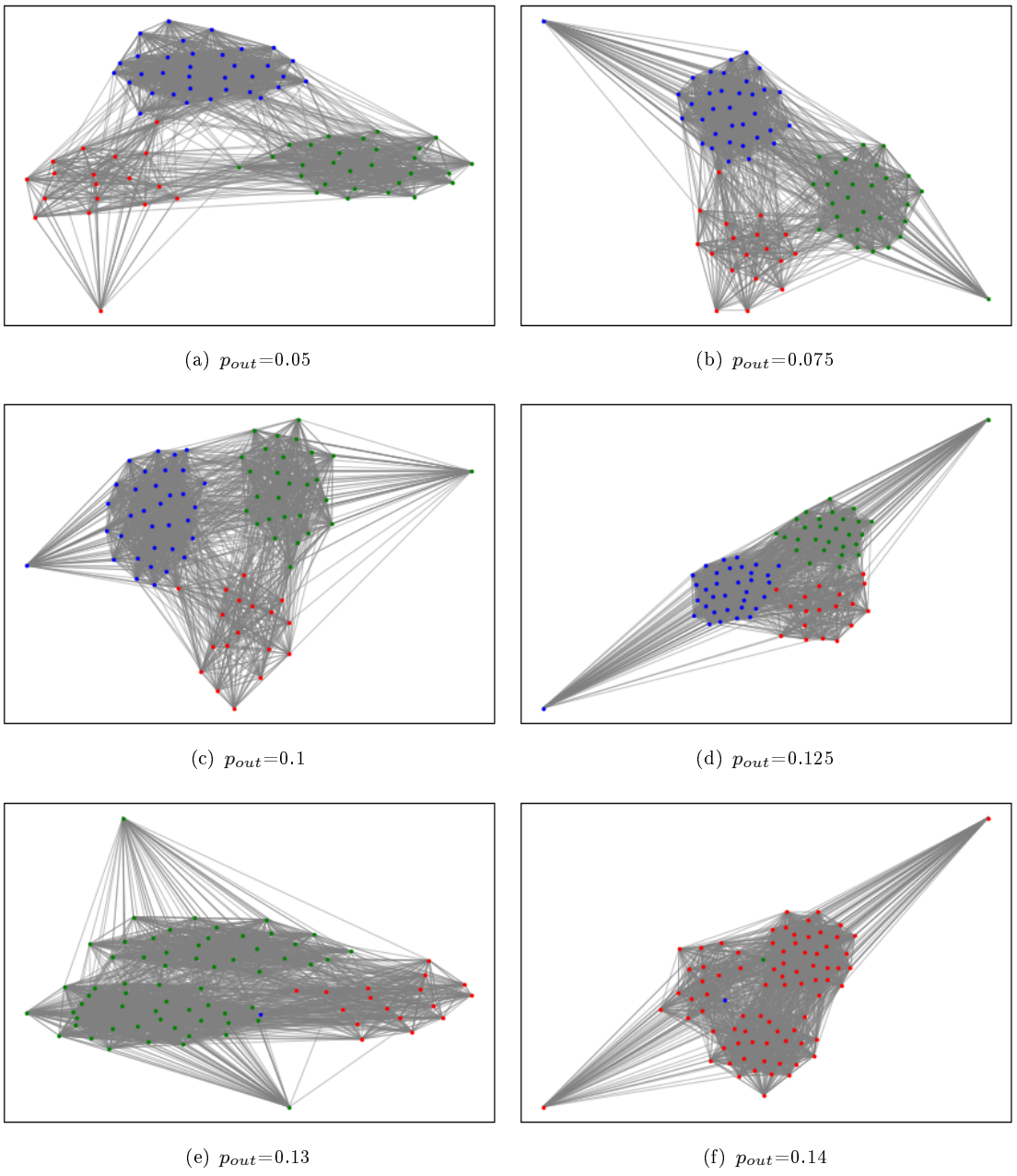
4.19.3 Gaussian Random Partition Graphs

The graphs created by the LPP model contain identical partitions by design, a feature unnatural to graphs of real systems. In such systems degree distributions are often skewed with vertices of varying degrees coexisting in the same partition. The graph generator in this section is a benchmark graph used to evaluate clustering mechanisms on graphs with similar features to real world systems. The implementation of the graphs discussed in this subsection were proposed by Brandes et al [43]. The implementation in [43] is an algorithm to generate gaussian random partition graphs (denoted as GRP) in which k partitions are created with sizes drawn from a normal distribution - vertices are similarly grouped to LPP using p_{in} and p_{out} parameters.

The first configuration we consider generates groups of varying sizes, specifying the number of vertices $n = 80$, the mean cluster size $s = 30$ and shape parameter $v = 3$, where the variance of the cluster sizes is s/v. Given these parameters GRP generates a

graph containing three partitions of sizes $p_1 = 34$, $p_2 = 18$ and $p_3 = 28$. These partitions are connected using using p_{in} and p_{out} parameters.

In the first graph, we create partitions using $p_{in} = 0.8$ increment p_{out} until the original partitions are no longer clearly found.



(a) p_{out}=0.05 (b) p_{out}=0.075

(c) p_{out}=0.1 (d) p_{out}=0.125

(e) p_{out}=0.13 (f) p_{out}=0.14

FIGURE 4.14: APC on Gaussian Random Partition Graph $p_{in} = 0.8$

Method	D	n	s/v	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	GRP	80	10	0.8	0.125	1.000	1.000	1.000	1.000
GN	GRP	80	10	0.8	0.125	0.931	0.925	0.909	0.917
LP	GRP	80	10	0.8	0.125	1.000	1.000	1.000	1.000
APC	GRP	80	10	0.8	0.13	0.436	0.917	0.500	0.647
GN	GRP	80	10	0.8	0.13	0.934	0.909	0.896	0.902
LP	GRP	80	10	0.8	0.13	1.000	1.000	1.000	1.000
APC	GRP	80	10	0.8	0.14	0.021	0.286	0.036	0.064
GN	GRP	80	10	0.8	0.14	0.931	0.925	0.909	0.917
LP	GRP	80	10	0.8	0.14	1.000	1.000	1.000	1.000
APC	GRP	80	10	0.8	0.17	0.021	0.286	0.036	0.064
GN	GRP	80	10	0.8	0.17	0.537	0.800	0.520	0.630
LP	GRP	80	10	0.8	0.17	0.622	1.000	0.607	0.755

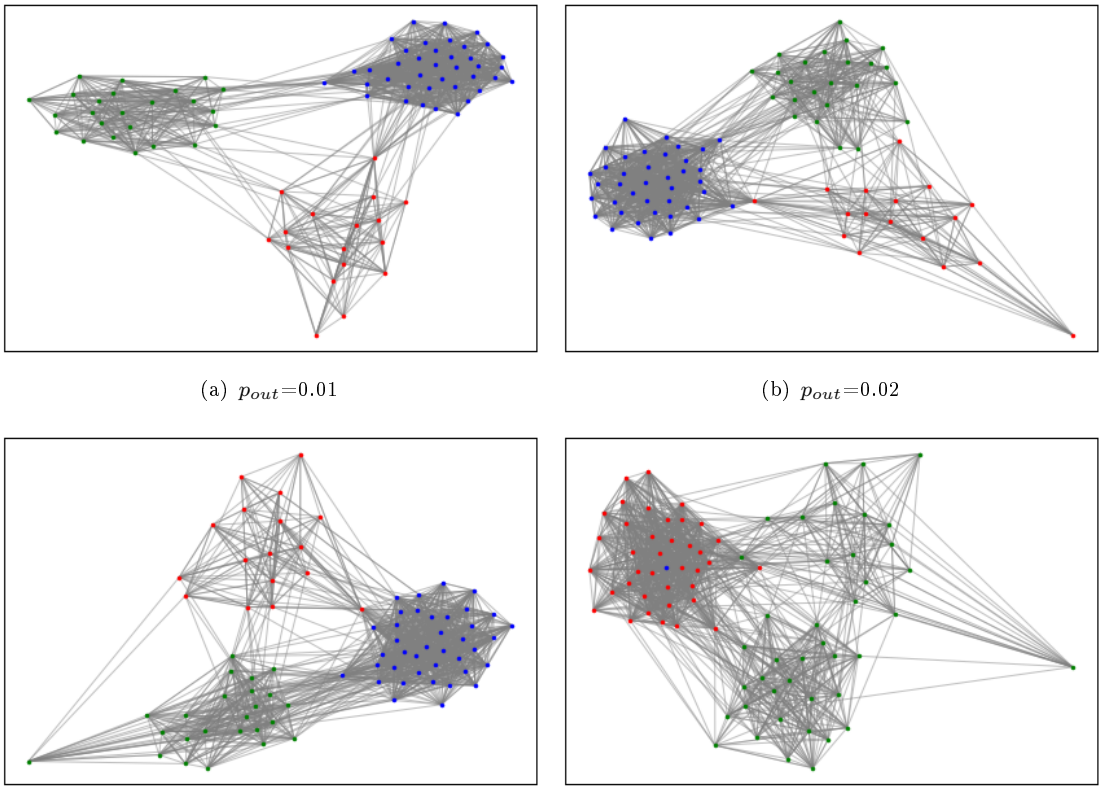
TABLE 4.30: APC on Gaussian Random Partition Graph $p_{in} = 0.8$

As expected the cluster algorithms cease to find the original partitions at lower values of p_{out} than the previous graph generators. A reason for this and a trait inherent in these topologies is that clusters are varying in sizes and the intra-cluster similarity is lower due to inconsistent cluster sizes. Smaller clusters will appear more sparse as there are comparatively more chances to create external edges than internal, creating difficult scenarios to discern partitions. The set of diagrams in Figure 4.14 show the performance of our method on graphs with increasing p_{out} and results in Table 4.30 contain the evaluation statistics of comparator methods.

