COMP390

2022/23

A Simulator for Protocols for Stable Construction of Networks

Student Name:	Xinbo Luo
Student ID:	201600700
Supervisor:	Prof Paul Spirakis
Second Marker:	Dr Navjot Kukreja

DEPARTMENT OF COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Dedicated to Shuhui Wang

Acknowledgements

I’d like to thank my parents, Shuhui and Xiangrong, for their support throughout my studies.

COMP390

2022/23

A Simulator for Protocols for Stable Construction of Networks

DEPARTMENT OF
COMPUTER
SCIENCE

University of
Liverpool Liverpool
L69 3BX

Contents

Abstract

This dissertation presents the design, implementation, testing, and evaluation of a protocol simulator software project. The project was motivated by the increasing complexity of modern networked systems and the challenges of analysing and optimizing their performance. It aims to model and simulate protocols for stable network construction, offering a more precise and scalable approach to understanding and predicting network performance dynamics.

The main body of this dissertation involves simulation of the protocols along with evaluation of their correctness. A comprehensive analysis of protocols' correctness with different sizes of the networks is included. Evaluation of protocols was performed using network structure and running time comparison. Future work should consider the implementation of universal algorithms to simulate any protocols and adding single step simulation functionality. Overall, this dissertation demonstrates the project is a valuable tool for network researchers and engineers, providing them with an advanced platform for understanding and improving network performance.

1 Introduction

1.1 Background

Distributed algorithms, also known as protocols, are sets of rules or procedures that guide the behaviour of a group of interconnected distributed systems. Unlike centralised algorithms, which assume there is a single agent that controls the entire system, distributed algorithms operate networks in a way that agents communicate and coordinate with each other to cooperate and perform tasks. Distributed systems must follow these protocols in order to transfer data or establish network communications [1]. Consequently, protocols decide how data can be transferred between distributed systems.

With protocols, distributed networks can be constructed. These distributed network models can be used in a variety of contexts, including peer-to-peer file sharing and distributed databases. The system's reliability, scalability, and fault tolerance can also be enhanced by implementing distributed networks. Additionally, they can reveal relationships between different objects in some fields, such as medical, neural, and social. Some real world problems can be solved by computing and predicting with distributed networks [2]. For example, timing a chemical reaction and capturing the stable structure of a chemical reaction network [3], [4], [5]. Various types of protocols are used to build communication between distributed systems, which allows them to cooperate and interact to help the system achieve overall larger capacity, and better performance to complete specialized tasks [3], [6].

1.2 Motivation

If the protocol forms a network that has a different structure than expected, issues related to communication and compatibility may arise. Furthermore, unverified correctness of protocols can lead to performance issues such as slow speed of data transfer, network congestion, or system crash. Hence, the utilization of a simulator is imperative to test and validation of protocols before being deployed in a practical environment. This can reduce the risk of errors and downtime. Another potential utilization of a protocol simulator lies in the research field. By simulating protocols, potential issues may be identified which helps with further optimisation of the performance [7].

1.3 Population Protocols

Population protocols are classes of distributed algorithms designed to use in networks with large scale but limited communication and computational power. A population of agents, sometimes referred to as nodes or entities, are involved in these algorithms. These agents collaborate through interactions to solve problems. Each agent or node possesses limited computational abilities, which means they cannot complete complex tasks or execute sophisticated algorithms independently. Instead, they use simple rules to interact locally with other agents to obtain the desired output. A population protocol operates in rounds, also known as time steps. Each round consists of a sequence of interactions between randomly nodes. In this distributed model, pairwise meeting mechanism is implemented, where two nodes are selected randomly and interact in every round (shown in Figure 1). Agents are supposed to move in an uncontrolled way in a common area and meet each other from time to time. This can refer to molecular can meet each other in a well-mixed solution in chemical reactions. During each interaction, the nodes exchange information about their states and update their own states based on a fixed set of local rules. This model refers to the occasions in which agents have limited mobility and control over which other agents they are interacting with. The input and output of a population protocol depend on the specific problem being solved [8].

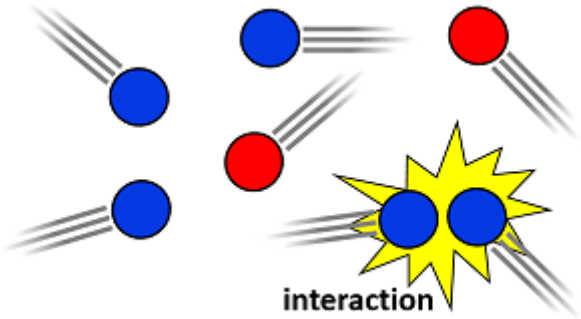


Figure 1. Pairwise meeting

The input of a population protocol can be a set of agents with initial states and the topology of the interaction network. The initial states determine the start of computation process. The topology of interaction network determines how nodes can interact with each other. The input can also include parameters, such as the number of nodes, the maximum degree of the graph, or the range of possible values for the node states. By interacting according to network topologies, agents can adapt to the network dynamically to achieve expected outputs.

The output of population protocols can be a global function of states of agents in the system or a value. The function can be a decision, a computation, or a transformation of the input which is reflected by interactions between agents in the system will. For example, if a population protocol is designed for leader election, the output may be a graph of nodes that indicates which node is the leader. An example is shown in Figure 2 [9].

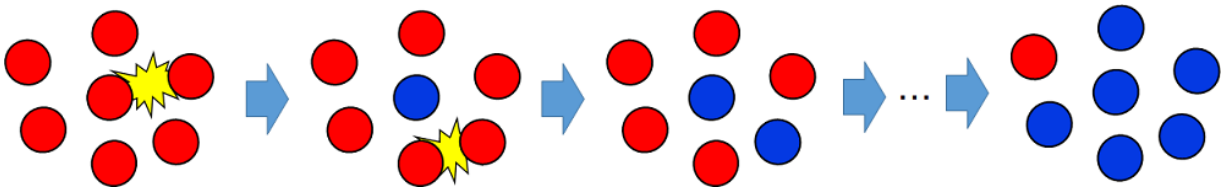


Figure 2. Example of leader election with population protocols

Population protocols have been employed to address a wide range of distributed computing problems, including consensus, majority determination, and sorting. These protocols are particularly useful in environments where nodes have limited resources or communication ability, such as wireless

sensor networks, ad-hoc networks, and mobile networks. For example, protocols can assist devices that are in nanoscale and have limited mobility cooperate when injected in veins to decide which direction to move to and perform tasks for medical purposes [3], [10].

Another important model that inspires the protocols in this project is mediated population protocols. As an extension to population protocols, mediated population protocols can maintain the uniformity and anonymity properties of population protocols, which respectively, allow them to be independent of the population size and not necessarily require unique identifiers for the agents [2]. They introduce a mediator to coordinate the interactions between agents. Additionally, edges in mediated population protocols have different states, while they do not have states in population protocols.

1.4 Project Description

The protocols implemented in this project are highly inspired by population protocols and mediated population protocols. Based on these two types of protocols, the protocols in this project are developed. They preserve the main properties of population protocols and mediated population protocols and are therefore similar to them. However, it is noteworthy that there are two significant differences between the protocols in this project and population protocols. The protocols in this project mainly focus on network construction, whereas the aim of population protocols is function computation. The other difference is that edges have only 2 states in this project, but they have multiple states in mediated population protocols. The protocols in this project also overcome the limitations of assuming the communication topology is already known (specifically the structure of the network), storing unique identities of nodes, unlimited capacities, and insufficient control on which nodes will have interactions [3], [11], [12]. Without the above assumptions, protocols are not assumed to be able to converge in this project. Hence, this project will have to detect the stable structures of networks and determine when to stop the simulation.

Three protocols from [3] are implemented in this project, which are Global-Star, Cycle-Cover, Simple-Global-Line (more analysis in **2 Design and Implementation** and **5 Evaluation**). Global-Star will generate a spanning star, Cycle-Cover generates one or more cycles of nodes, and Simple-Global-Line constructs networks with a spanning line. Another custom protocol is also implemented which constructs a clique (to construct a group of nodes where each node is directly connected to every other node) [13], [14].

In such a population of entities, an entity (nodes, or agents) can have several states and edges (connection between nodes) can be either active or inactive. A pairwise mechanism is implemented, under which two nodes are selected randomly in each simulation step. Under this mechanism, each node is assumed to be able to meet and interact with one other node at a time. There will also be edges being formed or removed during the simulation. The states of nodes and edges also change with the process of simulation according to the rules. These rules are in the format of, i.e., $(s_1, s_2, s) \rightarrow (s_1', s_2', s')$, where s_1 and s_2 are states of two nodes which are selected to interact in a round and s is the state of edge between them. If the states of two nodes and edge joining them meet one of the rules (on the left-hand side), the states will be updated to s_1', s_2', and s' according to the rule and therefore changes the network structure. Otherwise, the meeting of nodes does not result in changes of the structure of the network. When the program detects the network structure is stable and does not change with time, the simulation will terminate. The main aim of this project is to develop and create a simulator software which is able to execute protocols for stable network construction. The project will demonstrate the process of simulation via a graph of nodes and edges along with their states to show the structure of the network. The graph can be updated if necessary (before the network is stably constructed). The running time and stability are also displayed and updated in real-time.

2 Design and Implementation

2.1 Class Relationships

To assist in better understanding of the class relationships of this software, a UML diagram is provided below (shown in Figure 3).

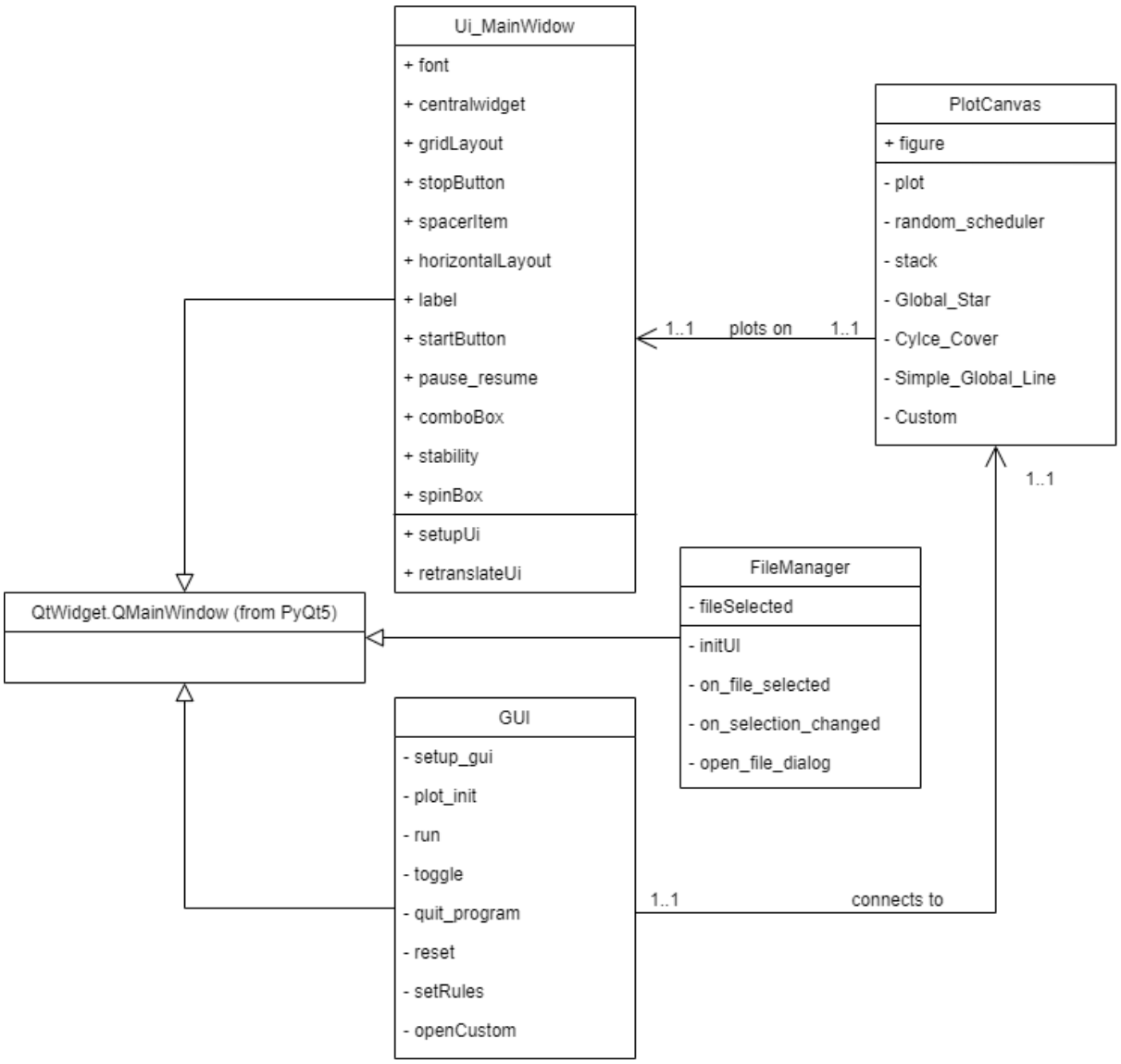


Figure 3. Class diagram

The class “Ui_MainWindow” creates the user interface. This part of the code is generated by Qt Designer. Another class called “PlotCanvas” is responsible for network construction and data visualisation. This class is connected to the graphics view component on the main window such that it can plot the structure of the network on the user interface using the function “plot”. Functions including “Global-Star”, “Cycle-Cover”, “Simple-Global-Line”, and “Custom” contained in this class are responsible for execution of protocols. This class also determines if the network is stable and when to terminate the simulation.

The class that controls both algorithms for protocol implementation and the user interface is “GUI” which is stored in the main file. This class connects functions to components on the user interface. It

also set constraints to some variables, for example, limitation of number of nodes that prevents the user from entering too many nodes. Functionalities such as starting or pausing simulation, choosing protocol to simulate, and speed adjustment are also achieved by this class. Class “FileManager” is to open a sub window that works as a file explorer to open the file in which the custom protocol is stored.

