

本科毕业设计（论文）

**二层网络协议仿真工具设计与实现**

学院（部、中心）：电子与信息学部

专 业：计算机科学与技术

班 级：计算机91班

学生姓名：叶文

学 号：2184214539

指导教师：张鹏

2023年06月

摘 要

随着互联网的普及和应用，计算机底层网络规模逐步扩大，计算机网络故障频发，网络资源浪费严重。从配置与协议出发来解决网络故障是一个新兴且充满挑战的领域。过去的研究主要聚焦在三层路由器转发设备，以此研究对应的硬件和流量异常，但这种方法往往工作量巨大。目前通过研究交换机等二层设备及协议进行链路诊断，能够帮助管理员快速了解网络运行情况，及时监测和诊断网络故障。这将提高网络的可靠性和性能，对于网络管理和运维人员具有重要意义。

本文以生成树协议簇为基础，构建了一个二层网络协议仿真工具，能够对真实网络拓扑进行建模，并进行多种生成树算法，MAC表算法检测，对底层网络链路进行优化。首先，本文解析了eNSP配置文件，选取部分重要字段封装为自定义配置文件，进行网络拓扑建模；其次根据自定义配置文件编写STP、MSTP、MAC表算法对自定义配置文件及网络拓扑进行处理，生成端口角色表、MAC表、日志文件，生成无环链路，可对域、VLAN等进行处理，并对链路进行优化；再者，优化了工具的操作性，本文使用QT套件实现了友好的可交互式界面，将以往用代码来生成配置文件的方式改为填表，方便操作；最后，通过华为eNSP工具对本文算法模块的正确性进行了验证，并进行算法效率评估。

通过对模型的测试验证，二层网络协议仿真工具STP、MSTP、MAC表算法正确性达到100%。该结果表明此仿真工具能够对真实网络进行生成树协议簇处理及MAC表处理，具有良好的生成无环拓扑及VLAN处理的能力，可以在真实网络监测中推广，在实际应用中具有价值。

**关 键 词：**生成树协议；多生成树协议；MAC表算法；二层网络配置文件

ABSTRACT

With the popularization of the Internet, the scale of the underlying computer network is gradually expanding, computer network failures are frequent, and network resources are wasted seriously. Solving network failures from configuration and protocols is an emerging and challenging area. Past research has focused on the third layer router forwarding devices as a way to study the corresponding hardware and traffic anomalies, but this approach is often a huge workload. Link diagnosis by studying second-layer devices and protocols such as switches can help administrators quickly understand the situation of network and monitor and diagnose network failures on time. This will improve the reliability and performance of the network, and it is important to network management and maintenance personnel.

Based on spanning tree protocol clusters, this paper builds a second-layer network simulation tool, which can simulate real network topology and perform two spanning tree algorithms, MAC table algorithm detection, and optimization of the underlying network links. Firstly, this paper parses the eNSP configuration file, and selects some important fields to encapsulate as custom configuration file for network topology modeling; secondly, writes STP, MSTP, and MAC table algorithms to process the custom configuration file and network topology according to the custom configuration file, generates port role table, MAC table, log files, generates loop-free links, can process domains, VLANs, etc., and optimizes the links; furthermore, the operability of the tool is optimized. In this paper, a friendly and interactive interface is implemented using the QT suite, and the previous way of using code to generate configuration files is replaced by filling in tables for easy operations; finally, the correctness and evaluation of the algorithm modules in the tool is verified and performed by Huawei eNSP tool.

Through testing and verification of the model, the second-layer network protocol simulation tool achieves 100% accuracy regarding STP, MSTP, and MAC table algorithms. This result shows that this simulation tool is capable of effectively handling spanning tree protocol cluster processing and MAC table processing for real networks. It exhibits good capability of generating loop-free topology and VLAN processing, which can be popularized in real network monitoring.

**KEY WORDS**: Spanning tree protocol; Multiple spanning tree protocol; MAC table algorithm; Second layer network configuration file

目 录

[1 绪论 1](#_Toc137215865)

[1.1 背景及意义 1](#_Toc137215866)

[1.2 国内外研究现状 2](#_Toc137215867)

[1.2.1 网络配置验证 2](#_Toc137215868)

[1.2.2 网络配置修改 2](#_Toc137215869)

[1.2.3 冗余链路及底层协议 2](#_Toc137215870)

[1.3 研究内容 3](#_Toc137215871)

[1.3.1 研究内容及目标 3](#_Toc137215872)

[1.3.2 研究设计 4](#_Toc137215873)

[1.4 本章小结 4](#_Toc137215874)

[2 网络配置文件解析及封装对象处理 5](#_Toc137215875)

[2.1 eNSP配置文件解析 5](#_Toc137215876)

[2.2 配置文件解析后封装对象 6](#_Toc137215877)

[2.3 网络拓扑生成模块 8](#_Toc137215878)

[2.4 本章小结 9](#_Toc137215879)

[3 生成树协议（STP）及MAC表生成模型 10](#_Toc137215880)

[3.1 生成树协议（STP）简介 10](#_Toc137215881)

[3.2 STP仿真模块及MAC表生成模块 10](#_Toc137215882)

[3.2.1 根桥选举 12](#_Toc137215883)

[3.2.2 根端口选举 12](#_Toc137215884)

[3.2.3 指定端口选举 13](#_Toc137215885)

[3.2.4 堵塞端口选举 13](#_Toc137215886)

[3.2.5 MAC表生成模块 14](#_Toc137215887)

[3.3 本章小结 15](#_Toc137215888)

[4 多生成树协议（MSTP）模块及MAC表生成模块 16](#_Toc137215889)

[4.1 多生成树协议（MSTP）简介 16](#_Toc137215890)

[4.2 MSTP仿真模块 17](#_Toc137215891)

[4.2.1 CIST算法模块 19](#_Toc137215892)

[4.2.2 域内多实例模块 22](#_Toc137215893)

[4.3 MAC算法处理模块 23](#_Toc137215894)

[4.4 本章小结 24](#_Toc137215895)

[5 模型评估实验与交互界面设计 25](#_Toc137215896)

[5.1 模型正确性验证 25](#_Toc137215897)

[5.2 模型效率验证 27](#_Toc137215898)

[5.3 交互界面设计 29](#_Toc137215899)

[5.4 本章小结 30](#_Toc137215900)

[6 结论与展望 31](#_Toc137215901)

[6.1 结论 31](#_Toc137215902)

[6.2 展望 31](#_Toc137215903)

[致 谢 32](#_Toc137215904)

[参考文献 33](#_Toc137215905)

[附 录 35](#_Toc137215906)

[外文原文 35](#_Toc137215907)

[附录 B 40](#_Toc137215908)

[外文译文 40](#_Toc137215909)

[附 录 C 47](#_Toc137215910)

[源程序 47](#_Toc137215911)

主要符号表

|  |  |
| --- | --- |
| *STP* | 生成树协议 |
| *MSTP* | 多生成树协议 |
| *G* | 真实网络拓扑结构 |
| *e* | *G*中对应链路 |
| *V* | 节点集合 |
| *E* | 链路集合 |
| *BPDU* | 网桥协议数据单元 |
|  |  |
|  |  |

章的MathType的章标记（打印前将其字体颜色变为白色，在打印预览中看不见即可）：

# 绪论

## 背景及意义

随着计算机网络的普及和应用，计算机网络规模飞速提高，但是在实际应用中，计算机网络故障频发，冗余链路的复杂化使得数据包传播效率低下，导致数据包的无限循环，导致网络资源的消耗，严重影响工作效率。

网络故障的诊断成为了一个突出的问题，传统的网络故障的诊断往往从设备与流量异常入手，排查方式往往集中在网络层。比如在路由器对其配置信息进行检查，或者对其接口状态及路由日志进行分析，或者通过Ping命令和TraceRoute命令（CLI命令）来检查链路的连通性以及跟踪路由信息，检查是否存在异常或延迟[1]，但是这样往往需要手动操作，或者需要对每一条路由进行操作，工作量巨大，同时这种检查方法不能让管理员快速掌握网络的整体运行情况，从而快速对故障作出反应。

目前对二层网络设备交换机及其协议研究以及链路排除来进行故障检测逐渐兴起。通过对二层交换机及其协议的仿真研究，例如对生成树协议簇进行研究，可以为节点间可达性提供判别依据，同时避免环路导致的无限循环带来的网络拥塞，根据备份节点可以在链路故障时及时切换链路，增强网络的恢复能力；生成树协议簇中MSTP协议可以提供层次化管理，将整个网络以域的概念缩小，生成多个区域及实例，提高了网络的伸缩性。通过在二层上进行研究以及建模能够为节点间的可达性提供判别依据，同时能为广播风暴等异常的发生进行预警，从而快速帮助管理人员定位链路错误。

本文旨在建立一个二层网络协议仿真工具，对真实的网络拓扑进行建模，设计并实现生成树检测算法，完成对网络设备端口仿真工作，确保网络处于无转发环路的安全状态，进而对链路进行优化，同时工具可以分析真实网络拓扑运行生成树协议（Spanning Tree Protocol，STP）、多生成树协议（Multiple Spanning Tree Protocol，MSTP）、MAC表算法的实际情况，让操作人员认识控制平面对网络运行的影响。通过仿真工具可以更好认识网络的结构，并通过生成树协议对链路进一定的优化，减少网络故障的发生，并且通过可视化工具有利于网络管理人员来检查网络运行状态，及时的发现链路故障，减少因为网络问题带来的资源消耗。

## 国内外研究现状

计算机网络故障诊断处理涉及众多方式。故障处理可以根据计算机网络底层协议，网络配置验证，网络配置修改，冗余链路优化等方面来进行处理。

### 网络配置验证

目前计算机网络诊断的研究集中在网络配置验证方向。在网络配置验证方向:Ryan Beckett及其小组在网络控制平面及网络数据方面，提出 Minesweeper，验证网络的关键属性，如节点的可达性，路径指向等，团队首次将网络配置文件转化为可以捕捉 OSPF，BGP 协议稳定状态的逻辑公式，解决了数据从控制层面出现的网络配置验证问题，并在实践中获得了较好的效率[2]。Santhosh Prabhu及其小组针对网络配置验证提出了Plankton模型，该模型结合等价划分与模型状态检查，使用符号来管理复杂的头部区域，并使用简洁的模型检查来扩充协议的行为，减少了网络配置验证的计算复杂度，同时支持丰富的协议且Plankton模型针对工业规模网络的配置验证有时能够比现有方法快10000倍[3]。

### 网络配置修改

在网络配置修改方面:目前的网络配置合成方法在网络解释性，协议的覆盖性和伸缩性上仍然存在很多缺陷，因此Ahmed El-Hassany小组提出了 NetComplete 框架，一种全新的可以接受标识参数的网络配置合成的框架，能够使用静态路由，OSPF，BGP 等完成网络结构的自动配置，可以在几分钟在多达200多个路由器的大型网络中自动完成网络配置，这种方法能够帮助网络管理员高效的修改网络[4]。但是这种有关网络配置的方法需要对网络行为有一个精确的描述，因此为了完成网络配置的修改和验证，需要一个正式的规范。 Rüdiger Birkner 小组提出了一种能够结合数据平面分析和控制平面验证的网络行为策略，能够动态的从网络配置和故障模型中挖掘网络规范，并且适用于大型网络[5]。

### 冗余链路及底层协议

计算机网络故障诊断处理可以通过对冗余链路进行诊断通过协议协议进行处理。最早的生成树协议可以追述到20世界80年代，其中最著名的算法：交换机的扩张树算法（Spanning Tree Algorithm，STA）由Radia Perlman在1985年提出，并在IEEE 802.1标准中作为生成树协议（Spanning Tree Protocol, STP）的基础算法[6]，根据算法可以对底层的链路进行优化，防止计算机网络故障的发生。

冗余链路在底层网络的应用特别广泛。多冗余链路保障底层网路的可靠性，冗余链路的存在可以使得极个别的网络适配器的传输通道故障不影响整体通信网络的运行，这种冗余链路尤其运用在军事与大型工业设施上[7][8]。目前对于物理设备及部分通信协议的故障研究集中在路由器等三层协议设备上[9]。网络传输通道和网络设备一般是冗余的，通过构成多重冗余的网络结构来保证整体网络的高效运行。为了解决冗余链路带来的环路问题、链路故障，需要从底层协议、物理设备、故障算法等多方面来解决问题。为了使管理员能够迅速掌握网络运行的整体情况， 同时为了避免链路帧在冗余链路的无限传播，广播风暴，研究交换机等二层设备就显得尤为重要，目前已经有了多种算法来解决二层数据帧转发的问题，比如MAC自学习，STP/RSTP/MSTP 等协议来解决网络环路带来的广播风暴等。同时此类算法已经广泛用在了计算机底层网络以及校园网，工业网络等。在广州地铁中运用了生成树协议提高了五号线SDH稳定的业务可靠性[10]。

为了进一步提高生成树算法的性能，张占国等人提出了一种基于网桥生成树协议 STP 的算法，能够利用简单网络管理协议(Simple Network Management Protocol, SNMP) 获得各个交换机 MIB 库中的生成树状态信息，根据生成树协议推导出网络的物理拓扑[11]。和已有算法相比，该物理拓扑发现算法不要求各个网桥的 FDB 表信息是完备的，因此提高了生成树的收敛速率。Lu Chang-jiang 及其小组改了生成树算法中BPDU的固定发送间隔，采取自适应性时间间隔发送 BPDU 数据包算法策略，同时考虑网络拓扑变化的随机性提高收敛速度[12]。还有部分研究人员采用 VRRP+MSTP 协议的网络来提高网络的安全性[13],同时Edoardo Longo小组受到二层协议的启发，研发以生成树进行组织的分布式代理架构协议MQTT-ST[14]。