Our method and *LP* find the correct partitions up to $p_{out} = 0.125$ as shown in the evaluation metrics. Surprisingly, *GN* misclassified one vertex into an incorrect clustering which explains the close to perfect evaluation scores, which highlights the issues with *GRP* generation. Our method encounters difficulty at $p_{out} = 0.13$, merging two clusters incorrectly - although as shown in Subfigure (e) in Figure 4.14 the merged partitions contain many overlapping edges. In this configuration of *GRP* graphs, our method classifies all vertices into one partition at $p_{out} = 0.14$ as shown in Subfigure (f). The partitions are very difficult to discern and partitions have many inter-cluster edges - this will produce similar vertex fitness values and increase the difficulty in separating clusters as this graph model can create difficult outliers, as shown by the inability of *GN* to discern entirely the correct partitions. The comparator methods *GN* and *LP* encounter issues at $p_{out} = 0.17$.

We consider another configuration to exacerbate the graph properties by reducing p_{in}. We specify the number of vertices $n = 80$, the mean cluster size $s = 30$ and shape parameter $v = 3$, where the variance of the cluster sizes is s/v. We also set $p_{in} = 0.5$ which generates a *GRP* graph containing three partitions of sizes $p_1 = 24$, $p_2 = 39$

and $p_3 = 17$. The resulting GRP is considerably more sparsely connected and again we increment p_{out} until we have a change in the resulting clustering.



(a) p_{out}=0.01 (b) p_{out}=0.02

(c) p_{out}=0.03 (d) p_{out}=0.035

FIGURE 4.15: APC on Gaussian Random Partition Graph $p_{in} = 0.5$

Method	D	n	s/v	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	GRP	80	10	0.5	0.03	1.000	1.000	1.000	1.000
GN	GRP	80	10	0.5	0.03	1.000	1.000	1.000	1.000
LP	GRP	80	10	0.5	0.03	1.000	1.000	1.000	1.000
APC	GRP	80	10	0.5	0.035	0.715	0.923	0.666	0.773
GN	GRP	80	10	0.5	0.035	1.000	1.000	1.000	1.000
LP	GRP	80	10	0.5	0.035	1.000	1.000	1.000	1.000
APC	GRP	80	10	0.5	0.057	0.003	0.213	0.027	0.049
GN	GRP	80	10	0.5	0.057	0.027	0.298	0.038	0.068
LP	GRP	80	10	0.5	0.057	0.022	0.293	0.045	0.078

TABLE 4.31: APC on Gaussian Random Partition Graph $p_{in} = 0.5$

The cluster sizes produced by the GRP models are heterogeneous and the degree distribution is also heterogeneous in that the expected degree of vertices depends on the number of vertices in the same cluster. Therefore, it is expected that clustering algorithms perform not so well on this difficult class of benchmark graphs. Although, the

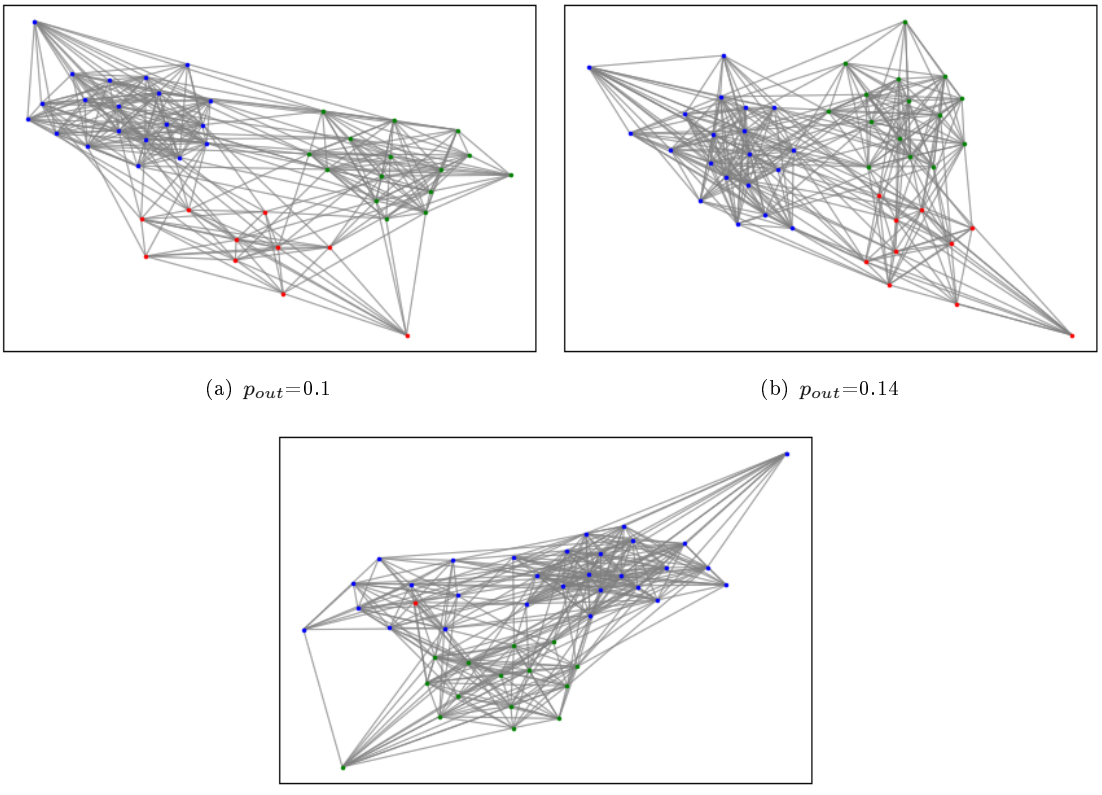
trend between our algorithm and comparator methods are consistent on this exacerbated configuration as shown in Figure 4.15 and in Table 4.31.

Our method begin to show signs of clusters being incorrectly merged at $p_{out} = 0.035$. The degradation in evaluation metrics is not as pronounced due to the higher variance in partition sizes as there are less vertices being incorrectly labelled.

4.19.4 Random Partition Graphs

In this section we consider Random Partition Graphs (denoted as RPG), a generalization of the planted-l-partition described in [63]. This model allows for the creation of groups of any size and therefore specify each partition exactly to generate more specific topologies. Using this model, graphs similar to those generated by GRP will be produced although with the caveat of exact control over the size of initial partitions. Using RPG we can demonstrate the symptomatic heterogeneous clusters and degree distributions.

Therefore, we demonstrate the performance of our method and comparator algorithms on RPG where $k = 3$ and the initial partitions $p_1 = 15$, $p_2 = 15$ and $p_3 = 20$. We also fix $p_{in} = 0.8$ and increment p_{out} until the original partitions are no longer returned.



(a) p_{out}=0.1 (b) p_{out}=0.14

(c) p_{out}=0.15

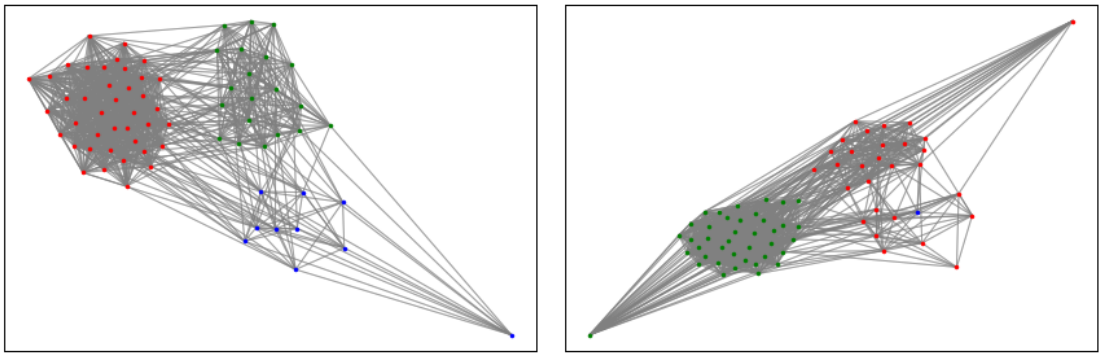
FIGURE 4.16: APC on Random Partition Graph $p_{in} = 0.8$

Method	D	n	$\|p_1\|,\|p_2\|,\|p_3\|$	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	RPG	80	10, 15, 20	0.8	0.14	1.000	1.000	1.000	1.000
GN	RPG	80	10, 15, 20	0.8	0.14	1.000	1.000	1.000	1.000
LP	RPG	80	10, 15, 20	0.8	0.14	1.000	1.000	1.000	1.000
APC	RPG	80	10, 15, 20	0.8	0.15	0.622	0.902	0.624	0.737
GN	RPG	80	10, 15, 20	0.8	0.15	0.936	0.931	0.916	0.923
LP	RPG	80	10, 15, 20	0.8	0.15	1.000	1.000	1.000	1.000
APC	RPG	80	10, 15, 20	0.8	0.16	0.683	0.895	0.659	0.759
GN	RPG	80	10, 15, 20	0.8	0.16	1.000	1.000	1.000	1.000
LP	RPG	80	10, 15, 20	0.8	0.16	0.607	1.000	0.600	0.750

TABLE 4.32: APC on Random Partition Graph $p_{in} = 0.8$

As the number of vertices is small, we have generated sparse graphs and show the resulting clustering in Figure 4.16 and evaluation metrics in Table 4.32. In this *RPG* graph, the variance in cluster sizes is increased by a small, constant value. Considering the difficulty of these graphs to discern the predefined partitions, our method performs well, identifying the smallest cluster with sparse intra-cluster edges and comparatively large inter-cluster edges. This class of graphs also creates difficult outlier vertices - this is shown in the configration of $p_{out} = 0.15$, our method incorrectly merges a sparse cluster with another and *GN* incorrectly classifies some vertices. *GN* proves to be less sensitive to this scenario as opposed to our more strict cluster building mechanism. The cluster quality of *APC* is maintained until $p_{out} = 1.6$ and finds similar clustering to *LP*.