2.2 Functionalities

The software is designed to simulate protocols for stable network construction. Nodes and edges will be displayed in a graph with their states. The graph can keep updating before the network is stable. The user can pause the simulation at any time during the simulation in order to inspect the structure of the network and the states of nodes and edges. The running time and stability are also paused to update.

Ten different simulation speeds are provided which vary from 1 second per time step to 0.1 second per time step. This allows the user to speed up the simulation if desired and reduces the time to wait for the result.

Users are allowed to simulate the 3 protocols in [3], which are Global-Star, Cycle-Cover, Simple-Global-Line (more details to be covered in **5 Evaluation**). Another protocol for clique construction is also implemented [13]. However, the customised protocol is mainly designed for demonstration purpose since this project also aims to simulate any protocols for stable network construction. The software will be able to convert the rules in a CSV file into strings to displayed on the main window. A sub window can pop up as a file explorer which can access any space in the computer to open the CSV file.

A flowchart is shown in Figure 4 illustrates the procedural steps involved in the software's workflow.

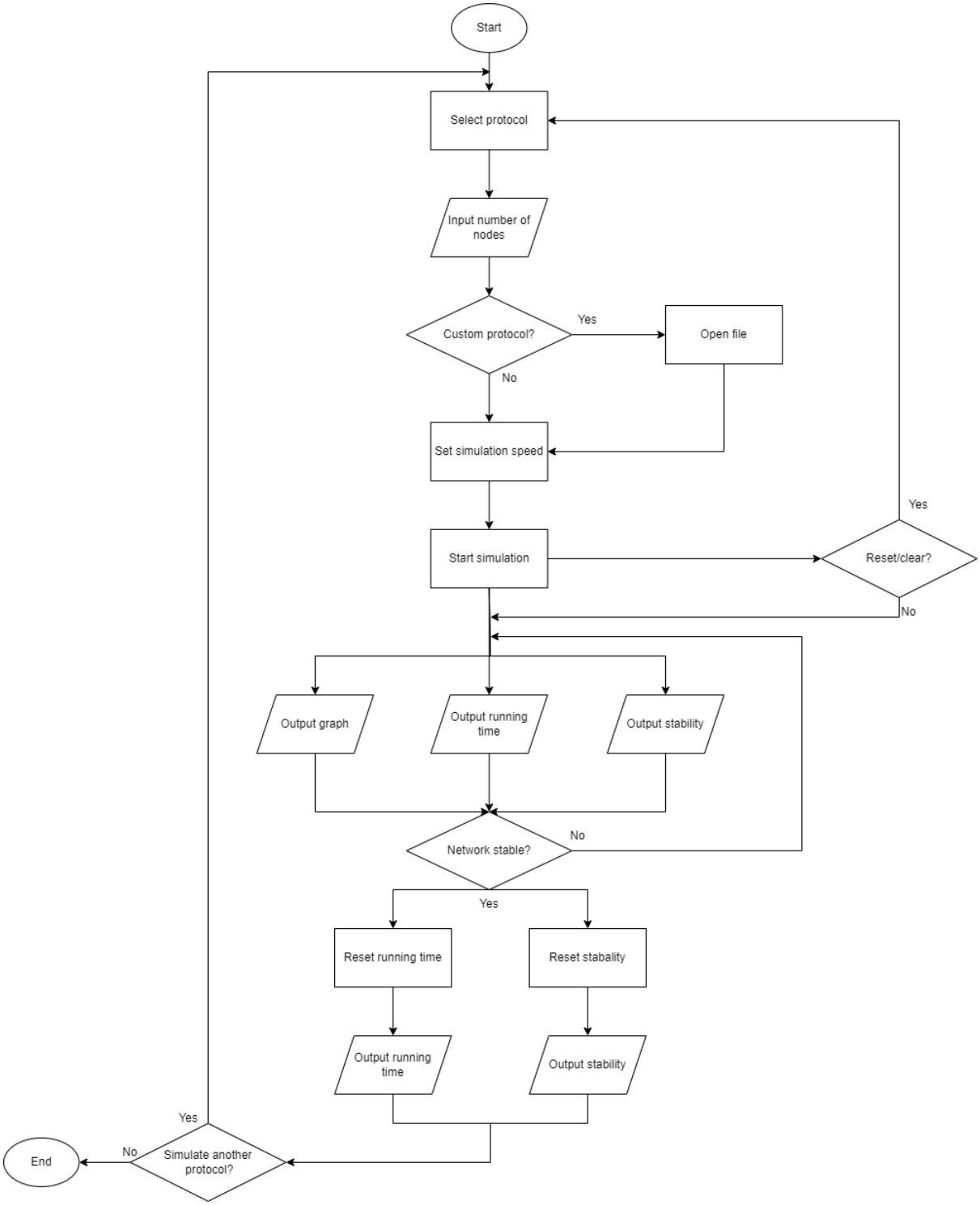


Figure 4. Flowchart of the software

The user can choose the protocol and decide the size of the network (number of nodes). The number of nodes is limited between 2 to 10, as 2 is the minimum number to achieve pairwise meeting and the network cannot be displayed clearly in the graph if too many nodes are added. Another reason for limited number of nodes is that a large number of nodes can lead to exponentially increased computational power (more details covered in **2.7 Data Structures**). The network structure can then be displayed on the main window. Figure 5 illustrates the main window of the software, featuring appropriate annotations to aid in its comprehension.

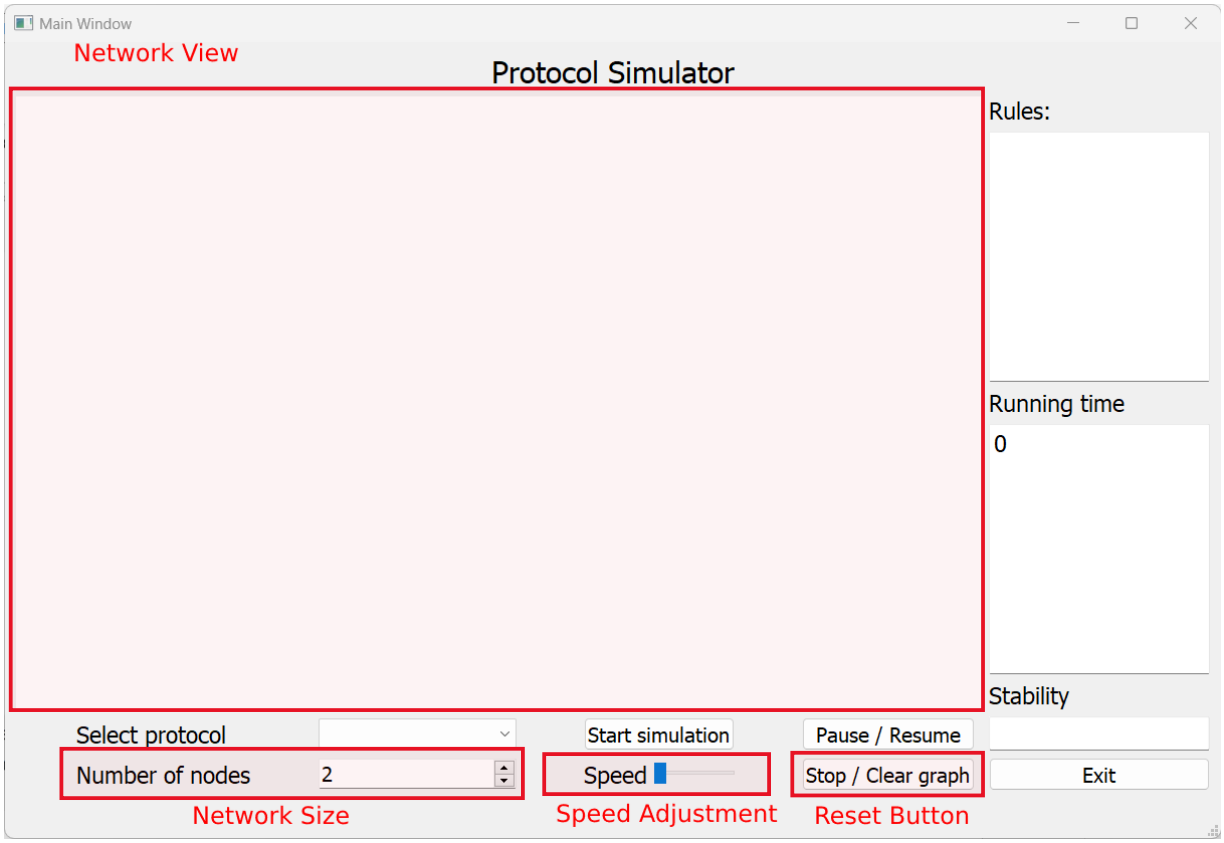


Figure 5. Main window of the software

2.3 Environment

The project is developed with Python 3.11. Required packages can be found in Figure 6.

```
Package         Version
--------------- -------
contourpy       1.0.6
cycler          0.11.0
fonttools       4.38.0
Jinja2          3.1.2
kiwisolver      1.4.4
MarkupSafe      2.1.2
matplotlib      3.6.2
networkx        2.8.8
numpy           1.24.0
packaging       22.0
Pillow          9.3.0
pip             23.0
pyparsing       3.0.9
PyQt5           5.15.7
PyQt5-Qt5       5.15.2
PyQt5-sip       12.11.0
python-dateutil 2.8.2
setuptools      65.5.0
six             1.16.0
```

Figure 6. Required packages

“matplotlib” and “network” are used for data visualisation. Packages such as “PyQt5”, “PyQt5-Qt5” are used for user interface.

2.4 Pairwise Meeting

Since the selection method of the protocols in this project is pairwise (as explained in **1.3 Project Description**), a pair of random nodes needs to be selected in every time step referring to the environment in which the agents have limited mobility and control on the other agent to meet. After the user chooses the protocol to simulate and decides the number of nodes, these nodes will be added to an array with their initial states. As the code shown in Figure 7, two random numbers between 0 and *n* (number of nodes) will be generated in every time step.

```
def random_scheduler(self):
    meeting = random.sample(range(0, globals.num), 2)
    meeting.sort()
    print(meeting[0] + 1, meeting[1] + 1)
    return meeting[0], meeting[1]
```

Figure 7. Pairwise meeting

The numbers are corresponding to the indexes of the nodes in the array. The output of this function is a two-element array whose elements will be returned by the function. The program can then use the generated numbers to locate and fetch the nodes from the arrays and enable them to interact according to protocols (details covered in **2.5 Protocol Implementation**). This design has achieved the randomness to implement pairwise meeting mechanism.

2.5 Protocol Implementation

Since the nodes meet and the check states in every time step, the procedures can be achieved by a group of *if* statements. After two random nodes are drawn, the program will analyse the states of these two nodes. If the states of two nodes and the edges between them meet the condition, the program will update the states and add or remove edges according to the rules. Specifically, if the states of two nodes engaged in a meeting and the edge between satisfy the conditions outlined in the left-hand side of a given rule, the states of these nodes will be updated in accordance with the rule's right-hand side. An example of the rule $(q_0, q_0, 0) \rightarrow (q_1, q_1, 1)$ of Cycle-Cover can be found below (shown in Figure 8). In the code below, the function for random scheduler is called first to determine which nodes will meet in a time step. The program then modifies the content of the arrays for storage of nodes and edges (details covered in **2.7 Data Structures**).

```
def Cycle_Cover(self, frame):
    self.figure.clf()
    num1, num2 = self.random_scheduler()
    if globals.labels[num1] == 'q0' and globals.labels[num2] == 'q0' and globals.G.has_edge(num1, num2) == False:
        globals.G.add_edge(num1, num2)
        globals.edges[num1][num2] = 1
        globals.edges[num2][num1] = 1
        globals.labels[num1] = 'q1'
        globals.labels[num2] = 'q1'
        globals.states[num1] = 'q1'
        globals.states[num1] = 'q2'
```

Figure 8. Example of protocol implementation

These three protocols' rules are listed in the table below (see Table 1):

Global-Star	Cycle-Cover	Simple-Global-Line
$(c,c,0) \to (c,p,1)$ $(p,p,1) \to (p,p,0)$ $(c,p,0) \to (c,p,1)$	$(q_0,q_0,0) \to (q_1,q_1,1)$ $(q_1,q_0,0) \to (q_2,q_1,1)$ $(q_1,q_1,0) \to (q_2,q_2,1)$	$(q_0,q_0,0) \to (q_1,l,1)$ $(l,q_0,0) \to (q_2,l,1)$ $(l,l,0) \to (q_2,w,1)$ $(w,q_2,1) \to (q_2,w,1)$ $(w,q_1,1) \to (q_2,l,1)$

Table 1. Protocol rules

After checking conditions and update states using *if* statements, the "stack" function is called to store the network structure in every round to compare the network structures (more details covered in **2.6 Termination of Simulation** and **2.7 Data Structures**). The program will then update the texts for both running time and stability. After the changes to the network and main window are made, a new graph of the network will be plotted.

The above procedure can only implement the protocol's rules for one round. To enable the simulation to run continuously, the procedure is needed to be done in every time step. This can be achieved by calling the function of a certain protocol from time to time after the program reads the content from the combo box to detect which protocol is to be simulated (details covered in **2.10 Data Visualisation**).

Another functionality that can pause and continue the simulation is implemented after the protocols are simulated continuously. It checks stability based on the text on main window. If the network structure is not stable, the simulation can be paused if desired. If the network structure is stable, the animation status is read as paused and the toggle function cannot be called. Figure 9 shows the details of the code.

```
def toggle(self):
    if (globals.form.stability.text() == "Unstable"):
        if globals.pause == False:
            globals.ani.pause()
            globals.pause = True
        else:
            globals.ani.resume()
            globals.pause = False
```

Figure 9. Pause/continue function

A variable "globals.pause" is a Boolean value which records the status of the network, in particular, if the network is stable or not. It is set to "True" if the network is stable, and vice versa. This variable prevents the simulation from continuing and affecting the final running time if the pause button is pressed and the network is stable.

2.6 Termination of Simulation

With the implementation in **2.5 Protocol Implementation**, the protocols can be simulated continuously but the simulation cannot terminate. Therefore, it is necessary to distinguish when the network is stably constructed to terminate the simulation. A solution to this problem is comparing a number of the latest network structures. If all the networks are the same within these time steps, it can indicate the network structure is stable as it does not change with time.