在现有的相关研究中，大部分工作主要集中在网络层，涉及网络配置验证与生成、配置修改策略等方面的方法。此外，还有研究者针对网络规范开展了创新性算法的探讨，试图通过这些算法验证网络是否符合相应规范。然而，在数据链路层的研究方面，主要侧重于基于算法的探索。例如，对于二层网络交换机数据帧传播算法模型的优化改进，以及新型协议算法的研发等。本文设计并实现一个协议仿真工具，采用多种生成树协议优化冗余链路，帮助使用人员了解网络整体结构。这种仿真工具能够帮助我们深入了解不同协议在相同网络拓扑下的工作原理，从而为网络故障诊断和学习做出重要贡献。

## 研究内容

### 研究内容及目标

本文主要的研究内容为从网络的控制平面出发，以配置文件与协议为起点，设计并实现生成树协议，MAC学习等二层协议的仿真工具，为交换机及PC主机节点的可达性提供依据，同时避免网络环路的产生，减少广播风暴发生的可能性。

本文主要对静态的网络拓扑结构进行分析研究，并不把故障中类似于链路中断，物理故障纳入考虑范围。算法过程中，对应交换机各端口生成的角色一经确定就不会再重新选拔，没有考虑网络拓扑中途改变的情况，因此并不涉及根桥故障，直连链路故障，非直连链路故障这三种情况，也不涉及具体的保护功能。本模型可以针对真实网络拓扑对多种生成树协议及MAC表算法进行仿真，且结果与企业网络仿真平台（Enterprise Network Simulation Platform, eNSP）仿真工具一致。

### 研究设计

本文运用文献研究法，先阅读文献总结数据链路层交换机配置文件，解析配置文件并抽象出需要的具体属性，总结生成树，多生成树算法的详细过程步骤，为后续实验作准备。后解析出重要属性完成生成树簇算法的编程设计，最终采用验证法利用华为eNSP工具完成正确性及效率的验证。

本文的具体工作可以详细分为以下几个模块：

第二节统筹梳理数据链路层交换机具体的配置文件信息，如读取到的交换机设备信息，对应的接口对象，边对象的信息，从接口中读取如根桥ID，路径开销等详细指标，并将其刻画封装为具体的对象，配置网络输入拓扑文件。并在第三节中以算法实现为导向，详细说明封装配置文件如何生成网络拓扑节点中的角色；分析桥协议数据单元（BPDU）文件在STP/MSTP算法修改传播以及互相影响的过程，设计选拔多种端口角色如根端口，指定端口，替代端口，备份端口，指定端口等的对应生成树算法；通过华为eNSP工具进行算法验证，评估算法的效率，最终通过PYQT5交互界面（GUI）封装，成为一个易操作的协议仿真工具，并对输入的多种结果进行分析。

## 本章小结

本章描述了二层网络仿真工具的研究背景意义，强调了仿真工具在面对多种冗余链路的功能，以及应对网络环路引发的网络风暴的重要意义。其次归纳总结了目前检查网络运行状态，监测链路故障的的国内外研究现状，并指出部分研究的缺口，以及本文的研究灵感来源，较为粗略的说明解释了需要解决的任务以及方法。

# 网络配置文件解析及封装对象处理

根据企业网络仿真平台（Enterprise Network Simulation Platform, eNSP）对生成树协议及多生成树协议对应的配置文件及生成树协议及多生成树协议桥协议数据单元进行分析，解析配置文件中与生成树协议有关的属性。图2-1说明了配置文件解析及封装的过程，将真实拓扑中重要属性封装为接收对象以供后续拓扑生成模块使用。

图 2‑1 配置文件解析封装及处理流程

## eNSP配置文件解析

eNSP配置文件主要用于定义真实网络环境仿真情况下设备连接情况及其配置。配置文件中可以指定设备（如路由器，交换机）的具体属性（如IP地址，优先级，接口模式，虚拟局域网 ( Virtual Local Area Network, VLAN)配置等）和设备间链路的连接属性。eNSP与生成树协议有关的配置文件对应的主要字段如表2-1，根据对应的配置文件可以运用生成树协议或多生成树协议生成网络拓扑结构中具体的生成树，表2-1中列举了eNSP中配置路由器及其端口对应配置字段及其说明。

表2‑1 eNSP配置文件重要字段

|  |  |
| --- | --- |
| 配置字段 | 字段说明 |
| stp mode {stp | rstp | mstp} | 配置交换机工作模式 |
| Stp [ instace instace-id ] priority *priority* | 配置实例及其映射关系 |
| Stp pathcost-standard{dot1d-1998 | dot1t | legacy } | 端口路径开销配置 |

表2‑1（续）

|  |  |
| --- | --- |
| 配置字段 | 字段说明 |
| stp[instace instace-id] cost *cost* | 设置当前端口在指定生成树实例中路径开销 |
| Stp port priority *priority* | 配置当前端口的优先级 |
| stp region-configuration | 激活MST域配置 |
| region-name *name* | 域名设置 |
| Instance instace- id vlan { vlan-id1 to vlan-id2} | 配置实例与VLAN映射 |
| Stp [ instace instace-id ] root {primary | secondary } | 配置指定MSTI根桥 |

对表2-1中对应配置文件进行解析得到关键字段及其功能，对部分关键字段进行整理封装，得到交换机，交换机端口，链路属性的详细描述，和多生成树协议对应域与VLAN的映射描述，在2.2节中通过4类描述可以生成生成树协议及多生成树协议对应的网络拓扑结构，作为后续生成树算法的计算基础结构。

## 配置文件解析与封装对象

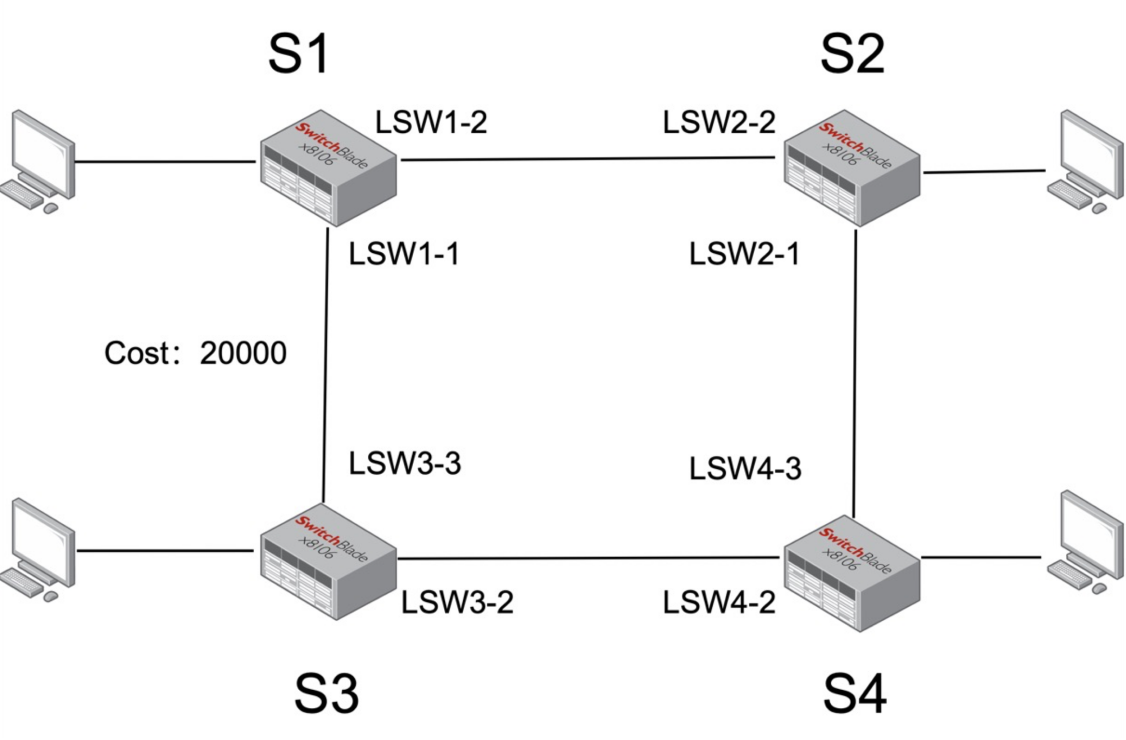
对表2-1配置文件中有关生成树协议的配置字段中的交换机配置，交换机端口配置，端口链路配置，实例与VLAN映射，进行属性分类，本文对原先的配置命令进行分析，选举与生成树协议强相关的字段并加以简化，将原先代码形式的配置行解析出4种对应的表格信息。图2-2表示了一个包含4个交换机的，9个交换机端口且在唯一域的真实网络拓扑示意图及域内VLAN与实例的映射关系。

图 2‑2 单一域4个交换机的网络拓扑结构

将图2-2包含的网络配置文件信息转化为表2-2，表2-3，表2-4，表2-5信息，分别代表交换机信息，交换机端口信息，链路信息，实例与VLAN映射信息，供图2-1后的拓扑生成模块生成网络拓扑。

表2-2表示网络拓扑结构中的交换机配置信息。字段从左到右分别表示为：交换机所具有的所有端口的端口号；交换机优先级；MAC信息；交换机名称；交换机所在域的域名；交换机所连接的终端网络。其中所有端口可直接用字符串，以逗号隔开；交换机优先级占16位，为桥优先级；MAC信息占48位，共同组成了交换机的桥ID，表示桥的优先级；交换机名称为交换机的唯一识别主键；所属域名为MSTP协议中交换机具体所属域； PC-MAC表示交换机所致连接的终端网络MAC地址。表2-2表示图2-2中4个交换机的具体信息。

表2‑2 交换机配置表

| 所有端口 | 优先级 | MAC信息 | 名称 | 所属域名 | PC-MAC |
| --- | --- | --- | --- | --- | --- |
| 1,2 | 32768 | 1 | LSW1 | RG1 | Mac1 |
| 1,2 | 32768 | 2 | LSW2 | RG1 | Mac2 |
| 2,3 | 32768 | 3 | LSW3 | RG1 | Mac3 |
| 2,3 | 32768 | 4 | LSW4 | RG1 | Mac4 |

表2-3表示交换机端口的详细配置信息。字段从左到右分别表示为端口ID；端口名称；端口优先级；端口可转发的VLAN范围。其中端口ID为端口的唯一识别主键；端口号为12位的值，用于唯一标识每个交换机的每个端口；端口优先级为一个4位的值，端口号和端口优先级组成端口标示符（Port Identifier）；VLAN表示为端口中可以进行VLAN转发的VLAN范围。表2-3表示图2-2中8个非PC端口的详细信息，PC端口会根据表2-2自动进行生成。

表2‑3 端口配置表

| 端口ID | 端口名 | 优先级 | 通过VLAN |
| --- | --- | --- | --- |
| LSW1-1 | 1 | 128 | 2-20 |
| LSW1-2 | 2 | 128 | 2-20 |
| LSW2-1 | 1 | 128 | 2-20 |
| LSW2-2 | 2 | 128 | 2-20 |
| LSW3-3 | 3 | 128 | 2-20 |
| LSW3-2 | 2 | 128 | 2-20 |
| LSW4-2 | 2 | 128 | 2-20 |
| LSW4-3 | 3 | 128 | 2-20 |

表2-4表示链路配置表为交换机端口之间连接链路的具体信息。链路配置表包含3个字段，分别为：端口；端口；两个端口链路之间的路径开销。端口，端口代表链路所连接交换机端口节点；路径开销表示链路端口的路径开销，默认为20000。表2-4表示图2-2中4条链路的详细信息，其中路径开销为默认开销。

表2-4 链路配置表

| 端口ID1 | 端口ID2 | 路径开销 |
| --- | --- | --- |
| LSW1-2 | LSW2-2 | 20000 |
| LSW1-1 | LSW3-3 | 20000 |
| LSW2-1 | LSW4-3 | 20000 |
| LSW3-2 | LSW4-2 | 20000 |

表2-5 VLAN映射表存储着多生成树协议中对应的多个实例和VLAN之间的映射关系。字段从左到右分别为：域名信息；域内实例号；转发VLAN范围；指定根桥。其中域名为表2-2对应域名；实例（MSTI）ID表示分配MSTI号，用于区分不同的生成树；VLAN ID表示此实例可以转发的VLAN范围；实例根桥表示对应MSTI中对应的根桥，根桥会将桥优先级置0。表2-5表示了图2-2中的域内映射的文字信息，包含了一个域内的两条映射关系，在域内指定两个实例，实例1转发VLAN在2-10的具体信息，实例1对应根桥为LSW1；实例2转发VLAN在11-20的具体信息，实例2对应根桥为LSW2。

表2‑5 实例与VLAN映射表

| 域名 | 实例ID | VLAN ID | 实例根桥 |
| --- | --- | --- | --- |
| RG1 | 1 | 2-10 | LSW1 |
| RG1 | 2 | 11-20 | LSW2 |

对于多生成树协议（MSTP）通过以上4种输入范例表可构建出完整的网络拓扑关系，供图2-1拓扑生成模块使用，生成初始网络拓扑图。对于生成树协议（STP）通过表2-2，表2-3，表2-4可以生成对应初始的网络拓扑图，供图2-1拓扑生成模块使用。

## 网络拓扑生成模块

拓扑生成模块主要采取Python中NetworkX库来接受表2-2，2-3，2-4，2-5数据，添加节点及边属性，构成网络拓扑图。NetworkX库是创建，操作和研究复杂网络的Python库，其中提供了广泛的网络分析工具及多种API来构建处理各种复杂网络结构。

