The Efficiency of Distributed Gossip Protocols

Joshua Oliver

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

October 7, 2021

Abstract

The goal of a gossip protocol is to spread information throughout a network of agents by means of one-to-one phone calls. Each agent has a corresponding piece of information, and initially this piece of information is only known by its respective agent. During phone calls, agents divulge all of their known information to the other party [5], and in the case of dynamic gossip agents also exchange the phone numbers that they know [21]. We will analyse five gossip protocols (ANY, CO, LNS, TOK and SPI) which each impose different restrictions when randomly selecting and executing calls in a distributed system. In particular, we wish to compare these protocols based on their efficiency. This includes the time taken to execute the protocol as well as how many calls are required to reach an all-expert state in which all agents know all possible secrets [18]. We also consider how often the execution of a protocol results in an all-expert state. We can then discuss which protocol is optimal given details of how fast they can successfully terminate, and how often they are able to do so. This analysis will be done using results produced by a gossip protocol simulator that we will design and implement in a high-level programming language. This simulator will be comprised of three main parts. Firstly we need to be able to randomly generate graphs that represent the network of agents; this will include complete networks, incomplete (connected) networks, and (weakly connected) digraphs in the case of dynamic gossip. These graphs will be used as experiment data. We then need to implement the protocol simulator which is able to perform a specified protocol on a generated graph, and report key findings such as the computation time and how many calls were performed. Finally we must be able to combine the previous two features in order to perform experiments on random graphs, and the results will be used to evaluate the suitability of the protocols.

Student Declaration

SIGNATURE Joshua Oliver
DATE October 7, 2021

Acknowledgments

I would like to thank my project supervisor, Louwe Kuijer. He has provided guidance and support throughout the process of completing this dissertation. This includes giving helpful suggestions and constructive criticism.

I would also like to thank my family and friends. In the context of the pandemic, I have been able to study under these unusual circumstances because of their encouragement and support.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Gossip Problem

The gossip problem describes a scenario in which n agents (or "gossips") each know a piece of information known as their "secret". An agents secret is different to the secrets of all other agents, and (initially) this secret is only known by its respective agent. The main goal of the gossip problem is to spread all available information throughout the network of agents. This is achieved via "one-to-one" telephone calls during which both agents reveal all of their known secrets to the other party. Once an agent has learnt all possible secrets, we can refer to them as an "expert" [18], and thus we wish to execute calls until all agents are experts [5].

Gossip can be visualised via a communication network which uses mathematical graph theory to represent possible calls between agents [14]. For example, if there exists an edge between nodes (agents) a and b, then a and b know each others phone numbers respectively. The earliest and most widely researched strand of gossip involves communication networks that are *complete*, and thus every agent within the network is familiar with the phone number of all others [5, 9, 19]. Incomplete graph topologies such as trees [14] and sun graphs [21] have also been studied.

Dynamic gossip is such that agents reveal all of the phone numbers that they know during calls as well as their known secrets. We can represent a dynamic communication network via a *digraph* such that an arc from agent a to b means that a has the phone number of b [20, 21]. An important distinction here is that an arc from a to b does not imply that b knows the phone number of a, unlike on complete and incomplete *un-directed* networks, where an edge between a and b means that there is mutual knowledge of each others phone number. As calls are executed and phone numbers are distributed, a dynamic communication network becomes closer to being complete as extra arcs are added.

1.2 Example

Consider the complete communication network featured in figure 1.1. We have five agents labelled a, b, c, d and e who each know pieces of information A, B, C, D and E respectively. The sequence of calls given in table 1.1 results in an all-expert state. The cells of the table show what secrets are known by the agents after the call in the corresponding row. Here a call between agents a and b is denoted ab [7]. We achieve an all-expert state in the minimum number of calls on a complete network of 5 agents according to the following theorem proposed by Baker and Shostak:

Theorem 1.2.1 *Denote $f(n)$ as the minimum number of calls required for a network of n agents. Then $f(n) = 2n - 4$ for $n \geq 4$ [9].*

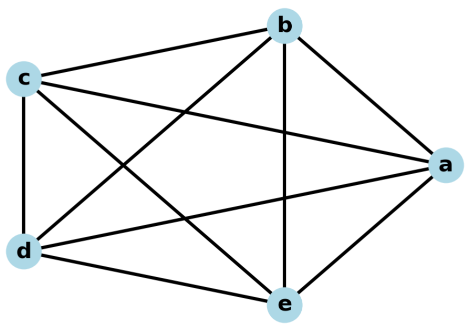


Figure 1.1: Communication network on five agents (produced using NetworkX [1]).

This minimum can be achieved via the following procedure:

1. Designate four “chief gossips”.
2. Choose one chief gossip to call the remaining $n-4$ gossips.
3. It is a known result that $f(4)=4$. Thus the four chief gossips can learn each other’s information in a further 4 calls. Since one of the chief gossips knows the information of the remaining $n-4$ agents, all chief gossips are now experts.
4. Finally, one of the chief gossips can proceed to call the remaining $n-4$ agents once more, in turn making all of them experts.
5. Thus in total there have been $(n-4)+4+(n-4)=2n-4$ calls.

Indeed this procedure is not a rigorous proof that $f(n)=2n-4$, though such proofs do exist [9].

Calls	a	b	c	d	e
	A	B	C	D	E
ae	AE	B	C	D	AE
ad	ADE	B	C	ADE	AE
bc	ADE	BC	BC	ADE	AE
ab	$ABCDE$	$ABCDE$	BC	ADE	AE
cd	$ABCDE$	$ABCDE$	$ABCDE$	$ABCDE$	AE
ae	$ABCDE$	$ABCDE$	$ABCDE$	$ABCDE$	$ABCDE$

Table 1.1: Example of the distribution of information in a 5 agent network [7].

1.3 Distributed Gossip Protocols

We act as a “central authority” in section 1.2 [21], and have complete control over the scheduling of calls to achieve an all-expert state in the most efficient manner possible. This paper will instead focus on *distributed* gossip [8] where the execution of calls is modelled as a random process and chosen calls must satisfy a pre-defined protocol condition. This random process can be carried out in one of two ways:

1. Randomly select an agent who can make a permitted call, followed by randomly choosing a call for that agent to make.
2. Randomly select a call to execute given a full list of permitted calls [19].

This project will use the second method. Furthermore, we will focus on five such gossip protocols:

- ANY - "Any protocol". We impose no extra restrictions on the calls that can be made. As long as a has the phone number of b, then a is able to call b.
- CO - "Call once". Two agents may only communicate with each other on one single occasion during the protocol execution. This means that if a were to call b, then the calls ab and ba are no longer permitted.
- LNS - "Learn new secrets". a may only call b if a does not know b's secret.
- TOK - "Token". Each agent starts the execution with a token that enables them to make a call. When agent a calls agent b, a loses its token, and b gains a token if it does not have one already.
- SPI - "Spider". Agents each start with a token that enables them to make a call. When agent a calls agent b, b loses its token should it possess one. Agents cannot retrieve another token once they have lost their original [18].

1.4 Related Work

Discrete mathematicians introduce the gossip problem indirectly assuming a complete network topology, and view the solution for minimising the number of calls required to reach an all-expert state from the perspective of a central controller. In particular, rigorous proofs are provided for Theorem 1.2.1 [9, 13, 17]. Harary and Schwenk introduce the concept of a *communication network*, and also explore the idea that a complete topology cannot always be assumed in a gossip scenario. A notable result from their contribution is the minimum number of calls required for a *tree* ($2n - 3$), and a connected graph that contains a quadrilateral ($2n - 4$). They also introduce the idea of "one-way-communications" in which information is only passed from one party of the call to the other; a situation which can be visualised via a *strongly connected digraph* [14]. More relevant to this project, Attamah et al provide a logical framework for epistemic gossip protocols that dispose of the need for a central controller [8]. Apt et al design a formal framework for distributed gossip protocols which is then used to argue about their correctness and termination [5]. Ditmarsch et al address which distributions of secrets are reachable by ANY, CO and LNS when assuming a complete network topology. Simulation results are also used to explore the expectation of LNS and CO [19]. Ditmarsch et al characterize various outcomes in terms of success when executing a distributed gossip protocol. The focus of their contribution is on dynamic gossip, for which they explore various network topologies and adapt the definitions of well-known protocols for the dynamic case [20, 21].

1.5 Outline and Contribution

This project will provide a design for a distributed gossip protocol simulator in Python. This will then be used to provide simulation results for ANY, CO, LNS, TOK and SPI on complete and incomplete gossip graphs, and dynamic gossip digraphs. We include both sequential gossip results, and round based gossip. We will report key findings such as average execution lengths for a set number of agents, average computation times, number of rounds required to reach an all-expert state, and success rates. Chapter 2 gives a background to the gossip problem, including definitions of concepts that will be reflected in the simulator. Chapter 3 gives an overview of the design of the system, and how the system will be evaluated. Chapter 4 explains how the design was implemented, including key changes that were made to the original plan during this process, and discusses how the software was tested. Chapter 5 provides the results that were produced by the simulator, including graphs of average execution lengths, number of rounds, and boxplots to show the spread of execution lengths on a single gossip graph. Chapter 6 gives an evaluation of the software and the results, as well as a discussion of the knowledge and skills that were acquired throughout the project. Finally, chapter 7 concludes the project by summarising the main simulation results and suggesting routes for further research.

Chapter 2

Background

This chapter will provide sufficient background to the gossip problem and distributed gossip protocols. All definitions and results are taken or adapted from gossip literature. Where available, these results can be later used to compare with claims made within this dissertation. As commonly used in other literature, we will denote agents by lower-case letters (a, b, c, ...), and their respective secrets by the upper-case equivalent. Referring to an arbitrary agent can also be done by using a lower case letter with a subscript (e.g. a_i) [18].

2.1 Calls and Call sequences

Ditmarsch et al formalise the concept of secret distributions:

Definition 2.1.1 (Distribution of Secrets) *For each agent a_i, we assign a set of known secrets $S_{a_i} \subseteq \{A_1, A_2, ..., A_n\}$ where $\{A_1, A_2, ..., A_n\}$ is the set of all secrets. For n agents, the distribution of all secrets can be written as the tuple $S = (S_{a_1}, ..., S_{a_n})$.*

Initially $S_{a_i} = A_i$ is true for all agents a_i before the execution of a protocol, and an agent is an expert given $S_{a_i} = \{A_1, ..., A_n\}$ [18]. We can also define the distribution of phone numbers in a similar way, where $N_{a_i} \subseteq \{a_1, ..., a_i, ..., a_n\}$ is the set of agents whose phone number is known by a_i [21]. All agents know their own phone number.

Apt et al highlight three possibilities for the exchange of information during a call:

- **Push-pull.** Both the caller and the callee divulge their known information during the call.
- **Push.** Only the caller divulges their information during the call.
- **Pull.** Only the callee divulges their information during the call [5].

This project is concerned with the *push-pull* method, although the push and pull methods are possible routes for extension. Ditmarsch et al define a call as an "*ordered pair* (a, b) with $a \neq b$". Note that it is important for this pair to be ordered since our protocols have conditions that specify the caller and callee (the first and second element of the pair respectively).

Definition 2.1.2 (Call) *A call is an ordered pair (a, b) with $a \neq b$ such that a and b exchange all of their known secrets (and phone numbers if applicable). Therefore after the call (a, b), S_a and S_b both become $S_a \bigcup S_b$ (and N_a and N_b both become $N_a \bigcup N_b$ in the dynamic case) [18].*

A call sequence is a series of calls to be executed sequentially, often denoted in the literature by σ [21]. Say we have call sequence σ made up of calls $(a, b), (c, d), (a, e)$.

Then we can write $\sigma = ab; cd; ae$. We can then use this notation to describe the effect of a call sequence on the set of secrets known by a particular agent. For example, from an initial state ($S_a = \{A\}$), we can apply call sequence σ as follows:

$$S_a^\sigma = \{A\}^{ab;cd;ae} = \{AB\}^{cd;ae} = \{AB\}^{ae} = \{ABE\}$$

We can also apply the same notation to a full distribution of secrets [18]:

$$S = (S_{a_1}, ..., S_{a_n}) \implies S^\sigma = (S_{a_1}^\sigma, ..., S_{a_n}^\sigma)$$

2.2 Gossip Situation and Gossip Graphs

Gossip graph definitions will be adapted from [21]. We will also make a distinction between a gossip graph and a gossip *digraph.*

2.2.1 Undirected Gossip Graphs

We use an undirected network in the non-dynamic case to represent the distribution of known phone numbers. The nodes of the graph represent agents, and the edge (a, b) exists if agents a and b know each others phone numbers.

Definition 2.2.1 (Undirected Gossip Graph) *An undirected gossip graph is a double $G = (A, E)$ such that $A = \{a_1, ..., a_n\}$ is the set of nodes (agents), and E is the set of all unordered pairs of agents (a, b) such that $a \in N_b$ and $b \in N_a$.*

There is no distinction between an initial (undirected) gossip graph and a gossip graph after a sequence of calls since phone numbers are not being exchanged.

2.2.2 Directed Gossip Graph

The dynamic scenario is represented via a directed graph since the knowledge of phone numbers between two agents is not always mutual. Agent a knows the phone number of agent b iff there exists an arc (a, b) in the graph.

Definition 2.2.2 (Directed Gossip Graph) *A directed gossip graph is a double $G = (A, O)$ such that $A = \{a_1, ..., a_n\}$ is the set of nodes (agents), and O is the set of all ordered pairs of agents (a, b) such that $b \in N_a$.*

Here we add extra arcs as agents learn new phone numbers. We can denote the effect of a call (i, j) on gossip digraph $G = (A, O)$ as

$$G^{ij} = (A, O^{ij}),$$

where $O \subseteq O^{ij}$ and O^{ij} is the set of ordered pairs of agents (a, b) such that $b \in N_a$ after the call ij has been executed. We informally define an initial gossip digraph G as a graph that has had no arcs added as a result of a call [21].

2.2.3 Gossip Situation

Instead of defining a gossip graph as a triple $G = (A, N, S)$ that incorporates information on the distribution of secrets like in [20], we instead define a gossip situation as a (directed or undirected) gossip graph paired with a mapping from the set of agents A to the secret distribution S.

Definition 2.2.3 (Gossip Situation) *A gossip situation is an (un)directed gossip graph $G = (A, E)$, paired with a mapping $f : A \longrightarrow S$ such that $A = \{a_1, ..., a_n\}$ is the set of agents, $S = (S_{a_1}, ..., S_{a_n})$ is the distribution of secrets, and $f(a_i) = S_{a_i}$ for all $i = 1, ..., n$. This is an initial gossip situation if $S_x = \{X\}$ for all agents $x \in A$ [6], and no arcs of G have been added as a result of a previous call.*

We denote a gossip situation as a pair (G, S) where G is a gossip (di)graph and S is the corresponding secret distribution.

2.3 Permitted Call Sequences and Protocol Success

The informal definitions of the gossip protocols given in section 1.3 are sufficient for the computational implementation. We will however give a simplification of a permitted call sequence. Take a call sequence σ and an arbitrary call $ab \in \sigma$. Define call sequence ϵ as the calls of σ up to (but not including) ab.

Definition 2.3.1 (P-Permitted Call Sequence) *Call sequence σ is p-permitted in gossip situation (G, S) if each call $ab \in \sigma$ satisfies the condition set out by protocol P in gossip situation (G^ϵ, S^ϵ). The call sequence σ is permitted in general if σ is p-permitted, and for each call $ab \in \sigma$, the edge/arc (a, b) exists in G^ϵ [20, 21].*

We also characterise scenarios in which a call sequence has been successful to best suit the aims of the project. Since we aim to evaluate efficiency, the execution of a protocol will have a pre-defined time limit τ that will be incorporated into the definition of success. Though not particularly rigorous, this helps us to evaluate the success of infinite call sequences which are not possible to test computationally.

Definition 2.3.2 (Successful Call Sequence) *Let σ be a (possibly infinite) call sequence and τ a pre-defined time limit. σ is successful in gossip situation (G, S) if computationally executing the calls of σ results in an all-expert state within time limit τ. Otherwise σ is unsuccessful.*

Thus an instance of protocol P is a permitted call sequence σ, and an instance of protocol P is successful if σ is successful. We can also distinguish between two types of failure: either the execution of a call sequence has exceeded the time limit, or the call sequence is finite and we have not reached an all-expert state once it has been fully executed. This definition of success is a simplification of that given in [21], and we make these changes to best suit the computational aspect of the project.

2.4 Rounds of Calls

A common variation of the gossip problem is for calls to be made at the same time within a single *round* [15]. We focus on an earlier implementation of this concept in which we designate pairs of agents (a, b) (with $b \in N_a$) such that the call ab is permitted under the relevant protocol. We choose such pairs (possibly at random) until no more exist. Each chosen pair is "mutually disjoint", meaning that each agent may only participate in at most one call within a single round [18]. Phone calls between designated pairs of agents are then executed "in parallel" [19].

Definition 2.4.1 (Round of Calls) *A round of calls is a set of permitted calls C such that no two calls of C feature the same agent. Computationally, we can also view a round of calls as a call sequence σ. Since the calls of C are disjoint, no two calls within σ will impact each other, and thus the order of call execution is not important.*

2.5 Dynamic Success Results

Here we highlight some results from [21] with regard to the success of ANY, CO, LNS, TOK and SPI in a dynamic setting. Ditmasrch et al distinguish between strong success, fair success, and weak success. Protocol P is strongly successful in a gossip situation if *all* p-permitted call sequences terminate in an all expert state, and weakly successful if *some* p-permitted call sequences terminate in an all expert state. Fair success assumes all *fair* call sequences terminate in an all-expert state. For example, the ANY protocol could repeat the same call indefinitely and never terminate successfully, but this sequence of calls is said to be "unfairly scheduled". In this project, infinite (unfair) call sequences will be handled by the timeout condition in Definition 2.3.2. By Theorems 13 and 15 from [21], and assuming (G, S) features a weakly connected gossip digraph, we have that:

- ANY, TOK and SPI are fairly successful in (G, S).
- CO is strongly successful in (G, S).

The success of LNS is dependent on the type of gossip graph used. LNS is "strongly successful on G iff G is a sun", and is "weakly successful on a weakly connected gossip graph G iff G is neither a bush nor a double bush" [21]. Here a sun, bush and double bush are three special types of graph that are unlikely to be the result of a random graph generator. These findings lead us to expect that our protocol simulator should be able to successfully execute ANY, CO, TOK and SPI in a dynamic setting 100% of the time (excluding timeouts), whereas LNS should only be successful some of the time. This is according to the definition of success given in definition 2.3.2.

Chapter 3

Software and Experiment Design

This chapter will highlight the objectives of the project, outline the design elements of the software, and explain how this software can be used to carry out experiments to evaluate the efficiency of the protocols. The original design document can be found in appendix D.