We now demonstrate the performance of our method and comparator algorithms on another RPG configration where k = 3 and the initial partitions $p_1 = 10$, $p_2 = 20$ and $p_3 = 40$. We also set $p_{in} = 0.8$ and increment p_{out} until the original partitions are no longer returned.



(a) p_{out}=0.1 (b) p_{out}=0.11

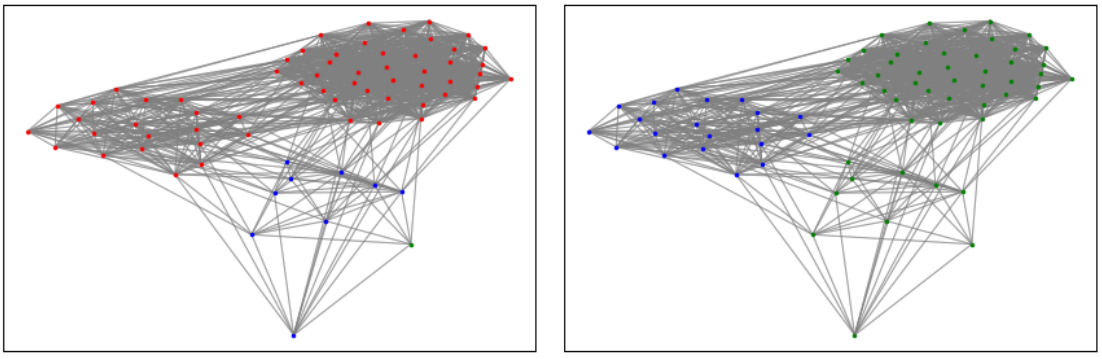
FIGURE 4.17: APC on Random Partition Graph $p_{in} = 0.8$

Method	D	n	$\lvert p_1\rvert,\lvert p_2\rvert,\lvert p_3\rvert$	p_{in}	p_{out}	ARI	$Comp$	H	V
APC	RPG	80	10, 20, 40	0.8	0.1	1.000	1.000	1.000	1.000
GN	RPG	80	10, 20, 40	0.8	0.1	0.959	0.950	0.930	0.940
LP	RPG	80	10, 20, 40	0.8	0.1	0.835	1.000	0.715	0.834
APC	RPG	80	10, 20, 40	0.8	0.11	0.843	0.938	0.732	0.822
GN	RPG	80	10, 20, 40	0.8	0.11	0.379	0.898	0.429	0.581
LP	RPG	80	10, 20, 40	0.8	0.11	0.678	1.000	0.626	0.770

TABLE 4.33: APC on Random Partition Graph $p_{in} = 0.8$

The configuration of the *RPG* graph in Figure 4.17 and Table 4.33 with doubling cluster sizes is to emphasize the point in that the cluster size does not impact the average out-degree. The difficulty inherent in these benchmark graphs is that at a certain threshold, clusters could be heuristically be classified as the same group. This is reinforced by the fact that the *GN* algorithm misclassifies a single vertex into another partition, contrary to the predefined truth labels.

Nevertheless, our algorithm and comparator algorithms perform well in identifying even the smallest cluster in this difficult, albeit unnatural graph. Based on the results in Table 4.33 our method out performs both *GN* and *LP*, though *GN* is only misclassifying a small number of vertices and the resulting clustering is still well defined and comparable - this is also true for *LP* but to a lesser extent.



(a) GN (b) LP

FIGURE 4.18: GN and LP on Random Partition Graph $p_{in} = 0.8$ and $p_{out} = 0.11$

In the event $p_{out} = 0.11$ all methods encounter difficulty in identifying the predefined partitions. In this configuration the smallest partition is indistinguishable due to the poor intra-cluster constitution. Interestingly (shown in Subfigure (b) in Figure 4.17 and Figure 4.18) *APC* merges the smallest partition with the second smallest, *LP* merges the smallest partition with the largest and *GN* merges the largest two clusters. The resulting size of the incorrect merges directly impacts the evaluation metrics in Table 4.33, i.e., *APC* maintains the highest scoring algorithm because the two smallest clusters are merged resulting in less vertices incorrectly classified.

In recent years new graph generators have been created to remedy the heterogeneous degree distribution and cluster sizes shown in the previous two experiments, known as LFR benchmark graphs. The defining property of these graphs is that degrees and communities are power laws with exponents τ_1 and τ_2, respectively [63]. Vertices then share a fraction of its edges with other members of the community and share another fraction of edges with vertices in other communities. The LFR benchmark graphs expand on these models in [84] introducing features of real networks, i.e., the heterogeneity in the distributions of node degree and community size.

4.20 Conclusion, Discussion and Future Work

In this chapter we have proposed a study of phylogenetic building rules to determine the viability in the context of clustering. We have modularised the process into three main components: constraint generation, constraint ranking and cluster building. The content of this chapter covers the evolution of our method from its early form, highly similar to standard phylogenetics to a highly flexible and augmented variant with multiple constraint generation functions, ranking procedure and building mechanism. We show our proposals, experiments, modifications and discussion of the methods successes and difficulties. Throughout our research and experimentation, we focused on the concepts of phylogenetics and relaxed these constraints as our method progressed, introducing multiple constraint generation functions and using these constraints to understand more about the graph structure as opposed to using them strictly syntactically. This process can be inherently distributed and can be parallelised to increase performance, with each vertex reporting its statistics and community fitness value, which can then be used in the cluster building phase.

The only area of phylogenetics that we did not relax is the tree building mechanism. The phylogenetic tree building process in its current form has proven to produce good clustering on a variety of benchmark graphs, but also has the potential to be too volatile to be used in the context of clustering. During the initial stages of the clustering procedures vertices are merged steadily, building clusters granularly. As the strongest constraints are used, the weaker triplets are considered which increases the likelihood of an undesirable triplet occurring and merging clusters which should be separate. Post evaluation we feel one triplet being interpreted in such a manner puts the onus of the resulting clustering on a single constraint (which is decidedly weak but the best of the remaining constraints in the set). Therefore, we propose various avenues of further research, which are plentiful and can all address this problem in various ways.

Firstly, the method we have proposed draws inspiration from phylogenetics and hierarchical clustering. The intention was to create an algorithm which did not need to calculate all pairwise similarities in the graph and instead focus on local information of each vertices neighbourhood. The unique feature of hierarchical clustering is that the pairwise similarities are updated subject to every merge performed - a step which is

not considered in our method. A possible area of experimentation would be to update the fitness values of a vertex given changes in the topology of the local neighbourhood, although this would be computationally expensive, understanding the resulting quality of clusters is worth considering. Using this method the cluster building function remains strict.

Secondly, we have compared our method against a near-linear time algorithm to solve clustering which is useful in large scale systems. This method, known as *label propagation* selects a vertex arbitrarily and begins propagating labels throughout the graph. The method is synchronous, with each vertex checking the majority of labels in the local neighbourhood - ties are broken randomly. This method is fast and produces best results on dense graphs, but the resulting clusters are fuzzy in nature. We believe a good route for further research is to focus on speed and cluster accuracy by creating a hybrid method that uses constraints to build clusters but check local neighbours to determine the majority before merging. In this processes the building process could be relaxed in that clusters are not strictly merged and instead vertices involved in a constraint can switch clusters.

Thirdly, we have considered the notion of *thresholding* for automatic termination instead of specifying a k value - knowing this value beforehand is a commodity not often granted in real systems. The Python code of our method already has this feature included, though it is beyond the scope of this chapter. The fundamental idea is to avoid the volatility of the building mechanism when constraints consist of *unfit* vertices. The intention would be to cluster vertices using the constraints up to a certain rank (threshold). The remaining vertices remain un-clustered but the primary partition centres will have already been defined. The remaining vertices can then be included through a k-nearest neighbour pass, or similarly label propagation.

Fourthly, subsequent to decreasing the volatility of the building mechanism earlier stages of the cluster merging process can be modified to approximate small cluster centers through the use of overlaying k-means with a granular number of centroids. This is based on the work in [39] in which the authors use this method to speed up the initial stages of hierarchical clustering, reducing the time needed to update similarities by approximating the cluster centers using k-means. In this work experiments are performed with different levels of k and assessing the overall quality produced by this method compared to standard hierarchical clustering.

Finally, the clustering algorithm is currently based in clustering graphs if the data can be represented as such. In machine learning, data samples often consist of multiple dimensions. Another area of adaptation would be to expand the algorithm to work on data of multiple dimensions.

Chapter 5

Conclusion

5.1 Overview

This thesis has presented multiple solutions and experiments to problems in the area of Distributed Computing and Clustering algorithms. The problems that we consider are grounded in determining consensus, interactions and local communities in networks. The models proposed throughout are influenced by other works in the field, using the concepts based in the areas of Random Walks, Population Protocols, Phylogenetics and Hierarchical Clustering.

The following results from Chapters 2 and 3 are from the published work carried out by the author, their supervisors and several collaborators from different institutions. Chapter 4 is experimental work carried out by the author and academic supervisors. The following sections are detailed conclusions obtained from the main results presented in this thesis.