拓扑生成模块中，模块接收表2-2的数据创建交换机（switch节点）节点，以及PC节点，构建无环图, 其中表示节点集合，表示边的集合，目前无环图中存在switch 节点及PC终端节点 ，公式2-1表示添加的内部链路，添加内部链路*e*用于连接某一个交换机对应的节点。

|  |  |  |
| --- | --- | --- |
|  |  | （2-1） |

式中：*e* ——链路；*Dnode* —— 终端节点；*Snode* —— 源节点；*cost* ——开销。

根据表2-3，及2-4的所有端口及端口号映射关系添加交换机间链路，以（2-1）中添加外部链路*e*:

|  |  |  |
| --- | --- | --- |
|  |  | (2-2) |

式中：*e* ——链路；*Dnode* —— 终端节点；*Snode* —— 源节点；*cost* ——开销。

根据表2-5添加MSTI及VLAN映射关系:

|  |  |  |
| --- | --- | --- |
|  |  | (2-3) |

式中:—— 域内实例；—— 域名； —— 根桥； —— 对应VLAN号。

模块生成对应节点及节点间链路即可构建网络拓扑。网络拓扑经生成树协议/多生成树协议，3.1节，3.2节处理生成STP及MSTP协议对应的无环链路后经3.3节处理生成MAC表。

## 本章小结

本章解析了eNSP配置文件中的对应生成树协议及多生成协议的配置字段，通过提取关键配置字段，对生成树协议及多生成树协议构建了生成拓扑范例模式，通过对应模式输入可以将真实网络拓扑结构输入仿真工具的拓扑生成模块，拓扑生成模块生成初始网络拓扑供后续算法仿真模块使用。

# 生成树协议（STP）及MAC表生成模型

## 生成树协议（STP）简介

生成树协议是一种用在以太网的数据链路层网络协议，其目的是为了在有环网络中创建一个无环链路同时进行路径优化。对于没有生成树的网络环境，每一个交换机均作为网桥，由于过多的冗余链路会导致环路的形成，比如ping命令时端口向网络发送ARP广播包，在其他交换机端口进行抓包网络中会瞬间产生大量的广播报，同时在交换机日志中可以发现出现了路由震荡的现象，交换机设备CPU利用率会远超正常值，这时引起了带宽等资源的大量消耗，引发了广播风暴。对于实施生成树协议的网络，通过生成以某个交换机为根桥的唯一活动路径，避免环路的产生，对于以太网中网络拓扑发生变化时，通过生成树协议进行收敛，达到保护的目的。

生成树协议的过程如下：首先，通过交换机的桥优先级（具有最低优先级及最低MAC地址优先级最高）选举根桥。根桥广播生成树对应的数据单元：桥协议数据单元（Bridge Protocol Data Unit ，BPDU），根据BPDU传递各个交换机及交换机端口的具体信息。非根桥通过接收传递的BPDU，根据BPDU优先级来更新自身的详细参数，同时将自身的参数通过BPDU传递给其他的根桥及端口。每个端口根据收到的BPDU确定端口角色，其中端口角色分为根端口、指定端口、阻塞端口。

生成树协议规则为：对网络拓扑，为对应节点，为对应的边。对于中具有最小桥ID（BID）的选举为根桥，对每个且中同一交换机，所在端口到根桥最小的开销的链路被选中，对应的非根桥的端口为根端口，对每一条链路，对两端节点选举指定节点，如果在给定交换机和根桥之间存在两条最低开销相同的路径，则使用路径中第二个交换机节点的选举角色。生成树选举的参数通过BPDU数据单元在网络中进行传播交换，最终确保所有的交换机均具有最新的且一致的生成树信息。

当拓扑稳定之后，根桥根据Hello定时器定时发送BPDU报文给下级节点，下级节点根据上游BPDU报文更新配置信息。

当网络拓扑发生变化时，下游设备发送TCN BPDU报文，上游指定端口对TCN BPDU报文进行处理，使用Flags字段中TCA（拓扑变化标志）置1的配置BPDU报文发送给下游设备，告知下游设备已接收，根桥收到复制TCN BPDU报文，根桥使用Flags字段中TC为1的配置BPDU报文向下游进行回送，告知下游设备删除故障的端口MAC地址表，以此快速收敛

## STP仿真模块和MAC表生成模块

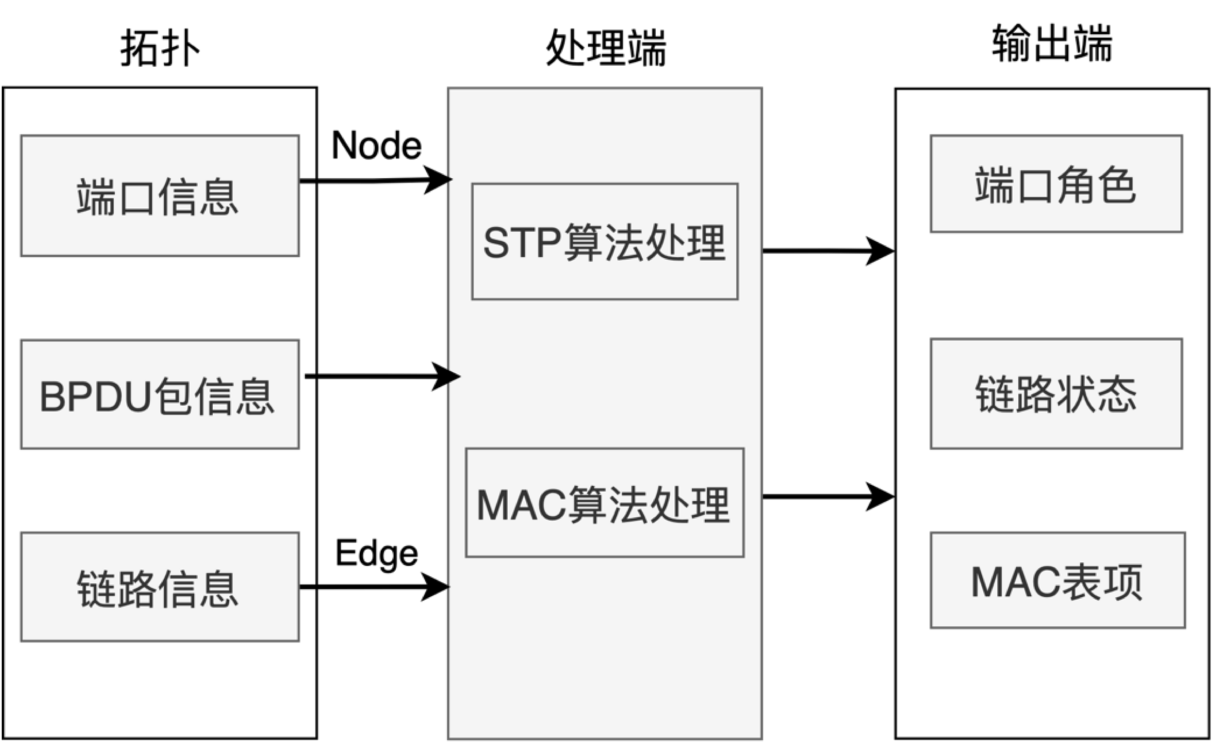
生成树(STP)仿真模块主要模拟IEEE 802.1D的选拔流程规范及陈金木学者对STP协议流程的选拔[15]。图3-1为STP仿真模块具体流程图。图3-1拓扑模块包含了初始网络拓扑的结构，处理端通过网络拓扑运行STP算法处理模块及MAC算法处理模块后生成对应结构，最后经过输出端对网络拓扑图, 中节点进行端口角色更新，对中链路是否阻塞进行更新，并生交换机对应的MAC表项。

图 3‑1 STP模拟算法结构图。

处理端中STP算法模拟IEEE 802.1D具体选拔流程。传统IEEE STP协议主要采用动态BPDU发现节点，而工具仿真模块通过输入即可知具体的网络连接，因此可以通过整体的比较来确定根桥与端口角色，收敛会更加迅速。

算法模块中定义了端口配置文件信息，本文中定义为一个四元组向量（3-1）：

|  |  |  |
| --- | --- | --- |
|  |  | (3-1) |

式中：——根桥ID，——累计跟路径开销，——发送者桥ID，——发送端口ID，优先级从左到右依次下降。

STP模块中在选举端口角色中为每个端口维护一个最优的四元组向量，不同端口的选举中根据端口维护的向量，比较优先级确定不同的端口角色。

图3-2表示一个宿舍楼层网络拓扑结构，该网络包含8个对应的交换机及若干终端节点，其中终端节点代表每个宿舍的具体局域网，经过2.3节网络拓扑生成模块，构建网络拓扑图 输入至处理模块中, 其中V是所有端口的集合，E为网络链路的集合。图3-2网络拓扑中V包含24个交换机端口节点及24个不同的宿舍信息，8个交换机节点，18个边缘及PC端口；网络拓扑中E包含21条不同端口开销的链路。

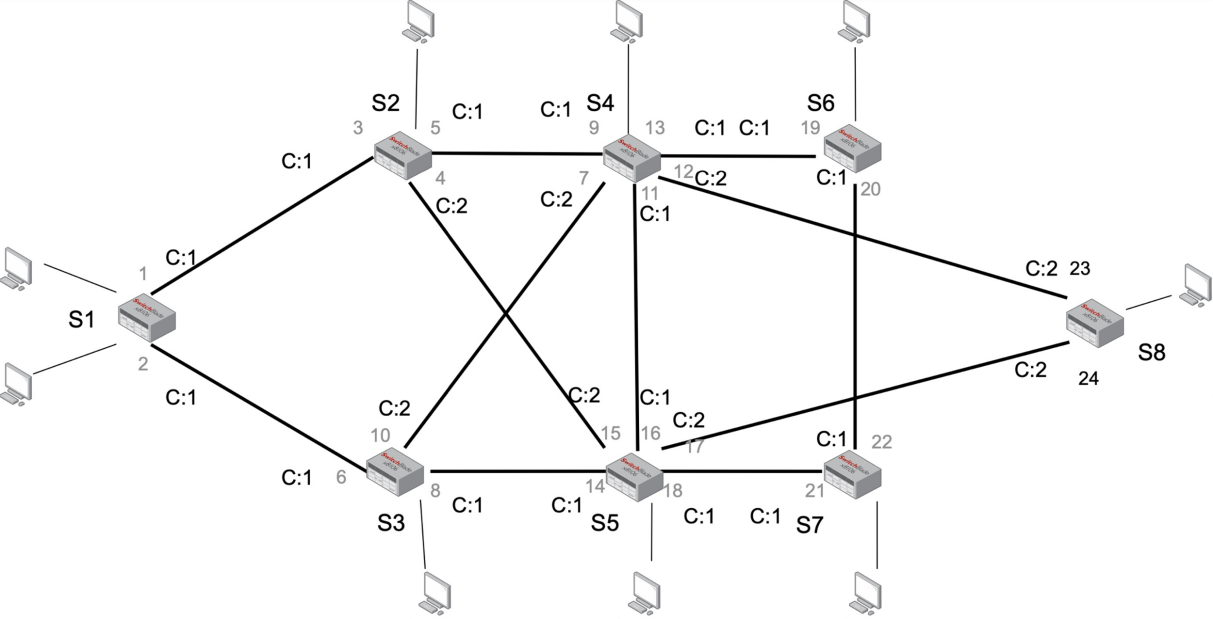
其中为交换机，对应端口开销，小黑体字对应端口号

图3‑2 8交换机网络拓扑结构示例。

### 根桥选举

算法模块首先进行根桥选举。根桥选举是在所有网络中交换机设备选举最优桥，即STP树最顶端的交换机。根桥的选举根据桥配置文件及BPDU中的桥ID（BID）字段确定，桥ID字段通过优先级与交换机MAC地址进行确定。根桥选举结束之后，为了模拟整个网络拓扑端口的BPDU交互更新，需要对所有端口进行一次数据更新以保持一致性：将所有端口的根桥ID置为选举根桥的桥ID（BID）。

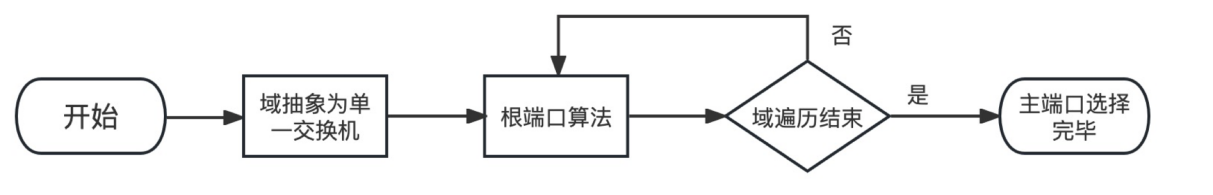
### 根端口选举

根端口的选举在所有非根桥的设备中选取唯一到根桥最近的端口。其衡量标准根据端口的累积路径开销进行决定。根端口其实质为非根桥接收上级配置文件的端口。为了模拟802.1D中STP算法中根端口的不断迭代过程，我们通过Dijkstra算法计算最短路径[16]，算法以端口路径开销为计算开销。公式3-2表示累计路径开销的计算方法，计算方法为端口到根桥所有开销之和的最小值：

|  |  |  |
| --- | --- | --- |
|  |  | (3-2) |

式中：RPC：累计路径开销，——端口到根桥的所有路径集合，——P中第i条路径，——路径中端口的端口开销。

通常情况下，同一交换机不同端口的RPC不相同，如果相同则比较3-1中优先级，选择发送者BID即公式3-1中SBID（对端）最小的端口作为根端口。图3-3表示根端口的选举过程，分别对每一个非根桥进行计算，对每一个交换机的每一个端口进行计算，最小的开销选举为根端口，且每个交换机只能选举一个端口为根端口。