3.1 Project Objectives

We aim to evaluate the suitability of five distributed gossip protocols (ANY, CO, LNS, TOK and SPI) in three scenarios; complete communication networks, incomplete networks, and dynamic directed networks. The suitability of a protocol is based on its ability to terminate successfully in an all-expert state, including how often it is able to do so, and how *fast* it is able to do so. Protocol efficiency can be broken down into execution length (i.e. the number of calls required to terminate in an all-expert state [19]) and execution time. As an extension to the original proposal, our consideration of efficiency will also extend to round-based gossip, where we will analyse the number of rounds required for each protocol to terminate successfully. A full outline of project objectives is as follows:

- Include formal definitions of concepts that are reflected in the software and experiments, including (but not limited to) gossip graphs, p-permitted call sequences and successful call sequences.
- Implement a distributed gossip protocol simulator in Python that allows the user to generate a random gossip (di)graph and execute one of ANY, CO, LNS, TOK or SPI. The simulator should be able to perform calls sequentially or in rounds.
- Use the simulator to test each protocol in a complete, incomplete and dynamic case. This will include generating gossip graphs across a varied number of agents n, simulating the protocols on these graphs and making note of important results such as execution length, computation time and whether the protocol was successful. We can produce plots of these results against the number of agents n. We can also perform repeated executions of a protocol on a single graph in order the evaluate the consistency of a protocol by analysing the spread of execution lengths. Full details of the experiment(s) are provided in section 3.2.
- Discuss the suitability of the five protocols in varying scenarios. This will include comparison of execution lengths, number of rounds required, computation times and rate of success.

3.2 Experiment Design

This section will discuss an experiment that will be used in order to evaluate the efficiency and success of the five protocols. The same experiment will be carried out three times; once for each of complete, incomplete, and dynamic gossip. For some fixed number of agents n, we wish to generate a collection of gossip graphs and perform one execution of each protocol on each graph from the collection. We will keep track of the number of successes for each protocol across the collection of graphs, as well as execution lengths/number of rounds and computation times for successful executions. For each protocol we now have a list of execution lengths/number of rounds and computation times from which we can compute an average. We can also determine the success rate using the number of successful executions. The experiment can be summarised as follows:

1. Choose a suitable range of n values to experiment on where n is the number of agents.
2. For each fixed value of n, generate a relevant gossip graph on n agents.
3. Execute each protocol on this gossip graph. Note the success status, and record execution length/number of rounds and computation time for successful executions.
4. Repeat steps 2 and 3 for a specified number of trials for each value of n (in error, the original proposal only stated that step 3 would be repeated for each fixed n).

When considering the spread of execution lengths, we instead generate a single gossip graph and execute each protocol on the graph t times, where t is a set number of trials. We can then, for example, produce boxplots to show the spread of execution lengths given by each protocol.

3.3 Software Outline

Here we discuss the design for the software. We give a breakdown of the functions, including details of the methods used within the code.

3.3.1 Graph Creation

The challenge posed by randomly generating a gossip graph is that it must be *connected* for an undirected graph, or *weakly connected* for a directed graph. In the dynamic case, Ditmarsch et al state that "the goal of all agents being expert can never be reached in graphs that are not weakly connected", and the same is true for disconnected undirected networks by the same logic [20].

NetworkX
This project makes use of the open source Python library NetworkX [1]. NetworkX allows us to create graph objects in Python for which we can easily manipulate the nodes and edges/arcs. Each node of the graph can also be assigned its own set of attributes, which in our case will include the following:

- A set containing all known secrets.
- A set of agents that the agent has been in contact with during the current protocol execution.
- A boolean variable to determine if the agent owns a token (True) or not (False).

We also have access to a full list of edges for the graph, methods that will be used for the partial creation of random graphs, and the ability to create customisable visualisations of networks (e.g. figure 1.1).

The nodes of an n agent graph will be labelled from 0 to $n-1$. As an abuse of notation, the respective secrets and phone numbers of the agents will also be labelled from 0 to $n-1$ since we need to use an unbounded set. Each agent will be given the following initial attributes: a secret set containing only itself, an empty set of previous contacts, and a true token variable. The following functions will create an initial gossip situation (G, S), where each agent a is assigned the set of known secrets $S_a = \{A\}$ to represent the initial distribution of secrets S.

Complete Graphs. NetworkX provides a method to create a complete graph on n nodes. We simply loop through the nodes of the graph and assign the required attributes.

Incomplete Graphs. Bondy et al state that "every connected graph contains a spanning tree" [11]. It is this corollary on which the method proposed by StackOverflow user Baugh is based [10], which involves first creating a random spanning tree, and then adding an arbitrary number of edges at random. We first use a method provided by NetworkX to create a random tree on n nodes (such a graph will have $n-1$ edges [11]). We then randomly generate the number of extra edges to add to the graph. A complete graph on n nodes has $nC2$ edges [11], and thus we can add a maximum of

$$e = nC2 - (n-1)$$

extra edges. We therefore generate a random integer $r \in [0, e]$, and add edges until we have a total of $(n-1)+r$. This is done by randomly choosing vertices u and v and adding the edge (u, v) to graph if it does not already exist, up until the edge limit has been reached. The resulting graph is randomly generated and connected as required. The pseudo-code is as follows where we input the desired number of agents n:

```
incompleteGraph(n)
(1) Generate random tree with n vertices G (using NetworkX)
(2) $e$ = nC2 - (n-1)
(3) Generate random number r between 0 and $e$
(4) While number of edges in G < (n-1)+r:
(5)     Choose random vertices u and v
(6)     If (u,v) is not an edge:
(7)         Add edge (u,v) to G
(8) od
(9) For every node of G:
```

```
(10)    Assign initial secret
(11)    Assign list of past contacts (initially empty)
(12)    Assign Token = True
(13) Output graph G
```

Directed graphs. The method for creating random weakly connected digraphs is similar to that for incomplete graphs, and is an extension of the method described by Baugh [10]. After producing a random tree, consider randomly orienting each of the edges into an arc. Then, as before, we randomly add arcs up until some limit. The maximum number of arcs we can add is

$$e = 2(nC2) - (n-1),$$

and we choose some $r \in [0, e]$. The pseudo-code to produce a random gossip digraph is as follows:

```
diGraph(n)
(1) Generate random tree with n vertices G (using NetworkX)
(2) Orient the edges of G
(3) $e$ = 2(nC2) - (n-1)
(4) Generate random number r between 0 and $e$
(5) While number of arcs in G < (n-1)+r:
(6)    Choose random (ordered) vertices u and v
(7)    If (u,v) is not an arc:
(8)       Add arc (u,v) to G
(9) od
(10) For every node of G:
(11)    Assign initial secret
(12)    Assign list of past contacts (initially empty)
(13)    Assign Token = True
(14) Output graph G
```

3.3.2 Protocol Simulation

We must now implement methods in order to execute each of the five protocols on an initial gossip situation created by the functions in section 3.3.1. Firstly, each of the five protocols will be assigned a unique function that returns a list of permitted calls from which a new call can be chosen and executed. This can be done by looping through the ordered edges/arcs of a gossip graph G, and adding any calls that satisfy the protocol P to a list of candidates. The original pseudo-code to do this can be found in appendix D.2.2.

Next we implement a function that takes an initial gossip situation (G, S) and executes a chosen protocol P, returning the execution length, execution time and success status. The function will terminate in two scenarios: all agents are experts (success) in which case $|S_a| = n$ for all agents a, or the list of permitted calls has become empty (failure) before an all-expert state is reached. The execution of a protocol is carried out in stages of a loop, wherein a list of permitted calls is retrieved using one of the five protocol functions, and a random call is chosen and executed from the list. This is equivalent to generating and executing a permitted call sequence σ on a call-by-call basis. All necessary operations are carried out which includes the following: exchange of secrets, add agents to each others list of contacts, exchange tokens (for

TOK and SPI), and add new arcs to the graph (in the dynamic case). The function also keeps track of the number of calls performed and the computation time. The pseudo-code can also be found in appendix D.2.2, namely the `executeProtocol` function.

3.3.3 Experiments

For the experiment, we combine the graph generator and protocol execution functions such that we can conveniently test each protocol on a wide range of gossip graphs. The user must be able to perform the experiment from section 3.2 by inputting the number of agents n, protocol P, network topology and number of trials t. The function will generate t gossip graphs on n nodes and test the protocol on each of these graphs, returning the execution lengths, computation times, and number of successes/failures. Note that we also wish to keep track of the number of timeouts, a feature that was not included until the implementation phase.

3.4 Evaluation Criteria

3.4.1 Software Evaluation

The software evaluation criteria is relatively unchanged from the original design document.

- **Validity.** The gossip graphs and protocol simulation results must be valid to the best of our knowledge. The output of the graph generators must be accurate computational representations of definitions 2.2.1 and 2.2.2, and the protocol simulation results must appear plausible and relatively un-contradictory to gossip literature. For example, from [21] we know that CO will always terminate successfully on a weakly connected digraph in a dynamic setting. Thus the simulator should show a 100% success rate for the CO protocol in this setting (excluding timeouts).

- **Efficiency.** Protocol simulations must terminate as expected within a suitable time frame. We wish to implement the most efficient methods possible, and the generation of a graph of 100 agents followed by a protocol execution on that graph should take no longer than a few seconds. A suitable timeout clause must be implemented to prevent protocols that become 'stuck' from hindering experimentation.

- **Convenience.** The experiment from section 3.2 must be easily adaptable to different variations of gossip. For example, we must be able to switch between graph topologies, sequential/round based gossip, and range of n values without changing the internal code of any of the functions. The `experiment` function must be able to receive inputs to specify which variation of gossip we wish to test, and results must be returned in a compact and well-organised manner.

- **Presentation.** The code must be appropriately commented, well-organised, and all functions must have a short description of their inputs and outputs.

3.4.2 Results Evaluation

Our main goal is to compare the five protocols in terms of their efficiency and success. Plotting the results must clearly show how the average execution length and computation time of a protocol changes as we vary the number of agents, as well as how successful the protocol has been throughout the experiments. We give an interpretation as to which protocol is optimal given a network topology, and this must be supported by the results of the simulator.

3.5 Ethical Use of Data

This project utilises data in the form of gossip graphs on which we test each of the protocols. These graphs are randomly generated in Python using the methods described in section 3.3.1. This random process partly involves the use of a NetworkX function `random_tree(n)` to create random trees, details of which can be found in the NetworkX documentation [1]. No other data has been retrieved from external sources. This project does not involve any human participation such as external software evaluation.

Chapter 4

Software Implementation and Testing

This chapter will describe the process of implementing the software design from chapter 3, including any issues that were encountered, as well as changes made to the original design. We also discuss how the software was tested to ensure that (to the best of our knowledge) the results we produce are accurate and meet the project objectives. The code listing can be found in appendix E.

4.1 Graph Creation

4.1.1 Implementation and Design Changes

We take the pseudo-code provided in section 3.3.1 and make it Python specific. We also implement a boolean input so that the user is able to plot the generated gossip graph using NetworkX [1]. An example of these plots for each of the three graph topologies is given in appendix B (figure B.1). The properties of node 0 from the digraph in figure B.1 are given in figure B.2. You can see how initially node 0 only knows its own secret, possesses a token, and that the set of known numbers corresponds to its out-neighbours in the graph. Changes to the original design are as follows:

- Each agent a is now assigned a set of known phone numbers N_a. Having dedicated sets to represent the distribution of phone numbers made the dynamic implementation more intuitive. Thus after a call (a, b), we must assign $N_a \bigcup N_b$ as the set of numbers known by agents a and b.
- When generating a digraph, we purposefully restrict the maximum number of arcs that can be added since a graph that is very well connected could make the concept of exchanging phone numbers somewhat redundant. Therefore consider changing e to the integer closest to $\frac{e}{10}$.

4.1.2 Testing

Testing of the graph generators was carried out manually. We can produce graphs on a small number of agents (say 10), and check the properties of each agent are as expected. This highlighted an implementation error in which agents did not know their own phone number, and thus in a dynamic setting, agents with only out-neighbours would never receive a call. Furthermore, we can randomly generate and plot a collection of graphs before visually inspecting them for (weak) connectedness.

4.2 Protocol Execution

4.2.1 Prerequisites

We first discuss some useful 'tools' that were written during the implementation phase.

Firstly consider a function `experts(G)` that takes a gossip situation (G, S) and returns `True` if all agents of G are experts. It will do this by looping through all agents of G, and checking if their corresponding set of known secrets has n elements where n is the number of agents. If we find an agent a with $|S_a| < n$, then we have not reached an all-expert state and the function will return `False`.

Next we design a function `pPermitted(G, P, caller, callee)` that takes a gossip graph G, a protocol P, and the index of a caller and callee as input. It will return `True` if the call (caller, callee) is p-permitted. This function will be used in the dynamic case when new arcs are being added to the graph; if a new arc $(i.j)$ is p-permitted then it can be added to the list of permitted calls.

4.2.2 Implementation and Design Changes

We implement the functionality to produce a breakdown of permitted calls and calls made in each stage/round via a boolean input variable. A basic example of this is given in figure B.3, where we perform the CO protocol on a complete graph with 4 agents. You can see how the list of CO-permitted calls changes throughout the execution. The final output corresponds to the execution length, computation time, success status, failure status, and timeout status.

Significant changes were made to the individual protocol functions. All five functions will have a common feature that returns the ordered edges/arcs of the gossip graph if we have an initial gossip situation. If the gossip situation is not initial, then the five functions have a unique method for selecting calls by taking the latest call made as input, and updating the previous list of permitted calls. This differs from the original proposal in which a list of permitted calls is generated from scratch at each stage, which is less efficient. The five functions have the following general structure:

```
P(Graph:  G, Latest call:  newcall, previous calls:  calls)
(1) If G is an initial gossip situation:
(2)     Return ordered edges/arcs of G
(3) Else:
(4)     Update calls according to P using newcall
(5)     Return updated list of calls
```

For each protocol we must describe a method to carry out line 4 of this pseudo-code where (i, j) is the latest call made:

- **ANY.** Always return the previous list of calls. This is because the list of permitted calls does not change as calls are executed.
- **CO.** Remove calls (i, j) and (j, i) from the previous list of permitted calls if they exist. Return the updated list.
- **LNS.** Loop through all secrets $k \in S_i$, and remove any calls (i, k) from the previous list of calls if they exist. Similarly remove calls (j, k) such that $k \in S_j$. Return the updated list.

- **TOK.** Since i has lost its token, remove all calls (i, k) such that $k \in N_i$. Since j may have gained a token, add any calls (j, k) such that $k \in N_j$ and (j, k) is not already in the previous list of permitted calls. Return the updated list.
- **SPI.** Since j has lost its token (if it owned one), remove all calls (j, k) such that $k \in N_j$ from the previous list of permitted calls. Return the updated list.

We also implement a function similar to `executeProtocol` that executes calls in *rounds*, namely `executeRounds`. Instead of selecting and executing a single call and incrementing the call count, we randomly choose as many disjoint calls from the list of permitted calls as possible, execute all of them, and increment the round count. `executeRounds` will return the number of rounds, computation time, success status, failure status, and timeout status, given an initial gossip situation and a protocol to execute.

We also implement a timeout clause when executing a protocol. This is to prevent infinite call sequences from jeopardizing the experiments. In particular, we assign a timer $\tau = \frac{n}{10}$ seconds to a protocol execution. Making the timer proportional to the number of agents means that we can give more time to executions on larger graphs which are expected to take longer.

4.2.3 Testing

Testing of protocol executions was done by examining the breakdown of calls 'by hand' for each combination of protocol and graph topology. The breakdown of calls revealed that loops were being added to the gossip graphs in the dynamic case. During a SPI execution, this would cause agents to call themselves resulting in the loss of their own token.

4.3 Experiments

4.3.1 Implementation and Design Changes

The experiment design from section 3.2 is implemented via two functions: one that will fix the number of agents n, generate gossip graphs, and test each of the protocols on the graph; another that will utilise the first function across a range of n values. Implementing a second function to perform experiments across a range of n values made the experimentation phase more convenient.

- `testProtocols(n, topology, trials, rounds)` - takes the following inputs: number of agents, graph topology, number of trials and a boolean to dictate if calls are executed in rounds or not. The function will loop through each trial, create a random graph for each, and execute each of the protocols on the graph. All results including execution lengths/number of rounds, computation times, and number of successes, failures and timeouts are stored in a dictionary. Finally the function computes averages for execution length and computation time for each of the five protocols, as well as rate of success, failure and timeout.
- `experiment(min n, max n, topology, trials, rounds)` - takes the same inputs as the `testProtocols` function, except we now specify a minimum and maximum n value. We loop through all n values from `min n` up to `max n` in steps of 5, and execute the `testProtocols` function for each. We store average execution lengths/number of rounds, computation times, and

success, failure and timeout rates in lists, before compiling all results into a single variable.

4.3.2 Testing

No specific method was used to test the experiment function(s). We simply used the functions to produce the desired results (see chapter 5), and evaluated these results keeping in mind the **validity** criteria from section 3.4.1.

Chapter 5

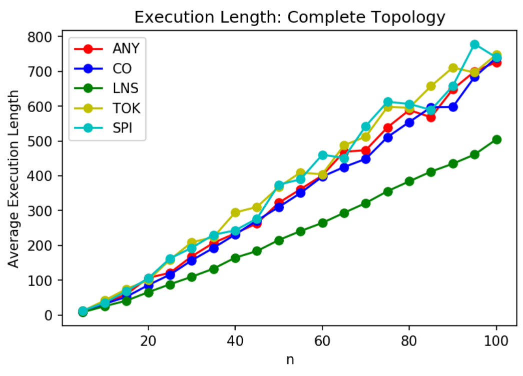
Results and Analysis

This chapter will be divided into three sections corresponding to results for complete networks, incomplete networks, and directed dynamic networks. Relevant plots [4] for execution length, computation time, success rate and number of rounds will be provided.

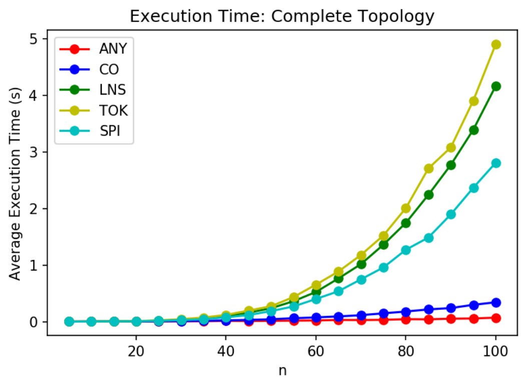
5.1 Complete Topology Results

5.1.1 Sequential Gossip

Figure 5.1 shows the results for executing the protocols in a sequential manner on complete graphs. We vary the number of agents n between 5 and 100 in steps of 5, and perform the experiment outlined in section 3.2. Every protocol has a 100% success rate across all values of n, and thus success is not a relevant factor when determining which protocol is optimal. Thus the question now becomes whether we give higher value to smaller execution length or smaller execution time.