5.2 Majority Color Problem

The work in Chapter 2 introduces various solutions to the Majority Color Problem on synchronous networks in restrictive models. The networks are unknown to the vertices and protocol, in that each vertex is aware of its edges, but unaware of the target vertex. The proposed protocols also operate in dynamic graphs where the adversary is benign. The protocols do not terminate but they will converge in finite time and we also show these protocols can terminate w.h.p if the value of n is known by the vertices.

Chapter 2 introduces the first protocol which is also the foundation of the other proposed solutions, to determine the consensus on the majority color in networks whose vertices are initially one of two colors. The protocol guarantees that if a majority color exists then each vertex will learn of the initial majority color. The initial algorithm in the original paper left an arbitrary distribution of colors throughout the graph if no majority existed. Based on further research the protocol was adapted such that if there exists no majority then each vertex will learn that there is equality in the network. The

protocol uses limited memory to solve this problem and disseminates the information to all vertices in the network.

We adapt this protocol and propose a variation in which multiple tokens can operate. The standard protocol is susceptible to deadlock, or concurrency issues in that if two agents each hold the last remaining token of a color then there will be no remaining vertex to complete a match, resulting in the vertices of the graph being infinitely recolored and never converging. The protocol is modified in such a fashion this problem is removed, with a small penalty to memory usage the process of solving MCP can be increased using multiple tokens.

Another protocol we propose is to solve the k-surplus problem, which disseminates the result across all vertices in the network if a majority of at least k is discovered. The original protocol can be reduced to this problem when $k = 2$. For any state where the token counter value $c < k$ then equality or neutrality is disseminated throughout the network, indicating there exists no k-surplus.

Finally we proposed a protocol for k-colors, an extension that uses the binomial coefficient $\binom{k}{2}$ instances of our original protocol. This modification is powerful in that not only the MCP is solved, but also other order statistics and computational tasks.

As a supplement to the theoretical models proposed in this chapter a software simulation of our protocols has been developed using Python that solves all problems defined in this chapter including equality, relative and absolute majority for k colors when $k \geq 2$. This software simulation is explained in Appendix B accompanied by small examples on various network topologies.

Interestingly, further research can be performed to study the MCP on non-trivial special classes on graphs as complete graphs can be solved in $O(n \log n)$ expected time and $O(n^2 m)$ time on any connected undirected graphs.

As a result of the utilization of random walks to solve majority problems as well as determine order statistics, similar to state transitions in population protocols, an avenue of research would be to determine whether all predicates of presburger arithmetic can be computed using the random walk model and determine the computational benefits of such a method.

Finally, another avenue of research would be to consider other variants of random walkers individually or working in synergy to solve the MCP problem in distributed, unknown networks whilst maintaining small memory usage.

5.3 Majority Problems in Population Protocols

The work in Chapter 3 discusses space-efficient deterministic population protocols for several variants of the *majority* problem including *plurality consensus*. We focused on space efficient majority protocols in populations with an arbitrary number of colours C represented by k-bit labels, where $k = \lceil \log C \rceil$. In particular, we presented asymptotically space-optimal (with respect to the adopted k-bit representation of colours) protocols for

(1) the *absolute majority* problem, i.e., a protocol which decides whether a single colour dominates all other colours considered together, and (2) the *relative majority problem*, also known in the literature as *plurality consensus*, in which colours declare their volume superiority versus other individual colours.

The new population protocols we proposed in this chapter rely on a *dynamic formulation* of the majority problem in which the colours originally present in the population can be changed by an *external force* during the communication process. The considered dynamic formulation is based on the concepts studied in [20] and [91] about *stabilizing inputs* and *composition of population protocols*. Also, the protocols presented in this chapter use a composition of some known protocols for static and dynamic majority.

Further work in regards to the work in this chapter would be to adapt our equality finding protocol to use less overall states and discover a space-optimal solution in regards to the number of states. Also in a wider context, in our solutions the emphasis was on asymptotic space optimality. One open problem, however, is to determine more exact bounds on the number of states required to compute the considered types of majorities for a given number of colours C. Another interesting problem refers to the time complexity and parallelism of considered majority problems in the presence of a random scheduler. Finally, one can ask what other computations are possible through a composition of several "partially self-stabilizing" (sub)protools.

5.4 Phylogenetic Clustering

The work in Chapter 4 introduces a clustering method that retains the hierarchical formation of the resulting partitions, as opposed to flat clustering methods were this information is lost. The method we proposed is a connectivity, edge based algorithm, although it could be augmented to work in other models depending on the distance metric used to determine vertex similarity. The method does not perform updates of the ranking mechanism as we rely on information derived from local communities. Therefore, our method shares similarities with hierarchical clustering, but also faster methods such as label propagation. The method performs well in a variety of graph types, predominantly in networks with well connected clusters as performance was shown to depreciate once a certain level of external edges had been reached.

We discuss the introduction of phylogenetic concepts and our algorithms parameters on a series of synthetic datasets. More detailed trace tables and clustering diagrams from early experiments are shown in Appendix C.

We proposed multiple ways in which constraints can be generated, which are graph dependent and result in different clusterings. A method in which constraints are generated through all pairwise comparisons between vertices and the local neighborhood and another which creates constraints based on the presence of local triangles in the graph. The first being more versatile as performance is maintained in sparse graphs and more information is created in dense graphs. The second performs comparatively well

in well-connected graphs, with the intention of removing the liklihood of troublesome triplets.

We propose a ranking algorithm to score and order the constraints accordingly based on a vertex fitness function - a function that benefits from negative and positive information equally. The method could further benefit from increased clustering accuracy if the ranking function is updated post merge, although a significant increase in computational efficiency will be incurred.

Finally, we compare our method with similar algorithms in the domain on set of real and synthetic data. We demonstrate the algorithms capabilities and evaluate using clustering specific evaluation metrics and clustering specific benchmark graphs, adhering to the call for standardisation in [63].

There are many avenues for potential research given the foundation laid in this chapter. We show that phylogenetic concepts can be adapted to detect communities in generic networks, and useful information can be derived from the generated constraints described as the notion of fitness.

The constraint generation methods can be inherently parallelized to further save computational time. The method can be further aligned with hierarchical clustering and other clustering methods in regards to updating the ranks of the constraints. Conversely, clustering accuracy can be increased utilising the proposed thresholding function to cluster highly similar vertices and post processing can be used in a similar way to label propagation, although clusters return may potentially be fuzzy.

The fitness function could potentially be adapted to aid in determining outliers in the network which isn't too dissimilar to the initial objectives - scoring these vertices low enough that the triplets containing two vertices positively is less likely to be used.

5.5 Final Remarks

The work in this thesis is presented in chronological order. Our first work was inspired by population protocols, how agents with limited memory and computational power can solve complex problems. We wanted to determine if these protocols can be adapted into a structured environment and communicate via walking entities. The results we produced with a random walking entity are promising as we show they are efficient in time and space to solve the MCP which is well studied in population protocols. We discovered combining multiple instances of our procedure allow us to solve more complex problems, such as relative and absolute majority for k colors. The results in this area ultimately inspired us to translate these findings into the context of population protocols, to solve the more complex problems by chaining processes as well as equality - which were our salient results in this area. Finally, our work in clustering although seemingly separate share similar motivations and concepts. Firstly, similarly to our random walk model to solve distributed problems, we consider structured environments. We also consider local relationships and interactions between vertices in the graphs, not global interactions

akin to other clustering methods. The local relationships between vertices contribute to forming clusters and local communities in our clustering method. Random walks are also used in determining cluster quality as an evaluation metric, as a random walker will more likely walk within a cluster as opposed to traversing external edges to another cluster.

Over the course of the research produced in this thesis we felt that we have contributed a set of new protocols and models that tackle majority problems in distributed systems utilising random walks and population protocols. We also hope that the work in this thesis prompts interest in researching further into random walk models as we have proven they are efficient in time and space to solve this set of problems. The model defined in the context of random walks should be researched further to discover if they can solve equivalent problems in population protocols. In the context of population protocols, much work has already been done and the models we propose allow for the chaining of dependent processes to solve more complex problems in which space is system constraint. Finally, we proposed a new clustering method in our work in this field. The method performed well given the benchmark pipelines defined in recent literature - more work needs to be done in this field to standardise evaluation processes of these mechanisms. We believe by pursuing concepts outlined in future work sections that our method will perform even stronger against comparable methods to find communities in difficult topologies.

Appendix A

Chapter Specific Definitions

The following chapter specific definitions are a concise glossary of terms used throughout the bodies of work. More technical definitions are found in the actual chapters themselves.

A.1 Match-Maker Majority Protocols

The definitions and notions described below are used in the work of Chapter 2.

Definition. Majority Color Problem: Consider an undirected, connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. Each node is colored one of two colors initially. With the restrictions of an unknown network and limited memory (a fixed number of bits per node and per token message), all nodes must eventually learn and agree on the initial majority color.

Definition. k-Surplus Problem: an adaptation of MCP with the additional caveat that there needs to be a majority color in the network that exceeds or is equal to k. Else, there is no k-surplus and each node learns and agrees on this fact.

Definition. Absolute Majority: The occurrence when an element has majority over all other elements combined, traditionally more than 50%.

Definition. Relative Majority: The occurrence when an element has the majority over all other elements in direct pairwise comparison, but does not achieve an absolute majority on its own.

Definition. Random Walk: The process of an entity simulating a walk based on random choice. Given an initial starting position in the network and a vertex degree d, every time interval the entity selects a random edge with equal probability $c = 1..d$ and traverses along it.