图3‑3根端口选举流程图

### 指定端口选举

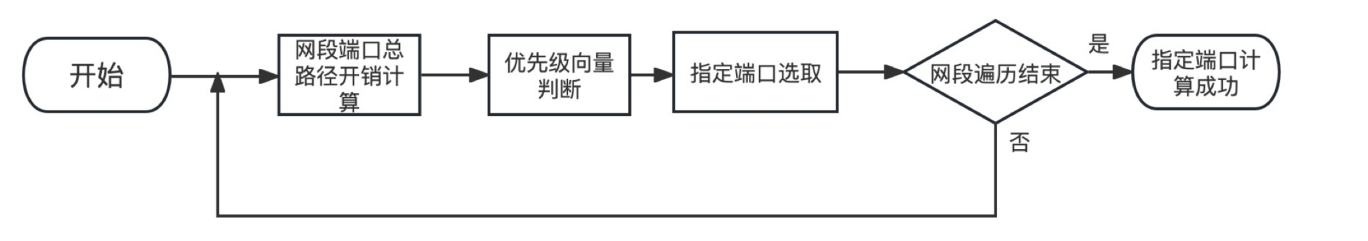
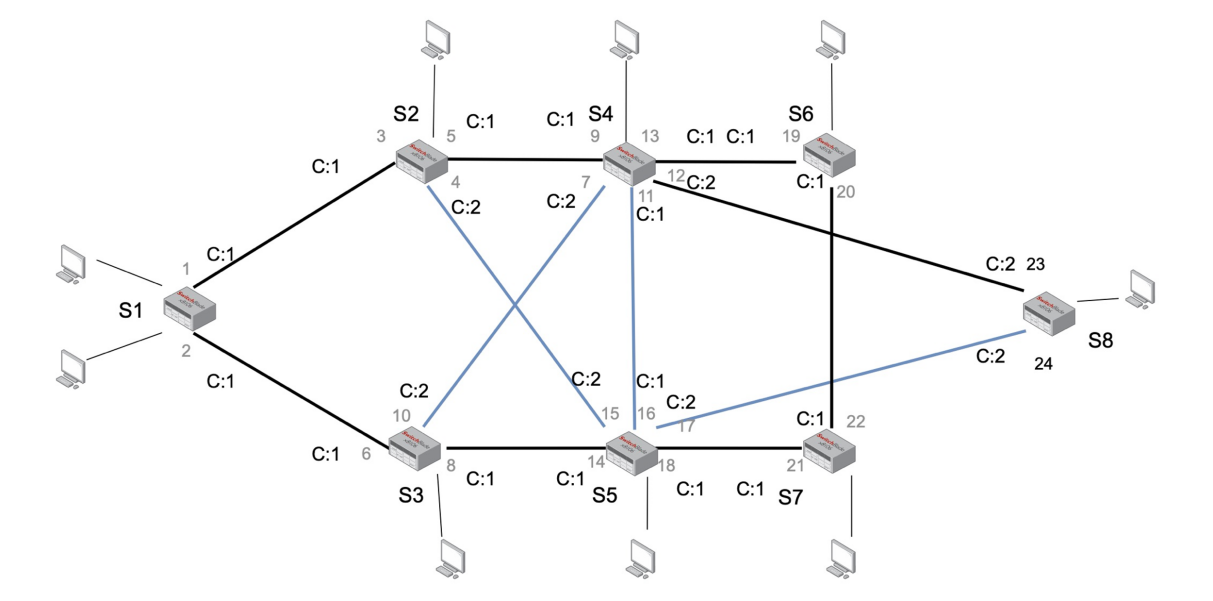
指定端口选举为每一个链路中针对两个端口之间选举出一个指定端口。指定端口即离下游设备最近的端口，为本链路中唯一可以接受下游设备数据的端口。为了模拟802.1D中STP算法中指定端口的迭代过程，我们需要重新用Dijkstra算法计算路径（可经过所属非根桥进行转发），用公式3-2重新计算端口的RPC。根据公式3-1中优先级向量优先级从累计路径开销，链路对端桥ID（SBID），发送方端口ID（SPID）依次进行判断，优先级不断降低。图3-4表示了指定端口选举流程，分别对网络中所有网段中的两个交换机计算根路径开销，比较更新后的优先级向量，其中优先级高的端口选举为指定端口。

图3‑4指定端口选举流程图。

### 堵塞端口选举

网络收敛后，选举出的根桥作为该生成树对应根，3种端口角色中根端口和指定端口可以向上级或下级转发数据为Forwarding态，其余的端口此时只能够监听不能向外转发数据，不能转发数据的端口为阻塞端口为Blocking态，不转发用户流量作为堵塞端口。

将图3-2所示的网络拓扑结构输入图3-1所示的处理端算法中，可以得到各端口的端口角色，角色分为根端口，指定端口，堵塞端口。图3-5表示了图3-2经过STP模拟算法的其中结果，其中S1选举为根桥，3，6，9，14，19，23为根端口，1，2，4，5，7，8，11，12，13，17，18，20为指定端口，剩余10，15，16，22，24端口不能转发用户流量为堵塞端口。将堵塞端口所在的链路堵塞，图3-5中黑色粗线为生成的转发路径，蓝色粗线为堵塞端口所在的链路，蓝色粗线表示被堵塞。

图3‑5 8交换机网络拓扑经处理后拓扑结果

### MAC表生成模块

根据STP算法流程，对*G*中所有端口计算出端口角色及端口状态，端口状态可分为Forwarding态及Blocking态，对于Blocking态的端口对应链路阻塞，无法转发数据包，因此MAC表生成算法对Forwarding的端口对应链路进行计算，使用Dijkstra算法计算交换机直连MAC地址对应最短路径，根据最短路径获得交换机的出接口作，目的MAC地址和出接口作为对应交换机中的一条路由表项，遍历所有可达的目的端口可以得到所有路有表项。表3-1表示路由表项的设计，表中值为图3-5结果。一条表项为目的MAC地址；VLAN-ID；出接口。其中目的MAC地址为终端网络对应MAC地址，VLAN-ID为为了兼容MSTP算法，出接口为到目的终端网络路径中本交换机转发接口。

表3-1 MAC地址表

| MAC地址 | VLAN—ID | 出接口 |
| --- | --- | --- |
| Mac1 | 0 | Pc1 |
| Mac2 | 0 | Pc2 |
| Mac3 | 0 | 1 |
| Mac4 | 0 | 2 |
| Mac5 | 0 | 1 |
| Mac6 | 0 | 2 |
| Mac7 | 0 | 1 |
| Mac8 | 0 | 2 |
| Mac9 | 0 | 1 |

图3-2中的网络拓扑经过STP算法处理后，根据MAC表算法对每一交换机生成对应的MAC表。图3-2中交换机S1 对应MAC表项，一共9条，分别代表了9种不同的转发路径。其中STP协议不涉及VLAN概念，为0（兼容之后MSTP算法）

## 本章小结

本章模拟IEEE 802.1D STP生成算法，对已知网络拓扑完成STP算法，构建二层协议仿真工具处理端的算法之一，模拟实现STP算法对网络拓扑结构计算出多种端口对应角色：根端口；指定端口；堵塞端口，并通过MAC表生成算法对各交换机生成对应MAC表项。最后通过输出模块对结果进行可视化输出。

# 多生成树协议（MSTP）模块及MAC表生成模块

## 多生成树协议（MSTP）简介

多生成树协议（MSTP）是基于RSTP（IEEE 802.1W）和VLAN（IEEE 802.1Q）的生成树协议之一[17]。MSTP协议兼容了STP和RSTP协议，既可以快速收敛，又能使不同的VLAN流量沿着各自的路径转发，使部分冗余链路分担负载。MSTP协议可以为数据帧提供稳定可靠的优化后路径，针对不同的域及实例生成不同的生成树，适应复杂的网络环境，满足不同群体的隔离及优化的需求。同时阻塞的冗余链路也会承载不同的VLAN流量, 优化网络资源，防止二层网络攻击[18]。

MSTP协议类似于生成树协议可以在有环的链路中确保只有一条不成环的活动路径，基于此避免广播风暴和数据包传输冲突。MSTP与STP协议相比，MSTP协议可以根据网络的需求和拓扑结构，将整个单一的网络结构划分为多个区域和实例，并在每个域内的多个实例中独立生成生成树，满足不同的网络需求，为不同的部分或者用户群组提供定制的生成树配置。同时MSTP协议支持快速收敛和故障恢复，在网络拓扑发生变化时重新计算生成树，并通过端口状态的转换来避免数据中断。

MSTP协议通过公共生成树（Common Spanning Tree, CST）连接所有多生成树区域（Multiple Spanning Tree, MST）内单独生成的多组独立的生成树实例（Multiple Spanning Tree Instance, MSTI）。其中域内MSTI对应的小生成树d的生成树方法可以兼容STP或者RSTP。在某一个区域内，多个VLAN可以映射到一个实例上，并以实例为基础实现负载的均等。通过多个MSTI可以帮助实现负载在不同生成树之间的均衡分布[17]。每个区域中，都有一个基础实例用实例0表示，即。公共与内部生成树（CIST）由连接所有区域的公共生成树CST和区域内部的生成树组成。其他的对应不同的生成树实例。如图4-1，其中包含了3个域以及CST与MST。每一个域内的多个交换机通过主端口和多个指定端口连接到CST，区域之间相互隔离，域外的拓

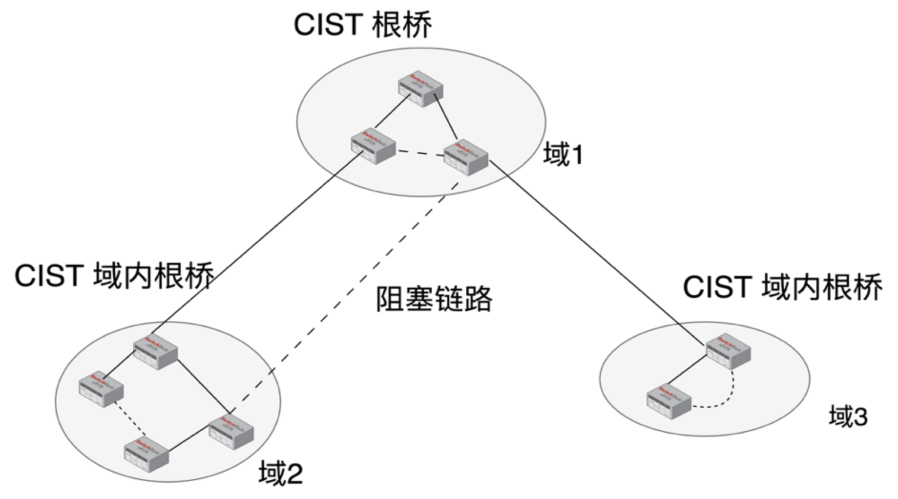
扑结构和域内的拓扑结构相互隔离。

图4‑1 MSTP网络拓扑结构

但是MSTP协议的配置比较繁琐，除了需要对每个交换机及端口配置对应的的信息，还需要对每个端口进行VLAN的配置，且其配置必须映射到具体实例，配置表对同一区域内网桥需要一致，否则会发生严重故障。目前IEEE 802.1S标准定义MSTP协议中，使用了配置标识符及配置表，修订号，摘要(MD5)来进行映射完整性检查[19]。 本文提出的MSTP仿真模块设计根据表2-5来进行VLAN与域中多实例MSTI关键字段进行映射，减轻部分工作量。

## MSTP仿真模块

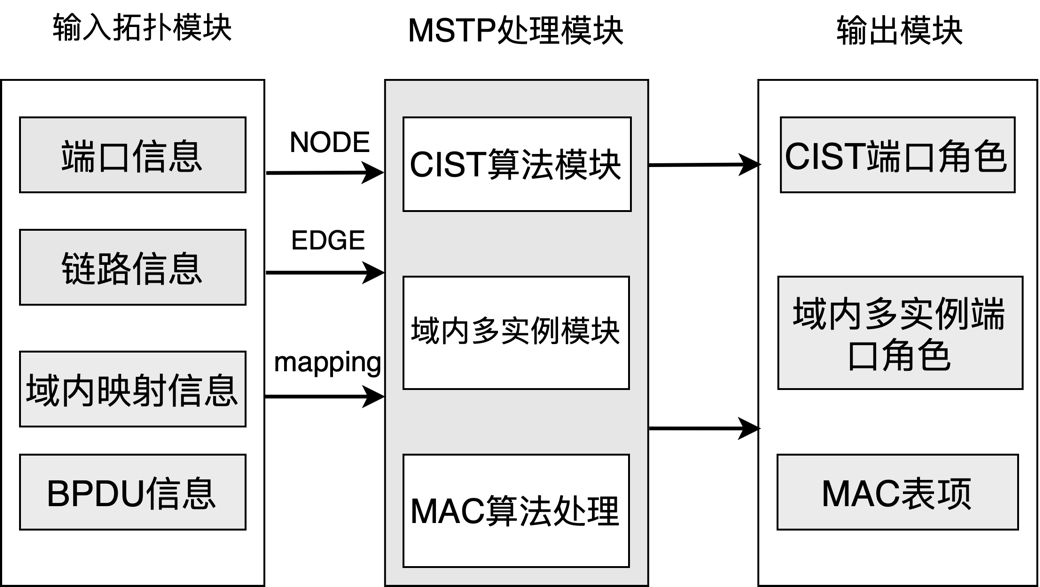
MSTP仿真模块实现了IEEE 802.1S的多生成树选拔流程规范。图4-2为为MSTP生成模块具体流程图。图4-2输入拓扑模块包含了初始网络拓扑中包含的4类信息，分别为端口信息：存储交换机及交换机端口中的具体信息，链路信息：存储网络拓扑中具体链路信息，域内映射信息：存储域内MSTI与VLAN之间的映射信息，BPDU信息：更新端口配置文件的具体信息，输入模块通过表2-2，表2-3，表2-4，表2-5中的信息生成网络拓扑图,存储网络拓扑中的所有节点信息，存储链路信息。MSTP处理模块包含3个模块：CIST算法模块对提供网络拓扑计算整体端口角色（不考虑VLAN），域内多实例模块针对域内实例VLAN映射计算实例对应端口角色，对中节点进行端口角色更新，对链路是否阻塞进行更新；MAC算法处理生成交换机对应的MAC表项。最后经输出模块进行端口角色，MSTI对应角色及MAC表对应表项进行输出。