However, there is no method to guarantee the simulation can always terminate at the right time. Since the two nodes are randomly selected in every time step, there is still a probability that some nodes are never selected before the simulation is terminated by the software. For example, in a network presented in Figure 10 (nodes are denoted with numbers to be easier distinguished from each other in this example, but there are no unique identifiers for nodes in these protocols). The only step needed to construct a stable network structure is that nodes 3 and 4 are selected and interact, in particular,

to change the state of the edge between node 3 and node 4 from active to inactive. Nevertheless, they may not be selected within additional time steps. After these additional time steps, the simulation will stop since the network structure does not change. This will lead to incorrect simulation results that the termination is before the network structure is stable. Hence, the number of additional time steps is critical to lower the risk of unexpected termination before the stable network structure is constructed.

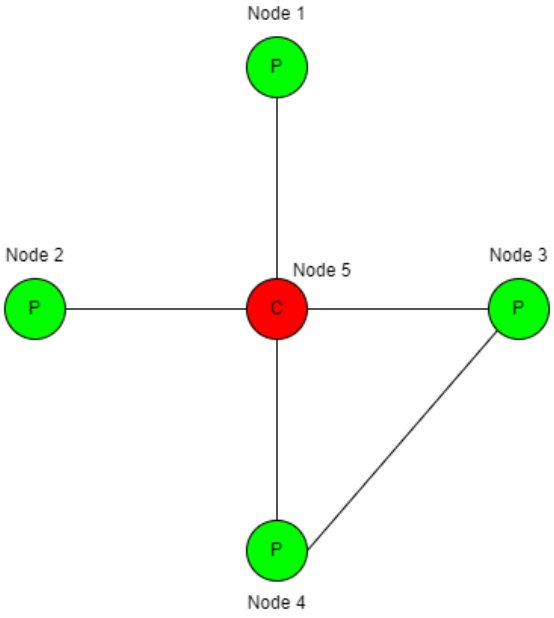


Figure 10. Example of unexpected termination

The number of networks to be stored and compared with is determined by various factors such as complexity of the protocol, the size of the network being constructed, and the desired level of accuracy of simulation are taken into consideration. The expected time and lower bound of 3 protocols in this project are listed as follow (shown in table 2):

Protocol	Expected Time	Lower Bound
Global-Star	$\Theta(n^2 log_2 n)$	$\Omega(n^2 log_2 n)$
Cycle-Cover	$\Theta(n^2)$	$\Omega(n^2)$
Simple-Global-Line	$\Omega(n^4)$ and $O(n^5)$	$\Omega(n^2)$

Table 2. Expected time and lower bound of protocols

A safe number of additional time steps after stabilisation is:

$$expected\ time \times log_2(expected\ time)$$

When this calculation method is applied to Simple-Global-Line protocol, the number of additional time steps can be large, i.e., the number for a network with 10 nodes is at least 10^4. This will keep the user waiting for a long time until the simulation stops. It also significantly increases the data of networks stored in memory for comparison purpose (further explanation in **2.7 Data Structures**). However, the number of additional time steps can be large in some cases. For example, additional time steps will be $10^5 \times log_2(10^5) \sim 1.66 \times 10^6$ for a network constructed by Simple-Global-Line protocol with 10 nodes. This will cause the user to wait for a long time. To solve this problem, the number of additional time steps limited to 1000 if it exceeds 1000 when being calculated with the formula above. Since the maximum number of nodes is limited to 10 and the probability for any two nodes to meet is $\frac{1}{n \times (n-1)}$, 1000 additional time steps is safe for the simulation to terminate with correct network structure.

Therefore, the additional running time for these protocols can be calculated as follow (shown in table 3):

Protocol	Additional Time Steps
Global-Star	$n^2 log_2 n \times log_2 (n^2 log_2 n)$ or 1000
Cycle-Cover	$n^2 \times log_2(n^2)$
Simple-Global-Line	$n^4 \times log_2(n^4)$ or 1000

Table 3. Additional time steps of protocols

The following code in Figure 11 shows the details of the stack.

```
def stack(self, name):
    if (name == "Global-Star"):
        if math.ceil(globals.num * math.log2(globals.num) * math.log2(globals.num * math.log2(globals.num))) < 1000:
            globals.additional = math.ceil(globals.num * math.log2(globals.num) * math.log2(globals.num * math.log2(globals.num)))
        else:
            globals.additional = 1000
        print(globals.additional)
    if (name == "Cycle-Cover"):
        if math.ceil(globals.num ** 2 * math.log2(globals.num ** 2)) < 1000:
            globals.additional = math.ceil(globals.num ** 2 * math.log2(globals.num ** 2))
        else:
            globals.additional = 1000
        print(globals.additional)
    if (name == "Simple-Global-Line"):
        if math.ceil(globals.num ** 4 * math.log2(globals.num ** 4)) < 1000:
            globals.additional = math.ceil(globals.num ** 4 * math.log2(globals.num ** 4))
        else:
            globals.additional = 1000
        print(globals.additional)
```

Figure 11. Stack function

A variable “global.flag” acts as a signal emitter in which is set to “True” if the network is stable and “False” when the network is not stable. This variable must be unidentical from the one for pause and continue functionality since they perform different tasks. The flag variable is then checked by protocols functions to determine the stability of the network. If the network is stable, the function will set new texts of stability and running time on new window.

```
if ([globals.stack[0]] * len(globals.stack) == globals.stack) and \
        ([globals.state_stack[0]] * len(globals.state_stack) == globals.state_stack) and (len(globals.stack) == globals.additional):
    print("Stable")
    globals.ani.pause()
    print(globals.runningTime-globals.additional+1)
    globals.flag = True
    globals.running = False
else:
    print("Unstable")
    globals.flag = False
    globals.running = True
```

Figure 12. Size constraint of stacks

A constraint is set to the size of the stack. If and only if the stack size equals to the number of additional time steps, the stack will compare its elements. This is due to the simulation may stop in the first few rounds as the same pair of nodes keeps being chosen, which cannot lead to network structure changes. The constraint is highlighted in Figure 12.

```
globals.runningTime = globals.runningTime + 1
self.stack("Cycle-Cover")
globals.form.runningTime.setText(str(globals.runningTime))
if globals.flag == True:
    globals.form.stability.setText("Stable")
    globals.form.runningTime.setText(str(globals.runningTime - globals.additional + 1))
if globals.flag == False:
    globals.form.stability.setText("Unstable")
self.plot()
```

Figure 13. Running time count back

With the above method to determine if the network structure is stable, the simulation runs for additional time steps. As shown in Figure 13, the highlighted code demonstrates how the software count back to the time steps in which the network is stable for the first time.

2.7 Data Structures

To solve the problem of termination of the simulation in **2.6 Termination of Simulation**, some data structures are implemented to store information about network structures. States of nodes are stored in arrays where elements are strings that represent the states of nodes. States of the edges between nodes are stored in 2-dimensional arrays which are also called matrices. The size of the matrices is $n \times n$ (n is the number of nodes). The elements are all initially 0 as the edges are all inactive at the beginning. If an edge changes its states from inactive to active, the elements at corresponding positions will be changed from 0 to 1.

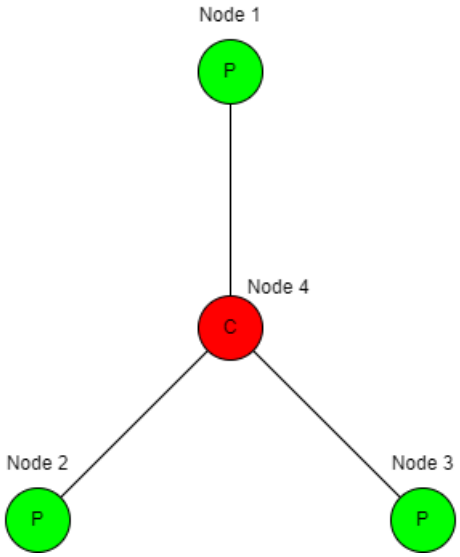


Figure 14. Network example for data structure clarification

For example, if a network constructed by Global-Star with 4 nodes has only 3 active edges which are between nodes 1 and 4, 2 and 4, 3 and 4 (index starts from 1, shown in Figure 14), the matrix will be:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Meanwhile, the array which stores states of nodes in this network will be:

$$[p, p, p, c]$$

The arrays above are created and can only store information about the network in one round. They are pushed into another two arrays which work as stacks to store the states of nodes and edges of the latest networks. Therefore, the stack which stores states of nodes is a 2-dimensional array which can be demonstrated below:

$$[[q_0, \dots q_0], \dots [q_0, \dots q_0]]$$

The elements in it will be arrays that are mentioned above to store states. The stack which stores states of edges is a 3-dimensional array, in which the elements are matrices that store states of edges. The data structure can be visualised as:

$$\left[\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}\right]$$

As shown in Figure x, the arrays that store edges and nodes are copied to the last position of the stacks. The stacks are in a FIFO (first in first out) order, also known as queues. They will then discard the first (oldest) items. In this way, the stack can always keep the latest network structures. Details are shown in Figure 15.

```
globals.stack.insert(0, deepcopy(globals.edges))
globals.state_stack.insert(0, deepcopy(globals.states))
if (len(globals.stack) > globals.additional):
    globals.stack.pop(globals.additional)
if (len(globals.state_stack) > globals.additional):
    globals.state_stack.pop(globals.additional)
```

Figure 15. Copying arrays to the stack

With the size of networks increases, the amount of data stored in stacks will increase exponentially since the data structures are 2 or 3-dimensional arrays. This requires significant computational power which also explains the reason for setting the limit of number of nodes to 2 to 10 (covered in **2.2 Functionalities**).

2.8 Sorted Arrays

As discussed in **1.4 Project Description**, the nodes for protocols implemented in this project do not have unique identifiers. In other words, node (or device) is not assigned a unique identifier that distinguishes it from other nodes in the network. Instead, nodes are identified by their network address, which can be a combination of a network prefix (to which the node is connected) and the address of the node. The network structure should therefore not be affected by the order of states of nodes and edges stored in the arrays. If the order of nodes or edges changes in a network but do not affect the overall structure, the network remains the same. When comparing these networks, the structure in its time steps is considered to be the same. For example, as shown in Figure 16, two networks of six nodes constructed by Simple-Global-Line protocol. The nodes are labelled with their index in the array that stores them.

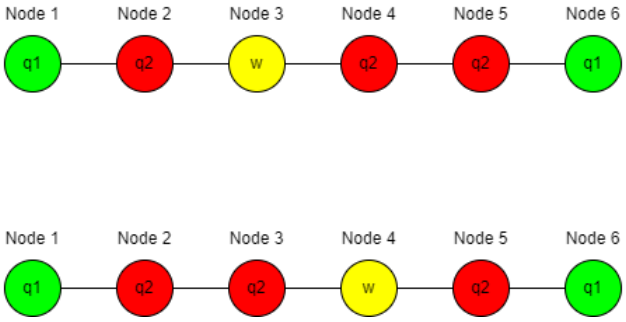


Figure 16. Sorted array network example

According to **2.7 Data Structures**, the edges of the network above in Figure 6 can be interpreted as:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The states of nodes of it will be stored as:

$$[q_1, q_2, w, q_2, q_2, q_1]$$

The edges of network below can be stored in matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Its states of nodes will be stored as:

$$[q_1, q_2, q_2, w, q_2, q_1]$$

Comparing both states of nodes and edges of these two networks, the original matrices of edges are the same, whereas the original arrays of nodes differ from each other. Nevertheless, these two networks are supposed to be the same in practical protocols as there are no unique identifiers.

The problem of not recognising networks with the same structure can only affect the overall results in Simple-Global-Line as the states change in an irreversible manner in other protocols. In Global-Star, states of nodes can only be updated from “c” to “p”. Once a “c” is eliminated, it cannot change back again. Nodes in Cycle-Cover are divided into three different states in degrees. Nodes can only upgrade their states to higher degrees. However, in Simple-Global-Line, if two spanning lines join, the node with “w” state can walk along the line in unpredictable directions. Every time this problem occurs in Simple-Global-Line, it can affect the final result by one time step. Specifically, in a certain time step, if the order of states of nodes is reversed in the next step and states of edges do not change.

To solve this problem, the elements in the stacks are sorted prior to being stored. The arrays in the stack are sorted such that when comparing elements in the stacks, if an array is the same as the reverse of another array, these two elements are considered to be identical. In the example above, the two networks’ array of states of nodes will be the same since one of them equal to the other’s reverse. In this way, the two networks are equivalent.

By implementing the method above to compare networks, the accuracy of running can be improved. However, when the network is stably constructed, no edges are added or removed, and the states should then not switch between nodes (nodes have internal unique identifiers within a network). The network structure is therefore unique. If meeting all the conditions above, a network is stable.

2.9 Custom Protocol

This protocol contains only one rule which is $(c, c, 0) \rightarrow (c, c, 1)$. Customised protocols can be stored in a CSV file and opened by the software. Each state of the node or edge should be stored in one cell with a blank cell in between of the left and right side of the arrow in the rule. For example, if one rule of a protocol is $(s_1, s_2, s) \rightarrow ({s_1}', {s_2}', s')$, its format in the CSV file is:

s_1	s_2	s		${s_1}'$	${s_2}'$	s'

File type is constrained to *.csv. The function “on_selection_changed” emits a signal that contains the path of the file when a file is selected. The content of the file is then converted into strings and the slot will be caught by main window to display the rules of the custom protocol. These functionalities are achieved by the following code (shown in Figure 17).