(a) Average Execution Length

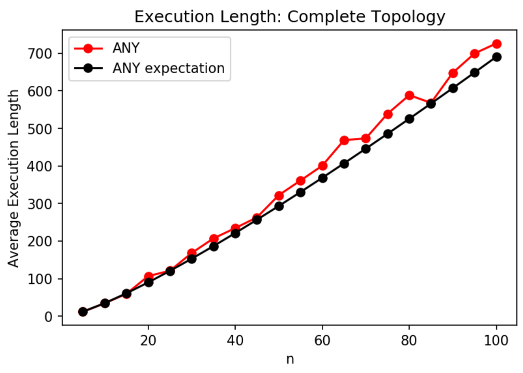


(b) Average Computation Time

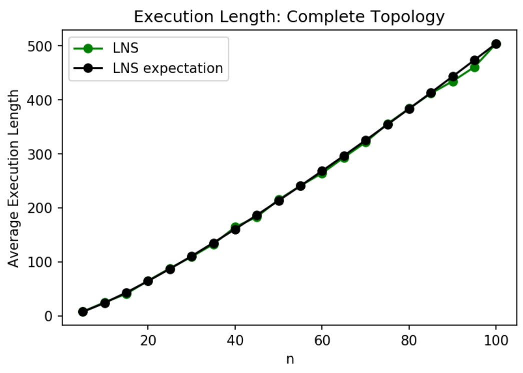
Figure 5.1: Complete topology results when varying the number of agents n between 5 and 100. Here we generate 10 graphs for each value of n.

We can see from figure 5.1a that LNS has superior execution lengths across the range of n values, and the other four protocols perform similarly to each other. In particular, the simulation results in the graph show that LNS has a smaller average execution length than ANY by a "constant factor", which is consistent with known results from the literature. From [19] we have that ANY has an expected length of $\frac{3}{2} \cdot n \cdot \log(n) + O(n)$ and LNS has an expected length of $1.0976 \cdot n \cdot \log(n) - 1.1330$. We can isolate the results from figure 5.1a for ANY and LNS, and plot against their expected length. This can be found in figures 5.2a and 5.2b respectively.

Figure 5.1b shows that ANY has the smallest average computation time for larger



(a) ANY expectation



(b) LNS expectation

Figure 5.2: Expected execution length of ANY and LNS paired with the simulation results.

values of n, taking only a fraction of a second on graphs with 100 agents. In comparison LNS has an average computation time of around 4.1 seconds on the same collection of 100 node graphs. This might be due to the simplicity of selecting ANY-permitted calls.

Now consider executing each of the protocols on a complete graph of 50 agents 50 times. A boxplot of the results can be seen in figure 5.3. We can see that not only does LNS have optimal execution length, but it is extremely *consistent* when compared to the other protocols as indicated by its narrow box.

Boxplots
Boxplots are produced using the `boxplot` function from the matplotlib library [4]. The box displays the distance between the first ($Q1$) and third ($Q3$) quartile of the data, and the orange line indicates the median. Furthermore the lower and upper whiskers correspond to $Q1 - 1.5(Q3 - Q1)$ and $Q3 + 1.5(Q3 - Q1)$ respectively, and any data points that lie outside of the whiskers are considered outliers.

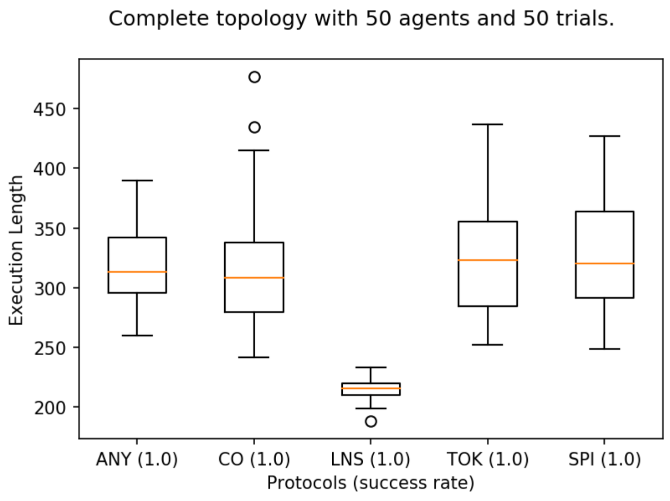


Figure 5.3: Boxplot of execution lengths on a complete graph

5.1.2 Round Based Gossip

All five protocols have a 100% success rate on complete graphs for round-based gossip. Performing calls in rounds produces the graph seen in figure 5.4. Token based protocols (TOK and SPI) require significantly more rounds to reach an all expert state. This could be due to agents losing tokens and being unable to make calls, resulting in less calls per round as the protocol is executed. An example of this can be seen in figure B.4, where we execute the TOK protocol in rounds on a ten agent complete gossip graph. You can see that we can select five calls in the first round, but only two in the tenth round. Isolating the results for ANY, CO and LNS (see figure 5.5), you can see that CO requires the least number of rounds on average, and that in general more rounds are required when the number of agents is odd.

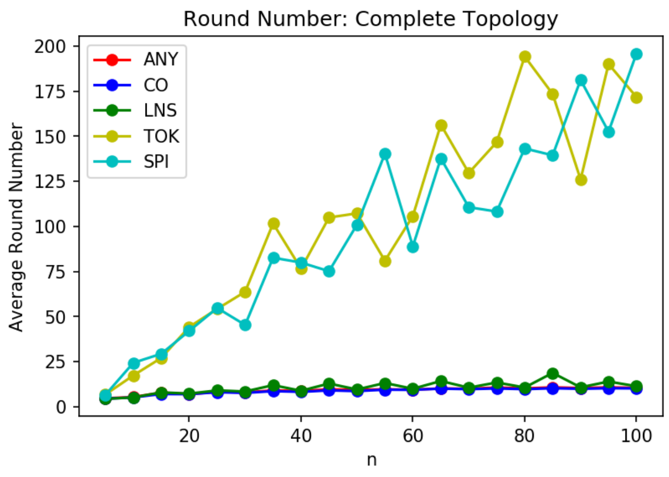


Figure 5.4: Average number of rounds required on complete graphs from 5 to 100 agents. Here we generate 10 graphs for each value of n.

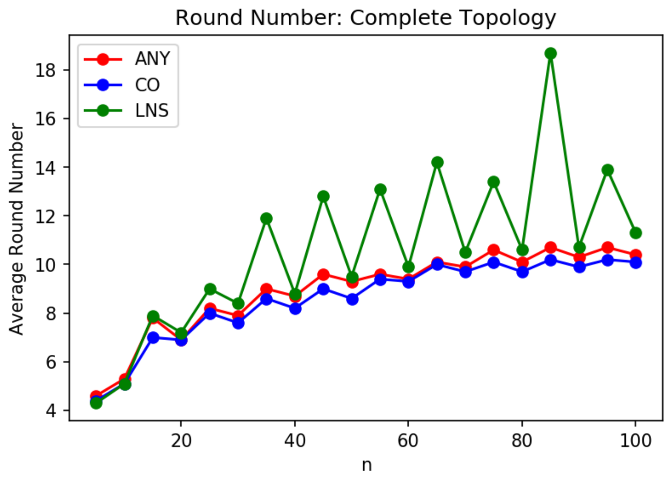
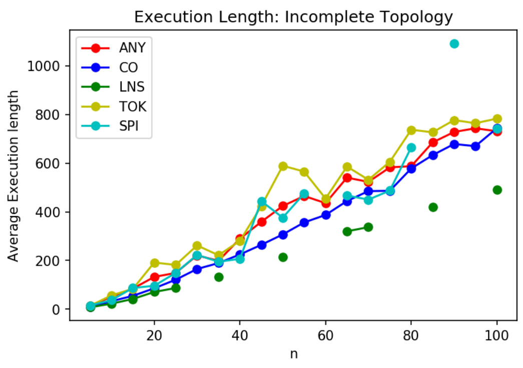


Figure 5.5: Average number of rounds required on complete graphs from 5 to 100 agents (excluding results for TOK and SPI).

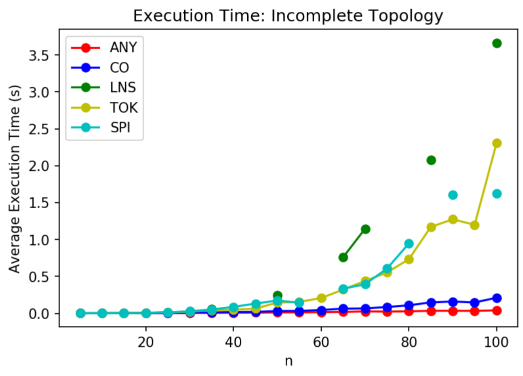
5.2 Incomplete Topology Results

5.2.1 Sequential Gossip

Figure 5.6 displays the main results for random incomplete graphs. The breaks in the graph are a result of there not being a single successful execution of the protocol for the corresponding value of n. We can see from figure 5.6a that LNS tends to have optimal execution length, however this is conditional on a successful termination which is not guaranteed. In particular we can see from figures 5.7b and 5.7c that LNS tends to fail (run out of permitted calls) on incomplete graphs rather than timeout. The other four protocols perform similarly in terms of execution length, however from figure 5.7a we can see that ANY and TOK are the only two protocols to have a 100% success rate (SPI often times out and CO fails). ANY has slightly better execution lengths than TOK, and vastly superior computation times (see figure 5.6b).



(a) Average Execution Length



(b) Average Computation Time

Figure 5.6: Incomplete topology results when varying the number of agents n between 5 and 100. Here we generate 10 graphs for each value of n.

Figure 5.8 gives a boxplot of execution lengths on a random incomplete graph of 50 agents. The results for four other gossip graphs can be found in appendix C (figure C.1). Figure 5.8 shows that when CO terminates successfully, it has the smallest spread of execution lengths.

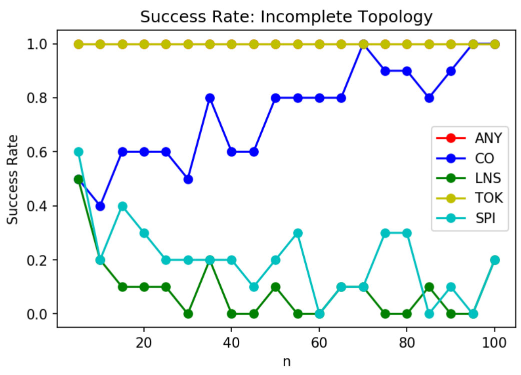
5.2.2 Round Based Gossip

Figure 5.9 shows the results for round based gossip on incomplete gossip graphs. Similar to complete graphs, token based protocols require more rounds on average to reach an all expert state, and CO requires the least number of rounds on average. The results for success are similar to those for sequential gossip on incomplete graphs; TOK timed out on one occasion, though this may be due to the fact that this specific implementation of round based gossip is less efficient than sequential gossip in terms of computation time.

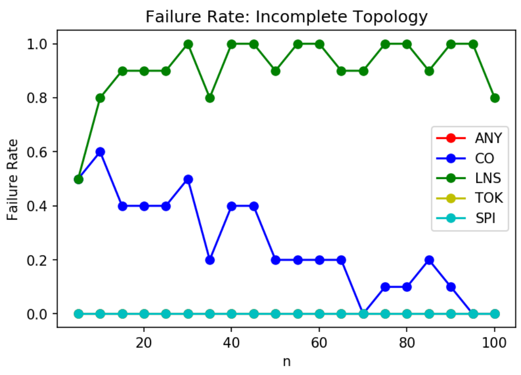
5.3 Dynamic Gossip Results

5.3.1 Sequential Gossip

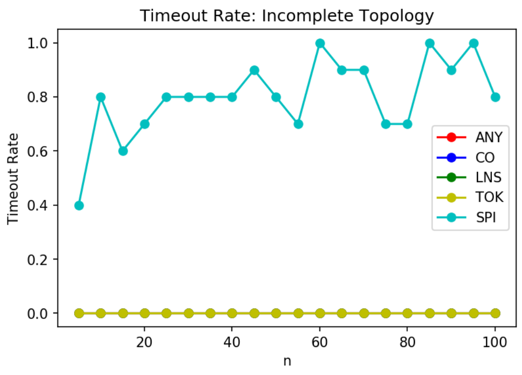
Figure 5.10 shows the results for dynamic gossip. LNS has optimal execution length, but is the only protocol of the five to have a success rate less than 100%. Furthermore LNS has the slowest computation time, being around two seconds slower (on average)



(a) Rate of Success



(b) Rate of Failure



(c) Rate of Timeout

Figure 5.7: Corresponding breakdown of success, failure and timeout rate for the results given in figure 5.6.

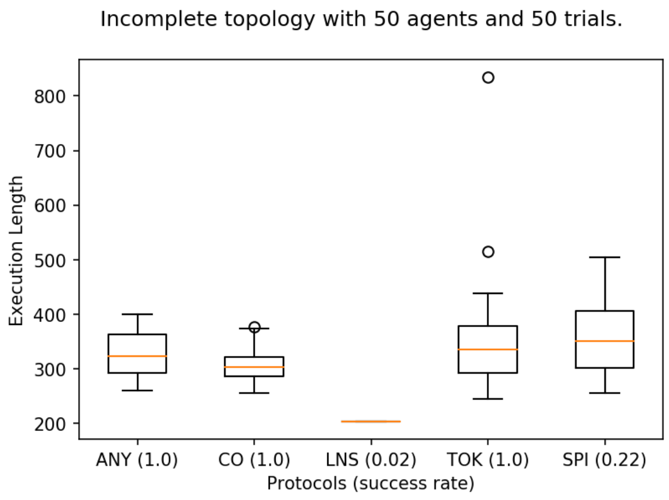
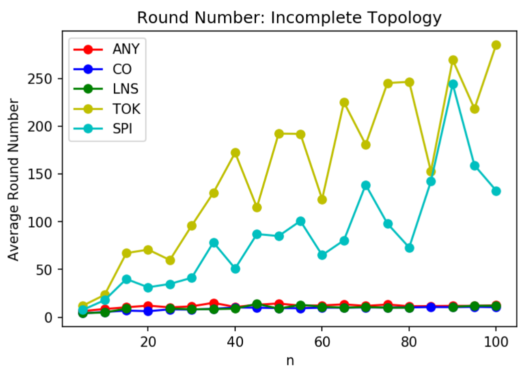
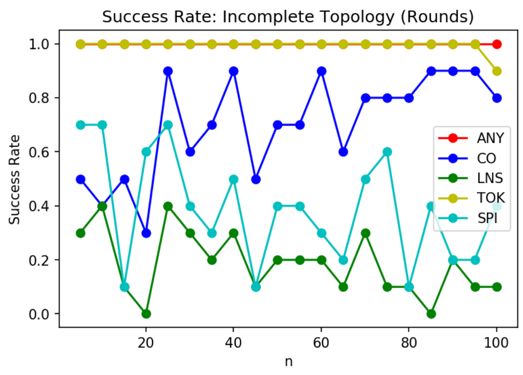


Figure 5.8: Boxplot of execution lengths on a random incomplete graph.

than the next slowest protocol on 100 agents. The execution lengths of the ANY protocol appear to be somewhat erratic as a result of outliers (see figure 5.11 for an example of the ANY protocol producing outliers). After LNS, TOK and SPI are relatively tied for the next most efficient in terms of execution length, however SPI has superior computation times. Figure 5.11 shows the results for the spread of execution lengths on a dynamic gossip graph of 50 agents. Four further examples can be found in appendix C (figure C.2). We can see that ANY has a tendency to produce large outliers, whereas TOK and SPI are very consistent.

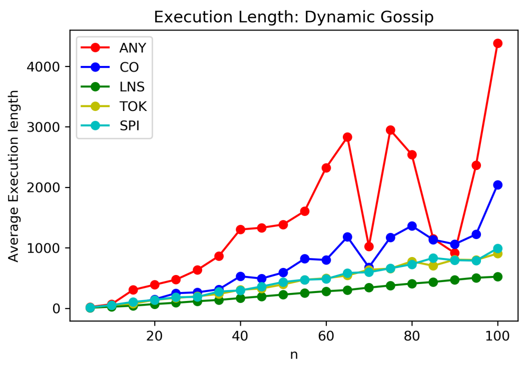


(a) Number of Rounds

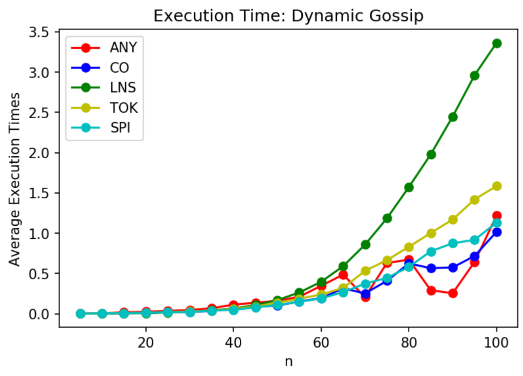


(b) Rate of Success

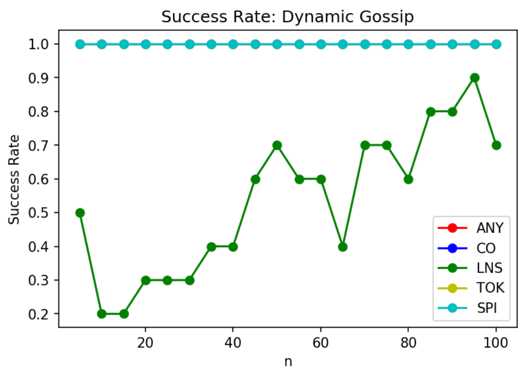
Figure 5.9: Average number of rounds required on incomplete graphs from 5 to 100 agents and the accompanying success rate. Here we generate 10 graphs for each value of n.



(a) Average Execution Length



(b) Average Computation Time



(c) Rate of Success

Figure 5.10: Dynamic gossip results when varying the number of agents n between 5 and 100. Here we generate 10 graphs for each value of n.