Definition. Cover Time: The time expected for a random walk to visit all nodes in a network at least once.

Definition. Token: A constant-sized message that maintains a state based on its interactions with network nodes - which is the entity that performs a random walk in the context of this work.

Definition. Dynamic Network: A network in which the edges between nodes can change during each time interval.

Definition. Benign Adversary: An adversary that can change the network structure within an allowable ruleset from round to round.

Definition. Convergence: All entities will converge on some correct value after some finite time.

A.2 Population Protocols

The definitions and notions described below are used in the work of Chapter 3, some definitions are also shared from the above definitions in Section A.1.

Definition. Population Protocol: A model that describes a collection of tiny mobile agents that interact with one another to carry out a computational task.

Definition. Agents: In this context agents are identically programmed finite state machines.

Definition. Plurality Consensus: A synonym of relative majority, studied in work regarding population protocols. The occurrence when an element has the majority over all other elements in direct comparison, but does not achieve an absolute majority on its own.

Definition. Finite State Machine: An abstract machine that can be in exactly one of a finite number of states at any given time.

Definition. Transition Function: Interactions between pairs of agents cause the two agents to update their states.

Definition. Adversary: These interactions defined by the transition function are scheduled by an adversary, subject to a fairness constraint.

Definition. Self-Stabilization: Input values are initially distributed to the agents, and the agents must eventually converge and stabilize to the correct output value (but not necessarily state).

A.3 Clustering

The definitions and notions described below are used in the work of Chapter 4.

Definition. Population: the total set of data points, represented as X in machine learning algorithms.

Definition. Labels: each sample has an associated label, the "'truth"' value, generally only provided in supervised learning algorithms and are used to compare algorithmic performance. Unsupervised algorithms do not initially have labels associated with the data and is the task of the algorithm to assign 'correct' labels to data points. Represented as y in machine learning algorithms.

Definition. Samples: a subset of the population.

Definition. Features: each sample (represented as a vector) in a population can have n dimensions, each one being a feature of the data.

Definition. Truth Label: the original label of a samples class/partition membership.

Definition. Partition: the original partitioning of data used as *truth labels* to determine clustering accuracy in evaluation procedures.

Definition. Clustering: the task of grouping data points or objects together that are similar, by varying definitions of similarity. These groupings are called clusters, communities or segments.

Definition. Similarity: the score which determines the similarity of two elements in a dataset.

Definition. Agglomerative: a bottom-up approach of grouping data points. Each data point initially starts in its own group, merged iteratively together to form increasingly larger groupings.

Definition. Divisive: a top-down approach of grouping data points. Each data point initially starts in one collective group, split iteratively into sub sets to form increasingly granular groupings.

Definition. K-Nearest Neighbor: a non-parametric algorithm used to determine the k nearest neighbors to some point in space.

Definition. Phylogenetics: an area of biology to study the evolutionary relationships between biological species.

Definition. Triplet: a ternary relationship between three data points in the form $((A, B), C)$ which implies A and B are more similar than C.

Definition. Forbidden Triplet: a ternary relationship between three data points in the form $((A, B), C)$ which implies A and B are more similar than C that contradicts another triplet, for example $((A, C), B)$.

Definition. Fan Triplet: a ternary relationship between three data points in the form (A, B, C) which implies A, B and C are equally similar.

Appendix B

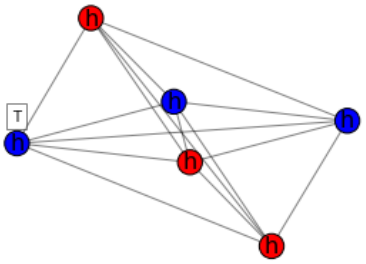
Random Walk Majority Experiments

This section demonstrates an example execution of the augmented variants of *BASIC* on a variety of graph types. The simulation uses graph generators to create various topologies, including complete, lollipop, barbell, star and line, though only examples using complete and line are shown here. The simulation also allows for the color distribution to be in left-right, sequential color lines (i.e. for k colors the first n/k elements are colored k_1, second n/k elements are colored k_2 etc.) or randomly distributed. The code for the simulation can be found at the URL in Reference [8].

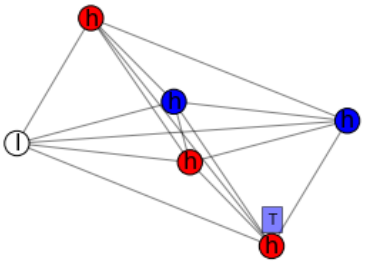
The figures in the following sections trace the execution from the initialization step (step 0), when the token t is placed at an arbitrary vertex, to the state when all vertices have been converted to the initial absolute majority or relative majority color. Each figure consists of the step number, the state of t in terms of the color that is being disseminated and the state of all the vertices (uppercase representing high influence and lowercase representing low influence).

B.1 Complete 6 Vertex: Graph: Equality

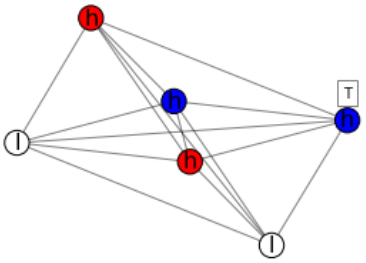
Here we consider a complete graph, where n = 6 and k = 2. There exists an equality among vertices in the graph. The following figures are displayed in order of the execution. There is no *IC* controller in this configuration as there is only one instance of *BASIC* with equality.



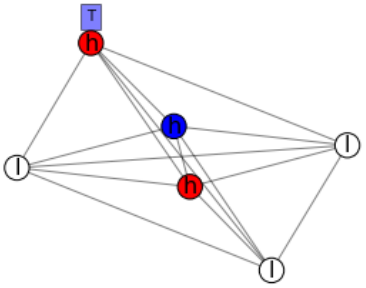
(a) Step 1



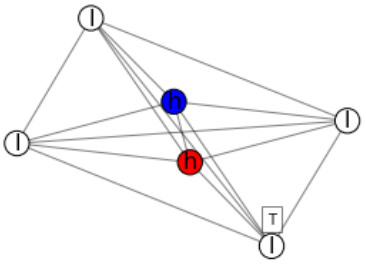
(b) Step 2



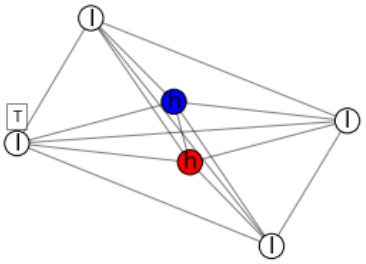
(c) Step 3



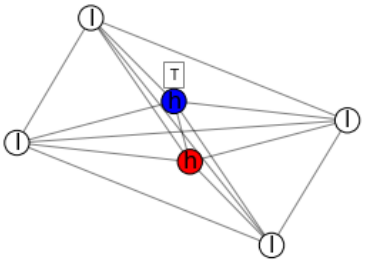
(d) Step 4



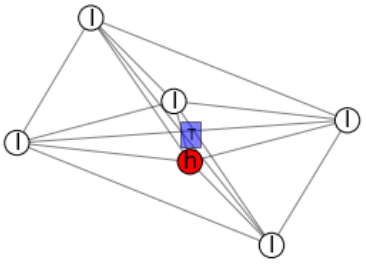
(e) Step 5



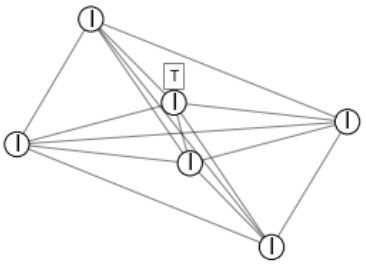
(f) Step 6



(g) Step 7



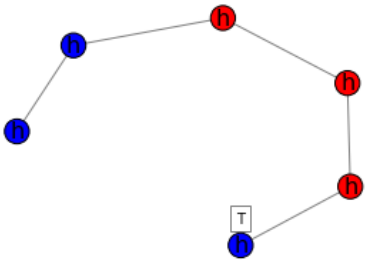
(h) Step 8



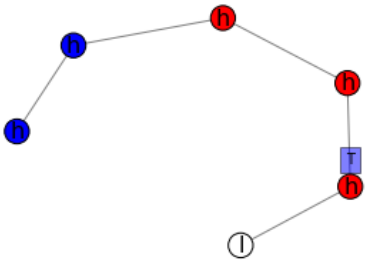
(i) Step 9

B.2 Path 6 Vertex Graph: Equality

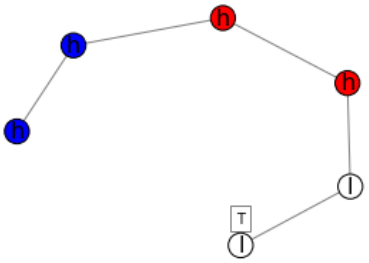
Here we consider a line graph or path with LR coloring mode, where n = 6 and k = 2. There exists an equality among vertices in the graph. The following figures are displayed in order of the execution. There is no IC controller in this configuration as there is only one instance of $BASIC$ with equality.