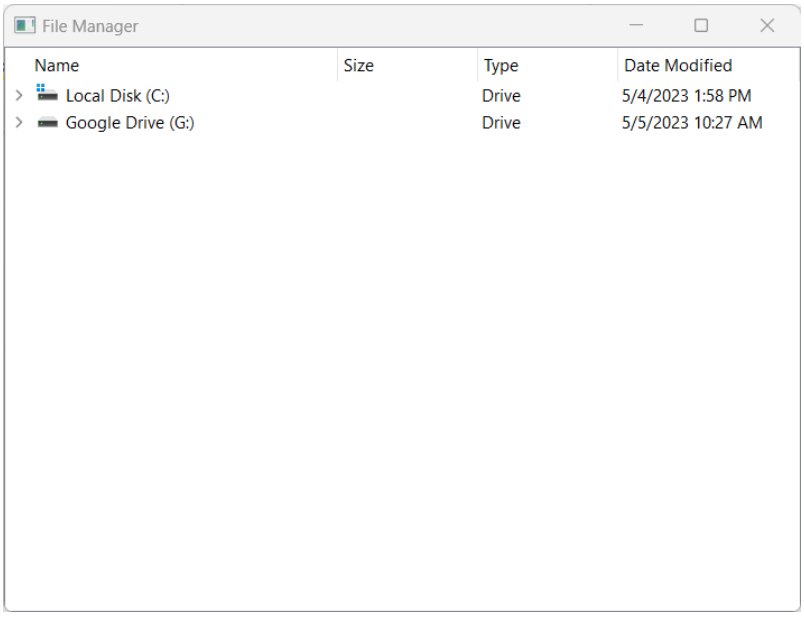


Figure 17. Sub window for custom protocol

As shown in Figure 18, the content of the file is read line by line in strings. The elements in different cells are separated by commas when reading the file. According to the format of the content, each state in the rules is at indexes 0, 1, 2, 4, 5, 6. They can then be picked out to form a new string that can be displayed as rules on main window. The exception that the format of the protocol is not correct is handled.

```
if (len(globals.custom_states) % 7 == 0):
    for x in range(0, int(len(globals.custom_states) / 7)):
        text = text + "(" + globals.custom_states[x * 7] + ", " + globals.custom_states[x * 7 + 1] + ", " +
        globals.custom_states[x * 7 + 2] + ") → (" + globals.custom_states[x * 7 + 4] + ", " +
        globals.custom_states[x * 7 + 5] + ", " + globals.custom_states[x * 7 + 6] + ")" + os.linesep
    globals.form.rules.setText(text)
    print(text)
else:
    print("error")
```

Figure 18. Code for data conversion

2.10 Data Visualisation

After all the protocols are implemented, the structure of networks can be visualised. This functionality can be achieved with libraries “matplotlib” and “networkx”. “matplotlib” plots the network and “networkx” is used to create, manipulate, and analyse networks. In “networkx” library, the states of nodes are represented by labels. These labels are added and stored in an array after the network is initialised. For example, nodes will be added with label “c” for networks being constructed by Global-Star (shown in Figure 19). This function can only plot the initial structure of the networks. An animation function will be called periodically to simulate the protocol continuously and update the graph.

```
if (globals.form.comboBox.currentText() == "Global-Star"):
    for x in range(0, globals.num):
        globals.G.add_node(x)
        globals.labels[x] = 'c'
        globals.states.append('c')
    globals.form.canvas.plot()
```

Figure 19. Network plotting example

```
globals.speed = globals.form.speed_slider.value()
if (globals.form.comboBox.currentText() == "Global-Star"):
        globals.ani = animation.FuncAnimation(globals.form.canvas.figure, globals.form.canvas.Global_Star, frames=None, interval=1000/globals.speed, repeat=True)
if (globals.form.comboBox.currentText() == "Cycle-Cover"):
        globals.ani = animation.FuncAnimation(globals.form.canvas.figure, globals.form.canvas.Cycle_Cover, frames=None, interval=1000/globals.speed, repeat=True)
if (globals.form.comboBox.currentText() == "Simple-Global-Line"):
        globals.ani = animation.FuncAnimation(globals.form.canvas.figure, globals.form.canvas.Simple_Global_Line, frames=None, interval=1000/globals.speed, repeat=True)
if (globals.form.comboBox.currentText() == "Custom"):
        globals.ani = animation.FuncAnimation(globals.form.canvas.figure, globals.form.canvas.custom, frames=None, interval=1000/globals.speed, repeat=True)
```

Figure 20. Recurringly calling functions

As shown in Figure 20, the “animation” function is a built-in function of “matplotlib” library which allows a certain function to be called for multiple times automatically. Time interval between two times the function being called is calculated as:

$$\frac{1000}{simulation\ speed \times 1000}$$

The simulation speed is taken from the slide bar from the main window with ten degrees vary from 1 to 10, and one second is divided to 1000 parts in this function. Therefore, the simulation speed is minimum 1 time step per second and maximum 10 time steps per second maximum.

3 Ethical Considerations

As a project which focuses on achieving the content in a theoretical research, no data is collected from public sources. The testing carried out by fellow students was done anonymously. The project does not contain personal information. Therefore, no ethical issues need to be considered for this project.

4 Testing

This project can be divided into 2 parts, protocol implementation and data visualisation. The protocol implementation part can be tested in unit testing and data visualisation is tested after the protocols are correctly implemented as integrated testing.

4.1 Unit Testing

The protocols were implemented before completely adding data visualisation functionalities during the process of development. Hence, it is critical to test and verify the protocols are implemented correctly.

To better inspect the network structure without visualisation, simulation results are printed out in the terminal in every time step. The results involve states of nodes, states of edges, number of additional time steps, which two nodes are interacting in a certain round, current running time, and the stability of the network. An example can be found in below in Figure 21.

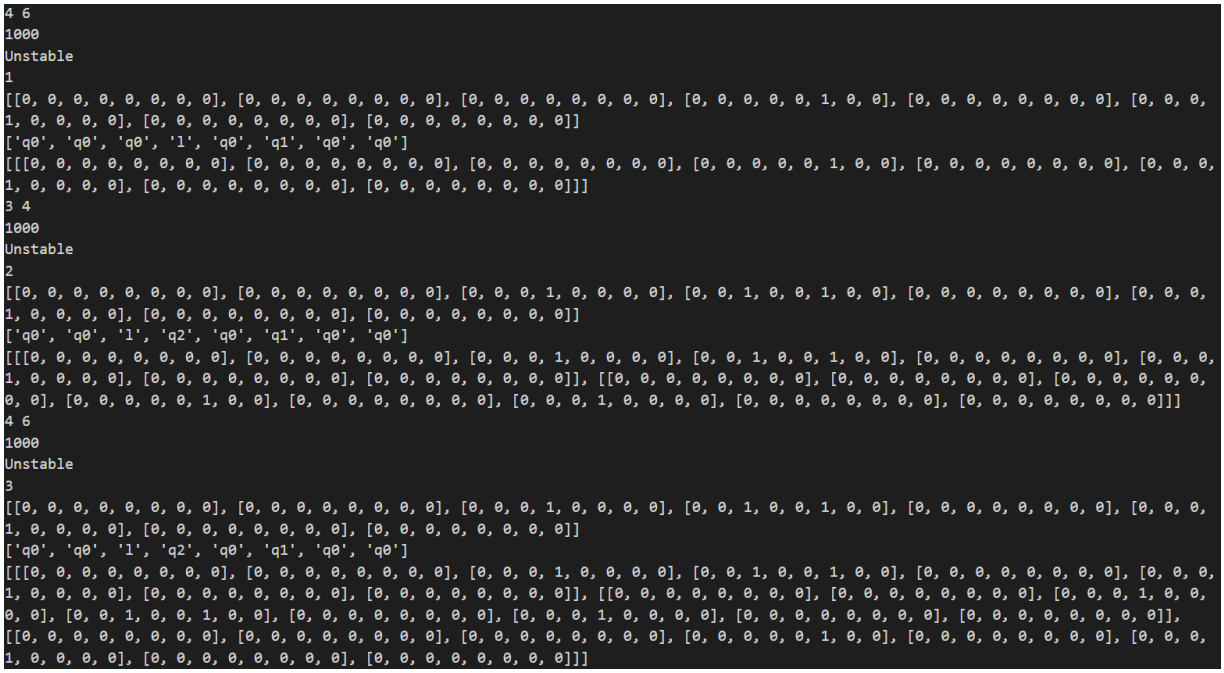


Figure 21. Terminal outputs example

After the network is stably constructed, the content in the arrays have to match the desired output. For example, if a network is stably constructed with Global-Star, there should be only one “c” in the arrays for nodes and others are “p”. In the edge storage arrays, only one array should have one 0 and other elements are 1s, and other arrays have all 0s except one 1. This is due to the stable structure of Global-Star has only one centre node and $n-1$ peripheral nodes. Similarly, the edges between the centre node and every peripheral nodes are active, while the edges between peripheral nodes are inactive. Therefore, there will be $n-1$ active connections. Figure 22 shows the output of a simulation of Global-Star which stops with a stable network. By comparing the output with conditions above, the network is stably constructed.

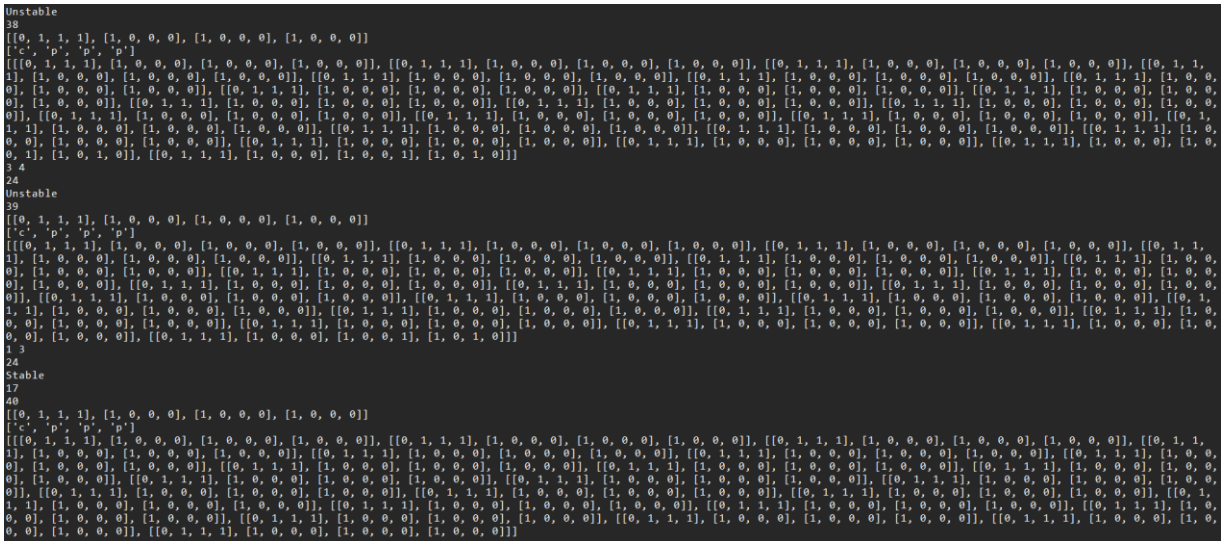


Figure 22. Terminal outputs of a stable network

Other protocols can be tested by the same method. However, the testing of Cycle-Cover must be done by considering arrays for both node and edge storage and derive the network structure. Since the Cycle-Cover protocol can have a waste of one or two nodes, it is impossible to predict the number of active edges and states of all the nodes. Thus, the network structure has to be drawn to determine if it is stable. More figures of testing of protocols are listed in **Appendices**.

By testing the output of the program in the terminal, the simulation results can be verified to be correct.

The custom protocol function is tested by creating files with different protocols and open them with the software. In Figure 23, a file with name “Custom.csv” is opened. The rule is displayed on the main window (shown in Figure 24). The output shown in this picture proves the rules are read and displayed correctly.

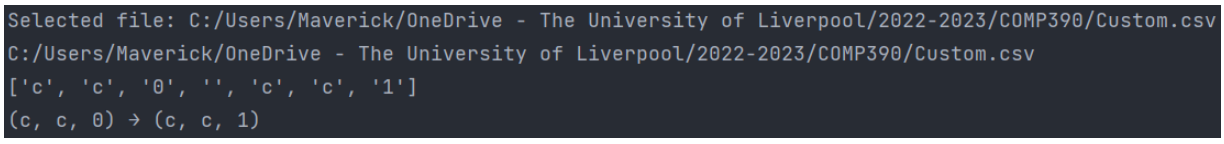


Figure 23. Terminal outputs of custom protocol reading

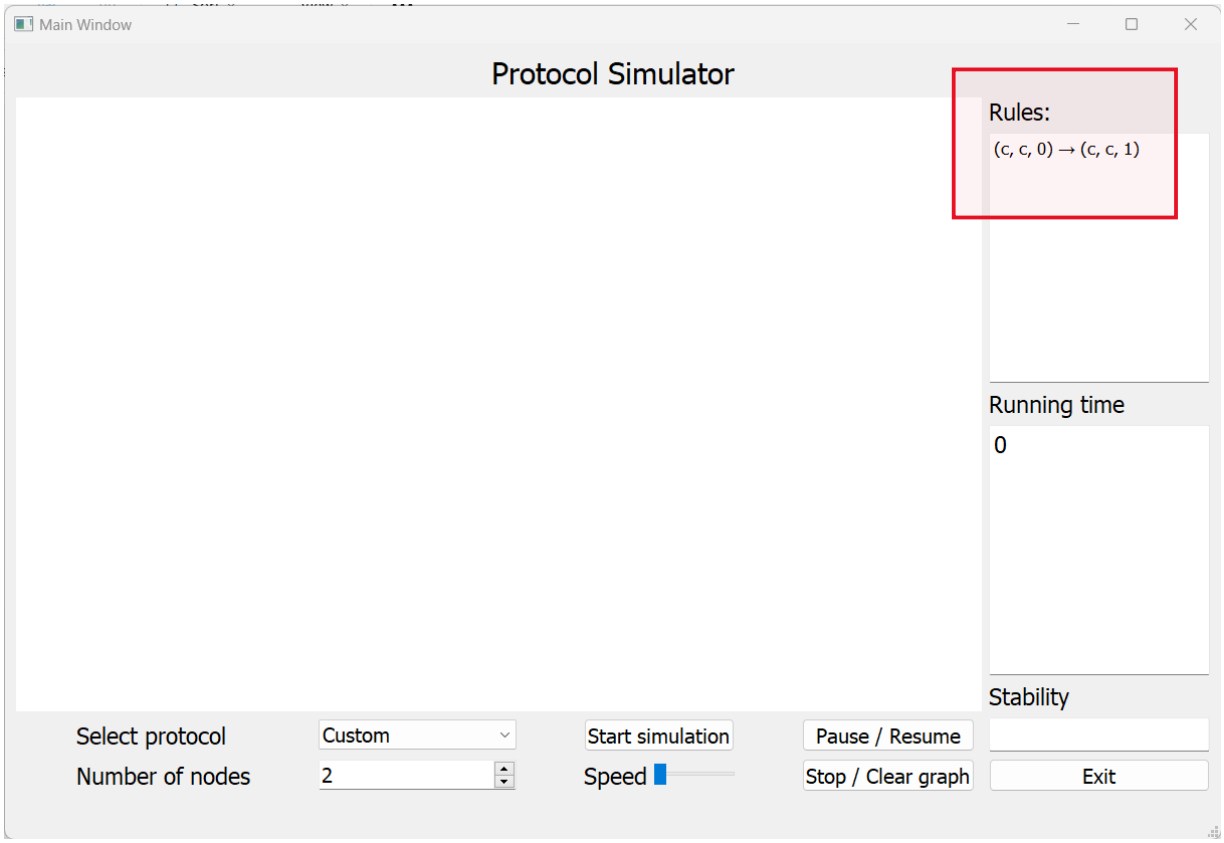


Figure 24. Main window of custom protocol displaying

Based on the results in above figures, the functionalities related to custom protocols are working correctly.