图4‑2 MSTP仿真模块流程结构

图4-3中网络拓扑为9交换机节点，3个域组成的的网络拓扑结构，经过2.3节网络拓扑生成模块，构建网络拓扑图 , 其中V是所有端口的集合，E为网络链路的集合。图4-3网络拓扑中V包含46个交换机端口节点，9个交换机节点，8个域边缘节点及9个PC节点；网络拓扑中E包含48条链路。

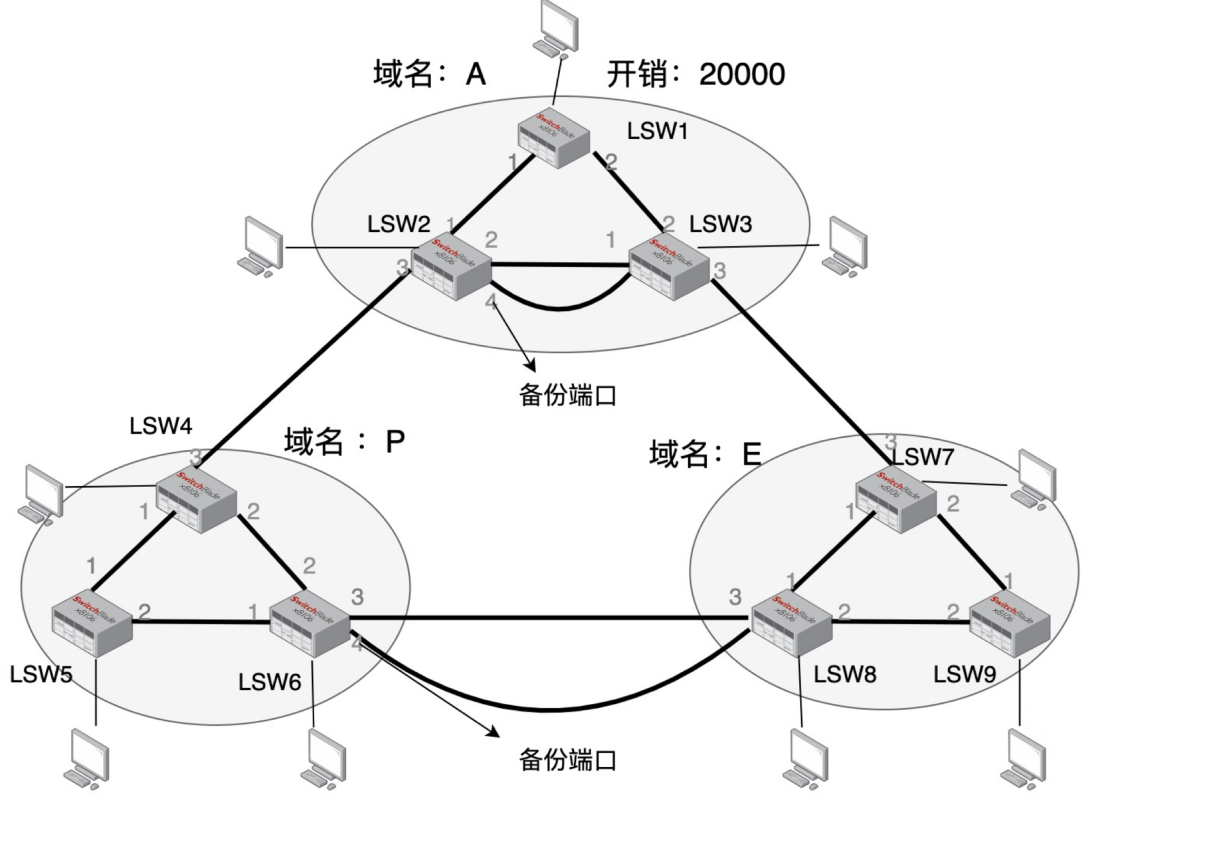
其中为交换机，对应端口开销，灰体小字对应端口号

图4‑3 MSTP协议仿真网络拓扑示例

表4-1表示了图4-3中域内多实例的映射关系。表内的一项表示在域内实例对应的转发VLAN及此实例对应的根桥。表4-1中每个域含两个实例，每个实例的根桥及转发VLAN路径不同。

表4-1 MAC地址表

|  |  |  |  |
| --- | --- | --- | --- |
| 域名 | 实例ID | VLAN ID | 实例根桥 |
| A | 1 | 2-10 | LSW1 |
| A | 2 | 11-20 | LSW2 |
| P | 1 | 2-10 | LSW4 |
| P | 2 | 11-20 | LSW5 |
| E | 1 | 2-10 | LSW7 |
| E | 2 | 11-20 | LSW9 |

其中处理模块中CIST算法模块模拟MSTP协议中CIST树（MSTI0 不涉及VLAN的属性）的生成过程, IEEE中 MSTP协议主要采用了动态BPDU更新及发现节点，但是仿真模块通过输入即可知具体的网络拓扑结构及邻居节点，因此可以通过整体的比较来确定根桥及端口角色，收敛会更加迅速。域内多实例模块根据表2-5生成对应域内的MSTI及VLAN映射，根据映射绘制单一域拓扑结构，在新的拓扑结构中进行 内部端口角色选取。执行完CIST算法模块及域内多实例模块后，MAC算法处理分两步生成所有的MAC表项：对CIST树生成MAC表项，对不同域内的生成MAC表项。输出模块中采取PYQT5和QTdesigner进行界面设计。

算法模块中定义了端口配置文件信息，本文中定义为一个七元组向量：

(4-1)

式中：——根桥ID，E——外部累计路径开销，——域根交换机BID，——指定桥，本交换机上游交换机，——指定桥对应端口为指定端口，——接收配置更新文件的端口。

通过元组向量来判断配置文件的优先级，进行端口及交换机配置文件的更新，优先级从左往右依次下降。

### CIST算法模块

CIST算法处理模块主要对输入端传递的网络拓扑生成了公共和内部生成树（Common and Internal Spanning Tree , CIST），连接内部所有的交换机的生成树，由内部生成树（Internal Spanning Tree , IST）及公共生成树（Common Spanning Tree , CST）组成。每个网络拓扑中只有一个CST，每个MST域内IST为CIST中对应不同域的一个片段。

算法模块会分别对域生成IST，及总的CST来生成CIST。其中端口角色分别对应：

根端口，指定端口，边缘端口，替代端口，备份端口，主端口，域边缘端口共7种端口角色。桥对应总根桥，域内根桥，非根桥设备三种角色。

#### 根桥选举

首先在网络范围内的所有区域中选取一个优先级最高的交换机作为CIST树根。根桥的选举依据配置文件中各交换机的桥ID（BID）字段：BID由交换机的优先级与对应交换机的MAC地址进行确定。其次CIST总根桥选举完毕后，需要单独对每个域选出域内最优根桥，即域内根桥作为IST的树根。根桥选举结束后为了保证配置文件的一致性需要对各交换机内配置文件中总根桥ID，域内根桥ID中字段进行更新：维护所有字段为总根桥BID，为对应域内的根桥BID。

图4-3中：LSW1选举为总根桥，在HCNA，HCNP，HCIE中域根分别为LSW1，LSW4，LSW7。

#### 备份端口选举：

备份端口是堵塞连接同一交换机的端口。同一交换机的两个不同端口同时连接同一设备，由于同一设备中链路开销为0，形成环路，交换机会堵塞一个端口，这个端口为备份端口。图4-3表示备份端口的选举流程，对于网络拓扑结构中链路进行遍历：如果同一交换机的两个不同端口同时连接同一设备，则将其中链路开销大的端口阻塞，如果链路开销一致则选择端口PID优先级低为备份端口（backup），当链路故障时会将备份端口重新启用。图4-3中：LSW2-4，LSW3-4，LSW6-4，LSW8-4为备份端口。

图4‑4 备份端口计算流程

#### 根端口，指定端口选举：

IEEE中定义根端口为接收到最优配置消息的端口，根端口为负责向树根方向转发数据的端口，非根桥到根桥距离开销最小的端口为本交换机的根端口。指定端口是向下游交换机转发数据的端口，连接下级交换机的端口为指定端口。根端口与指定端口的选举模拟MSTP算法中不同域内根端口计算中的不断迭代过程，本文通过Dijkstra算法计算最短路径，算法以各端口路径开销的参考值，且外部路径开销为0。

|  |  |  |
| --- | --- | --- |
|  |  | (4-2) |

式中：EPRC：外部路径开销。

向量4-1中内部累计路径开销（IRPC）的计算方法为同属于一个域中端口到根桥所有开销之和的最小值，即：

|  |  |  |
| --- | --- | --- |
|  |  | (4-3) |

式中：IRPC：内部累计路径开销，——端口到域内根桥的所有路径集合，——P中第i条路径，——路径中端口的端口开销，——路径中所有端口属于的域。

通常情况下，同一域内的同一交换机不同端口IRPC不相同，选择发送者BID即公示4-1中（对端）最小的端口作为根端口。即依据公式4-1中优先级向量进行选举比较。

对于指定端口计算，我们需要重新用Dijkstra算法计算路径（可经过所属非根桥进行转发及同交换机中端口之间路径开销为0），用公式4-3重新计算端口的IRPC。根据公式4-1中优先级向量优先级从内部累计路径开销，指定桥 ID（SBID），指定桥对应端口ID（SPID），接收端口ID（）依次进行判断，优先级不断降低。

图4-3中，对所有的非根桥设备进行选举，其中端口LSW2-1，LSW3-2，LSW5-1，LSW6-2，LSW8-1，LSW9-1为各个非根桥的根端口。其中LSW1-1，LSW1-2，LSW2-4，LSW4-1，LSW4-2，LSW5-2，LSW7-1，LSW7-2，LSW8-2为各桥的指定端口。

#### 替代端口选举

网络收敛后，对于每一个域中，域内根桥作为域内生成树的根，只有根端口和指定端口可以转发数据，其余的端口为替代端口，不转发用户流量，提供了从指定桥到根桥的一条备份路径。

图4-3中，每一个域中（除去边缘端口）剩下的端口为堵塞端口，其中LSW3-1，LSW6-1，LSW9-2为对应的替代端口。

CIST模块通过根桥、备份端口、根端口、指定端口、替代端口选举生成每个域的IST。

#### 域抽象及域边缘端口选举

CST生成需要将整个域抽象为单一交换机同时不破坏域外的链路。通过搜索所有链路并判断链路是否为连接多个域之间链路，标记域边缘端口。抽象整个域为单一的仅具有域边缘端口的交换机。剩余网络拓扑结构组成新网络拓扑结构

#### 主端口，指定端口及替代端口选举

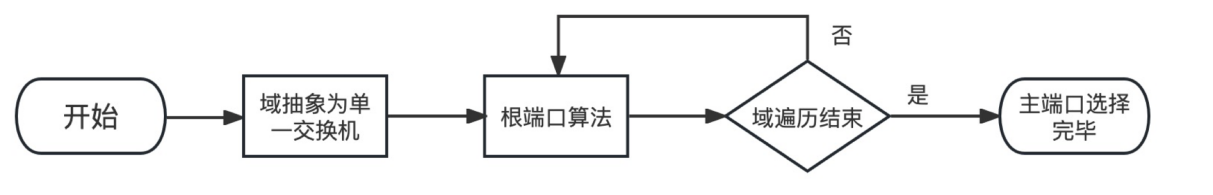
主端口（master端口）为MST域和总根相连最短路径上的端口，相当于IST的根端口。图4-5表示主端口的计算流程，其中根端口的选举方法同3）中根端口选举策略。指定端口和替代端口同3）中指定端口选举策略，4）中替代端口策略。

图4‑5 主端口计算流程

图4-3中对应CST端口角色为： LSW2-3，LSW3-3，LSW6-3为指定端口，LSW4-3，LSW7-3为主端口（master），LSW6-4为替代端口。