5.3.2 Round Based Gossip

Figure 5.12 shows the results for round based gossip in a dynamic setting, including a success rate graph. The apparent trend of token-based protocols requiring more rounds continues here. Furthermore we can see that ANY and CO sometimes fail to terminate in an all-expert state for larger values of n due to timeout. On the other hand, LNS often fails due to running out of permitted calls. None of the five protocols present as an 'obvious' choice in this scenario. Typically a 'safe' option is the ANY protocol, except here 3 out of 10 ANY executions on 100 node gossip

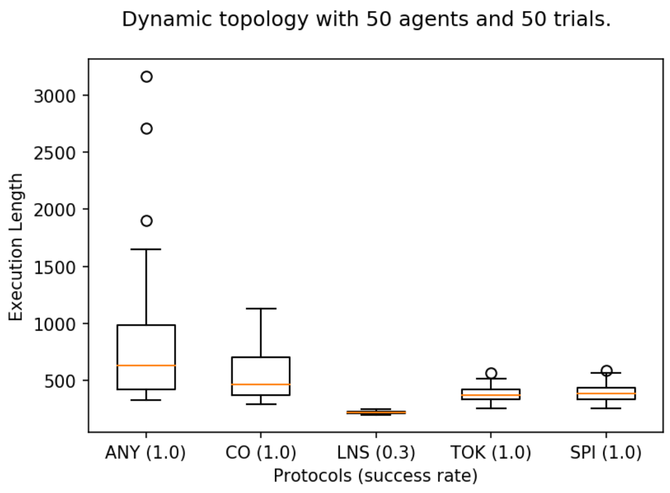
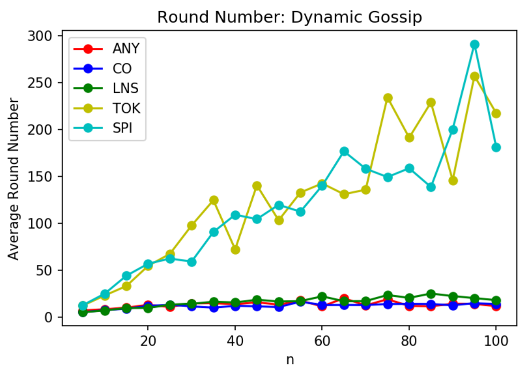
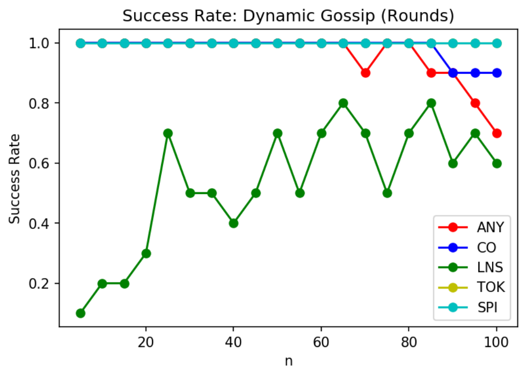


Figure 5.11: Boxplot of execution lengths on a random dynamic gossip graph.

graphs took longer than 10 seconds to terminate, resulting in timeout. This is further evidence of the unpredictability of the ANY protocol in a dynamic setting.



(a) Number of Rounds



(b) Rate of Success

Figure 5.12: Average number of rounds required on dynamic gossip graphs from 5 to 100 agents and the accompanying success rate.

Chapter 6

Project Evaluation and Learning Points

Here we give an evaluation of the strengths and weaknesses of the project with particular focus on the developed software and the results that were produced. We also discuss what has been learnt throughout the course of the project, including what could be improved in similar pieces of work. We finish by providing a brief summary of how the project complies with professional standards set out by the BCS code of conduct.

6.1 Project Evaluation

6.1.1 Software

We address the software evaluation criteria given in section 3.4.1.

- **Validity.** The final software performed valid gossip graph generation and protocol execution as indicated by manual inspection of some basic test cases. However manual inspection is not a viable method for verification of protocol execution results on larger gossip graphs. The simulation results for the execution length of ANY and LNS on complete graphs compare favourably to the known order of these protocols from [19]. Furthermore, defining protocol success as in 2.3.2 and recalling the conclusions given in section 2.5, we can see that the dynamic simulation results appear valid in terms of success rate. In particular, we expected ANY, CO, TOK and SPI to always terminate successfully (unless they timed out), and LNS to terminate successfully on a subset of executions. To the best of our knowledge, all other results cannot be verified via comparison to other sources, especially those for random incomplete gossip graphs.

- **Efficiency.** The generation of random graphs on 100 nodes of all three topologies typically takes 0.5 seconds or less. Regarding the computation time of protocol executions, we have a plethora of results given in chapter 5. The timeout criteria was implemented as required, and the maximum *average* time for a successful protocol execution was around 5 seconds. Round based executions took longer than for sequential gossip. This caused some protocols to timeout despite us knowing that it would have terminated successfully if given more time (e.g. the ANY protocol in the dynamic case).

- **Convenience.** The experiments were simple to carry out once all the component functions had been coded. We can produce a full set of results for a given range of n values, network topology, set number of trials per n value, and whether calls are made in rounds or not by simply inputting these variables into the experiment function. However, producing the graphs in chapter 5 required manual coding, including the boxplots. The experiment function is able to return a full set of results all stored within a single variable, however accessing specific subsets of the results requires knowledge of the inner-workings of this variable (e.g. key names).

- **Presentation.** All functions within the code have a description of their input and output, and brief notes are given throughout the code to explain the actions being performed.

6.1.2 Results

In most cases, the results enable us to make an informed judgement as to which protocols are optimal in terms of their success, however results regarding execution length/time can sometimes be hard to interpret from the graphs alone. For example, in figure 5.6a it is difficult to distinguish between ANY, CO, TOK and SPI. Perhaps performing more than ten trials per value of n would result in 'smoother' trends that are more easily comparable. The results become easier to interpret if we first discount protocols based on their poor success rate, and focus on the execution lengths/times of the remaining protocols. It can sometimes be difficult to determine the exact difference between the protocols from the graphs alone, however providing all results in a numerical form (say in a table) is not particularly viable. The boxplots provided do give a good indication as to the consistency of repeated executions of a protocol on a single graph, however the sample size of different graphs used is rather small.

6.2 Learning Points

This project required adequate background reading and preparation in order to produce the desired software. In particular, this project was a test of ones ability to tackle a difficult problem from the ground up, transferring knowledge acquired from reading into a gossip protocol simulator in a high-level programming language.

Knowledge. This project developed significant knowledge of the gossip problem, which entails the dissemination of information throughout a distributed system of agents. In particular, formalisation's of concepts such as call sequences, gossip graphs, and gossip protocols were provided by the literature. This allowed us to develop and alter these definitions in such a way as to best suit the aims of the project. As well as results provided by existing literature, the results provided within this dissertation were able to develop our understanding of how the protocols compare in varying scenarios. For example, we confirmed the reliability and simplicity of the ANY protocol in the complete and incomplete case. We also managed to show that LNS, a protocol which specifically promotes the spread of new information, often fares well in terms of execution length, despite its lack of success on incomplete and dynamic gossip graphs.

Skills. Tackling a substantial piece of work requires sufficient background knowledge and planning before fully executing the aims of the project. In particular, this project allowed the development of project planning skills, including the ability to set realistic goals that are also enlightening. From a more technical perspective, this project also developed problem solving and programming skills by requiring a full protocol simulator from scratch (except for the use of NetworkX). Finally, this project also developed the skills to design an experiment given a research problem, and include the results in a substantial piece of work that contains all necessary background and system design.

Downfalls. The process of testing the software was not very thorough, nor did we record any of the test cases to provide within the appendices of this dissertation. This harms the reliability of the software, despite test cases appearing to function as required. We provide detail of how the software operates, but readers would also benefit from more sample runs and examples of input/output. Evaluation of the software would have been more reliable if external participants were asked to provide comments on the user experience. The original design document did not specify how the spread of execution lengths was going to be analysed. This resulted in a late change to the method for producing the boxplots in chapter 5, which detracted from the time available to produce this dissertation. Key dates regarding the progress of the project were not recorded, meaning a project log could not be given.

Positives. The pseudo-code provided in the original design document was essential for implementing the software within the required time frame. In particular, starting out with a quick, possibly inefficient set of code highlighted avenues for improvement (e.g. the improved method for selecting permitted calls). Deciding to include computation time as well as execution length not only provided more content when discussing the efficiency of the protocols, but also provided data which could be used to argue for the efficiency of the program.

6.3 Professional Issues

Here we briefly address the four principles set out in the BCS Code of Conduct [2].

- **Public Interest.** This project does not impose any threat to the well-being of others or include discrimination of any kind, nor does it disrespect the rights of third parties.
- **Competence and Integrity.** This dissertation is an accurate representation of the writers level of competence in the subject. Furthermore, this piece of work is built upon the existing work of others who have been acknowledged. For example, the definitions in chapter 2 are heavily inspired or paraphrased from the relevant citations. Comments and criticisms from the project supervisors have been gladly accepted and incorporated into this piece of work where possible.
- **Duty to relevant authority.** Professional responsibility has been taken in the production of this work. The Code of Practice on Assessment provided by the University of Liverpool has been followed to maintain the integrity of the university and myself as an academic.
- **Duty to the profession.** All actions taken in the preparation of this work are in the best interests of the integrity and reputation of the profession.

Chapter 7

Conclusion

We designed a distributed protocol simulator in Python that is able to perform executions of ANY, CO, LNS, TOK and SPI on complete, incomplete, and dynamic gossip (di)graphs. In particular, this simulator was used to evaluate the efficiency and success of these protocols in varying scenarios, including the variation of round-based (or parallel) calls.

Results round-up. For sequential gossip, LNS typically has optimal execution length on all three types of gossip graph, however its lack of success on incomplete and dynamic gossip graphs means it is sub-optimal in these scenarios. LNS is also particularly consistent on complete gossip graphs in terms of the range of execution lengths it can produce. ANY performs extremely well in terms of computation time on complete and incomplete gossip graphs due to its simplicity. ANY typically has the best average execution length on random incomplete gossip graphs from the set of protocols with a 100% success rate in this scenario. However, ANY is unpredictable and unstable in the dynamic case, having a tendency to produce large outliers. LNS is the only protocol to have a success rate below 100% for dynamic gossip. TOK and SPI perform very similarly in this scenario, both having smaller average execution lengths than ANY and CO. SPI has better computation times than TOK, however.
For round-based gossip, results regarding success are similar to sequential gossip, although protocols are more likely to timeout. Token based protocols require more rounds to reach an all-expert state. CO has the smallest average number of rounds required on complete graphs, and ANY is optimal on incomplete gossip graphs from the set of protocols with a 100% success rate. On dynamic gossip graphs, it is unclear from the results whether ANY or CO is optimal in terms of number of rounds required.
A personal interpretation of which protocol to use in varying scenarios is given in appendix A.

Possible extensions. Many extensions to this work are possible, such as including analysis of the wCO (weak call once) [21] and HMS (hear my secret) [5] protocols. Furthermore we can include the push and pull modes for exchanging secrets [5]. For round based gossip on graphs up to 100 agents, there is not much to separate ANY, CO and LNS, and thus it could be worth analysing if there is any disparity between the number of rounds required by these three protocols when testing beyond 100 agents.

Bibliography

[1] Networkx - network analysis in python. https://networkx.org/, 2020. Accessed: 2021-06-23.

[2] Bcs code of conduct. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/, 2021. Accessed: 2021-09-24.

[3] Igraph - the network analysis package. https://igraph.org/, 2021. Accessed: 2021-06-23.

[4] Matplotlib - matplotlib.pyplot.boxplot. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html, 2021. Accessed: 2021-09-21.

[5] Krzysztof R Apt, Davide Grossi, and Wiebe van der Hoek. Epistemic protocols for distributed gossiping. *arXiv preprint arXiv:1606.07516*, 2016.

[6] Krzysztof R Apt and Dominik Wojtczak. Verification of distributed epistemic gossip protocols. *Journal of Artificial Intelligence Research*, 62:101–132, 2018.

[7] Maduka Attamah. *Epistemic Gossip Protocols.* PhD thesis, Department of Computer Science, University of Liverpool, 2015.

[8] Maduka Attamah, Hans Van Ditmarsch, Davide Grossi, and Wiebe van der Hoek. Knowledge and gossip. In *ECAI*, pages 21–26, 2014.

[9] Brenda Baker and Robert Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.

[10] Wesley Baugh. Stackoverflow - random simple connected graph generation with given sparseness. https://stackoverflow.com/questions/2041517/random-simple-connected-graph-generation-with-given-sparseness, 2017. Accessed: 2021-06-26.

[11] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[12] Alan Frieze and Michał Karoński. *Introduction to random graphs.* Cambridge University Press, 2016.

[13] A Hajnal, EC Milner, and E Szemerédi. A cure for the telephone disease. *Canad. Math. Bull.*, 15:447–450, 1972.

[14] Frank Harary and Allen J Schwenk. The communication problem on graphs and digraphs. 1974.

[15] Walter Knodel. New gossips and telephones. *Discrete Mathematics*, 13:95, 1975.

[16] Diana Ramos. How to make a gantt chart in word. https://www.smartsheet.com/content/make-gantt-ms-word, 2020. Accessed: 2021-07-13.

[17] R Tijdeman. On a telephone problem. *Nieuw Archief voor Wiskunde*, 3:188–192, 1971.

[18] Hans Van Ditmarsch, Malvin Gattinger, Ioannis Kokkinis, and Louwe B Kuijer. Reachability of five gossip protocols. In *International Conference on Reachability Problems*, pages 218–231. Springer, 2019.

[19] Hans van Ditmarsch, Ioannis Kokkinis, and Anders Stockmarr. Reachability and expectation in gossiping. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 93–109. Springer, 2017.

[20] Hans van Ditmarsch, Jan van Eijck, Pere Pardo, Rahim Ramezanian, and François Schwarzentruber. Epistemic protocols for dynamic gossip. *Journal of Applied Logic*, 20:1–31, 2017.

[21] Hans van Ditmarsch, Jan van Eijck, Pere Pardo, Rahim Ramezanian, and François Schwarzentruber. Dynamic gossip. *Bulletin of the Iranian Mathematical Society*, 45(3):701–728, 2018.

Appendix A

Choice of Protocol

Topology	**Call mode**	**Protocol**	**Justification**
Complete	Sequential	LNS or ANY	Both have a 100% success rate. LNS has optimal execution length whereas ANY has optimal computation time. Choice of protocol depends on which is given higher value.
	Rounds	CO	CO requires the least number of rounds on average.
Incomplete	Sequential	ANY	ANY and TOK are the only protocols to have a 100% success rate, however ANY has superior execution lengths and computation times.
	Rounds	ANY	ANY and TOK are the only protocols to have a 100% success rate (except for TOK timing out on one occasion), however ANY requires less rounds to successfully terminate.
Dynamic	Sequential	SPI	From the set of protocols with a 100% success rate, TOK and SPI have the best average execution lengths. However SPI has superior computation times compared to TOK.
	Rounds	CO	I believe that CO would have a 100% success rate if the timeout criteria was more lenient. It has vastly superior average round counts than TOK and SPI, making CO a compelling choice despite it timing out on a few occasions.

Appendix B

Example Output

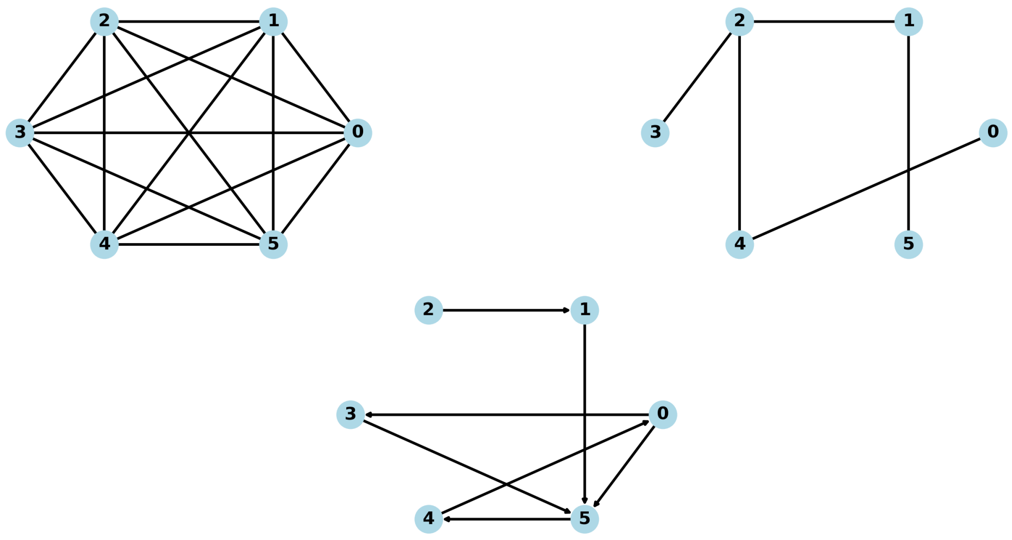
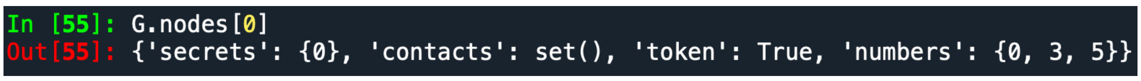
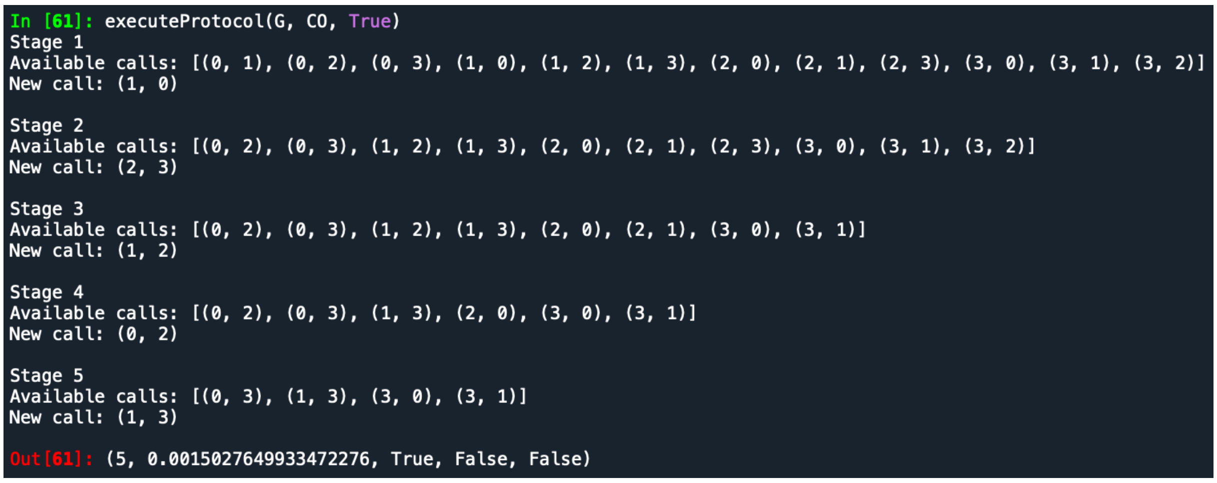


Figure B.1: Example gossip graphs produced by `completeGraph`, `incompleteGraph` and `diGraph` respectively.