4.2 Integrated Testing

After the protocols are implemented tested successfully, data visualisation can be tested. Based on the correct results in **4.1 Unit Testing**, a reliable method to test data visualisation is comparing the network structure shown in the graph with outputs in terminal. If the network structure can match the arrays in terminal outputs, the visualisation is implemented correctly. For example, a network with six nodes is constructed by Global-Star protocol. The data visualisation results are shown in Figure 25 as a graph of nodes and edges. The terminal outputs are shown in Figure 26.

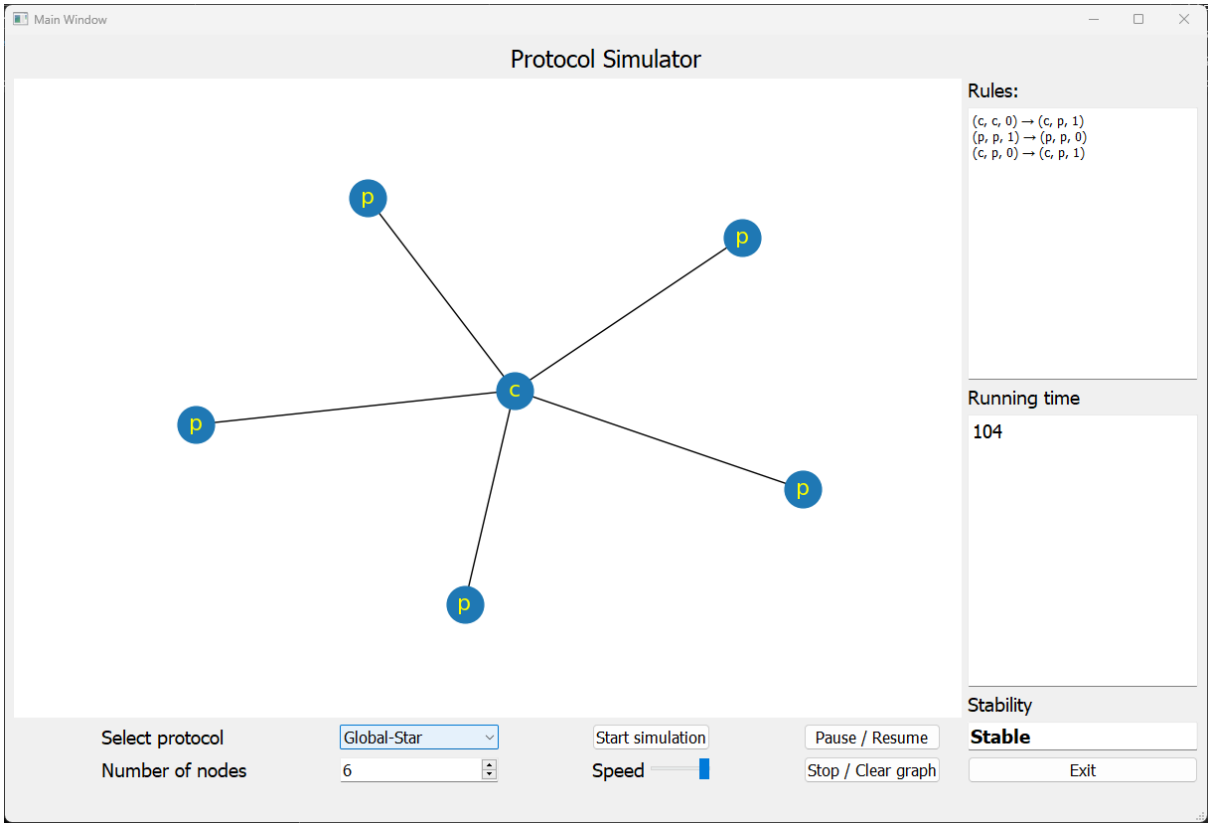


Figure 25. Global-Star visualisation

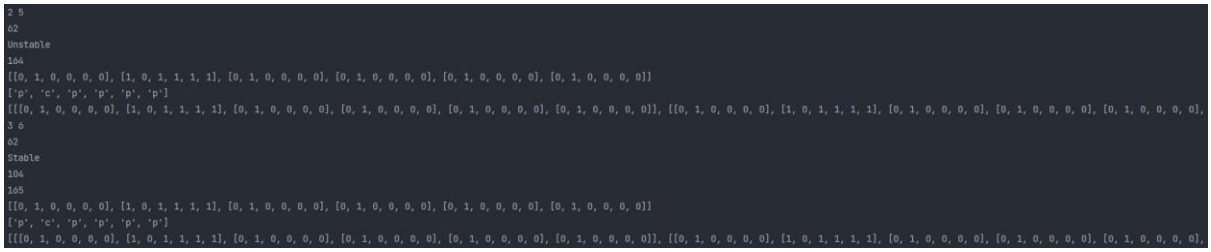


Figure 26. Global-Star terminal output

Other protocols are tested in the same way and more figures of testing results can be found in **Appendices**.

The integrated testing also includes testing of usability of the software. For example, testing if the buttons can work correctly under different conditions to identify exceptions. After usability testing and modifying the code, all potential issues are addressed.

4.3 Fellow Students Testing

The project is also tested by 3 fellow students after with the background information of the topic with suitable explanations provided. The testers are students with computer science knowledge. No issues were found during the fellow student testing.

5 Evaluation

To evaluate the correctness of the protocols, two commonly employed criteria are running time comparison and network structure comparison, as described in [16], [17]. Comparing network

structure with the desired structure can ensure the simulation terminates correctly and comparing the running time with expected time assists in analysing the performance of protocols. By using these two criteria in conjunction, it is possible to effectively verify the simulation results and ensure the correctness of the protocols.

5.1 Network Structure Comparison

An important measure of the stability of networks is the structure of them. Analysing the structure of a network can ensure its stability. The network structures are investigated and desired structures are shown in [3]. By comparing the states of both nodes and edges between them can
Each protocol will produce a certain structure when the network is stably constructed. Global-Star protocol will construct a stable network that contains one centre node and others are peripheral. All nodes are initially centres and when two centre nodes meet, one of them will become peripheral.

Therefore, the edges between the central node and every peripheral nodes are all active and edges between any two peripheral nodes are inactive (as shown in Figure 27).

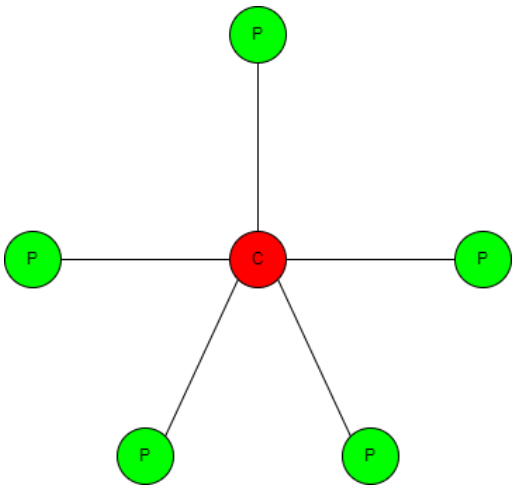


Figure 27. Stable network example (Global-Star)

Cycle-cover protocol can construct a stable network that contains one or more cycles of nodes. The nodes can have three degrees in states which are q_0, q_1, and q_2. Since the degree can only increase according to rules and at least three nodes are needed to form a cycle, some nodes can be left out from cycles. This leads to a potential maximum waste of 2 nodes, that is, an isolated node with state q_0 or two nodes in state q_1 or q_2 with an active edge between them [3]. Therefore, the networks shown in Figure 28, Figure 29, and Figure 30 are all considered to be stable.

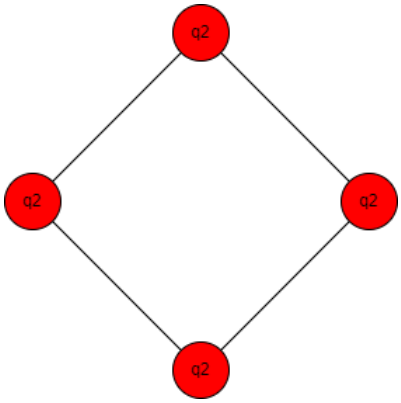


Figure 28. Cycle-Cover example

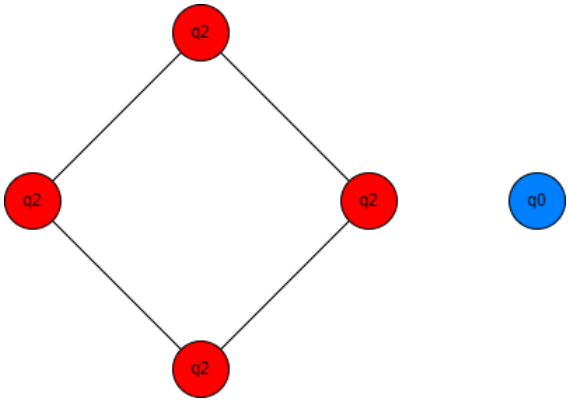


Figure 29. Cycle-Cover with waste of 1

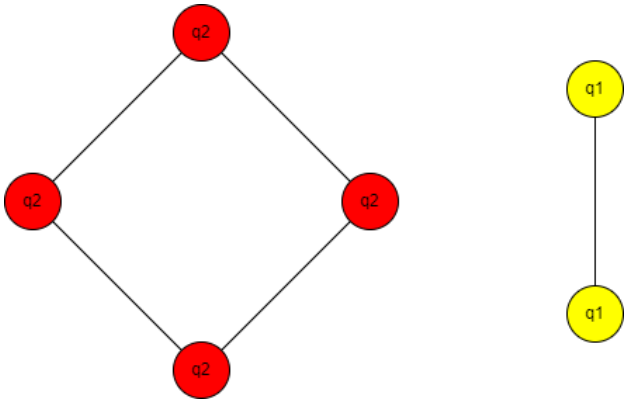


Figure 30. Cycle-Cover with waste of 2

Simple-Global-Line protocol can eventually construct a spanning line if the network structure is stable. One of two nodes on the sides of the spanning line has state l, which represents "leader", and the other node has state q_1. The nodes in between all have state q_2. An example of stable network of Simple-Global-Line is shown in Figure 31.

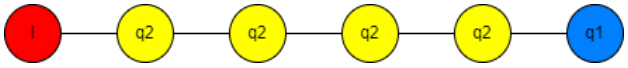


Figure 31. Simple-Global-Line example

If the final network structure formed by simulation can match the desired structure, the simulation can be considered successful. In other words, the simulation terminates correctly.

After simulating Global-Star protocol to construct networks from 2 nodes to 10 nodes and comparing network structures, the results can be verified to be correct. Sample figures of the simulation are included in **Appendices**.

By implementing the data structures and suitable numbers of additional time steps for different protocols, the probability for wrong results, for instance, unexpected early terminations, to occur is significantly decreased. No wrong results were found during rigorous testing. However, as explained in **2.6 Termination of Simulation**, the probability of unexpected terminations still exists.

5.2 Running Time Comparison

When evaluating the performance of a protocol for stable network construction, it can be useful to compare the actual running time of the protocol with the expected time and lower bound.

The expected time can represent the average performance of protocols under ideal conditions, while the lower bound represents the best performance that protocols can achieve. Therefore, the closer a protocol's lower bound and expected time d, the more efficiency and scalability it can obtain. On the contrary, if a protocol's expected time is significantly higher than its lower bound, the protocol is more likely and has more rooms to be optimised such that its performance can be improved [18], [19]. In practice, some factors including but not limited to memory usage, network traffic, and scalability can also affect the performance of protocols [17].

Both expected time and lower bound can be used to determine the correctness of a protocol. If the actual running time is between the expected time and lower bound, the correctness can be verified. However, the actual running time can go below the lower bound in some conditions. The lower bound is also determined by expected time [20] [21]. Another factor that can cause the lower bound to not be the best criterion is that the formular for lower bound calculation may have hidden constants. These hidden constants are difficult to be figured out which will affect the accuracy of results [22] [23]. Therefore, a better and more efficient method to verify protocol's correctness is only comparing the running time with expected time.

To avoid the randomness of pairwise meeting mechanism, each protocol is simulated 10 times and the average running time is calculated. The average running for a certain size of a network can then be used to compare with the expected time. If the actual running time is lower than the expected time, the simulation results can be proved to be correct.

With network structure comparison (details in **5.1 Network Structure Comparison**), wrong results are discarded. Running times of correct final network structures are used to compare with expected time.