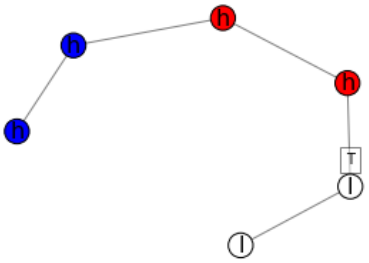
(a) Step 1



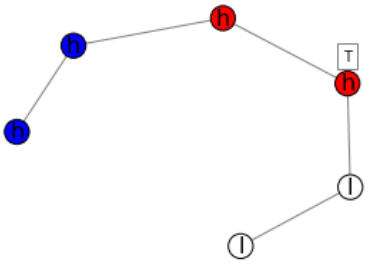
(b) Step 2



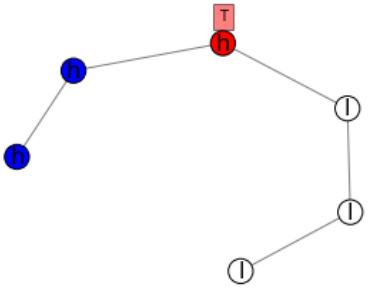
(c) Step 3



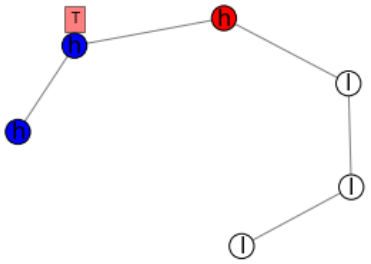
(d) Step 4



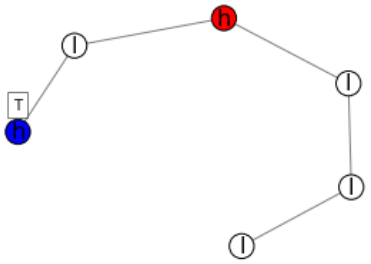
(e) Step 5



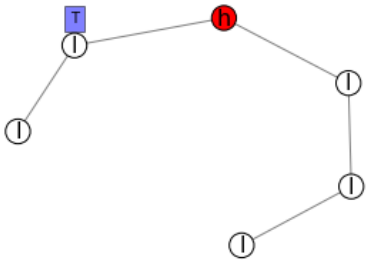
(f) Step 6



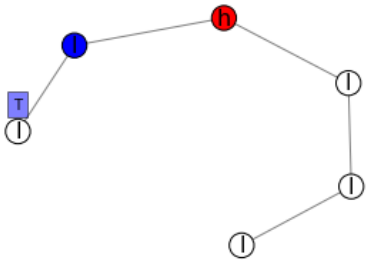
(g) Step 7



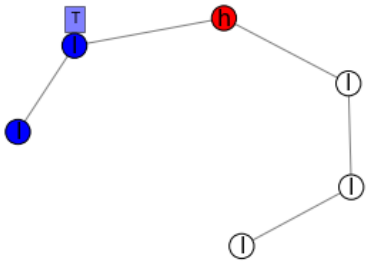
(h) Step 8



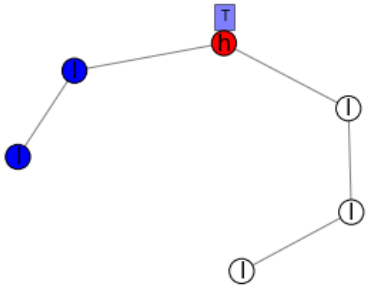
(i) Step 9



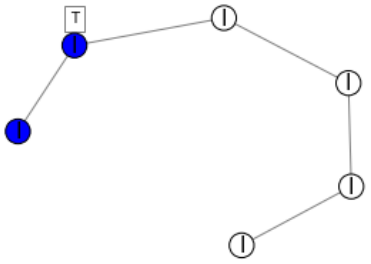
(j) Step 10



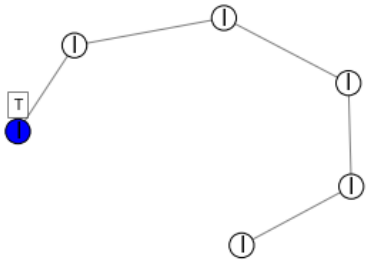
(k) Step 11



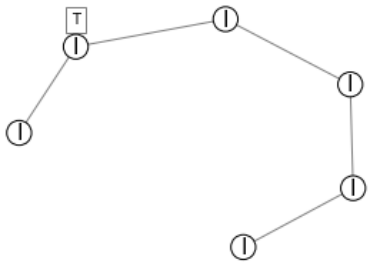
(l) Step 12



(m) Step 13



(n) Step 14



(o) Step 15

Appendix C

Appendix C

C.1 Experiment 2b - Full Trace Table

Step	T_{res}	Sim(T)	C
0	((0, 3), 2)	0.516	{0, 3}{1}{2}{4}{5}{6}{7}{8}{9}{10}{11}
1	((0, 3), 4)	0.516	{0, 3}{1}{2}{4}{5}{6}{7}{8}{9}{10}{11}
2	((1, 3), 4)	0.316	{0, 1, 3}{2}{4}{5}{6}{7}{8}{9}{10}{11}
3	((2, 3), 4)	0.316	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
4	((0, 3), 8)	0.516	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
5	((1, 3), 8)	0.316	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
6	((2, 3), 8)	0.316	{0, 1, 2, 3}{4}{5}{6}{7}{8}{9}{10}{11}
7	((4, 7), 3)	0.516	{0, 1, 2, 3}{4, 7}{5}{6}{8}{9}{10}{11}
8	((4, 6), 3)	0.316	{0, 1, 2, 3}{4, 6, 7}{5}{8}{9}{10}{11}
9	((4, 6), 8)	0.316	{0, 1, 2, 3}{4, 6, 7}{5}{8}{9}{10}{11}
10	((4, 7), 8)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8}{9}{10}{11}
11	((8, 9), 3)	0.316	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9}{10}{11}
12	((4, 7), 6)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9}{10}{11}
13	((8, 11), 3)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 11}{10}
14	((8, 11), 4)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 11}{10}
15	((0, 3), 1)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 11}{10}
16	((8, 10), 4)	0.316	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 10, 11}
17	((8, 11), 9)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 10, 11}
18	((8, 11), 10)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 10, 11}
19	((4, 7), 5)	0.516	{0, 1, 2, 3}{4, 6, 7}{5}{8, 9, 10, 11}
20	((4, 5), 8)	0.316	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}
21	((4, 5), 3)	0.316	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}
22	((8, 10), 3)	0.316	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}
23	((8, 9), 4)	0.316	{0, 1, 2, 3}{4, 5, 6, 7}{8, 9, 10, 11}

TABLE C.1: Experiment 2b Trace Table No Fans

Step	T_{res}	Sim(T)	C
-	-	-	{0}, {1}, {2}, {3} {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
1	((0, 3), 1)	0.516	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
1	((0, 3), 2)	0.516	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
2	((0, 3), 4)	0.516	{0, 3}, {1}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
3	((1, 3), 4)	0.316	{0, 1, 3}, {2}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
4	((2, 3), 4)	0.316	{0, 1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
5	((0, 3), 8)	0.516	{0, 1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
6	((1, 3), 8)	0.316	{0, 1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
7	((2, 3), 8)	0.316	{0, 1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}
8	((4, 7), 3)	0.516	{0, 1, 2, 3}, {4, 7}, {5}, {6}, {8}, {9}, {10}, {11}
9	(4, 5, 6)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8}, {9}, {10}, {11}
10	((4, 7), 6)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8}, {9}, {10}, {11}
11	((4, 6), 8)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8}, {9}, {10}, {11}
12	((4, 7), 8)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8}, {9}, {10}, {11}
13	((8, 9), 3)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9}, {10}, {11}
14	((4, 5), 8)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9}, {10}, {11}
15	((8, 11), 3)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 11}, {10}
16	((8, 11), 4)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 11}, {10}
17	(0, 1, 2)	0.408	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 11}, {10}
18	((8, 10), 4)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
19	(8, 9, 10)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
20	((8, 11), 10)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
21	((8, 11), 9)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
22	(1, 2, 3)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
23	(9, 10), 11)	0.408	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
24	((4, 5), 3)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
25	((4, 6), 3)	0.316	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
26	((4, 7), 5)	0.516	{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}
27	(3, 4, 8)	0.200	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
28	(5, 6, 7)	0.408	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
29	((8, 10), 3)	0.316	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
30	((8, 9), 4)	0.316	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

TABLE C.2: APC(PCG, FORB, FAN) Trace Table

Bibliography

[1] *NetworkX asynchronous label propagation*, https://networkx.readthedocs.io/en/latest/reference/algorithms.community.html#module-networkx.algorithms.community.asyn_lpal, Accessed: 2017-01-23.

[2] *NetworkX documentation*, http://networkx.readthedocs.io/en/latest/, Accessed: 2016-01-23.

[3] *NetworkX gaussian random partition*, https://networkx.github.io/documentation/development/reference/generated/networkx.generators.community.gaussian_random_partition_graph.html#networkx.generators.community.gaussian_random_partition_graph, Accessed: 2017-01-23.

[4] *NetworkX girvan newman clustering*, https://networkx.readthedocs.io/en/latest/reference/generated/networkx.algorithms.community.centrality.girvan_newman.html, Accessed: 2017-01-23.

[5] *NetworkX planted l-parition*, https://networkx.github.io/documentation/development/reference/generated/networkx.generators.community.planted_partition_graph.html#networkx.generators.community.planted_partition_graph, Accessed: 2017-01-23.

[6] *NetworkX random parition*, https://networkx.github.io/documentation/development/reference/generated/networkx.generators.community.random_partition_graph.html, Accessed: 2017-01-23.

[7] *Sci-kit Learn machine learning datasets*, http://scikit-learn.org/stable/datasets/, Accessed: 2016-01-23.

[8] *Thesis experimental simulations*, http://csc.liv.ac.uk/~ddh/thesis/experiments, Accessed: 2017-03-03.

[9] C. C. Aggarwal and C. K. Reddy, *Data clustering : Algorithms and applications.*, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series: v.31, Boca Raton : CRC Press, 2013., 2013.