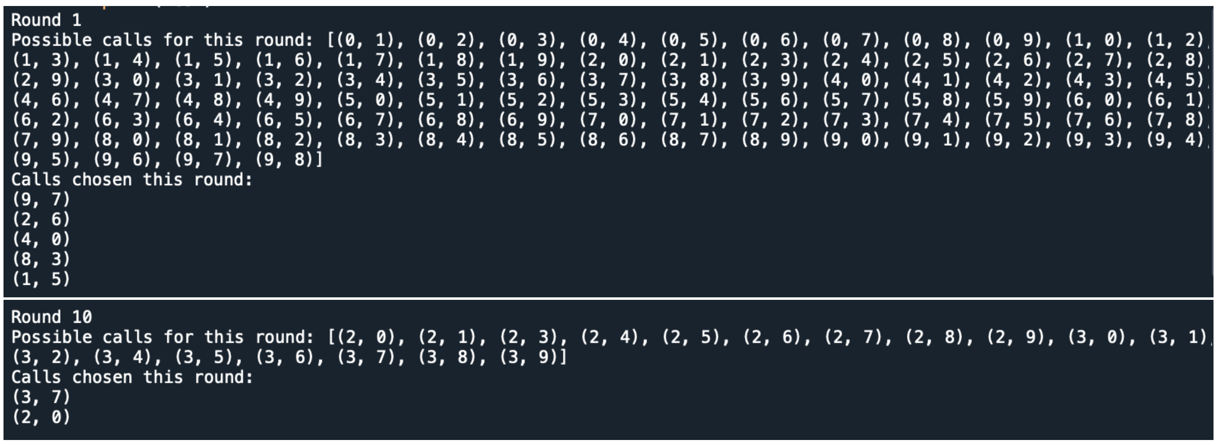
```
In [55]: G.nodes[0]
Out[55]: {'secrets': {0}, 'contacts': set(), 'token': True, 'numbers': {0, 3, 5}}
```

Figure B.2: Example properties of a node (namely node 0 from the digraph in figure B.1).



```
In [61]: executeProtocol(G, CO, True)
Stage 1
Available calls: [(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]
New call: (1, 0)

Stage 2
Available calls: [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]
New call: (2, 3)

Stage 3
Available calls: [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2, 1), (3, 0), (3, 1)]
New call: (1, 2)

Stage 4
Available calls: [(0, 2), (0, 3), (1, 3), (2, 0), (3, 0), (3, 1)]
New call: (0, 2)

Stage 5
Available calls: [(0, 3), (1, 3), (3, 0), (3, 1)]
New call: (1, 3)

Out[61]: (5, 0.0015027649933472276, True, False, False)
```

Figure B.3: Example breakdown of calls on a complete graph with 4 agents (CO protocol).



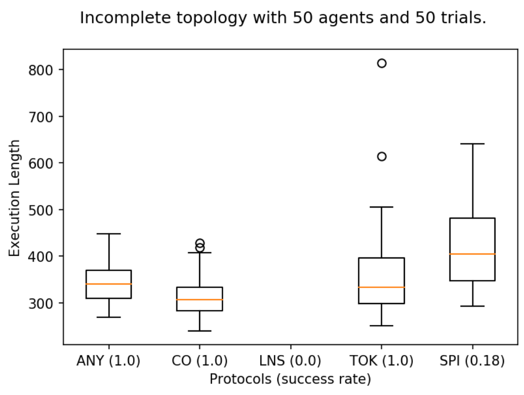
```
Round 1
Possible calls for this round: [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 0), (1, 2),
(1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 0), (2, 1), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8),
(2, 9), (3, 0), (3, 1), (3, 2), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (4, 0), (4, 1), (4, 2), (4, 3), (4, 5),
(4, 6), (4, 7), (4, 8), (4, 9), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 6), (5, 7), (5, 8), (5, 9), (6, 0), (6, 1),
(6, 2), (6, 3), (6, 4), (6, 5), (6, 7), (6, 8), (6, 9), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 8),
(7, 9), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 9), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4),
(9, 5), (9, 6), (9, 7), (9, 8)]
Calls chosen this round:
(9, 7)
(2, 6)
(4, 0)
(8, 3)
(1, 5)
```

```
Round 10
Possible calls for this round: [(2, 0), (2, 1), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (3, 0), (3, 1),
(3, 2), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9)]
Calls chosen this round:
(3, 7)
(2, 0)
```

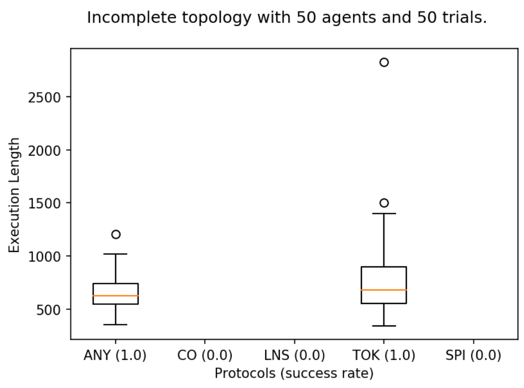
Figure B.4: Example output of the TOK protocol being executed in rounds (Complete topology, 10 agents).

Appendix C

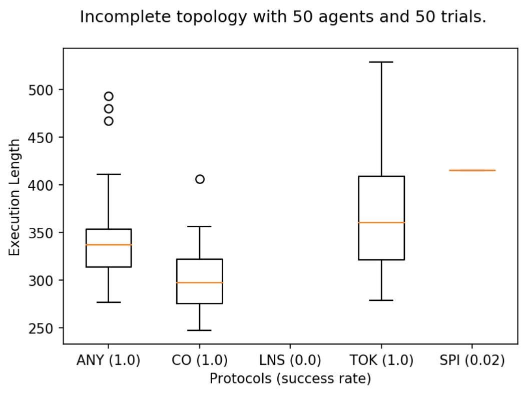
Extra Boxplots



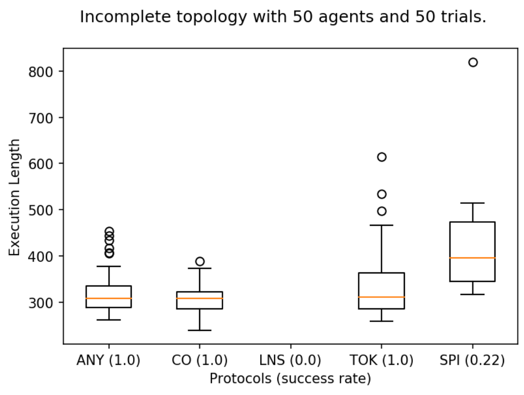
(a)



(b)

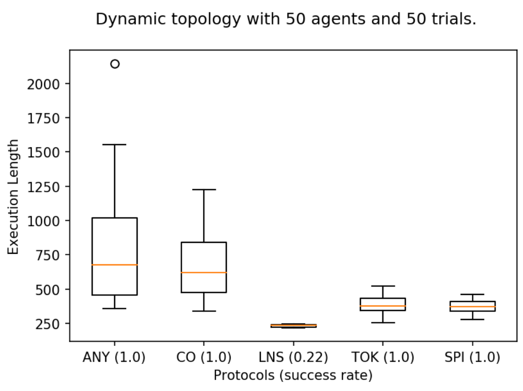


(c)

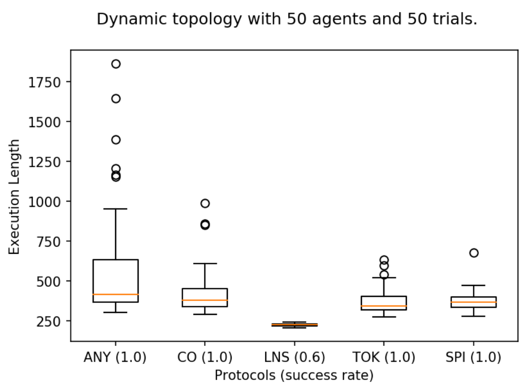


(d)

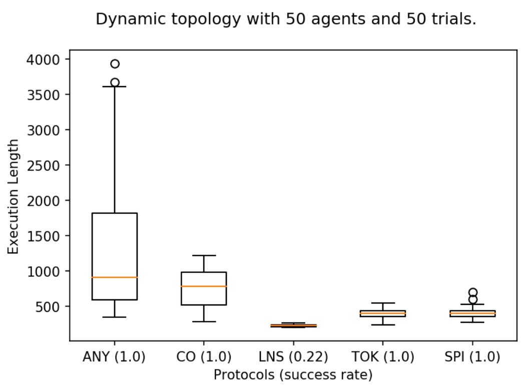
Figure C.1: Extra boxplots for the execution of the protocols on random incomplete graphs.



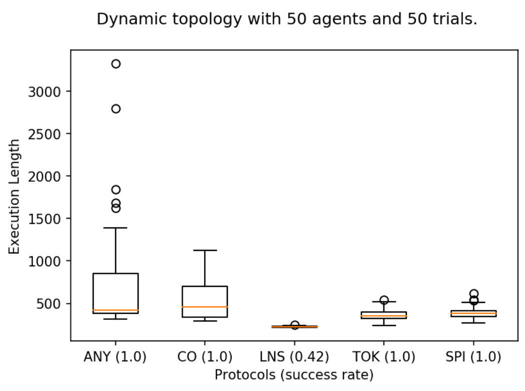
(a)



(b)



(c)



(d)

Figure C.2: Extra boxplots for the execution of the protocols on random dynamic gossip graphs.

Appendix D

Original Design Document

Note. This design document was originally submitted to the University of Liverpool on 17/07/2021 in partial fulfilment of the module COMP702. It is being given here as a reference to the original design elements of the project.

D.1 Summary of Proposal

D.1.1 Background

The Gossip Situation: Initially, a group of n agents ("gossips") each know a piece of information that is unique and not known by the others. The aim of a gossip scenario is to spread the information (or "secrets") amongst the agents such that all agents learn all possible secrets. This is achieved via "one-to-one" telephone calls between agents, at which time both agents participating in the call will divulge all the secrets they know to each other. At any given time if a call is taking place, exactly two agents are participating in the call, and all other agents are idle [5].

Example. Agents a, b and c each know a piece of information A, B and C respectively. The initial distribution of information is $a : \{A\}$, $b : \{B\}$, $c : \{C\}$. If a were to call b (denoted ab) then the distribution would become $a : \{A, B\}$, $b : \{A, B\}$, $c : \{C\}$ since a and b have exchanged secrets. If then a calls c followed by b calling c, the final distribution of secrets would be $a : \{A, B, C\}$, $b : \{A, B, C\}$, $c : \{A, B, C\}$, and all agents know all secrets.

The above gives a brief summary of a gossip situation in one of its most basic forms. Key questions still remain such as:

- Do agents know the telephone numbers of all other agents? [14] An agent may only call another if they have their telephone number. A common assumption in gossip theory is that all agents have the numbers of all other agents, but this is not always the case.
- If agents only know a subset of the telephone numbers of other agents, can information on numbers also be exchanged during calls? [21]
- In what order are calls executed? Are calls organised be a "central controller" or are the agents themselves responsible for making their own calls (referred to as "distributed gossip")? [21]

Answers to these questions, amongst others, dictate the various strands of research on gossip. After an agent has learnt all possible secrets, it is considered an *expert* [18]. In general, the goal is the make all agents experts. In particular, if we are able to make all agents experts after a sequence of calls, how many calls are required?

In reference to the third question posed in the list above, this project will focus mainly on *distributed gossip*; the case where "outside regulation" is not required [21]. This can be achieved by modelling the selection of calls as a random process that adheres to some pre-defined protocol condition [19]. The protocols in question will match those defined in [18]; namely ANY, CO, LNS, TOK and SPI (details to follow in sections D.2.2 to D.2.2).

Harary and Schwenk (1974) relate the idea of gossiping to mathematical graph theory, where nodes represent agents and edges represent possible calls [14]. In the most simple form of the problem, every agent can call every other agent, and thus the "communication network" for n agents is represented via the complete graph K_n. In general this is not always the case, and so the graph might be incomplete but connected. Here we assume that an edge (i.e. not a directed edge) between agents a and b means that a and b know each others number respectively. This is not necessarily the case when we consider dynamic gossip (to follow).

Finally we introduce the idea of *dynamic* gossip, where telephone numbers are exchanged during calls as well as secrets. We use a digraph to represent this scenario. Typically we use a dashed arc from agent a to agent b if a knows b's number, and a solid arc if a knows b's secret. This visualisation differs from a simple communication network since it portrays information on the distribution of secrets as well as the distribution of telephone numbers, and is referred to in the literature as a gossip graph [21].

D.1.2 Proposal

This project will aim to provide a Python implementation of five distributed epistemic gossip protocols for various network topologies, and return the number of calls required for a given protocol. We will consider complete gossip graphs (where all agents know everyone's telephone number), and incomplete gossip graphs (each agent knows a subset of other agents telephone numbers). A more complex implementation for *dynamic gossip* will also be given. In particular, we are interested in simulating various protocols, and inspecting which protocols are successful on each of the network topologies. A protocol is considered successful if performing a call sequence that satisfies that protocol results in all agents becoming experts [18]. From those protocols that are successful, we wish to determine which protocol is the most efficient (i.e. requires the least number of calls). More specifically, for a given communication network and for each of the five protocols, we will report how often the protocol terminates successfully (it may only be successful on a subset of executions), and from the set of successful terminations, which protocol is the most efficient. This means that a protocol is not required to be 100% successful to be compared to the others, but it's success rate must be reported.

Project Objectives

1. Provide sufficient non-technical background to the gossip problem, including some standard results for non-distributed gossip for the purpose of context. We also want to give a technical background to the gossip problem, including rigorous definitions to concepts such as (but not limited to) call sequences, gossip protocols, dynamic gossip and gossip graphs. Finally, other literature on gossip will be summarised so that the reader has an understanding of the foundations of gossip and the key results on the topic.

2. Write a Python program that allows the user to generate complete graphs with a specified number of vertices, incomplete (connected) graphs with a specified

number of vertices, and weakly connected directed graphs for use in dynamic gossip. Ideally this will also include the ability to produce visualisations of connection networks. The exact functionality depends on the library used, examples of which are NetworkX [1] and igraph [3].

3. Write a Python program that allows the user to simulate the following five network protocols on complete connected, incomplete connected, and weakly connected gossip (di)graphs: ANY, CO, LNS, TOK, SPI.

4. Recall known results of protocol success on various network topologies, and test these results using the protocol simulator. If the success of a protocol on a network topology is unknown (or at least we are ignorant to it), the simulator will be used to predict the result instead.

5. For a given network topology, analyse which protocol is most efficient from the set of successful protocols, and provide success rates for each protocol. This analysis might include reporting average execution lengths, and looking at the spread of execution lengths. We can also report the execution time of each protocol, and give thought to the order of execution time with respect to the number of agents. It may be interesting to compare the most efficient protocol in terms of execution time to that of shortest execution length.

D.1.3 Related work

Baker et al introduce the idea of gossip and provide a rigorous proof for the minimum number of calls required for a system with n agents in the non-distributed scenario [9]. Apt et al introduce the concept of *distributed* gossip, and formally define protocols "in terms of correctness, termination, and fair termination" [5]. The reachability of various protocols have been explored in [18] and [19]. In particular, Ditmarsch et al simulate protocols on complete connected graphs, and give results on the expected run time of LNS and CO [19]. Harary and Schwenk explore the gossip problem on incomplete (connected) graphs, and determine that the minimum number of required calls is equal to that of a given graphs spanning tree [14]. Ditmarsch et al focus on *dynamic* gossip, and define several gossip protocols in such a scenario. Results regarding the success of these protocols is given for different initial communication network topologies [21]. [20] also explores dynamic gossip, but for protocols in which agents also have information on what knowledge other agents already possess, which "cannot be as straightforwardly related to a graph property".

D.2 System Design

As mentioned in the previous section, the main software implementation for this project will allow the user to simulate a selection of gossip protocols on two different network topologies in Python. These topologies are connected graphs (including the specialisation of complete connected graphs) and weakly connected digraphs (for dynamic gossip). Indeed the scenario where a graph is not (weakly) connected has a trivial outcome: it is impossible for all agents to become experts and the protocol will never terminate [21].

Whilst the code will only be intended for use within the context of this project, possible extensions include the development of a Python library and supporting documentation to allow other users to simulate these protocols.

The Python code can be split in to two main components:

- The creation of randomly generated (weakly) connected networks. Ensuring that these graphs are (weakly) connected is not a straight-forward task.
- The execution of gossip protocols on initial gossip graphs, including updating the information held by each agent throughout the protocol, and updating the connection network in the case of dynamic gossip.

D.2.1 Graph Creation

To save on time, the Python library NetworkX will be used for this section [1]. NetworkX provides a nice implementation of mathematical graph theory in Python, and contains several features that will become useful for this project. In particular:

1. The ability to create a (di)graph of a specified number of nodes. This will allow us to vary the number of nodes, and analyse each protocol as the number of agents increases.
2. The ability to store information alongside each node. This will become particularly useful when we want to keep track of the distribution of secrets after each call. We can also store other information such as which agents a given agent has previously spoken to (this can be used in CO protocol for example; see section D.2.2).
3. A selection of methods to automate the random creation of graphs. There are also elementary methods for creating graphs such as adding individual nodes and edges; this might be useful for implementing a random graph generator from scratch.
4. The ability to create customisable visualisations of networks.

NetworkX provides a Python structure for a graph object, and allows us to access information on each individual node, edge, and other useful graph features such as adjacency matrices.

Undirected graphs

Note. The creation of complete connected graphs on n vertices is trivial. NetworkX contains a method "`complete_graph(n)`" to create such a graph.

The idea behind the chosen method for generation of random connected graphs is based on Corollary 2.4.1 from [11]: "Every connected graph contains a spanning tree." If we first create a random spanning tree G, and then add an arbitrary number of edges such that G remains simple, we can ensure that the resulting graph is connected. This method was suggested by StackOverflow user Wesley Baugh [10]. NetworkX contains a method for the creation of random trees on n vertices. Also note from [11] that a tree with n vertices has $n-1$ edges and the complete graph K_n has $nC2$ edges. Hence once we have created a tree with n vertices, we need to add at most

$$\epsilon := nC2 - (n-1)$$

extra edges. The pseudo-code might look as follows.

```
GenerateGraph(n)
(1) Generate random tree with n vertices G (using NetworkX)
(2) $\epsilon$ = nC2 - (n-1)
(3) Generate random number r between 0 and $\epsilon$
```

```
(4) While number of edges in G < (n-1)+r:
(5)     Choose random vertices u and v such that (u,v) is not an
edge
(6)     Add edge (u,v) to G
(7) od
(8) Assign initial secret to each node in G
(9) Assign list of past contacts to each node (initially empty)
(10) Assign Token = True for each node
(11) Output graph G
```

The condition on line 5 can be checked using the adjacency matrix of G (i.e. $A_{u,v} = 0$ if there is no edge between u and v). For each agent a we need to store the following information:

- Set of known secrets (initially each agent only knows its own secret)
- List of agents a has been in contact with (initially an empty list)
- Each agent starts with a token, signified by a variable token = True (purpose of tokens explained later)

Directed graphs

Frieze and Karoński describe a random digraph $\mathbb{D}_{n,p}$ which has n vertices, and "each of the $n(n-1)$ possible edges occurs independently with probability p" [12]. Such a graph is straightforward to generate, although there is no guarantee that the resulting digraph would be weakly connected. We could implement a check for weak connectedness and randomly generate digraphs until this condition is satisfied, but this could be very inefficient for a large number of agents.