#### CIST合成：

CIST为连接整个MSTP网络中所有交换机的单生成树。交换机网络结构中所有MST域和IST，CST一起构成一棵完整的生成树即CIST。

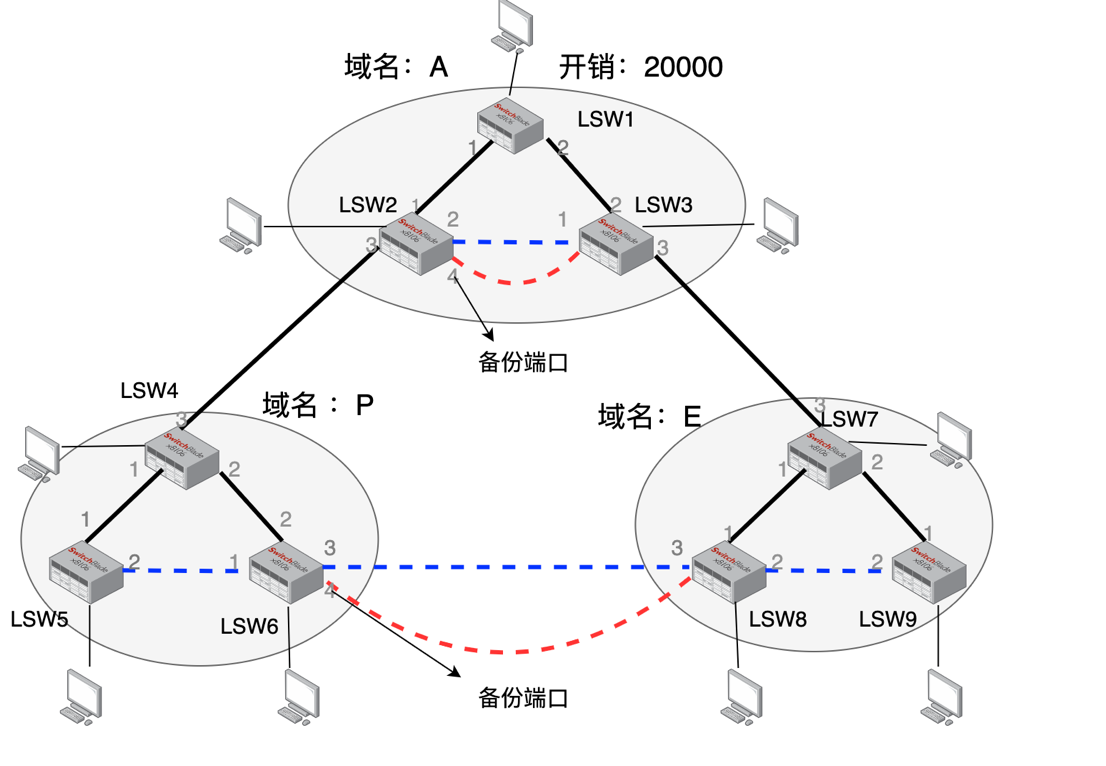
如图4-4中为CIST示意图，其中黑色粗线为流量转发唯一路径，构成了一棵总生成树，红色虚线为对应备份端口所在路径，蓝色虚线为替代端口所在的堵塞线路。

图4‑6 MSTP协议仿真网络拓扑结果示例（黑色粗线为转发路径）

### 域内多实例模块

MST域内，根据不同VLAN和生成树实例的映射关系，针对不同的VLAN生成不同的生成树实例。其中域内的多生成树实例之间独立成树，互不干扰。MSTI计算方法和RSTP基本相同。

#### 配置文件及拓扑链路配置

首先通过表2-5进行配置文件配置，对应实例中根桥设备的优先级置为0，对链路对应VLAN进行筛选。通过重置优先级为0，使得配置根桥设备优先级最高，根据桥ID（BID）选拔出对应域内多实例的根桥。同时为了仿真IEEE MSTP算法，修改域内的配置文件域内根桥ID为对应根桥ID，保证一致性。 通过表2-5及端口对应通过VLAN情况对域内的VLAN进行筛选，对于不能通过MSTI中指定VLAN及通过部分VLAN情况的链路进行链路删除。

#### 端口角色选拔

域内MSTI对应的端口角色分为根端口，指定端口。IEEE中定义根端口为接收到最优配置消息的端口。为了仿真根端口，通过Dijkstra算法计算内部路径开销：

|  |  |  |
| --- | --- | --- |
|  |  | （4-4） |

式中：：内部累计路径开销，——端口到域内根桥的所有路径集合，——P中第i条路径，——路径中端口的端口开销，——属于的域。

通过4-1公式中优先级向量对每个交换机端口进行比较，模拟收到最优配置信息的端口，确定根端口。

指定端口是向下游交换机转发数据的端口，连接下级交换机的端口为指定端口。为了模拟对应的选拔过程为每个端口维护两个优先级比较向量。分两步对公式4-5中的向量分别进行更新操作和比较。

|  |  |
| --- | --- |
|  | (4-5） |

式中：：存储着未进行更新的向量，下标r为未更新的指标。：根据生成的标准配置向量，其中 替换为：根端口的路径开销加上根端口链路的路径开销；SPID替换为：本交换机ID；SPID替换为：自身端口ID。

对公式4-5中的两类标准配置向量生成后进行比较。若端口配置消息更优，此端口为替代端口即阻塞端口，此端口只用来接收具体的配置消息；若更优，交换机端口为指定端口。

对图4-3进行域内实例配置得到图4-3其中，对于实例1端口角色为：LSW1-1，LSW1-2，LSW2-4，为指定端口，LSW2-1，LSW3-2为根端口，LSW3-1为阻塞端口，LSW2-2为备份端口。对于实例2端口角色为：LSW1-2，LSW2-1，LSW2-4为指定端口，LSW1-1，LSW3-1为根端口，LSW3-2为阻塞端口，LSW2-2为备份端口。

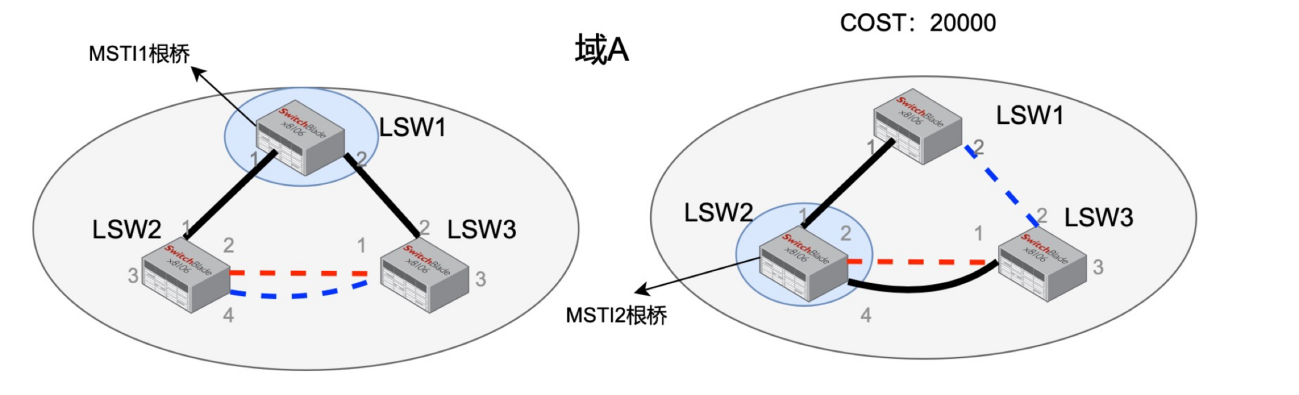
图4-5表示域A对应的多实例结果。其中黑色实现为对应VLAN转发路径，红色虚线为备份端口所在的路径，蓝色虚线为堵塞端口所在路径，链路堵塞。

图4‑5 域A内2实例结果（黑色粗线为转发路径）

## MAC算法处理模块

MAC表算法生成模块分两步生成所有的MAC表项。首先对CIST生成MAC表项。如图4-3对网络拓扑经过CIST生成模块处理后，MAC表算法对每一交换机生成对应的MAC表项。其中MAC表项由：MAC地址，VLAN-ID，出接口组成。MSTP算法中CIST生成为为公共和内部生成树。

根据CIST算法模块流程，可对中所有端口得到对应的端口角色，对备份端口与替代端口所在的链路进行堵塞，无法转发数据包，因此MAC表生成算法剩余端口所在的链路进行参考，使用Dijkstra算法计算交换机直连MAC地址最短路径，获得出接口。其中VLAN-ID为0。

其次对每一个域内的多实例分别处理。根据域内多实例模块流程对域内单独拓扑生成角色后分实例对替代及备份端口所在链路阻塞，使用Dijkstra算法计算交换机直连MAC地址最短路径，获得出接口。此时对应VLAN-ID为对应配置文件中MSTI域VLAN映射中VLAN范围。

对图4-3及4-4域中交换机LSW1进行MAC生成，对应的MAC表项如表4-2。

表4-2 LSW1对应MAC地址表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MAC地址 | VLAN-ID | 出接口 | MAC地址 | VLAN-ID | 出接口 |
| Mac1 | 0 | pC1 | Mac9 | 0 | LSW1-2 |
| Mac2 | 0 | LSW1-1 | Mac1 | 2-10 | PC1 |
| Mac3 | 0 | LSW1-2 | Mac2 | 2-10 | LSW1-1 |
| Mac4 | 0 | LSW1-1 | Mac3 | 2-10 | LSW1-2 |
| Mac5 | 0 | LSW1-1 | Mac1 | 11-20 | PC1 |
| Mac6 | 0 | LSW1-1 | Mac2 | 11-20 | LSW1-1 |
| Mac7 | 0 | LSW1-2 | Mac3 | 11-20 | LSW1-1 |
| Mac8 | 0 | LSW1-2 |  |  |  |

## 本章小结

本章模拟了IEEE 802.1S MSTP协议算法，根据自定义配置文件，实现了MSTP算法，分CIST，CST两步模拟MSTP算法计算过程，构建工具的处理端算法。二层网络协议仿真工具中处理端算法分别计算CIST生成树及域内多实例对应生成树。算法计算出端口角色，并通过MAC表生成算法得到各交换机生成对应MAC表项。最后通过输出模块对结果进行可视化输出。

# 模型评估实验与交互界面设计

## 模型正确性验证

采用华为eNSP工具[20]与本文仿真模型输入相同的网络拓扑结构进行模型的正确性验证。本模块对模型的STP模块和eNSP对应STP模块进行了2到5个交换机，5到37个交换机端口的5组不同拓扑结构的平行对照实验，对模型的STP协议、MAC表进行正确性验证。本模块对模型的MSTP模块和eNSP中MSTP模块进行了4、6、7、8、9、16、50个端口的5组不同拓扑结构的平行对照实验，对模型MSTP协议、MAC表生成协议进行了正确性验证。一共进行10组对照实验且结果均与华为eNSP一致。

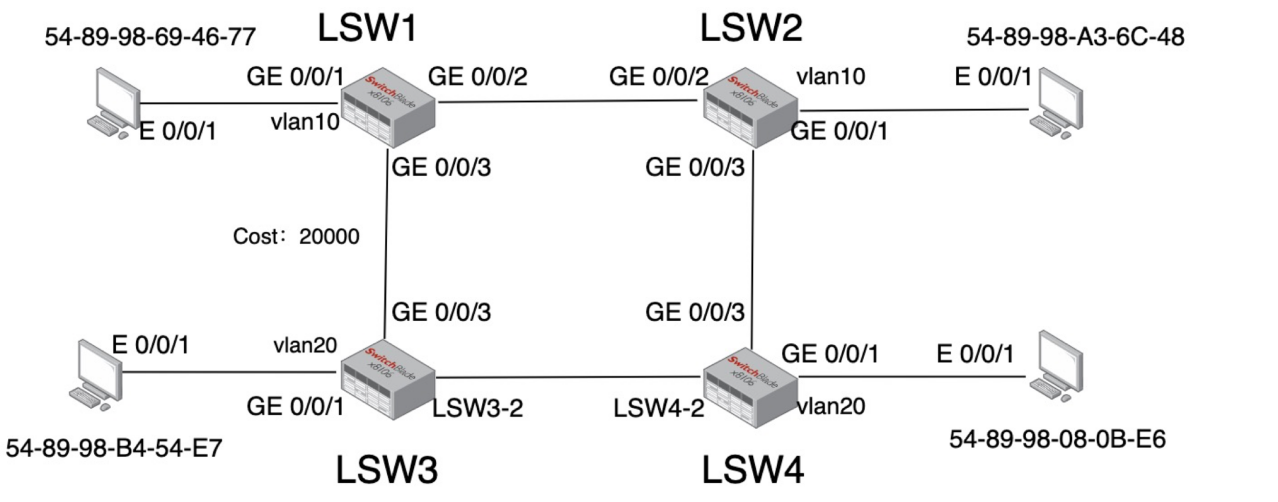
首先对华为eNSP工具输入测试网络拓扑结构，对MSTP协议测试输入拓扑，分别进行STP协议及MSTP协议进行测试，测试完之后，通过对每个PC定义IP地址后通过ping命令使交换机学习MAC表项（MSTP协议需分VLAN操作），最后输出查看对应端口角色，MAC地址。其次对本文提出的模型输入相同的玩过拓扑结构并进行STP、MSTP、MAC表处理获得对应的结果；最终对比eNSP工具结果与本文模型对应的结果进行正确性验证。图5-1表示对4个交换机的拓扑结构进行MSTP测试，4个交换机分为同一域，端口vlan分为vlan10与vlan20。

图5‑1 eNSP 4交换机的MSTP协议测试（eNSP建模图)

对图5-1所示的网络拓扑结构使用MSTP算法进行测试，最后PC之间相互ping通查看MAC路由表项，判断结果。将获得结果与模型仿真工具结果对照判断准确性。

对图5-1所示网络拓扑结构进行结果对比，针对vlan10，vlan20的端口角色分别端口角色与对应的MAC表项。对照结果中PC端口连接端口仿真工具为PC端口，而eNSP工具为指定端口，其他表示均一致，PC端口角色此时可以承载指定端口向下转发的功能，可认为此时工具内PC端口角色等同于指定端口角色。如表5-1为图5-1执行MSTP协议后所示为对应的端口角色对照表。