The running times for Global-Star protocol with 2 to 10 nodes are listed below (see Table 4):

Trial	2 nodes	3 nodes	4 nodes	5 nodes	6 nodes	7 nodes	8 nodes	9 nodes	10 nodes
1	1	8	24	28	42	75	96	123	177
2	1	4	32	37	21	78	79	120	198
3	1	6	22	28	48	127	117	201	147
4	1	7	25	50	59	69	160	164	244
5	1	4	17	23	68	62	98	213	275
6	1	7	3	18	66	133	97	122	268
7	1	4	17	53	52	107	130	135	200
8	1	11	29	20	76	61	160	99	179
9	1	9	17	34	65	67	136	123	150
10	1	4	5	23	55	57	214	244	173
Avg	**1**	**6.4**	**19.1**	**31.4**	**55.2**	**83.6**	**128.7**	**154.4**	**201.1**

Table 4. Global-Star running times

The data in Table 5.1 Global-Star simulation results can be converted into graph (shown in Figure 32).

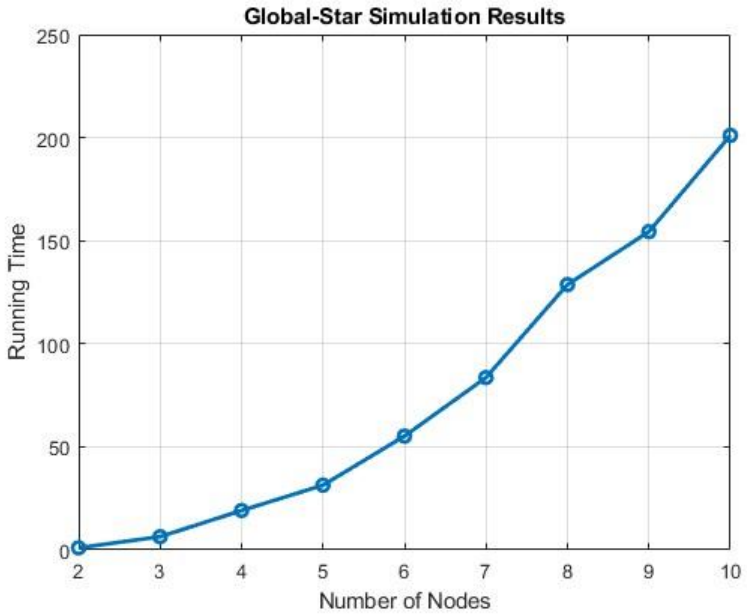


Figure 32. Global-Star simulation results

According to the expected time of protocols introduced in **2.6 Termination of Simulation**, the expected time of Global-Star protocol can be plotted as below (shown in Figure 33):

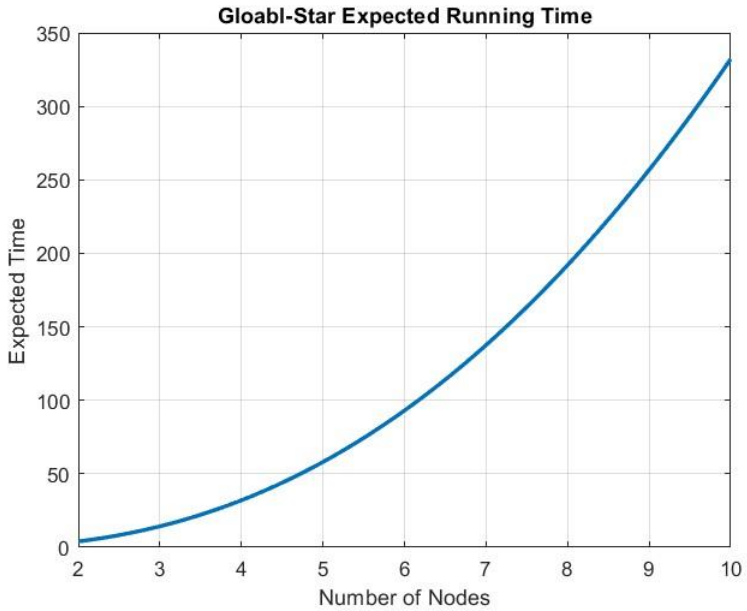


Figure 33. Global-Star expected running time

Among these three protocols, the lower bound and expected running time of Global-Star are the same. According to [3], the eventual centre node in Global-Star must meet every other node. Additionally, every other peripheral nodes must meet at least another peripheral to construct a stable network. To summarise these procedures in formular, the expected time should be:

$$\Theta(n^2) + \Theta(n^2 log_2 n)$$

For networks of large sizes, the term $\Theta(n^2)$ can be eliminated. However, the size of networks being tested in this project is relatively small. Therefore, the best criterion is comparing the expected time of Global-Star to its running time.

By comparing both Figure 32 and Figure 33, it can be noticed that the actual running time is below but close to the expected time. Therefore, the correctness of Global-Star can be verified.

After simulating Cycle-Cover protocol, the running times of it with different size of networks are listed below (see Table 5):

Trial	2 nodes	3 nodes	4 nodes	5 nodes	6 nodes	7 nodes	8 nodes	9 nodes	10 nodes
1	1	5	6	25	45	8	21	36	133
2	1	10	3	15	8	10	11	49	125
3	1	5	5	13	12	125	41	146	61
4	1	3	6	6	39	53	29	66	133
5	1	5	11	21	20	9	41	98	107
6	1	3	5	4	21	71	87	55	103
7	1	3	4	22	6	48	32	91	51
8	1	4	8	21	14	32	99	33	29
9	1	11	7	8	6	40	17	79	100
10	1	3	8	40	35	72	34	48	138
Avg	**1**	**5.2**	**6.3**	**17.5**	**20.6**	**46.8**	**41.2**	**70.1**	**98**

Table 5. Cycle-Cover simulation results

The data in Table 4. Cycle-Cover simulation results are plotted in Figure 34.

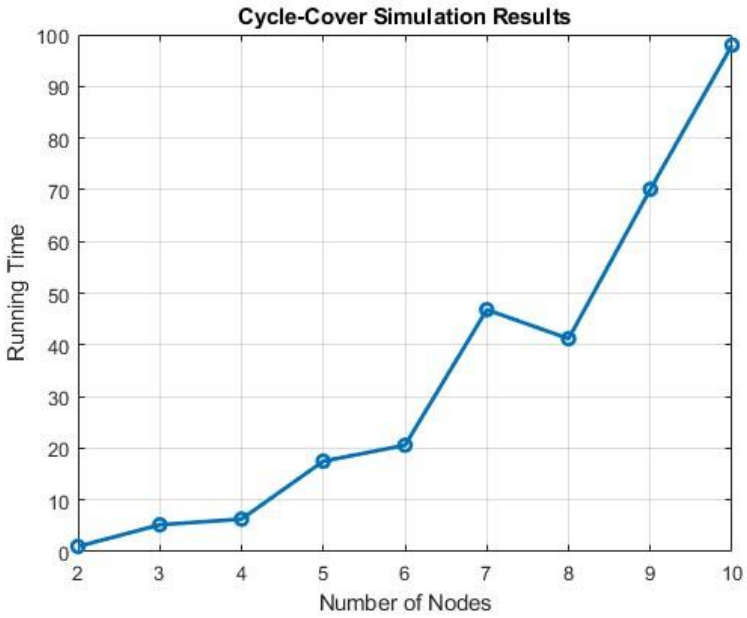


Figure 34. Cycle-Cover simulation results

The expected time of Cycle-Cover is plotted below in Figure 35.

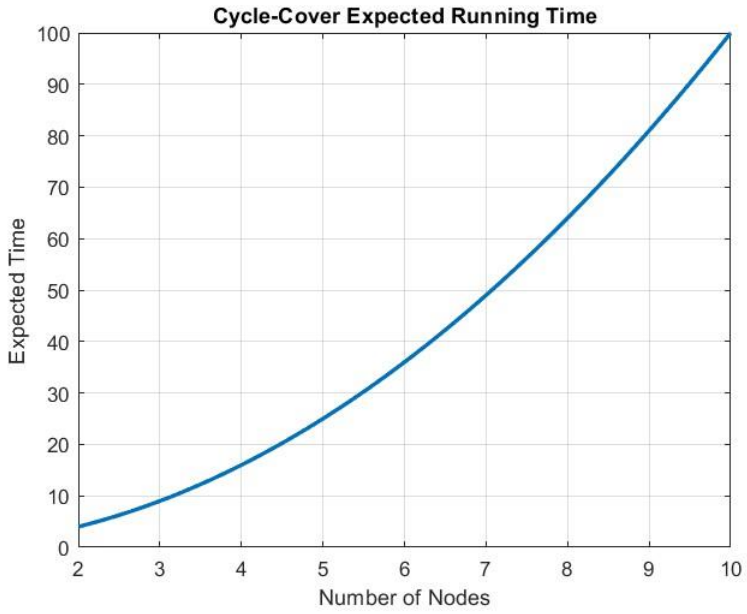


Figure 35. Cycle-Cover expected running time

Comparing graphs in Figure 34 and Figure 35, the simulation results are close to expected time. However, average time for networks with eight nodes is relatively lower than expected. This is due to the nature of the pairwise meeting mechanism. When some nodes can change the structure of the network by interacting with each other, they are exactly chosen in the time steps. Therefore, randomness allows the simulation to be on the optimistic side.

After simulating Simple-Global-Line protocol, the running times of it with different size of networks are listed below (see Table):

Trial	2 nodes	3 nodes	4 nodes	5 nodes	6 nodes	7 nodes	8 nodes	9 nodes	10 nodes
1	1	7	11	11	21	157	154	243	744
2	1	3	12	48	15	36	283	186	480
3	1	3	2	9	146	53	113	243	936
4	1	2	12	29	64	117	307	195	690
5	1	2	7	36	33	152	202	119	846
6	1	2	8	19	16	256	311	543	379
7	1	4	8	26	53	48	521	154	571
8	1	3	5	17	62	357	180	304	468
9	1	3	21	27	33	96	181	301	730
10	1	2	19	30	115	190	428	415	706
Avg	**1**	**3.1**	**10.5**	**30**	**55.8**	**146.2**	**268**	**270.3**	**655**

Table 6. Simple-Global-Line simulation results

The simulation results of Simple-Global-Line can be plotted as below (Figure 36):

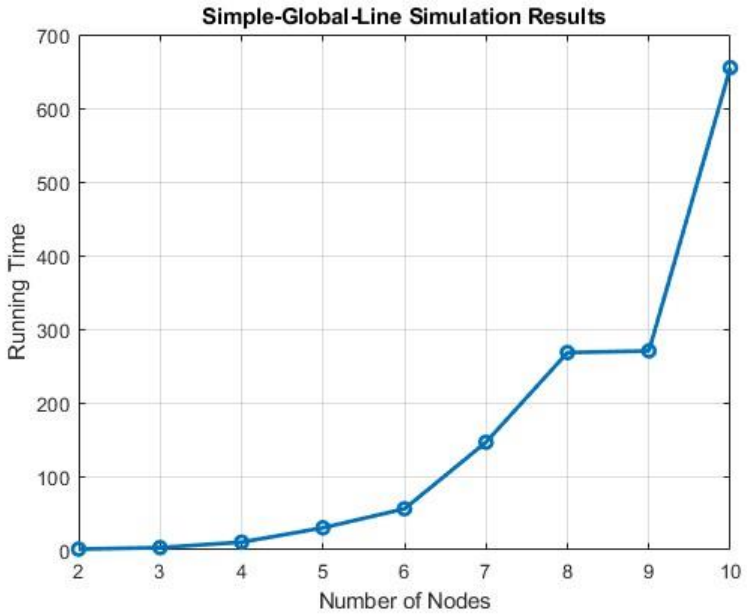


Figure 36. Simple-Global-Line simulation results

The expected time of Simple-Global-Line is plotted as below (see Figure 37):

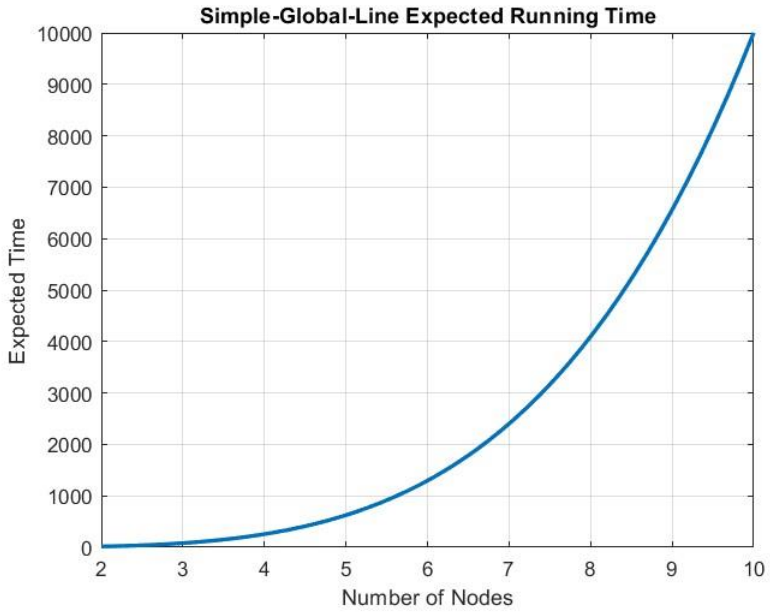


Figure 37. Simple-Global-Line expected running time

The running time in simulation of Simple-Global-Line is significantly lower than expected time. This may due to two reasons. One is that not enough sample data is collected. The number of trials for each size of the network is not large enough, which may cause the deviations. The other reason is that the sizes of networks are relatively small. In distributed network construction, as the network size grows larger, the impact of individual variations or outliers on the overall running time is reduced, and the average behaviour of the system tends to align more closely with the expected behaviour. Additionally, larger networks often have more paths and opportunities for parallelism, which can help to distribute tasks and maintain consistent performance [24].

The custom protocol is tested in the same way. Below is the table of simulation results of custom clique construction protocol (shown in Table 7).