Instead, I opt to extend the idea given in [10] for digraphs. Consider generating a random *undirected* tree as in section D.2.1, and then for each of the $n-1$ edges choosing a random direction. Then arbitrarily add a random number directed edges up to a maximum of $2(nC2)$ in total. Pseudo-code:

```
GenerateDigraph(n)
(1) Generate random tree with n vertices G (using NetworkX)
(2) For each edge in G:
(3)     Choose random direction, update G
(4) $\epsilon$ = 2*(nC2) - (n-1)
(5) Generate random number r between 0 and $\epsilon$
(6) While number of edges in G < (n-1)+r:
(7)     Choose random vertices u and v such that (u,v) is not an
arc
(8)     Add arc (u,v) to G
(9) od
(10) Assign initial secret to each node in G
(11) Assign list of past contacts to each node (initially empty)
(12) Assign Token = True for each node
(13) Output digraph G
```

D.2.2 Gossip Protocols

Ditmarsch et al outline two possible methods for selecting a call to be made in a distributed system:

- From the set of permitted calls that can be made, select one at random and execute that call. This is equivalent to selecting an edge from the communication network such that the edge represents a permitted call.
- Alternatively, randomly select an agent that can make at least one permitted call, and then select a call to be made by that agent [19].

We will opt for the first option. The the pseudo-code of a generic protocol P might look as follows. The function will take two arguments: an initial communication network G, and a pre-defined protocol function P(G). The protocol function will take a graph object G as an argument, and return a list of available calls. Individual protocol functions will be discussed next.

```
executeProtocol(Communication Network G, Protocol function P)
(1) Number of calls c = 0, Success = False, Terminate = False
(2) While Terminate == False:
(3)     If all agents are experts:
(4)         Success = True
(5)         break
(6)     Calls = P(G)
(7)     If Calls is non-empty:
(8)         Select random call from Calls
(9)         Execute chosen call, update G accordingly
(10)         c = c + 1
(11)     Else if Calls is empty:
(12)         Terminate = True
(13) Return (c, Success)
```

A variable 'Calls' is populated with a list of available calls via the protocol function P. A random call is selected and executed from 'Calls' if it is non-empty, otherwise another variable 'Terminate' is set to True. Once the protocol has terminated, a variable 'Success' is set to true if all agents are experts.

Now we consider each protocol condition separately. The following definitions and pseudo-code are based on [18]. Each protocol will have its own function, taking a communication network G as input. Each node (agent) of G, say a, will have the following attributes:

- a['secrets'] - a list of secrets known by agent a
- a['token'] - true if a possesses a token, false otherwise
- a['contacts'] - the set of agents that a has previously been in contact with

ANY

Under the ANY protocol, there are no extra restrictions on calls that can be made. i.e. agent a can call b as long as a has b's telephone number. Pseudo-code:

```
ANY(Communication Network G)
(1) A = adjacency matrix of G
```

```
(2) n = length of first row of A (number of agents)
(3) Calls = [] (empty list)
(4) for i in the range 1 to n:
(5)     for j in the range 1 to n:
(6)         if $A_{i,j} = 1$:
(7)             Add (i,j) to Calls
(8) Return Calls
```

The pseudo-code of the remaining protocols only differ to ANY by line 6, and so only this line will be provided.

CO

"Call once": each pair of agents may only speak once. Thus if a and b have been in direct contact in the past, the calls ab and ba are not candidates for selection.

```
(6) if $A_{i,j} = 1$ and node i not in node j['contacts']:
```

LNS

"Learn new secrets": Calls can only be made if the caller does not know the callee's secret. i.e. using notation from the example in section D.1.1, a may only call b if a does not know B.

```
(6) if $A_{i,j} = 1$ and node j's secret not in node i['secrets']:
```

TOK

"Token": Initially each agent possesses a token. Agent a may only make a call if a possesses a token. When agent a calls b, a's token is passed over to b. At the end of a call, each agent may only have at most one token (any agent with two must discard one). Therefore we will use a boolean variable for each agent to denote whether it has a token (true) or not (false).

```
(6) if $A_{i,j} = 1$ and node i['token'] = True:
```

SPI

"Spider": This protocol is similar to TOK, except the callee must pass their token to the caller instead. Furthermore, agents cannot retrieve a new token once they have lost theirs. Agents must still discard tokens at the end of each call if they have more than one.

```
(6) if $A_{i,j} = 1$ and node i['token'] = True:
```

Note. Protocol functions are only for finding suitable calls. Other important actions take place on line 9 of the function `executeProtocol` after a call has been chosen. When agent a calls b, the following actions must be taken:

- a's known secrets must be added to b's known secrets and vice versa (numbers are also exchanged in the dynamic case)
- a must be added to b['contacts'] and vice versa
- For TOK, set a['token'] = False and b['token'] = True. For SPI, set b['token'] = False

D.2.3 Central Method

Finally, the two key components of the program must be able to operate independently for convenience, but they will also be merged into a central function that will allow us to bring together all components of testing protocols on random networks. The function will need four inputs: choice of protocol, choice of network topology, number of agents, and the number of iterations. The function will output the success rate, and a list containing the execution length of each iteration.

D.3 Evaluation Design

D.3.1 System Evaluation

First we discuss evaluation of the system in terms of it's success and efficiency. The following criteria will be used to assess the system:

- **Validity.** Results produced by the system must be realistic, and reproducible (within reason). Though random executions of protocols will produce different results, the number of trials should be sufficiently large such that redoing the experiment should output similar results. Furthermore, results should be consistent with known results from the literature. For example, from [21] we know that TOK should successfully terminate on weakly connected digraphs in a dynamic setting. Therefore our system should be consistent with this analysis for a sufficiently small number of agents (for a very large number of agents the protocol could fail to successfully terminate due to computational limitations).

- **Efficiency.** We want to use the most efficient methods possible, and minimise the number of redundant operations. For example, ANY will return the same list of possible calls throughout the protocols execution in a non-dynamic setting, and so there is no need to run the ANY function each time we want to select a call. Efficiency evaluation also includes analysing whether a protocol has terminated appropriately. For a given communication network of 100 or so agents, we aim for each of the protocols to terminate within a maximum of a few seconds.

- **Convenience.** Switching between protocols and network topologies should require minimal effort, e.g. changing the value of an input variable for a function. This criteria would be failed should it feel tedious to change various features of the system. The functions and methods must be usable separately, but also usable in conjunction with minimal editing. This is the purpose of the central function; so that the experiment and analysis stage of the project can be carried out in a more straightforward manner.

- **Presentation.** Though the code is not specifically intended for external use, it will still be provided in the appendices of the paper, and thus must be clear, concise and appropriately commented.

The final paper should discuss each of these topics and give sound reasoning as to why a particular criteria has (or has not) been met, including what steps and thought processes have been followed to ensure the system is as fit-for-purpose as possible. The criteria will be evaluated during the results phase (see section D.5), so that important changes can be made before any analysis is conducted.

D.3.2 Results Evaluation

Once the system has been fine tuned, we then produce the desired set of results. The results and analysis will aim to answer the research question:

Which protocol is the most efficient on each of the network topologies?

In general, if the results cannot be used to answer this question (e.g. by calculating average execution lengths), then they are not satisfactory. We want to make a comparison between protocols based separately on success rate, and average execution length. For example, if protocol P has a very high success rate but high execution length, and Q has a low success rate but low execution length, one could argue P is still more suitable since it terminates successfully much more often. As results are being produced, we can also time how long it takes for each of the protocols to run, and plot an average execution time for each protocol as we increase the number of agents.

The main purpose of the central method is to efficiently simulate various protocols on various network topologies. For each of the three cases (complete, incomplete and dynamic), we wish to conduct the following experiment:

1. Vary the number of agents n within a suitable range (actual bounds for n are to be determined).
2. For each fixed n, generate an initial connection network of the relevant topology.
3. For each protocol P, simulate the exchange of information until termination, note if the protocol was successful, and record the execution length if so. Repeat this step for a fixed (specifiable) number of trials.
4. Now for various values of n, we have a list of execution lengths for each protocol, and a success rate for each protocol.

D.4 Required Data

The only data to be used in the project are the various networks used in the testing of the system and production of results. These networks will be randomly generated in Python, and will not be retrieved from external sources. Furthermore, no human participants will be evaluating the system, and no human data will be used.

D.5 Project Plan

D.5.1 Work Breakdown and Milestones

Figure D.1 shows a breakdown of the tasks to be completed during the project. I have identified the following milestones:

- **M1: Completion of Background reading.** Completion of this stage is of paramount importance for my own understanding of the project topic. Poor understanding of gossip carried forward from this stage would negatively affect the rest of the project.
- **M2: Completion of Design Document.** This stage will set a framework for the project which will allow progress to be monitored in the following weeks.

- **M3: Completion of Python Scripts.** Once this stage has been completed, all tools needed to simulate the protocols should be available and working as anticipated.
- **M4: Completion of Experiments and Analysis.** This stage will give purpose to the system that has been created.
- **M5: Completion of Dissertation.** The final deliverable for the project which will bring together everything that has been achieved.

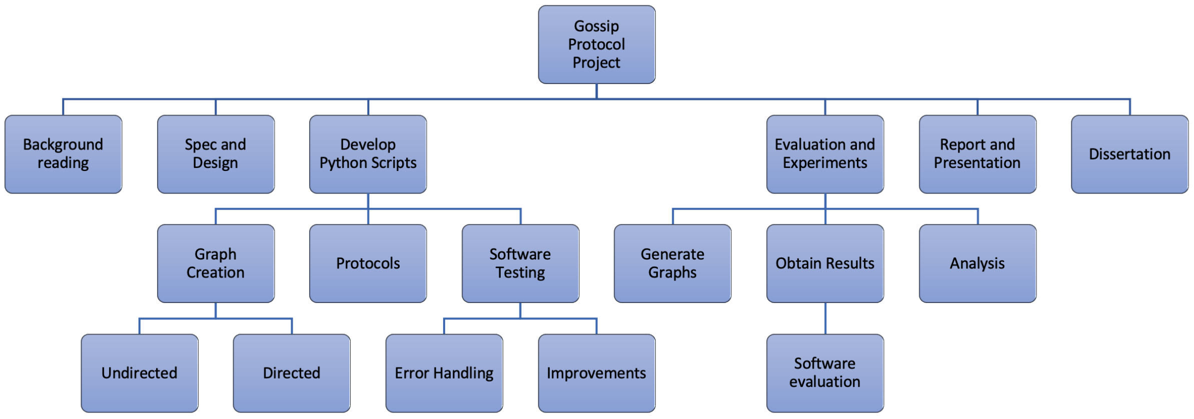


Figure D.1: Work Breakdown Structure

Activity	Effort	Duration
1 Background Reading	1 week	2 weeks
2 Spec and Design	1.5 weeks	4 weeks
3 Undirected Graph Function	1 day	4 days
4 Directed Graph Function	1 day	4 days
5 Protocol Functions	1.5 weeks	3 weeks
6 Error Handling	4 days	1 week
7 Code Testing and Improvements	4 days	1 week
8 Generate Graphs	1 day	4 days
9 Obtain Results	1 day	4 days
10 Analyse Results	1.5 weeks	3 weeks
11 Report and Final Presentation	1 week	2 weeks
12 Dissertation	2 weeks	3 weeks

Table D.1: Duration and Effort for each task

D.5.2 Time estimates

Table D.1 gives a breakdown of the project activities along with the expected effort required and predicted duration of each task. The sequencing of these activities is also given in the Gantt chart in figure D.2, where the number given in each bar is the expected number of days required for each task.

Note that the "Background Reading" section has already been completed, and the duration of the "Spec and Design" section has been extended by a week due to unforeseen circumstances. Furthermore, the "Protocol Functions" section can be further broken down into each of the five protocols, followed by the "`executeProtocol`" function and the central function. These protocol functions are not given a specific order in which they will be written because they directly influence each other, and changes will be made to each of them throughout this section.

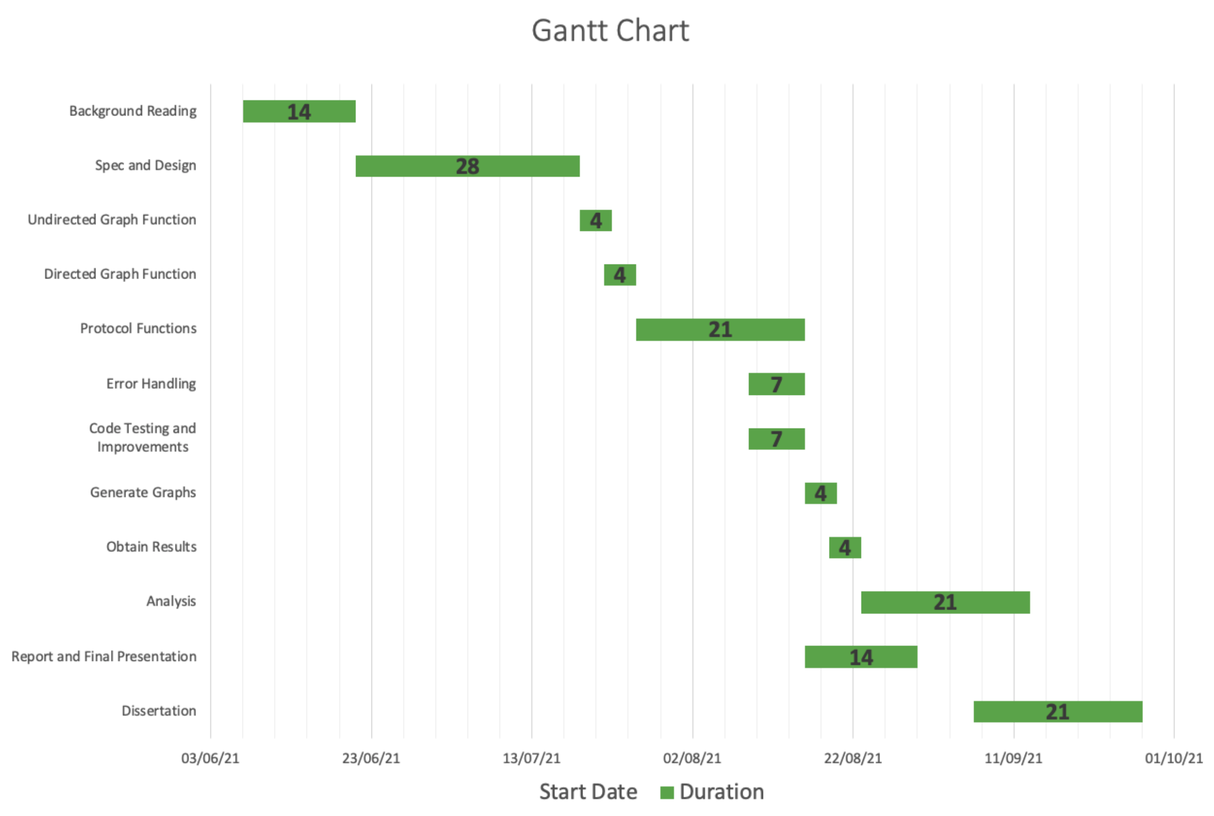


Figure D.2: Gantt chart showing the sequencing of project activities (made following the guide in [16]).

Appendix E

Installation and Code Listing

E.1 Installation

This project uses Python 3.7, preferably within the Spyder IDE (version 5.0.5). The following modules are required: networkX, matplotlib.pyplot (as plt), matplotlib, numpy (as np), math, random, time, copy and pickle. These can be imported using the following code preamble:

```python
# Install modules
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams["figure.dpi"] = 150
import numpy as np
from math import factorial
import random
import time
import copy
import pickle
```

E.2 Graph Generation

We define three functions in order to generate complete, incomplete, and dynamic directed gossip graphs. Agents are also assigned a set of known secrets, and we output an initial gossip situation (G, S).

```python
# ==============================================================
#  Complete (gossip) graph generator
# ==============================================================

# completeGraph(n, plot) - generate a complete gossip graph on
#                          n vertices
# Parameters: n - number of agents/nodes
#             plot - boolean, dicates if final network is plotted
# Returns - complete (gossip) graph object with n nodes

def completeGraph(n, plot = False):
    # Initialise complete graph using nx
    G = nx.complete_graph(n)
    # Indicate G is an initial gossip graph
    G.graph['initial'] = True

    # Add attributes to nodes (agents)
    for agent in list(range(n)):
        # Initial secret
        G.nodes[agent]['secrets'] = {agent}
        # Contact list (initially empty)
        G.nodes[agent]['contacts'] = set()
        # Token (boolean initially set to true)
        G.nodes[agent]['token'] = True
        # Initial list of known numbers
        G.nodes[agent]['numbers'] = set(G.adj[agent]).union({agent})

    # Plot the graph
    if (plot):
        options = {'node_color': 'lightblue',
                   'node_size': 1000,
                   'width': 3,
                   'with_labels' : True,
                   'font_weight' : 'bold',
                   'font_size' : 20
                   }
        nx.draw_circular(G, **options)

    # Return the graph
    return G


# ================================================================
#  Random incomplete (gossip) graph generator
# ================================================================

# incompleteGraph(n) - generate a random incomplete gossip graph
#                      on n vertices
# Parameters: n - number of agents/nodes
#             plot - boolean, dicates if final network is plotted
# Returns - graph object with n nodes

def incompleteGraph(n, plot = False):
    # Generate initial random tree using nx
    G = nx.random_tree(n)
    # Indicate G is an initial gossip graph
    G.graph['initial'] = True

    # Calculate maximum number of extra edges to add
    nC2 = factorial(n)/(2 * factorial(n - 2))
    epsilon = nC2 - (n-1)

    # Generate random number of edges to add
    r = random.randint(0, epsilon)

    # Add edges to G until limit reached
    while (G.number_of_edges() < (n-1) + r):
        # Pick random pair of nodes in G
        i = random.randint(0, n-1)
```

```
            j = random.randint(0, n-1)

            # Add edge (i,j) to G if edge doesn't already exist
            if (((i,j) not in G.edges) and (i != j)):
                G.add_edge(*(i,j))

        # Add attributes to nodes
        for agent in list(range(n)):
            # Initial secret
            G.nodes[agent]['secrets'] = {agent}
            # Contact list (initially empty)
            G.nodes[agent]['contacts'] = set()
            # Token (boolean initially set to true)
            G.nodes[agent]['token'] = True
            # Initial list of known numbers
            G.nodes[agent]['numbers'] = set(G.adj[agent]).union({agent})

        # Plot the graph
        if (plot):
            options = {'node_color': 'lightblue',
                       'node_size': 1000,
                       'width': 3,
                       'with_labels' : True,
                       'font_weight' : 'bold',
                       'font_size' : 20
                       }
            nx.draw_circular(G, **options)

        # Return the graph
        return G


    # ================================================================
    #  Random (gossip) digraph generator
    # ================================================================

    # diGraph(n) - generate a random gossip digraph on n vertices
    # Parameters: n - number of agents/nodes
    #             plot - boolean, dicates if final network is plotted
    # Returns - graph object with n nodes

    def diGraph(n, plot = False):
        # Generate intial random tree
        G = nx.random_tree(n)

        # Create digraph using the edges of G (each edge becomes two arcs)
        G = nx.DiGraph(G)

        # Fetch adjacency matrix of G
        A = np.array(nx.adjacency_matrix(G).todense())

        # Loop through all pairs of vertices
```

```
for i in list(range(n)):
    for j in list(range(i)):
        # Check if an arc exists between nodes i and j
        # (and hence vice versa)
        if (A[i,j] == 1):
            # If so, remove either arc (i,j) or (j,i) (at random)
            rand = random.randint(0,1)
            if (rand == 0):
                A[i,j] = 0
            else:
                A[j,i] = 0

# Convert matrix back into NetwrokX digraph object
G = nx.convert_matrix.from_numpy_matrix(A, create_using=(nx.DiGraph))
# Indicate G is an initial gossip graph
G.graph['initial'] = True

# Calculate maximum number of extra arcs to add (restricted)
nC2 = factorial(n)/(2 * factorial(n - 2))
epsilon = int(((2 * nC2) - (n-1))/10)

# Generate random number of arcs to add
r = random.randint(0, epsilon)

# Add arcs to G until limit reached
while (G.number_of_edges() < (n-1) + r):
    # Pick random pair of nodes in G
    i = random.randint(0, n-1)
    j = random.randint(0, n-1)

    # Add arc (i,j) to G if arc doesn't already exist
    if (((i,j) not in G.edges) and (i != j)):
        G.add_edge(*(i,j))

# Add additional attributes to nodes
for agent in list(range(n)):
    # Initial secret
    G.nodes[agent]['secrets'] = {agent}
    # Contact list (initially empty)
    G.nodes[agent]['contacts'] = set()
    # Token (boolean initially set to true)
    G.nodes[agent]['token'] = True
    # Initial list of known numbers
    G.nodes[agent]['numbers'] = set(G.adj[agent]).union({agent})

# Plot graph
if (plot):
    options = {'node_color': 'lightblue',
               'node_size': 1000,
               'width': 3,
               'with_labels' : True,
               'font_weight' : 'bold',
               'font_size' : 20
               }
    nx.draw_circular(G, **options)

# Return the graph
return G
```