表5-1 4交换机对应的端口角色表与MSTI映射

|  |  |  |  |
| --- | --- | --- | --- |
| MSTI\_ID | PORT | eNSP：ROLE | 仿真：ROLE |
| 1 | LSW1-G0/0/1 | DESI | PC |
| 1 | LSW1-G0/0/2 | DESI | DESI |
| 1 | LSW1-G0/0/3 | DESI | DESI |
| 1 | LSW2-G0/0/1 | DESI | PC |
| 1 | LSW2-G/0/0/2 | ROOT | ROOT |
| 1 | LSW2-G/0/0/3 | DESI | DESI |
| 1 | LSW3-G/0/0/2 | DESI | DESI |
| 1 | LSW3-G/0/0/3 | ROOT | ROOT |
| 1 | LSW4-G/0/0/2 | ROOT | ROOT |
| 1 | LSW4-G/0/0/3 | ALTE | ALTE |
| 2 | LSW1-G/0/0/2 | ALTE | ALTE |
| 2 | LSW1-G/0/0/3 | ROOT | ROOT |
| 2 | LSW2-G/0/0/2 | DESI | DESI |
| 2 | LSW2-G/0/0/3 | ROOT | ROOT |
| 2 | LSW3-G/0/0/1 | DESI | PC |
| 2 | LSW3-G/0/0/2 | ROOT | ROOT |
| 2 | LSW3-G/0/0/3 | DESI | DESI |
| 2 | LSW4-G/0/0/1 | DESI | PC |
| 2 | LSW4-G/0/0/2 | DESI | DESI |
| 2 | LSW4-G/0/0/3 | DESI | DESI |

对比结果，表5-1中对图5-1中关于eNSP工具与仿真工具对于MSTP协议模拟后的端口角色结果一致。

对图5-1执行ping操作学习MAC表项，表5-2表示eNSP工具对应部分MAC表：

表5-2 eNSP工具中4交换机中LSW1对应的MAC表

|  |  |  |
| --- | --- | --- |
| MAC\_address | VLAN/SI | Port |
| 5489-9869-4677 | 10 | G/0/0/1 |
| 5489-98a3-6c48 | 10 | G/0/0/2 |
| 5489-98b4-54e7 | 20 | G/0/0/3 |
| 5489-9808-0be6 | 20 | G/0/0/3 |

表5-3表示本文模型对同样的网络拓扑处理生成的LSW1的MAC表项：

表5-3二层协议仿真工具中4交换机中LSW1对应的MAC表

|  |  |  |
| --- | --- | --- |
| MAC\_address | VLAN/SI | Port |
| MAC1(PC1) | 10 | G/0/0/1 |
| MAC3(PC3) | 10 | G/0/0/2 |
| MAC2(PC2) | 20 | G/0/0/3 |
| MAC4(PC4) | 20 | G/0/0/3 |

本文提出的二层网络协议工具与华为eNSP工具进行对照，所有的端口角色正确，MAC表生成正确，二层网络协议仿真工具内置算法结果与华为eNSP工具一致。

通过对STP模块，MSTP模块多组平行测试得出，本文构建的二层协议仿真工具对于的网络系统具有很高的算法准确度，可以运用在一些真实的网络结构中进行算法测试。

## 模型效率验证

模型效率验证通过分析算法时间复杂度，绘制并分析不同的规模的网络拓扑执行STP模块、MSTP模块的算法执行时间图像，并与eNSP工具执行STP与MSTP算法效率进行比较。

二层网络协议仿真工具中STP算法模块采取图3-3、图3-4流程，逐步计算根端口，指定端口，堵塞端口。图3-3中根端口的计算存在两层循环，遍历所有网络拓扑结果中的交换机，在交换机中遍历交换机所有端口，执行Dijkstra算法算法。指定端口对每个网段中两个端口节点执行Dijkstra算法，公式5-1表示总的STP模块的算法复杂度：

|  |  |  |
| --- | --- | --- |
|  |  | (5-1) |

式中：—— 交换机集合；—— 交换机端口集合；—— 交换机；—— 端口；—— 链路；—— 链路集合；m—— 交换机端口节点的数量；n—— 交换机节点数量。

效率验证模块对二层网络协议仿真工具中的STP算法执行多组对照测试，通过改变算法输入的网络拓扑结构大小，统计不同网络拓扑大小的计算时间，并生成对应的时间图像。效率验证中对STP模块共采取了节点（交换机端口数量）数量在5-50范围内10组不同的网络拓扑结构进行测试，图5-2表示10组不同的网络拓扑结构对应的算法处理时间。

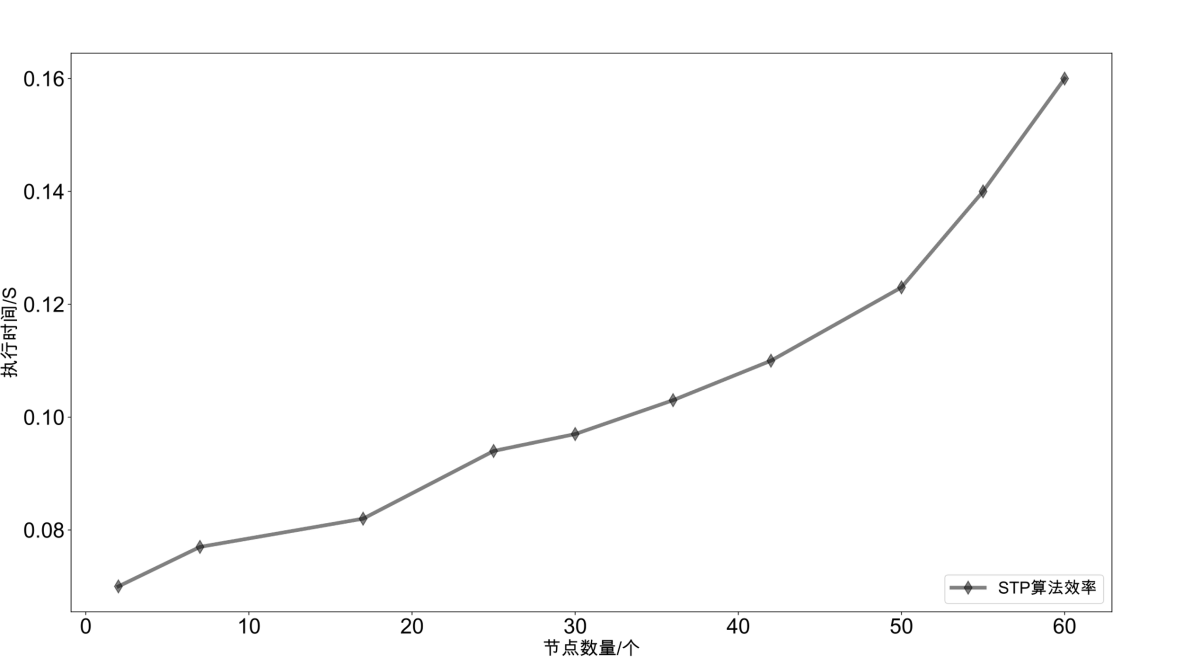
 图5‑2 STP算法处理时间随节点数量的图像

图5-2显示STP算法规模对于包含40个节点以内的网络拓扑，二层网络协议仿真工具可以在0.1s内计算完成且算法的执行时间随着输入网络拓扑规模的增加，算法所需的时间也逐步增加，由公式5-1和图5-2算法时间复杂度不为指数增加，计算时间可承受。

二层网络协议仿真工具中MSTP算法模块采取第四章的流程，分别计算CIST、域内的多MSTI。效率验证模块对二层网络协议仿真工具中的MSTP算法执行多组对照测试实验，改变MSTP算法输入的网络拓扑规模大小，统计对应网络规模的计算时间，并生成对应的时间图像。效率验证中对MSTP模块进行节点（交换机端口数量）数量在4-60范围内11组不同的网络拓扑结构进行测试，图5-3表示11组不同的网络拓扑结构对应的算法处理时间。

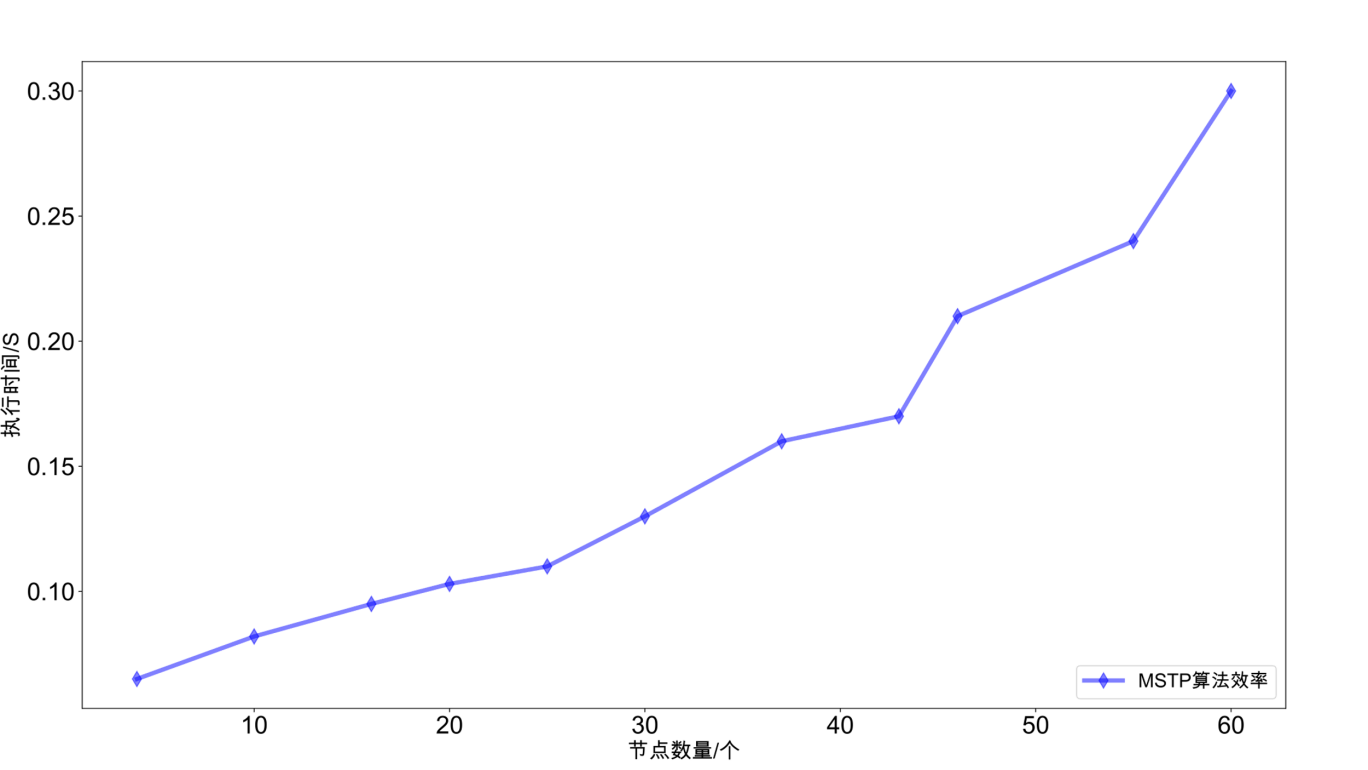
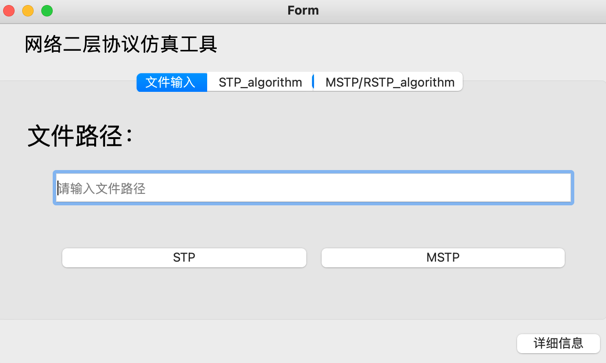
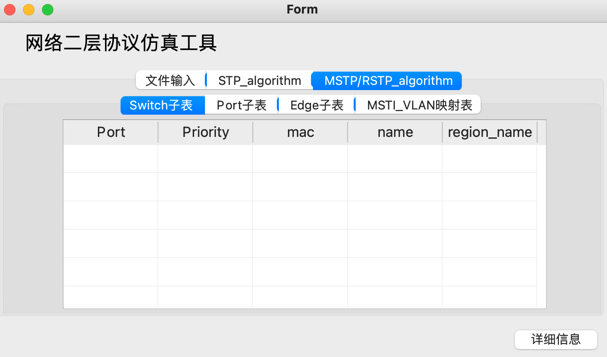
图5‑3 MSTP算法处理时间随节点数量的图像

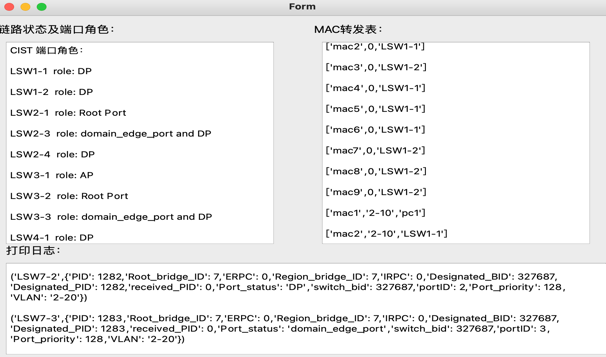
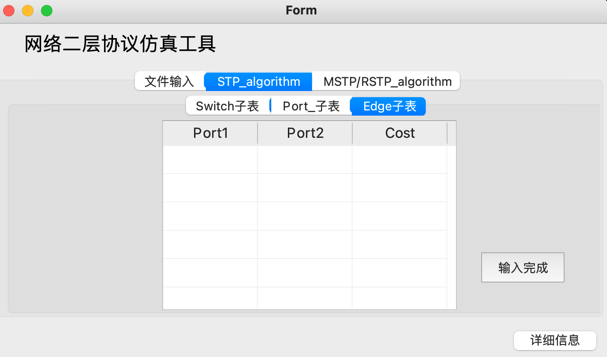
图5-3显示二层网络协议仿真工具对于包含60个节点的小型网络拓扑规模能够在1s内完成CIST，域内多实例处理，同时算法执行时间随着网络输入规模的增加，算法计算时间也增加，由图5-3MSTP算法时间复杂度不为指数增加，计算时间可以承受。