Trial	2 nodes	3 nodes	4 nodes	5 nodes	6 nodes	7 nodes	8 nodes	9 nodes	10 nodes
1	1	5	10	27	63	88	109	96	211
2	1	3	13	34	38	77	67	186	200
3	1	10	33	25	55	127	120	243	192
4	1	5	7	26	32	56	108	195	276
5	1	5	10	17	73	133	112	119	205
6	1	4	14	22	36	42	152	543	313
7	1	4	10	30	21	100	95	154	141
8	1	3	12	31	44	68	152	304	174
9	1	6	8	27	68	116	96	301	186
10	1	5	11	26	44	83	93	415	427
Avg	**1**	**5**	**12.8**	**26.5**	**55.8**	**89**	**110.4**	**270.3**	**232.5**

Table 7. Custom protocol simulation results

The results in Table 7 can be plotted as below (shown in Figure 38):

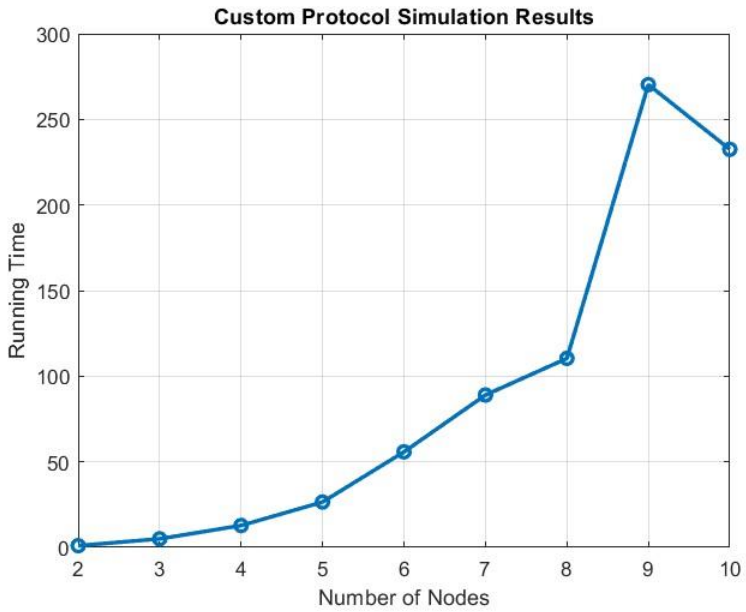


Figure 38. Custom protocol simulation results

However, since this protocol is customised and there are no academic papers to proof the correctness of this protocol, its lower bound and expected time is unknown.

6 Limitations

6.1 Universal Algorithms

This project was aimed at simulating protocols which are introduced in [3]. An expected functionality is that it can simulate any protocols in the same format. However, the custom protocol is merely for demonstration purpose. Only one fixed custom protocol can be simulated using this function. This project can be improved by implementing a universal algorithm which is able to simulation any protocols in the same format.

A potential method to achieve this functionality is generating *if* statements. The software can open a CSV file and read its content line by line. The rules can then be converted into text and analysed based on the format. Some *if* statements can be generated according to the rules. Conditions like random state change need to be taken into consideration. For example, one of Global-Star's rule is $(c, c, 0) \rightarrow (c, p, 1)$, where the node that becomes "p" from "c" is random.

This functionality is inspired by the custom protocol. Since the main aim of this project is to develop protocol simulator to execute protocols in [3], these protocols are implemented before developing the custom functionality. If this functionality can be achieved, the software will be simplified and contains less lines of code since the algorithm can also be used for protocols including Global-Star, Cycle-Cover, and Simple-Global-Line. All the protocols to be simulated can be stored in a separate CSV file. With this functionality, the code can also be optimised. Only one function is needed to simulate protocols instead of applying one function to one protocol.

6.2 Single Step Simulation

Another limitation of this project is one step simulation is not implemented. This functionality can be useful during the simulation. It allows the user to continue the simulation by one time step each time. For example, another button can be added to the main window. Once the user stops the simulation to inspect the structure of the network, running time, and stability, the button can be pressed to run the simulation for one step further. This functionality benefits in both testing and debugging of protocols during the development stage. Furthermore, simulating protocols for one time step can offer a mean for user to gain deeper insights into the underlying workings of the protocols, ultimately leading to enhanced performance and reliability.

7 BCS Criteria & Self-refection

This section will demonstrate how the software project meets the criteria established by the British Computer Society (BCS) for software development. The BCS criteria are widely recognized as a benchmark for quality and professionalism in the software industry. By adhering to these standards, the project not only delivers a high-quality software product but also follows industry best practices. The project meets relevant BCS criteria in the following aspects:

- An ability to apply practical and analytical skills gained during the degree programme: This project is developed based on Python, which is a practical skill. Moreover, data analysis skills are employed in **5.2 Running Time Comparison** to test the effectiveness of the project and evaluate the correctness of protocols.
- Innovation and/or creativity: As discussed in 2.4 Pairwise Meeting, the random scheduler is achieved by fetching nodes at two random indexes in the array. In **2.6 termination of simulation**, stacks are used to determine if a network is stable. Additionally, certain data structures are deployed store network structures (covered in **2.7 Data Structures**).
- Synthesis of information, ideas, and practices to provide a quality solution together with an evaluation of that solution: This project is to simulate protocols for stable network construction, which demands a solid understanding of topics of distributed protocols, network topologies, and algorithm, etc. The knowledge in these fields is obtained by reading research papers.
- The project meets a real need in a wider context: As discussed in **1.2 Motivation**, the project can test and verify protocols before deploying them in a practical environment.
- An ability to self-manage a significant piece of work: This project was developed following the waterfall model. Throughout the development of the project, the timeline was strictly following the Gantt chart shown in Figure 39. In the figure, all the tasks were finished on time.

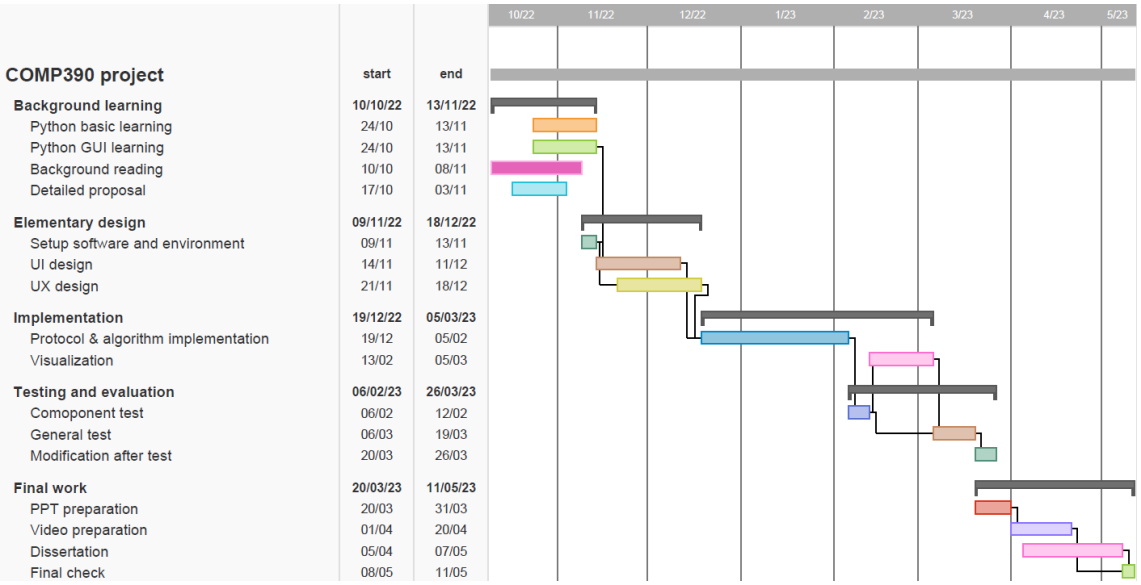


Figure 39. Project plan

- Critical self-evaluation of the process: The strengths and weakness of this project are identified. This project has achieved all the desired functionalities. However, as discussed in **6 Limitations**, that the project can be improved in some aspects if given more time.

Considering the above dimensions, this project has met the six BCS criteria by showcasing my ability to apply practical and analytical skills, innovation and creativity, synthesis of information, addressing a real need, self-management, and critical self-evaluation.

8 Conclusion

To conclude, the project has achieved all the desired functionalities, including protocol simulation, data visualisation, and custom protocol (for demonstration purposes). Stacks are used to compare structures of network and terminate the simulation. Certain data structures are implemented to store information about networks. The functionalities of the software are tested and verified. Protocols' correctness is validated by analysing data of simulation results. If given more time, the software can be optimised by deploying a universal algorithm that can execute any protocols and adding a single step simulation functionality. Last but not least, the project satisfies the BCS criteria in six aspects.

References

[1] P. Green, "An introduction to network architectures and protocols," *IEEE Transactions on Communications,* vol. 28, no. 4, pp. 413-424, 1980.

[2] N. A. Lynch and M. J. Fischer, "On describing the behavior and implementation of distributed systems," *Theoretical Computer Science,* vol. 13, no. 1, pp. 17-43, 1981.

[3] O. Michail and P. G. Spirakis, "Simple and efficient local codes for distributed stable network construction," *Distributed Computing,* vol. 29, pp. 207–237, 2016, doi: 10.1007/s00446-015-0257-4.

[4] D. Doty, "Timing in chemical reaction networks," in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, 2014: SIAM, pp. 772-784.

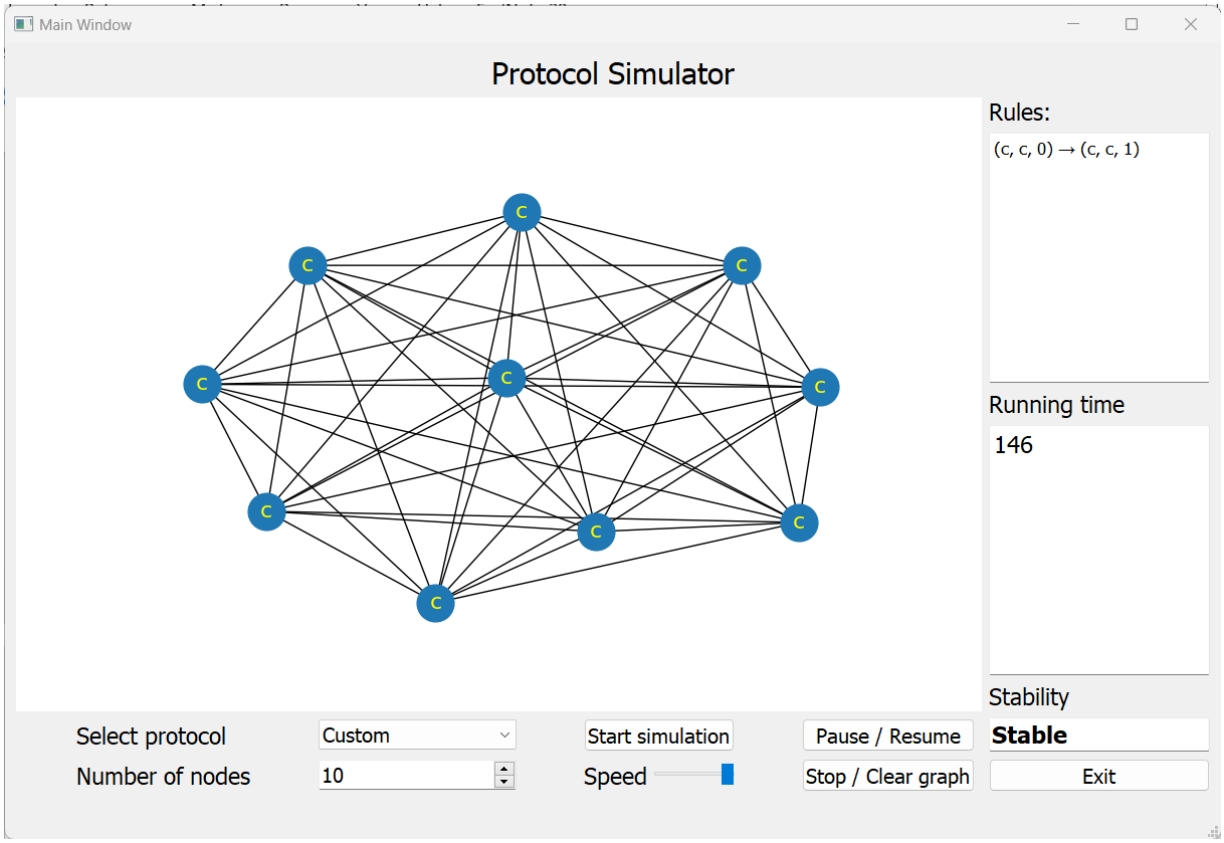
[5] M. Barrio, A. Leier, and T. T. Marquez-Lago, "Reduction of chemical reaction networks through delay distributions," *The Journal of chemical physics,* vol. 138, no. 10, pp. 104-114, 2013.

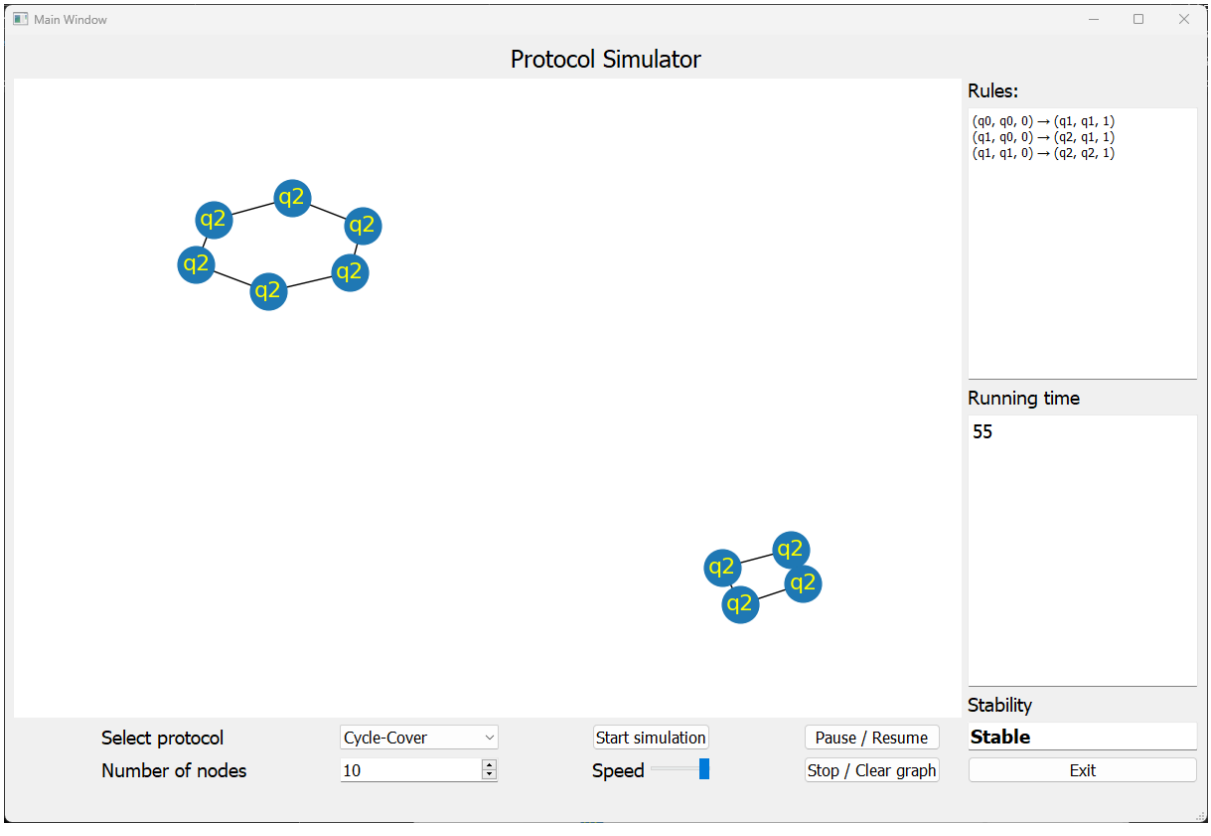
[6] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design," *IEEE transactions on Communications,* vol. 28, no. 4, pp. 624-631, 1980.