E.3 Useful Tools

We define a function that checks for an all-expert state, and a function that checks if an inputted call (i, j) is p-permitted.

```
# ================================================================
#  Useful tools
# ================================================================

# experts(G) - checks if all agents of G are experts
# Parameters: G - graph object, generated in "graph_generator.py"
# Returns: s - boolean, true if all agents experts, false otherwise

def experts(G):
    # Fetch number of nodes
    n = G.number_of_nodes()

    # Intialise s as true
    s = True

    # Loop through all nodes of G
    for agent in list(range(n)):
        # If an agent does not know all secrets,
        # set s to false and break
        if(len(G.nodes[agent]['secrets']) < n):
            s = False
            break

    return s


# pPermitted(G, P, caller, callee) - checks if (caller, callee) is a P
#                                    permitted call
# Parameters: G - graph object,
#             P - predefined protocol function,
#             caller - int, index of caller agent
#             callee - int, index of callee agent
# Returns: boolean - true if call is permitted, false otherwise

def pPermitted(G, P, caller, callee):
    # ANY - always p permitted
    if (P == ANY):
        return True
    # CO - p permitted if agents have not been in contact
    elif (P == CO):
        if (caller not in G.nodes[callee]['contacts']):
            return True
    # LNS - p permitted if caller does not know callee's secret
    elif (P == LNS):
        if (callee not in G.nodes[caller]['secrets']):
            return True
    # TOK and SPI - p permitted if caller has a token
    elif (P == TOK or P == SPI):
        if (G.nodes[caller]['token']):
            return True
    return False
```

E.4 Protocol Execution

We define a function that executes a protocol sequentially and a function to execute a protocol in rounds. We also define five individual protocol functions that return a list of permitted calls at each stage of a protocol execution.

```
# ================================================================
#  Protocol execution function
# ================================================================

# executeProtocol(G, P, breakdown) - executes protocol P on graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             P - function,  returns list of P permitted calls
#             breakdown - boolean, if true a breakdown of calls made and
#                         available calls at each stage given
# Returns: c - execution length
#          timer - execution time
#          success - boolean, indicates if all agents are experts
#                    at termination
#          failure - boolean, indicates if protocol has failed
#          timeout - boolean, indicates if protocol has timed out

def executeProtocol(G, P, breakdown = False):
    # Intitialise number of calls and success boolean
    c, success = 0, False
    # Initialise timeout and failure booleans
    timeout, failure = False, False

    # Initialise list of available calls and newest call
    calls = []
    newcall = 0

    # Record initial time
    initTime = time.perf_counter()

    # Execute protocol (each loop corresponds to one call or termination)
    while (True):
        # Check if all agents are experts
        if (experts(G)):
            success = True
            break
```

```
# Fetch list of available calls
calls = P(G, newcall, calls)

if (breakdown):
    print("Stage "+str(c + 1))
    print("Available calls: "+str(calls))

# If calls is non-empty
if (len(calls) != 0):
    # Select random call
    newcall = random.choice(calls)

    if (breakdown):
        print("New call: "+str(newcall)+"\n")
    # i is the caller, j is the callee
    i, j = newcall[0], newcall[1]

    # Execute new call

    # Exchange secrets
    G.nodes[i]['secrets'] = G.nodes[i]['secrets'].union(
                            G.nodes[j]['secrets'])
    G.nodes[j]['secrets'] = G.nodes[i]['secrets']

    # Add each agent to each others past contacts
    G.nodes[i]['contacts'].add(j)
    G.nodes[j]['contacts'].add(i)

    # Exchange tokens
    # TOK
    if (P == TOK):
        G.nodes[i]['token'] = False
        G.nodes[j]['token'] = True
    # SPI
    if (P == SPI):
        G.nodes[j]['token'] = False

    # Increment call counter
    c += 1

    # Indicate G is no longer an initial gossip graph
    G.graph['initial'] = False

    # Update arcs of G in dynamic case
    if (type(G) == nx.classes.digraph.DiGraph):
        # Update phone number lists
        G.nodes[i]['numbers'] = G.nodes[i]['numbers'].union(
                                G.nodes[j]['numbers'])
        G.nodes[j]['numbers'] = G.nodes[i]['numbers']

        # Add relevant arcs to graph
        # Loop through known numbers of caller
```

```
                for number in G.nodes[i]['numbers']:
                    # Add any new arcs to G
                    if (((i, number) not in G.edges) and (i != number)):
                        G.add_edge(i, number)
                        # Add new P permitted calls to calls
                        if pPermitted(G, P, i, number):
                            calls.append((i, number))

                # Similarly for the callee
                for number in G.nodes[j]['numbers']:
                    if (((j, number) not in G.edges) and (j != number)):
                        G.add_edge(j, number)
                        # Add new P permitted calls to calls
                        if pPermitted(G, P, j, number):
                            calls.append((j, number))

        # Else if calls is empty, break from loop
        elif (len(calls) == 0):
            failure = True
            break

        # Record current time
        curTime = time.perf_counter()
        # Time elapsed
        timer = curTime - initTime

        # Timeout
        if (timer > G.number_of_nodes()/10):
            timeout = True
            break

    return (c, timer, success, failure, timeout)


# ===============================================================
#  Protocol execution function (rounds variant)
# ===============================================================

# executeRounds(G, P) - executes protocol P on gossip graph G
#                       (in rounds of calls)
# Parameters: G - graph object, generated in "graph_generator.py"
#             P - function,  returns list of P permitted calls
#             breakdown - boolean, if true function will return
#                         breakdown of calls made in each round
# Returns: r - number of rounds performed
#          timer - execution time
#          success - boolean, indicates if all agents are experts
#                    at termination
#          failure - boolean, indicates if protocol has failed
#          timeout - boolean, indicates if protocol has timed out

def executeRounds(G, P, breakdown=False):
```

```
# Intitialise number of rounds and success boolean
r, success = 0, False
# Initialise failure and timeout booleans
failure, timeout = False, False

# Initialise list of available calls and newest call
newcall = 0
calls = []
calls = P(G, newcall, calls)

# Record initial time
initTime = time.perf_counter()

# Execute protocol (each loop corresponds to one round)
while (True):
    # Check if all agents are experts
    if (experts(G)):
        success = True
        break

    # If calls is empty, break from loop
    if (len(calls) == 0):
        failure = True
        break

    if (breakdown):
        print("Round "+str(r + 1))
        print("Possible calls for this round: "+str(calls))

    # Create copy of available calls
    roundCalls = copy.deepcopy(calls)
    # Initialise set of agents that have participated
    # in this round
    participants = set()

    if (breakdown):
        print("Calls chosen this round:")

    # Perform calls whilst list of round calls in non-empty
    while (len(roundCalls) != 0 and len(participants) <
            (G.number_of_nodes() - 1)):
        # Select random call
        newcall = random.choice(roundCalls)

        # i is the caller, j is the callee
        i, j = newcall[0], newcall[1]
        # If i or j already participated, remove (i,j) from calls
        # and select new pair
        if ((i in participants) or (j in participants)):
            roundCalls.remove(newcall)
            continue
        # Else execute new call
```

```
else:
    participants.add(i)
    participants.add(j)
    if (breakdown):
        print(newcall)

# Exchange secrets
G.nodes[i]['secrets'] = G.nodes[i]['secrets'].union(
                        G.nodes[j]['secrets'])
G.nodes[j]['secrets'] = G.nodes[i]['secrets']

# Add each agent to each others past contacts
G.nodes[i]['contacts'].add(j)
G.nodes[j]['contacts'].add(i)

# Exchange tokens
# TOK
if (P == TOK):
    G.nodes[i]['token'] = False
    G.nodes[j]['token'] = True
# SPI
if (P == SPI):
    G.nodes[j]['token'] = False

# Indicate G is no longer an initial gossip graph
G.graph['initial'] = False

# Update initial list of calls for next round
calls = P(G, newcall, calls)

# Update arcs of G in dynamic case
if (type(G) == nx.classes.digraph.DiGraph):
    # Update phone number lists
    G.nodes[i]['numbers'] = G.nodes[i]['numbers'].union(
                            G.nodes[j]['numbers'])
    G.nodes[j]['numbers'] = G.nodes[i]['numbers']

    # Add relevant arcs to graph
    # Loop through known numbers of caller
    for number in G.nodes[i]['numbers']:
        # Add any new arcs to G
        if (((i, number) not in G.edges) and (i != number)):
            G.add_edge(i, number)
            # Add new P permitted calls to calls
            if pPermitted(G, P, i, number):
                calls.append((i, number))

    # Similarly for the callee
    for number in G.nodes[j]['numbers']:
        if (((j, number) not in G.edges) and (j != number)):
            G.add_edge(j, number)
            # Add new P permitted calls to calls
```

```
                        if pPermitted(G, P, j, number):
                            calls.append((j, number))

        if (breakdown):
            print("\n")

        # Increment round counter
        r += 1

        # Record current time
        curTime = time.perf_counter()
        # Time elapsed
        timer = curTime - initTime

        # Timeout
        if (timer > G.number_of_nodes()/10):
            timeout = True
            break

    return (r, timer, success, failure, timeout)



# ================================================================
#  Individual protocol functions
# ================================================================

# ANY(G, newcall, calls) - returns ANY permitted calls for graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             newcall - tuple, latest call made
#             calls - list of permitted calls in the previous step
# Returns - calls, updated list of permitted calls

def ANY(G, newcall, calls):

    # If G is an initial gossip graph, add all edges/arcs from G
    if (G.graph['initial'] or newcall == 0):
        H = copy.deepcopy(G)
        H = nx.DiGraph(H)
        calls = list(H.edges)

    return calls




# CO(G, newcall, calls) - returns CO permitted calls for graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             newcall - tuple, latest call made
#             calls - list of permitted calls in the previous step
# Returns - calls, updated list of permitted calls
```

```
def CO(G, newcall, calls):

    # If G is an initial gossip graph, add all edges/arcs from G
    if (G.graph['initial'] or newcall == 0):
        H = copy.deepcopy(G)
        H = nx.DiGraph(H)
        calls = list(H.edges)

    # Latest call is (i,j)
    # If G not initial, remove (i,j) and (j,i) from calls
    else:
        calls.remove(newcall)
        if (newcall[::-1] in calls):
            calls.remove(newcall[::-1])

    return calls



# LNS(G, newcall, calls) - returns LNS permitted calls for graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             newcall - tuple, latest call made
#             calls - list of permitted calls in the previous step
# Returns - calls, updated list of permitted calls

def LNS(G, newcall, calls):

    # If G is an initial gossip graph, add all edges/arcs from G
    if (G.graph['initial'] or newcall == 0):
        H = copy.deepcopy(G)
        H = nx.DiGraph(H)
        calls = list(H.edges)

    # Say the latest call is (i,j)
    # If G not initial, remove calls (i,k) where i knows k's secret
    # Similarly remove calls (j,k) such that j knows k's secret
    else:
        i = newcall[0]
        # For each secret k that i knows
        for secret in G.nodes[i]['secrets']:
            # Check if (i,k) in calls and remove if so
            if ((i, secret) in calls):
                calls.remove((i, secret))

        j = newcall[1]
        # For each secret k that j knows
        for secret in G.nodes[j]['secrets']:
            # Check if (j,k) in calls and remove if so
            if ((j, secret) in calls):
                calls.remove((j, secret))

    return calls
```

```
# TOK(G, newcall, calls) - returns TOK permitted calls for graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             newcall - tuple, latest call made
#             calls - list of permitted calls in the previous step
# Returns - calls, updated list of permitted calls

def TOK(G, newcall, calls):

    # If G is an initial gossip graph, add all edges/arcs from G
    if (G.graph['initial'] or newcall == 0):
        H = copy.deepcopy(G)
        H = nx.DiGraph(H)
        calls = list(H.edges)

    # Say the latest call is (i.j)
    # If G not initial, remove calls (i,k) where i knows k's number
    # and add calls (j, k) such that j knows k's number
    else:
        i, j = newcall[0], newcall[1]

        # For each neighbour k of i, remove call (i,k) if it exists
        for neighbour in G.adj[i]:
            if ((i, neighbour) in calls):
                calls.remove((i, neighbour))

        # For each neighbour k of j, add new calls (j,k)
        for neighbour in G.adj[j]:
            if ((j, neighbour) not in calls):
                calls.append((j, neighbour))

    return calls


# SPI(G, newcall, calls) - returns SPI permitted calls for gossip graph G
# Parameters: G - graph object, generated in "graph_generator.py"
#             newcall - tuple, latest call made
#             calls - list of permitted calls in the previous step
# Returns - calls, updated list of permitted calls

def SPI(G, newcall, calls):

    # If G is an initial gossip graph, add all edges/arcs from G
    if (G.graph['initial'] or newcall == 0):
        H = copy.deepcopy(G)
        H = nx.DiGraph(H)
        calls = list(H.edges)

    # Say the latest call is (i.j)
    # If G not initial, remove calls (j,k) such that j knows k's number
    else:
        j = newcall[1]
        # For each neighbour k of j, remove call (j,k) if it exists
        for neighbour in G.adj[j]:
            if ((j, neighbour) in calls):
                calls.remove((j, neighbour))

    return calls
```