[10] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman, *Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions*, SIAM Journal on Computing **10** (1981), no. 3, 405–421.

[11] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász, and C. Rackoff, *Random walks, universal traversal sequences, and the complexity of maze problems*, Proc. 20th Annual Symposium on Foundations of Computer Science, 1979, pp. 218–223.

[12] D. Alistarh, J. Aspnes, D. Eisenstat, R. Gelashvili, and R.L. Rivest, *Time-space trade-offs in population protocols*, CoRR abs/1602.08032, 2016.

[13] D. Alistarh and R. Gelashvili, *Polylogarithmic-time leader election in population protocols*, Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP), 2015, pp. 479–491.

[14] D. Alistarh, R. Gelashvili, and M. Vojnovic, *Fast and exact majority in population protocols*, Proc. 33rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2015, pp. 47–56.

[15] D. Alistarh, J. Aspnes, D. Eisenstat, R. Gelashvili, and R L. Rivest, *Time-space trade-offs in population protocols*, CoRR **abs/1602.08032** (2016).

[16] D. Alistarh and R. Gelashvili, *Polylogarithmic-time leader election in population protocols*, pp. 479–491, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[17] N. Alon, C. Avin, M. Koucký, G. Kozma, Z. Lotker, and M.R. Tuttle, *Many random walks are faster than one*, Combinatorics, Probability & Computing **20** (2011), no. 4, 481–502.

[18] N. Alon, J.H. Spencer, and P. Erdös, *The probabilistic method*, Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley, 1992.

[19] Amazon, *Amazon Web Services*, `http://aws.amazon.com/`, 2016, Accessed 2017-04-05.

[20] D. Angluin, J. Aspnes, M. Chan, M.J. Fischer, H. Jiang, and R. Peralta, *Stably computable properties of network graphs*, Proc. 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), 2005, pp. 63–74.

[21] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta, *Computation in networks of passively mobile finite-state sensors*, Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), 2004, pp. 290–299.

[22] D. Angluin, J. Aspnes, M. Chan, M.J. Fischer, H. Jiang, and R. Peralta, *Computation in networks of passively mobile finite-state sensors*, Distributed Computing **18(4)** (2006), 235–253.

[23] D. Angluin, J. Aspnes, and D. Eisenstat, *A simple population protocol for fast robust approximate majority*, Distributed Computing **21** (2008), no. 2, 87–102 (English).

[24] D. Angluin, J. Aspnes, and D. Eisenstat, *A simple population protocol for fast robust approximate majority*, Distributed Computing **21** (2008), no. 2, 87–102.

[25] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert, *The computational power of population protocols*, Distributed Computing **20** (2007), no. 4, 279–304.

[26] D. Angluin, J. Aspnes, M.J. Fischer, and H. Jiang, *Self-stabilizing population protocols*, ACM Trans. Auton. Adapt. Syst. **3** (2008), no. 4, 1–28.

[27] D. Angluin, J. Aspnes, Zoë Diamadi, M. J. Fischer, and R.Peralta, *Computation in networks of passively mobile finite-state sensors*, Distributed Computing **18** (2006), no. 4, 235–253.

[28] Apache, *Distributed Processing of Large Datasets*, `http://hadoop.apache.org/`, 2017, (Accessed 2017-03-03.

[29] A. Arenas, A. Diaz-Guilera, and C. J. Pérez-Vicente, *Synchronization reveals topological scales in complex networks*, Physical review letters **96** (2006), no. 11, 114102.

[30] L. Gąsieniec, D.D. Hamilton, R. Martin, and P.G. Spirakis, *The match-maker: Constant-space distributed majority via random walks*, Proc. 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), 2015, pp. 67–80.

[31] J. Augustine, G. Pandurangan, and P. Robinson, *Distributed computing: 29th international symposium, disc 2015, tokyo, japan, october 7-9, 2015, proceedings*, ch. Fast Byzantine Leader Election in Dynamic Networks, pp. 276–291, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[32] C. Avin, M. Koucký, and Z. Lotker, *How to explore a fast-changing world (cover time of a simple random walk on evolving graphs)*, Automata, Languages and Programming (L. Aceto and et al, eds.), Lecture Notes in Computer Science, vol. 5125, Springer Berlin Heidelberg, 2008, pp. 121–132 (English).

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

[34] L. Becchetti, A. Clementi, E. Natale, F. Pasquale, R. Silvestri, and L. Trevisan, *Simple dynamics for plurality consensus*, Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2014, pp. 247–256.

[35] L. Becchetti, A.E.F. Clementi, E. Natale, F. Pasquale, and R. Silvestri, *Plurality consensus in the gossip model*, Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2015, pp. 371–390.

[36] M.A. Bender and D.K. Slonim, *The power of team exploration: Two robots can learn unlabeled directed graphs*, Proc. 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 75–85.

[37] P. Berenbrink, T. Friedetzky, G. Giakkoupis, and P. Kling, *Efficient plurality consensus, or: The benefits of cleaning up from time to time*, Proc. 43rd International Colloquium on Automata, Languages, and Programming (ICALP), 2016, pp. 1–14.

[38] Y. Birk, L. Liss, A. Schuster, and R. Wolff, *A local algorithm for ad hoc majority voting via charge fusion*, Distributed Computing (Rachid Guerraoui, ed.), Lecture Notes in Computer Science, vol. 3274, Springer Berlin Heidelberg, 2004, pp. 275–289 (English).

[39] A Bouguettaya, Q. Yu, X. Liu, X. Zhou, and A. Song, *Efficient agglomerative hierarchical clustering*, Expert Systems with Applications **42** (2015), no. 5, 2785–2797.

[40] J. M. Bower and H. Bolouri, *Computational modeling of genetic and biochemical networks (computational molecular biology)*, The MIT Press, 2004.

[41] J. M. Bower and H. Bolouri, *Computational modeling of genetic and biochemical networks*, MIT Press, 2004.

[42] U. Brandes, *A faster algorithm for betweenness centrality*, Journal of mathematical sociology **25** (2001), no. 2, 163–177.

[43] U. Brandes, M. Gaertler, and D. Wagner, *Experiments on graph clustering algorithms*, European Symposium on Algorithms, Springer, 2003, pp. 568–579.

[44] I. Chatzigiannakis, O. Michail, S. Nikolaou, and P.G. Spirakis, *The computational power of simple protocols for self-awareness on graphs*, Theoretical Computer Science **512** (2013), 98–118.

[45] H.-L. Chen, R. Cummings, D. Doty, and D. Soloveichik, *Speed faults in computation by chemical reaction networks*, Distributed Computing (2014), 16–30.

[46] H.-L. Chen, R. Cummings, D. Doty, and D. Soloveichik, *Speed faults in computation by chemical reaction networks*, pp. 16–30, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[47] A. Condon and R. M. Karp, *Algorithms for graph partitioning on the planted partition model*, Random Structures and Algorithms **18** (2001), no. 2, 116–140.

[48] C. Cooper, R. Elsaesser, and T. Radzik, *The power of two choices in distributed voting*, Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP), 2014, pp. 435–446.

[49] Intel Corporation, *Shareholder Report*, `http://files.shareholder.com/downloads/INTC/867590276x0xS50863-16-105/50863/filing.pdf`, 2016, [Online; accessed 21-April-2016].

[50] Microsoft Corporation, *Azure*, `https://azure.microsoft.com/en-gb/?WT.srch=1&WT.mc_ID=SEM_WS1bucqG`, 2016, [Online; accessed 21-April-2016].

[51] J. Czyzowicz, L. Gąsieniec, D.D. Hamilton, E. Kranakis, R. Martin, and P.G. Spirakis, *Match Maker: Journal*, TBD.

[52] L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas, *Comparing community structure identification*, Journal of Statistical Mechanics: Theory and Experiment **2005** (2005), no. 09, P09008.

[53] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert, *When birds die: Making population protocols fault-tolerant*, IEEE 2nd Intl. Conference on Distributed Computing in Sensor Systems (DCOSS), Lecture Notes in Computer Science, vol. 4026, Springer-Verlag, 2006, pp. 51–56.

[54] B. Doerr, L.A. Goldberg, L. Minder, T. Sauerwald, , and C. Scheideler, *Stabilizing consensus with the power of two choices*, Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2011, pp. 149–158.

[55] D. Doty and D. Soloveichik, *Stable leader election in population protocols requires linear time*, Proc. 29th International Symposium on Distributed Computing (DISC), 2015, pp. 602–616.

[56] D. Doty, *Timing in chemical reaction networks*, Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), SODA '14, Society for Industrial and Applied Mathematics, 2014, pp. 772–784.

[57] D. Doty and D. Soloveichik, *Distributed computing: 29th international symposium, disc 2015, tokyo, japan, october 7-9, 2015, proceedings*, ch. Stable Leader Election in Population Protocols Requires Linear Time, pp. 602–616, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[58] M. Draief and M. Vojnovic, *Convergence speed of binary interval consensus*, SIAM Journal on Control and Optimization **50** (2012), no. 3, 1087–1109.

[59] M. Draief and M. VojnoviÄĞ, *Convergence speed of binary interval consensus*, SIAM Journal on Control and Optimization **50** (2012), no. 3, 1087–1109.

[60] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, *Randomized multivalued consensus*, Proc. of Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001., 2001, pp. 195–200.

[61] U. Feige, *Collecting coupons on trees, and the analysis of random walks*, Technical report CS93-20 of the Weizmann Institute, 1993.

[62] W. Feller, *An introduction to probability theory and its applications : Vol 1, 3rd ed*, Wiley, New York, 1968.