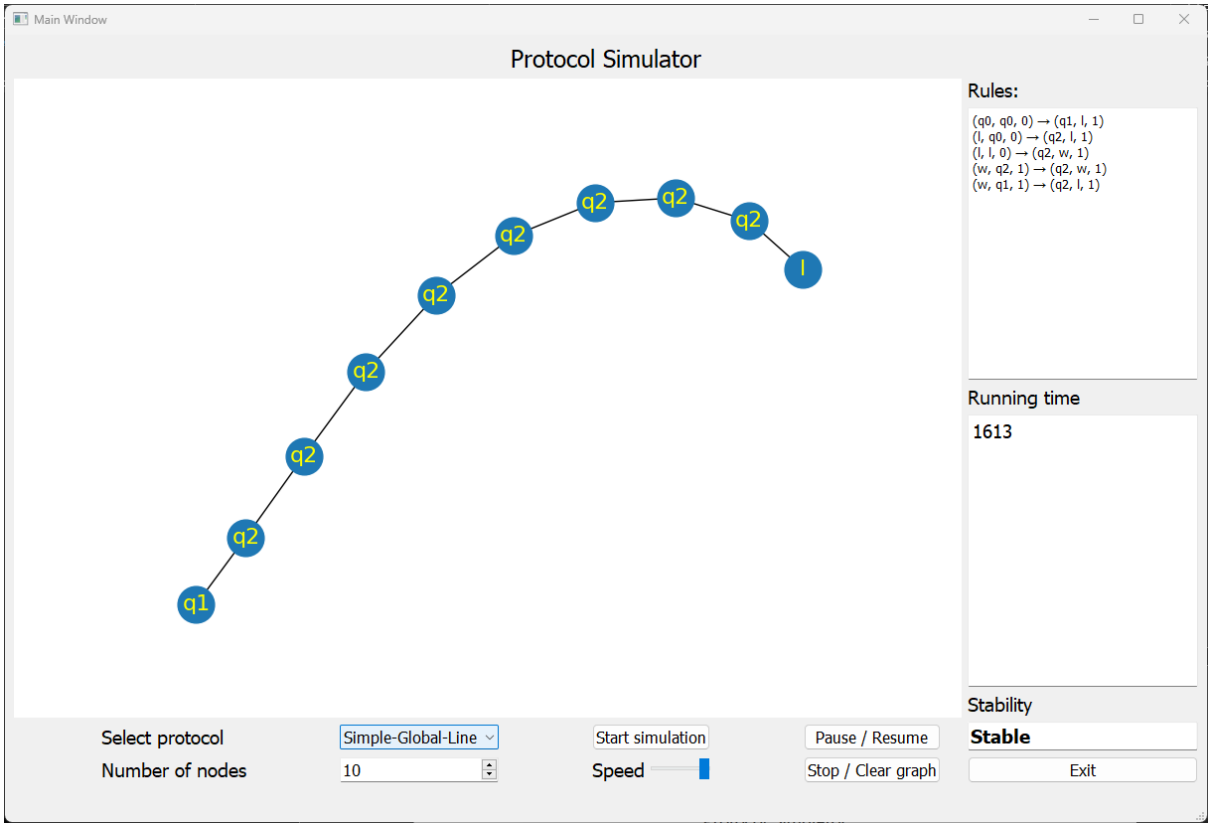
[7] M. Abdulkadhim and K. S. Korabu, "Future system: Using manet in smartphones the idea the motivation

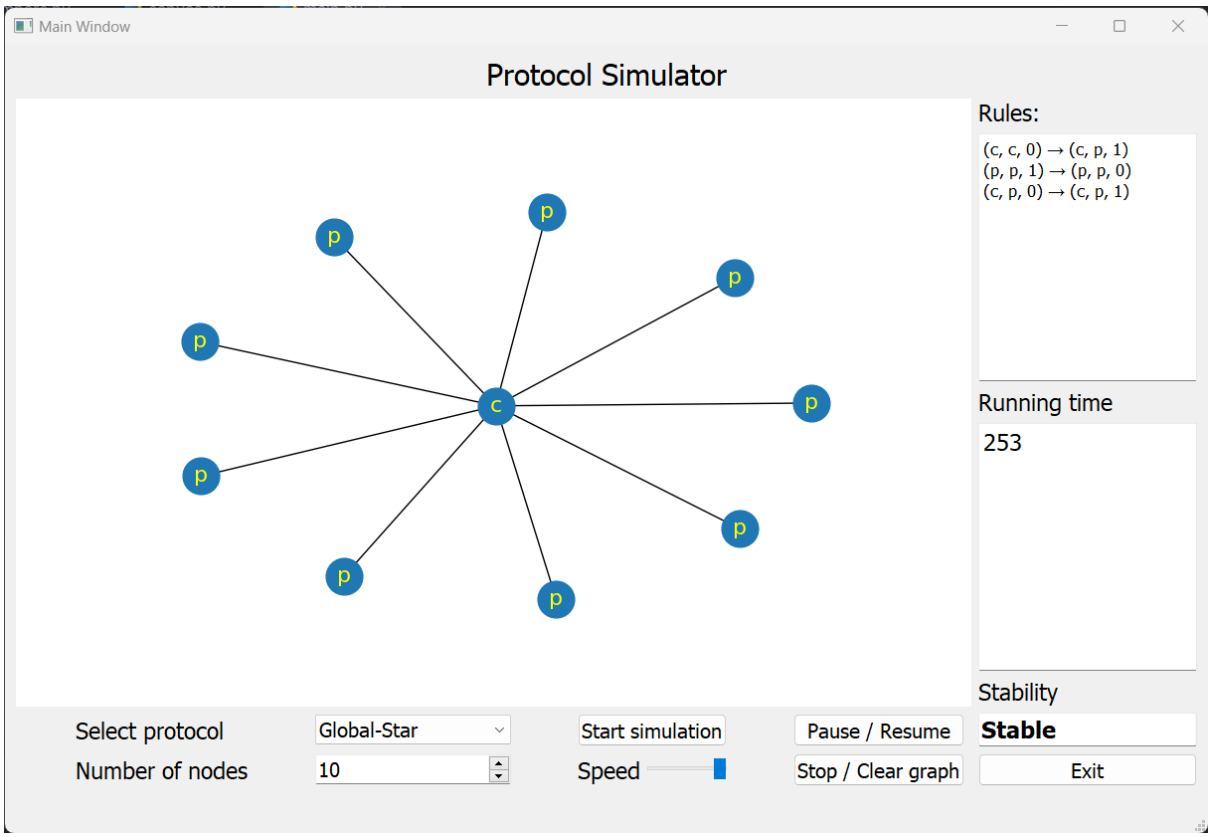
and the simulation," 2011: Springer, pp. 716-721.
[8] J. Aspnes and E. Ruppert, "An introduction to population protocols," *Middleware for Network Eccentric and Mobile Applications,* pp. 97-120, 2009.
[9] Y. Sudo, F. Ooshita, H. Kakugawa, T. Masuzawa, A. K. Datta, and L. L. Larmore, "Loosely-stabilizing leader election for arbitrary graphs in population protocol model," *IEEE Transactions on Parallel and Distributed Systems,* vol. 30, no. 6, pp. 1359-1373, 2018.
[10] A. F. McGuire, F. Santoro, and B. Cui, "Interfacing cells with vertical nanoscale devices: applications and characterization," *Annual Review of Analytical Chemistry,* vol. 11, pp. 101-126, 2018.
[11] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert, "The computational power of population protocols," *Distributed Computing,* vol. 20, no. 4, pp. 279-304, 2007.
[12] N. A. U. h. b. g. c. u. b. i. w.-x. Lynch, *Distributed Algorithms*. Elsevier Science, 1996.
[13] H. K. Qureshi, S. Rizvi, M. Saleem, S. A. Khayam, M. Rajarajan, and V. Rakocevic, "An energy efficient clique-based CDS discovery protocol for wireless sensor networks," 2010: IEEE, pp. 1-6.
[14] C. Mengucci *et al.*, "K-clique multiomics framework: a novel protocol to decipher the role of gut microbiota communities in nutritional intervention trials," *Metabolites,* vol. 12, no. 8, p. 736, 2022.
[15] H. Jiang, Z. Ge, S. Jin, and J. Wang, "Network prefix-level traffic profiling: Characterizing, modeling, and evaluation," *Computer Networks,* vol. 54, no. 18, pp. 3327-3340, 2010.
[16] G. Anastasi, A. Bartoli, and F. Spadoni, "A reliable multicast protocol for distributed mobile systems: Design and evaluation," *IEEE Transactions on Parallel and Distributed Systems,* vol. 12, no. 10, pp. 1009-1022, 2001.
[17] A. Salvi, S. Santini, and A. S. Valente, "Design, analysis and performance evaluation of a third order distributed protocol for platooning in the presence of time-varying delays and switching topologies," *Transportation Research Part C: Emerging Technologies,* vol. 80, pp. 360-383, 2017.
[18] I. Abraham, D. Dolev, and J. Y. Halpern, "Lower bounds on implementing robust and resilient mediators," 2008: Springer, pp. 302-319.
[19] M. Elkin, "Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem," 2004, pp. 331-340.
[20] J. Esparza, P. Ganty, J. Leroux, and R. Majumdar, "Verification of population protocols," *Acta Informatica,* vol. 54, no. 2, pp. 191-215, 2017.
[21] C.-H. F. Fung, K. Tamaki, and H.-K. Lo, "Performance of two quantum-key-distribution protocols," *Physical Review A,* vol. 73, no. 1, 2006.
[22] M. Elkin, "A faster distributed protocol for constructing a minimum spanning tree," *Journal of Computer and System Sciences,* vol. 72, no. 8, pp. 1282-1308, 2006.
[23] O. Michail and P. G. Spirakis, "Network constructors: A model for programmable matter," 2017: Springer, pp. 15-34.
[24] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science,* vol. 286, no. 5439, pp. 509-512, 1999.

Appendices


Main Window
Protocol Simulator
Rules:
(c, c, 0) → (c, c, 1)
Running time
146
Stability
Select protocol
Custom
Start simulation
Pause / Resume
Stable
Number of nodes
10
Speed
Stop / Clear graph
Exit


Main Window
Protocol Simulator
Rules:
(q0, q0, 0) → (q1, q1, 1)
(q1, q0, 0) → (q2, q1, 1)
(q1, q1, 0) → (q2, q2, 1)
Running time
55
Stability
Select protocol
Cycle-Cover
Start simulation
Pause / Resume
Stable
Number of nodes
10
Speed
Stop / Clear graph
Exit


Main Window
Protocol Simulator
q2
q2
q2
q2
q2
q2
l
q2
q1
Rules:
(q0, q0, 0) → (q1, l, 1)
(l, q0, 0) → (q2, l, 1)
(l, l, 0) → (q2, w, 1)
(w, q2, 1) → (q2, w, 1)
(w, q1, 1) → (q2, l, 1)
Running time
1613
Stability
Stable
Select protocol
Simple-Global-Line
Start simulation
Pause / Resume
Number of nodes
10
Speed
Stop / Clear graph
Exit


Main Window
Protocol Simulator
p
p
p
p
p
c
p
p
p
p
p
Rules:
(c, c, 0) → (c, p, 1)
(p, p, 1) → (p, p, 0)
(c, p, 0) → (c, p, 1)
Running time
253
Stability
Stable
Select protocol
Global-Star
Start simulation
Pause / Resume
Number of nodes
10
Speed
Stop / Clear graph
Exit

```
42
[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]]
['p', 'c', 'p', 'p']
[[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0,
0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1,
1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0,
0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0,
0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1,
0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0,
1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1,
0, 0], [0, 1, 0, 0]], [[0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 0], [1, 1, 0, 0]]]
1 4
24
Unstable
43
[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]]
['p', 'c', 'p', 'p']
[[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0,
0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1,
1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0,
0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0,
0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1,
0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0,
1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1,
0, 0], [0, 1, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0], [0, 1, 0, 0]]]
1 4
24
Stable
21
44
[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]]
['p', 'c', 'p', 'p']
[[[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0,
0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1,
1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0,
0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0,
0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1,
0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0,
1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1,
0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [1, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 0]]]
```

```
1 4
64
Unstable
64
[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
['q2', 'q1', 'q0', 'q0']
[[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0,
1 2
64
Unstable
65
[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
['q2', 'q1', 'q0', 'q0']
[[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0,
1 3
64
Unstable
66
[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
['q2', 'q1', 'q0', 'q0']
[[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0,
1 2
64
Stable
4
67
[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
['q2', 'q1', 'q0', 'q0']
[[[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0]], [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 0, 0], [0, 0,
```