E.5 Experiment Functions

We define two functions; one which tests each protocol on graphs for a fixed value of n; another which utilises the first function across a range of n values.

```
# ================================================================
#  Test protocols function
# ================================================================

# testProtocols(n, top, trials, rounds) - executes the five
#                 protocols on a set topology and number of agents
# Parameters: n - number of agents
#             top - network topology (complete, incomplete, dynamic)
#             trials - int, number of trials
#             rounds - boolean, dictates if calls are made in
#                      rounds (true) or not (false)
# Returns - dictionary, contains all results for set n

def testProtocols(n, top, trials, rounds = False):
    # Case where calls are made sequentially
    if (not rounds):
        # Intialise results dictionary
        results = {'ANY':{'execLengths':[], 'execTimes':[], 'avgTime': 0,
                          'avgLength': 0, 'sNumber': 0, 'fNumber': 0,
                          'tNumber': 0},
                   'CO':{'execLengths':[], 'execTimes':[], 'avgTime': 0,
                         'avgLength': 0, 'sNumber': 0, 'fNumber': 0,
                         'tNumber': 0},
                   'LNS':{'execLengths':[], 'execTimes':[], 'avgTime': 0,
                          'avgLength': 0, 'sNumber': 0, 'fNumber': 0,
                          'tNumber': 0},
                   'TOK':{'execLengths':[], 'execTimes':[], 'avgTime': 0,
                          'avgLength': 0, 'sNumber': 0, 'fNumber': 0,
                          'tNumber': 0},
                   'SPI':{'execLengths':[], 'execTimes':[], 'avgTime': 0,
                          'avgLength': 0, 'sNumber': 0, 'fNumber': 0,
                          'tNumber': 0}}

        # For set number of trials
        for _ in range(trials):
            # Generate graph
            if (top == 'complete'):
                G = completeGraph(n)
            elif (top == 'incomplete'):
```

```
    G = incompleteGraph(n)
elif (top == 'dynamic'):
    G = diGraph(n)

# Create copy of graph
H = copy.deepcopy(G)

# Run each protocol on H and record results
# (number of successes, failures and timeouts)
#ANY
(c, timer, success, failure, timeout) = executeProtocol(H, ANY)
if (success):
    results['ANY']['execLengths'].append(c)
    results['ANY']['execTimes'].append(timer)
    results['ANY']['sNumber'] += 1
elif (failure):
    results['ANY']['fNumber'] += 1
elif (timeout):
    results['ANY']['tNumber'] += 1
# CO
H = copy.deepcopy(G)
(c, timer, success, failure, timeout) = executeProtocol(H, CO)
if (success):
    results['CO']['execLengths'].append(c)
    results['CO']['execTimes'].append(timer)
    results['CO']['sNumber'] += 1
elif (failure):
    results['CO']['fNumber'] += 1
elif (timeout):
    results['CO']['tNumber'] += 1
# LNS
H = copy.deepcopy(G)
(c, timer, success, failure, timeout) = executeProtocol(H, LNS)
if (success):
    results['LNS']['execLengths'].append(c)
    results['LNS']['execTimes'].append(timer)
    results['LNS']['sNumber'] += 1
elif (failure):
    results['LNS']['fNumber'] += 1
elif (timeout):
    results['LNS']['tNumber'] += 1
# TOK
H = copy.deepcopy(G)
(c, timer, success, failure, timeout) = executeProtocol(H, TOK)
if (success):
    results['TOK']['execLengths'].append(c)
    results['TOK']['execTimes'].append(timer)
    results['TOK']['sNumber'] += 1
elif (failure):
    results['TOK']['fNumber'] += 1
elif (timeout):
    results['TOK']['tNumber'] += 1
    # SPI
    H = copy.deepcopy(G)
    (c, timer, success, failure, timeout) = executeProtocol(H, SPI)
    if (success):
        results['SPI']['execLengths'].append(c)
        results['SPI']['execTimes'].append(timer)
        results['SPI']['sNumber'] += 1
    elif (failure):
        results['SPI']['fNumber'] += 1
    elif (timeout):
        results['SPI']['tNumber'] += 1

# Calculate averages and success rates
# ANY
if (len(results['ANY']['execLengths']) > 0):
    results['ANY']['avgLength'] = sum(
        results['ANY']['execLengths'])/results['ANY']['sNumber']
    results['ANY']['avgTime'] = sum(
        results['ANY']['execTimes'])/results['ANY']['sNumber']
else:
    results['ANY']['avgLengths'] = None
    results['ANY']['avgTime'] = None
results['ANY']['sRate'] = results['ANY']['sNumber']/trials
results['ANY']['fRate'] = results['ANY']['fNumber']/trials
results['ANY']['tRate'] = results['ANY']['tNumber']/trials
# CO
if (len(results['CO']['execLengths']) > 0):
    results['CO']['avgLength'] = sum(
        results['CO']['execLengths'])/results['CO']['sNumber']
    results['CO']['avgTime'] = sum(
        results['CO']['execTimes'])/results['CO']['sNumber']
else:
    results['ANY']['avgLengths'] = None
    results['ANY']['avgTime'] = None
results['CO']['sRate'] = results['CO']['sNumber']/trials
results['CO']['fRate'] = results['CO']['fNumber']/trials
results['CO']['tRate'] = results['CO']['tNumber']/trials
# LNS
if (len(results['LNS']['execLengths']) > 0):
    results['LNS']['avgLength'] = sum(
        results['LNS']['execLengths'])/results['LNS']['sNumber']
    results['LNS']['avgTime'] = sum(
        results['LNS']['execTimes'])/results['LNS']['sNumber']
else:
    results['LNS']['avgLengths'] = None
    results['LNS']['avgTime'] = None
results['LNS']['sRate'] = results['LNS']['sNumber']/trials
results['LNS']['fRate'] = results['LNS']['fNumber']/trials
results['LNS']['tRate'] = results['LNS']['tNumber']/trials
# TOK
if (len(results['TOK']['execLengths']) > 0):
    results['TOK']['avgLength'] = sum(
                results['TOK']['execLengths'])/results['TOK']['sNumber']
            results['TOK']['avgTime'] = sum(
                results['TOK']['execTimes'])/results['TOK']['sNumber']
        else:
            results['TOK']['avgLengths'] = None
            results['TOK']['avgTime'] = None
        results['TOK']['sRate'] = results['TOK']['sNumber']/trials
        results['TOK']['fRate'] = results['TOK']['fNumber']/trials
        results['TOK']['tRate'] = results['TOK']['tNumber']/trials
        # SPI
        if (len(results['SPI']['execLengths']) > 0):
            results['SPI']['avgLength'] = sum(
                results['SPI']['execLengths'])/results['SPI']['sNumber']
            results['SPI']['avgTime'] = sum(
                results['SPI']['execTimes'])/results['SPI']['sNumber']
        else:
            results['SPI']['avgLengths'] = None
            results['SPI']['avgTime'] = None
        results['SPI']['sRate'] = results['SPI']['sNumber']/trials
        results['SPI']['fRate'] = results['SPI']['fNumber']/trials
        results['SPI']['tRate'] = results['SPI']['tNumber']/trials

    # Case where calls are made in rounds
    if (rounds):
        # Intialise results dictionary
        results = {'ANY':{'rounds':[], 'avgRounds': 0,
                         'times':[], 'avgTime': 0, 'sNumber': 0,
                         'fNumber': 0, 'tNumber': 0},
                   'CO':{'rounds':[], 'avgRounds': 0,
                        'times':[], 'avgTime': 0, 'sNumber': 0,
                        'fNumber': 0, 'tNumber': 0},
                   'LNS':{'rounds':[], 'avgRounds': 0,
                         'times':[], 'avgTime': 0, 'sNumber': 0,
                         'fNumber': 0, 'tNumber': 0},
                   'TOK':{'rounds':[], 'avgRounds': 0,
                         'times':[], 'avgTime': 0, 'sNumber': 0,
                         'fNumber': 0, 'tNumber': 0},
                   'SPI':{'rounds':[], 'avgRounds': 0,
                         'times':[], 'avgTime': 0, 'sNumber': 0,
                         'fNumber': 0, 'tNumber': 0}}

        # For set number of trials
        for _ in range(trials):
            # Generate graph
            if (top == 'complete'):
                G = completeGraph(n)
            elif (top == 'incomplete'):
                G = incompleteGraph(n)
            elif (top == 'dynamic'):
                G = diGraph(n)

            # Create copy of graph
```

```
H = copy.deepcopy(G)

# Run each protocol on H and record results
# (number of successes, failures and timeouts)
#ANY
(r, timer, success, failure, timeout) = executeRounds(H, ANY)
if (success):
    results['ANY']['rounds'].append(r)
    results['ANY']['times'].append(timer)
    results['ANY']['sNumber'] += 1
elif (failure):
    results['ANY']['fNumber'] += 1
elif (timeout):
    results['ANY']['tNumber'] += 1
# CO
H = copy.deepcopy(G)
(r, timer, success, failure, timeout) = executeRounds(H, CO)
if (success):
    results['CO']['rounds'].append(r)
    results['CO']['times'].append(timer)
    results['CO']['sNumber'] += 1
elif (failure):
    results['CO']['fNumber'] += 1
elif (timeout):
    results['CO']['tNumber'] += 1
# LNS
H = copy.deepcopy(G)
(r, timer, success, failure, timeout) = executeRounds(H, LNS)
if (success):
    results['LNS']['rounds'].append(r)
    results['LNS']['times'].append(timer)
    results['LNS']['sNumber'] += 1
elif (failure):
    results['LNS']['fNumber'] += 1
elif (timeout):
    results['LNS']['tNumber'] += 1
# TOK
H = copy.deepcopy(G)
(r, timer, success, failure, timeout) = executeRounds(H, TOK)
if (success):
    results['TOK']['rounds'].append(r)
    results['TOK']['times'].append(timer)
    results['TOK']['sNumber'] += 1
elif (failure):
    results['TOK']['fNumber'] += 1
elif (timeout):
    results['TOK']['tNumber'] += 1
# SPI
H = copy.deepcopy(G)
(r, timer, success, failure, timeout) = executeRounds(H, SPI)
if (success):
    results['SPI']['rounds'].append(r)
```

```
            results['SPI']['times'].append(timer)
            results['SPI']['sNumber'] += 1
        elif (failure):
            results['SPI']['fNumber'] += 1
        elif (timeout):
            results['SPI']['tNumber'] += 1

# Calculate averages and success rates
# ANY
if (len(results['ANY']['rounds']) > 0):
    results['ANY']['avgRounds'] = sum(
        results['ANY']['rounds'])/results['ANY']['sNumber']
    results['ANY']['avgTime'] = sum(
        results['ANY']['times'])/results['ANY']['sNumber']
else:
    results['ANY']['avgRounds'] = None
    results['ANY']['avgTime'] = None
results['ANY']['sRate'] = results['ANY']['sNumber']/trials
results['ANY']['fRate'] = results['ANY']['fNumber']/trials
results['ANY']['tRate'] = results['ANY']['tNumber']/trials
# CO
if (len(results['CO']['rounds']) > 0):
    results['CO']['avgRounds'] = sum(
        results['CO']['rounds'])/results['CO']['sNumber']
    results['CO']['avgTime'] = sum(
        results['CO']['times'])/results['CO']['sNumber']
else:
    results['CO']['avgRounds'] = None
    results['CO']['avgTime'] = None
results['CO']['sRate'] = results['CO']['sNumber']/trials
results['CO']['fRate'] = results['CO']['fNumber']/trials
results['CO']['tRate'] = results['CO']['tNumber']/trials
# LNS
if (len(results['LNS']['rounds']) > 0):
    results['LNS']['avgRounds'] = sum(
        results['LNS']['rounds'])/results['LNS']['sNumber']
    results['LNS']['avgTime'] = sum(
        results['LNS']['times'])/results['LNS']['sNumber']
else:
    results['LNS']['avgRounds'] = None
    results['LNS']['avgTime'] = None
results['LNS']['sRate'] = results['LNS']['sNumber']/trials
results['LNS']['fRate'] = results['LNS']['fNumber']/trials
results['LNS']['tRate'] = results['LNS']['tNumber']/trials
# TOK
if (len(results['TOK']['rounds']) > 0):
    results['TOK']['avgRounds'] = sum(
        results['TOK']['rounds'])/results['TOK']['sNumber']
    results['TOK']['avgTime'] = sum(
        results['TOK']['times'])/results['TOK']['sNumber']
else:
    results['TOK']['avgRounds'] = None
```

```
                results['TOK']['avgTime'] = None
            results['TOK']['sRate'] = results['TOK']['sNumber']/trials
            results['TOK']['fRate'] = results['TOK']['fNumber']/trials
            results['TOK']['tRate'] = results['TOK']['tNumber']/trials
            # SPI
            if (len(results['SPI']['rounds']) > 0):
                results['SPI']['avgRounds'] = sum(
                    results['SPI']['rounds'])/results['SPI']['sNumber']
                results['SPI']['avgTime'] = sum(
                    results['SPI']['times'])/results['SPI']['sNumber']
            else:
                results['SPI']['avgRounds'] = None
                results['SPI']['avgTime'] = None
            results['SPI']['sRate'] = results['SPI']['sNumber']/trials
            results['SPI']['fRate'] = results['SPI']['fNumber']/trials
            results['SPI']['tRate'] = results['SPI']['tNumber']/trials

    return results

# ================================================================
#  Experiment function
# ================================================================

# experiment(maxN, top, trials) - produces results up to maxN agents
# Parameters: maxN - maximum number of agents to test for
#                    (agents given in multiples of 5)
#             top - network topology (complete, incomplete, dynamic)
#             trials - int, number of trials for each value of n
#             rounds - boolean, dictates if calls are made in rounds
#                      (true) or not (false)
#             minN - minimum number of agents to test for
#                   (must be multiple of 5)
# Returns - dictionary - contains all results for all n

def experiment(maxN, top, trials, rounds = False, minN = 5):
    # Case where calls are made sequentially
    if (not rounds):
        # Execution lengths
        elANY, elCO, elLNS, elTOK, elSPI = [],[],[],[],[]
        # Execution times
        etANY, etCO, etLNS, etTOK, etSPI = [],[],[],[],[]
        # Success rates
        srANY, srCO, srLNS, srTOK, srSPI = [],[],[],[],[]
        # Failure rates
        frANY, frCO, frLNS, frTOK, frSPI = [],[],[],[],[]
        # Timeout rates
        trANY, trCO, trLNS, trTOK, trSPI = [],[],[],[],[]

        # Produce results for all values of 5n up to maxN
        for n in range(minN, maxN+1, 5):
            # Calculate execution length, time e.t.c.
            results = testProtocols(n, top, trials, rounds)
```

```python
print("Progress: "+str(n)+" agents complete.")

# Record ANY results
if (len(results['ANY']['execLengths']) > 0):
    elANY.append(results['ANY']['avgLength'])
    etANY.append(results['ANY']['avgTime'])
    srANY.append(results['ANY']['sRate'])
    frANY.append(results['ANY']['fRate'])
    trANY.append(results['ANY']['tRate'])
else:
    elANY.append(None)
    etANY.append(None)
    srANY.append(results['ANY']['sRate'])
    frANY.append(results['ANY']['fRate'])
    trANY.append(results['ANY']['tRate'])

# Record CO results
if (len(results['CO']['execLengths']) > 0):
    elCO.append(results['CO']['avgLength'])
    etCO.append(results['CO']['avgTime'])
    srCO.append(results['CO']['sRate'])
    frCO.append(results['CO']['fRate'])
    trCO.append(results['CO']['tRate'])
else:
    elCO.append(None)
    etCO.append(None)
    srCO.append(results['CO']['sRate'])
    frCO.append(results['CO']['fRate'])
    trCO.append(results['CO']['tRate'])

# Record LNS results
if (len(results['LNS']['execLengths']) > 0):
    elLNS.append(results['LNS']['avgLength'])
    etLNS.append(results['LNS']['avgTime'])
    srLNS.append(results['LNS']['sRate'])
    frLNS.append(results['LNS']['fRate'])
    trLNS.append(results['LNS']['tRate'])
else:
    elLNS.append(None)
    etLNS.append(None)
    srLNS.append(results['LNS']['sRate'])
    frLNS.append(results['LNS']['fRate'])
    trLNS.append(results['LNS']['tRate'])

# Record TOK results
if (len(results['TOK']['execLengths']) > 0):
    elTOK.append(results['TOK']['avgLength'])
    etTOK.append(results['TOK']['avgTime'])
    srTOK.append(results['TOK']['sRate'])
    frTOK.append(results['TOK']['fRate'])
    trTOK.append(results['TOK']['tRate'])
else:
```

```
            elTOK.append(None)
            etTOK.append(None)
            srTOK.append(results['TOK']['sRate'])
            frTOK.append(results['TOK']['fRate'])
            trTOK.append(results['TOK']['tRate'])

        # Record SPI results
        if (len(results['SPI']['execLengths']) > 0):
            elSPI.append(results['SPI']['avgLength'])
            etSPI.append(results['SPI']['avgTime'])
            srSPI.append(results['SPI']['sRate'])
            frSPI.append(results['SPI']['fRate'])
            trSPI.append(results['SPI']['tRate'])
        else:
            elSPI.append(None)
            etSPI.append(None)
            srSPI.append(results['SPI']['sRate'])
            frSPI.append(results['SPI']['fRate'])
            trSPI.append(results['SPI']['tRate'])

    # Compile results into a dictionaries
    executionLengths = {'ANY': elANY, 'CO': elCO, 'LNS': elLNS,
                        'TOK': elTOK, 'SPI': elSPI}
    executionTimes = {'ANY': etANY, 'CO': etCO, 'LNS': etLNS,
                      'TOK': etTOK, 'SPI': etSPI}
    successRates = {'ANY': srANY, 'CO': srCO, 'LNS': srLNS,
                    'TOK': srTOK, 'SPI': srSPI}
    failureRates = {'ANY': frANY, 'CO': frCO, 'LNS': frLNS,
                    'TOK': frTOK, 'SPI': frSPI}
    timeoutRates = {'ANY': trANY, 'CO': trCO, 'LNS': trLNS,
                    'TOK': trTOK, 'SPI': trSPI}

    # Final results
    finalResults = (executionLengths, executionTimes, successRates,
                    failureRates, timeoutRates)

# Case where calls are made in rounds
if (rounds):
    # Round lengths
    rlANY, rlCO, rlLNS, rlTOK, rlSPI = [],[],[],[],[]
    # Times
    tANY, tCO, tLNS, tTOK, tSPI = [],[],[],[],[]
    # Success rates
    srANY, srCO, srLNS, srTOK, srSPI = [],[],[],[],[]
    # Failure rates
    frANY, frCO, frLNS, frTOK, frSPI = [],[],[],[],[]
    # Timeout rates
    trANY, trCO, trLNS, trTOK, trSPI = [],[],[],[],[]

    # Produce results for all values of 5n up to maxN
    for n in range(minN, maxN+1, 5):
        # Calculate execution length, time e.t.c.
```

```
results = testProtocols(n, top, trials, rounds)
print("Progress: "+str(n)+" agents complete.")

# Record ANY results
if (len(results['ANY']['rounds']) > 0):
    rlANY.append(results['ANY']['avgRounds'])
    tANY.append(results['ANY']['avgTime'])
    srANY.append(results['ANY']['sRate'])
    frANY.append(results['ANY']['fRate'])
    trANY.append(results['ANY']['tRate'])
else:
    rlANY.append(None)
    tANY.append(None)
    srANY.append(results['ANY']['sRate'])
    frANY.append(results['ANY']['fRate'])
    trANY.append(results['ANY']['tRate'])

# Record CO results
if (len(results['CO']['rounds']) > 0):
    rlCO.append(results['CO']['avgRounds'])
    tCO.append(results['CO']['avgTime'])
    srCO.append(results['CO']['sRate'])
    frCO.append(results['CO']['fRate'])
    trCO.append(results['CO']['tRate'])
else:
    rlCO.append(None)
    tCO.append(None)
    srCO.append(results['CO']['sRate'])
    frCO.append(results['CO']['fRate'])
    trCO.append(results['CO']['tRate'])

# Record LNS results
if (len(results['LNS']['rounds']) > 0):
    rlLNS.append(results['LNS']['avgRounds'])
    tLNS.append(results['LNS']['avgTime'])
    srLNS.append(results['LNS']['sRate'])
    frLNS.append(results['LNS']['fRate'])
    trLNS.append(results['LNS']['tRate'])
else:
    rlLNS.append(None)
    tLNS.append(None)
    srLNS.append(results['LNS']['sRate'])
    frLNS.append(results['LNS']['fRate'])
    trLNS.append(results['LNS']['tRate'])

# Record TOK results
if (len(results['TOK']['rounds']) > 0):
    rlTOK.append(results['TOK']['avgRounds'])
    tTOK.append(results['TOK']['avgTime'])
    srTOK.append(results['TOK']['sRate'])
    frTOK.append(results['TOK']['fRate'])
    trTOK.append(results['TOK']['tRate'])
```

```
            else:
                rlTOK.append(None)
                tTOK.append(None)
                srTOK.append(results['TOK']['sRate'])
                frTOK.append(results['TOK']['fRate'])
                trTOK.append(results['TOK']['tRate'])

            # Record SPI results
            if (len(results['SPI']['rounds']) > 0):
                rlSPI.append(results['SPI']['avgRounds'])
                tSPI.append(results['SPI']['avgTime'])
                srSPI.append(results['SPI']['sRate'])
                frSPI.append(results['SPI']['fRate'])
                trSPI.append(results['SPI']['tRate'])
            else:
                rlSPI.append(None)
                tSPI.append(None)
                srSPI.append(results['SPI']['sRate'])
                frSPI.append(results['SPI']['fRate'])
                trSPI.append(results['SPI']['tRate'])

        # Compile results into a dictionaries
        roundLengths = {'ANY': rlANY, 'CO': rlCO, 'LNS': rlLNS,
                        'TOK': rlTOK, 'SPI': rlSPI}
        times = {'ANY': tANY, 'CO': tCO, 'LNS': tLNS,
                 'TOK': tTOK, 'SPI': tSPI}
        successRates = {'ANY': srANY, 'CO': srCO, 'LNS': srLNS,
                        'TOK': srTOK, 'SPI': srSPI}
        failureRates = {'ANY': frANY, 'CO': frCO, 'LNS': frLNS,
                        'TOK': frTOK, 'SPI': frSPI}
        timeoutRates = {'ANY': trANY, 'CO': trCO, 'LNS': trLNS,
                        'TOK': trTOK, 'SPI': trSPI}

        # Final results
        finalResults = (roundLengths, times, successRates,
                           failureRates, timeoutRates)

    return finalResults
```