[63] S. Fortunato, *Community detection in graphs*, Physics Reports **486** (2010), no. 3–5, 75 – 174.

[64] E.B. Fowlkes and C.L. Mallows, *A method for comparing two hierarchical clusterings*, Journal of the American statistical association **78** (1983), no. 383, 553–569.

[65] P. Fraigniaud and E. Natale, *Noisy rumor spreading and plurality consensus*, Proc. 34th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2016, pp. 127–136.

[66] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1, Springer series in statistics Springer, Berlin, 2001.

[67] L. Gąsieniec, D.D. Hamilton, R. Martin, and P.G. Spirakis, *Stabilization, safety, and security of distributed systems: 17th international symposium, SSS 2015, proceedings*, ch. The Match-Maker: Constant-Space Distributed Majority via Random Walks, pp. 67–80, Springer International Publishing, Cham, 2015.

[68] L. Gąsieniec, D.D. Hamilton, R. Martin, P.G. Spirakis, and G. Stachowiak, *Deterministic Population Protocols for Exact Majority and Plurality*, 20th International Conference on Principles of Distributed Systems (OPODIS 2016), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[69] M. Ghaffari and M. Parter, *A polylogarithmic gossip algorithm for plurality consensus*, Proc. 34th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2016, pp. 117–126.

[70] S. Ginsburg and E.H. Spanier, *Semigroups, presburger formulas, and languages*, Pacific J. Mathematics **16** (1966), 2434–2450.

[71] M. Girvan and M.E.J. Newman, *Community structure in social and biological networks*, Proceedings of the national academy of sciences **99** (2002), no. 12, 7821–7826.

[72] M.X. Goemans, *Lecture notes on bipartite matchings*, `http://www-math.mit.edu/~goemans/18433S09/matching-notes.pdf`, 2009.

[73] S. Gregory, *An algorithm to find overlapping community structure in networks*, European Conference on Principles of Data Mining and Knowledge Discovery, Springer, 2007, pp. 91–102.

[74] I.A. Targio Hashem, I. Yaqoob, N. Badrul Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, *The rise of "big data" on cloud computing: Review and open research issues*, Information Systems **47** (2015), 98 – 115.

[75] R. M. Henzinger, V. King, and T. Warnow, *Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology*, Algorithmica **24** (1999), no. 1, 1–13.

[76] P. Holme, M. Huss, and H. Jeong, *Subnetwork hierarchies of biochemical pathways*, Bioinformatics **19** (2003), no. 4, 532–538.

[77] L. Hubert and P. Arabie, *Comparing partitions*, Journal of Classification **2** (1985), no. 1, 193–218.

[78] B.D. Hughes, *Random walks and random environments: Random walks*, Oxford science publications, no. v. 1, Clarendon Press, 1995.

[79] B.D. Hughes, *Random walks and random environments: Random walks*, Oxford science publications, Clarendon Press, 1996.

[80] IBM, *What is big data?*, `http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`, 2016, [Online; accessed 21-April-2016].

[81] S. Ikeda, I. Kubo, and M. Yamashita, *The hitting and cover times of random walks on finite graphs using local degree information*, Theoetical Computer Science **410** (2009), no. 1, 94–100.

[82] J. Jansson, A. Lingas, and E.M. Lundell, *The approximability of maximum rooted triplets consistency with fan triplets and forbidden triplets*, pp. 272–283, Springer International Publishing, Cham, 2015.

[83] S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan, *On the complexity of universal leader election*, J. ACM **62** (2015), no. 1, 7:1–7:27.

[84] A. Lancichinetti, S. Fortunato, and F. Radicchi, *Benchmark graphs for testing community detection algorithms*, Physical review E **78** (2008), no. 4, 046110.

[85] I. Letunic and P. Bork, *Interactive tree of life v2: online annotation and display of phylogenetic trees made easy*, Nucleic Acids Research **39** (2011), no. suppl_2, W475.

[86] L. Levine and Y. Peres, *Spherical asymptotics for the rotor-router model in $\mathcal{Z}^d$*, arXiv:math/0503251.

[87] P. Lind and M. Alm, *A database-centric virtual chemistry system*, Journal of Chemical Information and Modeling **46** (2006), no. 3, 1034–1039, PMID: 16711722.

[88] M. Meila, *Comparing clusterings–an information based distance*, Journal of Multivariate Analysis **98** (2007), no. 5, 873 – 895.

[89] G.B. Mertzios, S. E. Nikoletseas, C.L. Raptopoulos, and P.G. Spirakis, *Determining majority in networks with local interactions and very small local memory*, Automata, Languages, and Programming (J. Esparza and et al, eds.), Lecture Notes in Computer Science, vol. 8572, Springer Berlin Heidelberg, 2014, pp. 871–882 (English).

[90] G.B. Mertzios, S.E. Nikoletseas, C. Raptopoulos, and P.G. Spirakis, *Determining majority in networks with local interactions and very small local memory*, Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP), 2014, pp. 871–882.

[91] O. Michail, I. Chatzigiannakis, and P.G. Spirakis, *New models for population protocols*, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2011.

[92] O. Michail and P.G. Spirakis, *Simple and efficient local codes for distributed stable network construction*, Proc. 32nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2014, pp. 76–85.

[93] O. Michail, I. Chatzigiannakis, and P.G. Spirakis, *New models for population protocols*, Synthesis Lectures on Distributed Computing Theory **2** (2011), no. 1, 1–156.

[94] Y. Mocquard, E. Anceaume, J. Aspnes, Y. Busnel, and B. Sericola, *Counting with population protocols*, Proc. 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA), 2015, pp. 35–42.

[95] E.F. Moore, *Gedanken-experiments on sequential machines*, Automata Studies (C.E. Shannon and J. McCarthy, eds.), Princeton University Press, 1956, pp. 129–153.

[96] A. Mostefaoui, M. Raynal, and F. Tronel, *From binary consensus to multivalued consensus in asynchronous message-passing systems*, Information Processing Letters **73** (2000), no. 5-6, 207 – 212.

[97] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, New York, NY, USA, 1995.

[98] M.E.J. Newman, *A measure of betweenness centrality based on random walks*, Social networks **27** (2005), no. 1, 39–54.

[99] M.E.J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Physical review E **69** (2004), no. 2, 026113.

[100] S. Pandey and S. Nepal, *Cloud computing and scientific applications âĂŤ big data, scalable analytics, and beyond*, Future Generation Computer Systems **29** (2013), no. 7, 1774 – 1776, Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services &; Cloud Computing and Scientific Applications âĂŤ Big Data, Scalable Analytics, and Beyond.

[101] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*, Ch 11, exercise 5(c), Prentice-Hall, Inc. NJ, USA, 1982.

[102] E. Perron, D. Vasudevan, and M. Vojnovic, *Using three states for binary consensus on complete graphs*, Proc. 28th Conference on Computer Communications (INFOCOM), 2009, pp. 2527–2535.

[103] E. Perron, D. Vasudevan, and M. Vojnovic, *Using three states for binary consensus on complete graphs.*, INFOCOM, IEEE, 2009, pp. 2527–2535.

[104] I PRESENT, *Cramming more components onto integrated circuits*, Readings in computer architecture **56** (2000).

[105] V.B. Priezzhev, D. Dhar, A. Dhar, and S. Krishnamurthy, *Eulerian walkers as a model of self-organized criticality*, Phys. Rev. Lett. **77** (1996), 5079–5082.

[106] U. Nandini Raghavan, R. Albert, and S. Kumara, *Near linear time algorithm to detect community structures in large-scale networks*, Phys. Rev. E **76** (2007), 036106.

[107] W.M. Rand, *Objective criteria for the evaluation of clustering methods*, Journal of the American Statistical association **66** (1971), no. 336, 846–850.

[108] S. Raschka, *Python machine learning*, Packt Publishing, 2015.

[109] A. Rosenberg and J. Hirschberg, *V-measure: A conditional entropy-based external cluster evaluation measure.*, EMNLP-CoNLL, vol. 7, 2007, pp. 410–420.

[110] A. L. Samuel, *Some studies in machine learning using the game of checkers*, IBM Journal of Research and Development **44** (2000), no. 1.2, 206–226.

[111] J.M. Santos and M. Embrechts, *On the use of the adjusted rand index as a metric for evaluating supervised classification*, pp. 175–184, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[112] A. Das Sarma, A. Rahaman Molla, and G. Pandurangan, *Distributed computation in dynamic networks via random walks*, Theoretical Computer Science **581** (2015), 45 – 66.

[113] S. S Skiena, *The algorithm design manual*, Springer-Verlag London, New York, NY, 2008.

[114] A. Telcs, *The art of random walks*, Lecture Notes in Mathematics, Springer, 2006.

[115] C.J. Van Rijsbergen, *A non-classical logic for information retrieval*, The computer journal **29** (1986), no. 6, 481–485.

[116] N. Xuan Vinh, J. Epps, and J. Bailey, *Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance*, J. Mach. Learn. Res. **11** (2010), 2837–2854.

[117] D.J. Watts, *Small worlds: the dynamics of networks between order and randomness*, Princeton university press, 1999.

[118] Y. Zhao and G. Karypis, *Criterion functions for document clustering: Experiments and analysis*, Tech. report, Citeseer, 2001.

[119] H. Zhou, *Distance, dissimilarity index, and network community structure*, Physical review e **67** (2003), no. 6, 061901.

[120] H. Zhou and R. Lipowsky, *Network brownian motion: A new method to measure vertex-vertex proximity and to identify communities and subcommunities*, International conference on computational science, Springer, 2004, pp. 1062–1069.