华为eNSP工具构建网络拓扑后执行STP、MSTP算法采取动态发现邻居节点且不断根据邻居节点维护最优的BPDU选拔生成树，所需时间往往是秒级。在协议中影响拓扑收敛的参数与网络直径、超时时间、定时器、链路聚合带宽有关，网络拓扑改变后，端口角色稳定需要较长时间，往往是十秒级。通过本文提出的二层网络协议仿真工具，通过输入整体的网络拓扑，算法对整体结构有一个整体把控，可以根据第3章、第4章流程以一个整体或者分割后的小整体执行STP、MSTP、MAC表算法，本文提出的二层网络协议仿真工具的执行速度更快。

## 交互界面设计

交互界面采用PyQt5与Qt Designer联合实现[21]，完成了二层网络协议仿真工具中多个仿真算法通用界面设计。PyQt5可以通过编程实现界面的设计、界面布局管理，业务逻辑实现等，Qt Designer可以快速设计界面，通过槽与槽函数的互调完成界面的业务逻辑实现。通过PyQt5h和Qt Designer联合编程实现界面及业务。

整个交互界面如图5-4，提供2种输入模式，对应为文件路径输入，表格数据输入，可以选择STP算法与MSTP算法，详细信息存储着二级界面。二级界面设计为输出界面：输出界面分为端口角色，MAC转发表，日志打印3个模块（如图5-4）。



(c)STP模式输入

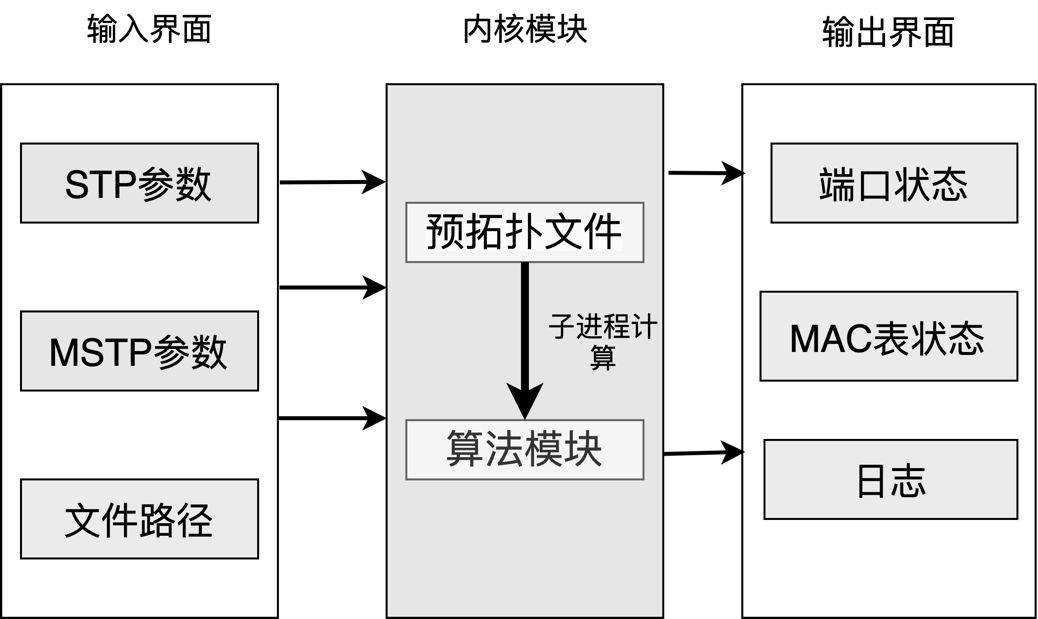
(d)输出结果页面

（b）MSTP模式输入

(a)文件路径输入

图 5-4 仿真工具多级界面设计

模型生成网络拓扑需要向图5-4输入正确参数（文件路径或者正确数据）后生成预拓扑文件。预拓扑文件中主要记录网络仿真拓扑生成的具体数据。图5-5显示整个工具的执行算法流程。内核模块子进程进行STP/MSTP算法运算，计算MAC表及端口角色，日志并写入文件。通过输出界面对端口状态，MAC表状态，日志进行输出打印，方便使用者进行网络协议结果的验证查看。

图5‑5界面与工具内核运行流程

华为eNSP工具需要手动建立网络拓扑结构并对建模的网络拓扑中的每一个交换机进行代码配置，对每一个交换机内的所有端口进行代码配置，结果的查看也需要对网络拓扑结构中的每一个交换机编程查看，且容易配置出错，难以检查。本文提出的二层网络协议仿真工具可以通过填表法完成网络拓扑的建立的同时完成每一个交换机及端口的配置，仅仅通过字段的填写替代了eNSP工具的代码配置，更加简单易读，最终的结果也可以直接图形化的输出，方便查看，对比eNSP工具的操作更加简单直观。

## 本章小结

本章主要介绍了二层协议仿真工具的正确性验证，工具效率验证，交互界面的构建算法。正确性验证通过华为eNSP工具与本文提出的二层协议仿真模型分别进行10组相同的网络拓扑下的对照实验，在STP、MSTP协议下对端口的角色，与MAC表进行验证，结果表明本文提出的二层网络协议仿真工具的端口角色结果及MAC表结果均与eNSP工具得到的结果一致；效率验证模块对比二层网络仿真工具的计算时间与eNSP工具计算的时间，本文提出的二层网络仿真工具针对同样的网络拓扑能够在更短的时间完成STP、MSTP、MAC表算法并输出；交互界面通过PyQt5与Qt Designer实现了界面与业务功能，实现了结果的可视化输出且通过填表法配置操作更加简便易读，方便查看。

# 结论与展望

## 结论

本文的主要任务是设计出二层网络协议仿真工具，用于仿真模拟不同网络拓扑结构，在不同生成树协议下的链路及端口状态、MAC表，检查当前网络的运行情况，并为可能出现的广播风暴等异常行为进行预警。本文对以往的文献及IEEE 802.1中STP、MSTP、MAC表协议流程总结提取，完成配置文件的解析及自定义配置文件的封装，代码实现适用于自定义配置文件的生成树，多生成树协议仿真算法并完成MAC表算法，设计可交互的可视化界面，并完成eNSP下的多组正确性对照及效率分析。

从实验的结果来看，整个模型的算法仿真正确性为100%，表明整个模型的算法正确性良好且适用于一般的规模的网络拓扑仿真分析，能够对网络拓扑进行生成树协议簇处理及MAC表处理，且具有良好的生成无环拓扑及VLAN处理的能力。且通过eNSP工具进行多组仿真验证，端口状态角色验证均一致。传统的仿真工具如eNSP需要手动输入代码来输入配置，本模型可直接输入对应的数字字段进行配置，更具有操作性与可读性。但是类似于eNSP协议仿真过程中的的抓包功能暂时未能完善。

从实际应用结果来看,本文构建了一个可交互的二层网络协议仿真工具，能够对网络拓扑结构迅速计算对应的生成树、多生成树和MAC表，同时提供了友好的可交互界面，在实际应用中具有一定的价值。

## 展望

互联网发展推动了网络拓扑规模的巨大化，但是同时网络故障层出不穷，本文构建的网络二层网络协议工具虽然能够对二层网络中生成树协议簇进行精确仿真计算且提供一个可交互的平台，但本文对于一些可深入研究的问题缺乏进一步研究。本文建立的模型只能对网络二层协议中生成树协议簇及MAC生成协议进行分析，可以添加实现PPP等其他协议的支持；可以进一步实现拓扑失效后重新收敛的功能，使得模型更加贴合IEEE 中真实的生成树协议簇。或者提高模型的隐私性，比如通过并行隐私计算[22]。对于模型的准确性，可以增加一些其他的指标，比如收敛速度，更好的评估模型。同时可以对MSTP算法进行改进，优化参数配置等，比如建立最佳生成树[23]。

时代造就了互联网如今的规模，底层的物理网络是人们使用网络进行生活娱乐的基础。本文提出的网络协议仿真工具能够更好的帮助网络管理员管理真实网络拓扑情况，增加网络的利用效率，网络协议仿真工具的发展显示出其具有良好的前景。将此类工具运用在真实网络的预警监测中，具有广阔的利用空间。本文建立的模型希望可以促进网络协议仿真工具在真实网络拓扑上的应用，提高网络管理员的管理能力，为广播风暴等异常行为进行预警，以提高网络的利用能力。

# 致 谢

始于2019，终于2023，大学时光即将落幕。回望4年时间如花火，转瞬即逝。大学生活中有苦恼，有快乐，有迷茫，也有感激。整个大学过程中，我得到了很多人的支持，我能清晰感知到，是他们改变了我的未来，在此向他们表示我最诚挚的谢意。

首先，我要感谢我的家人们。感谢父母二十余年的培养，是你们的支持培养才能让我树立坚定的目标，不断冲破重重困难，在风雨中不断前进。

其次，我要感激帮助恩师们。4年时间遇到了很多老师，和他们的交流让我受益匪浅。张鹏教授为人谦和，学识渊博。在毕设的选题及毕设的进展中，张老师倾注了极大的精力及鼓励。李保红老师平易近人，对计算机底层有着独到的见解。在C++ 及嵌入式课上收获匪浅。也要感谢何亮老师，张兴军老师, 陈妍老师等大学期间所有的老师，感谢你们对我的教诲，关心。

再者，我也感谢李禹陪同学，我们彼此促学，同时他的聪明努力深深影响到了我, 让我不断成长，让我打开了眼界，能遇到他是我的荣幸。我还要感谢张梓健同学，每次和他的讨论都会让我茅塞顿开，帮助了我很多。也感谢王崇智学长，感谢他不辞辛劳的对我的毕设进行指导。

同时也感谢付思齐同学，我们彼此督促，不断成长，她以极大的热情，一直在我身边，我们度过了一段美好的时光，感谢她对我的一切支持。

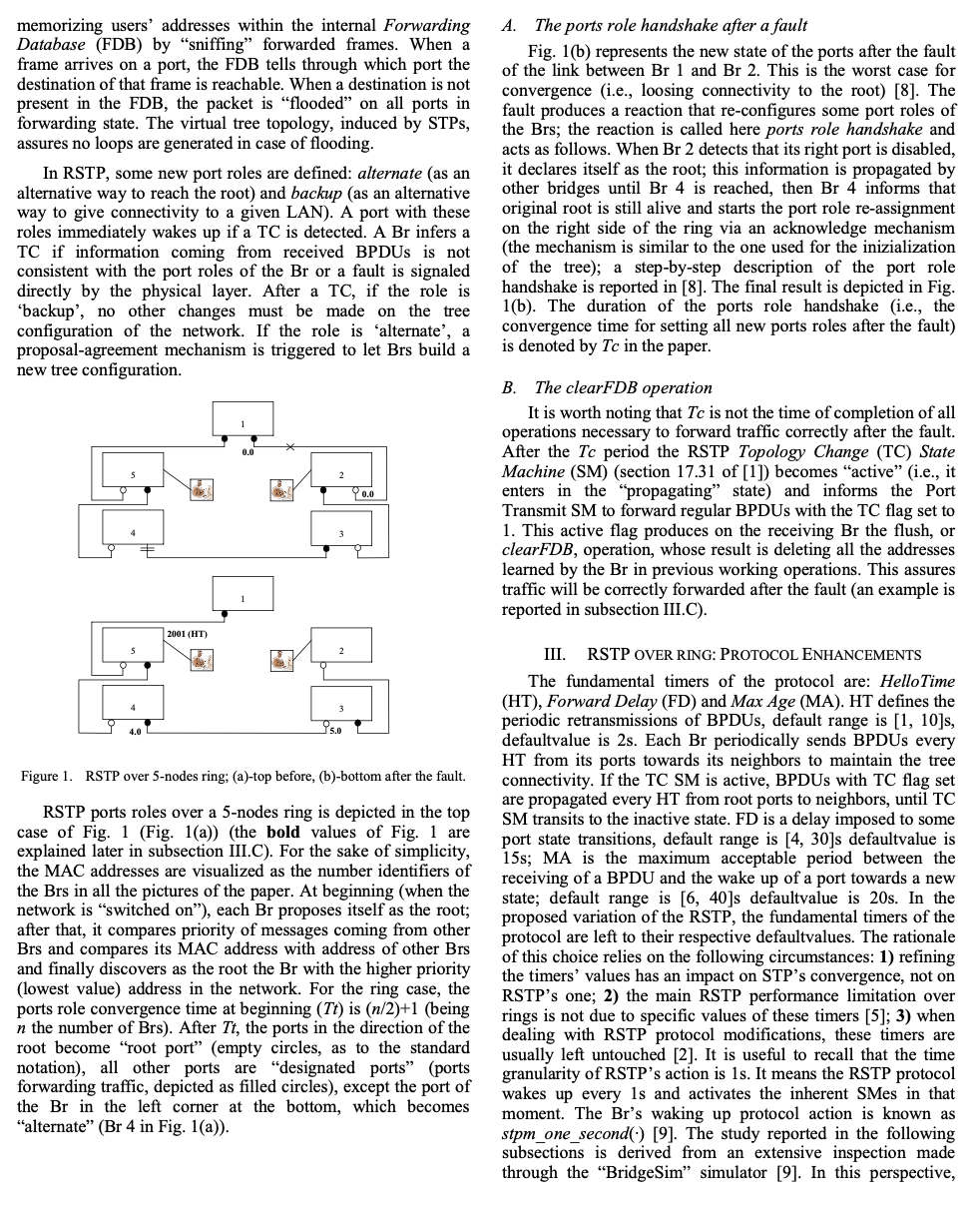
最后，我还要感谢论文答辩和评审中的各位教授，他们给了我未来努力的方向，是我发展路上的明灯。感谢以上的各位恩师，同学！在未来的路上，我一定为乘风破浪，不断远航。

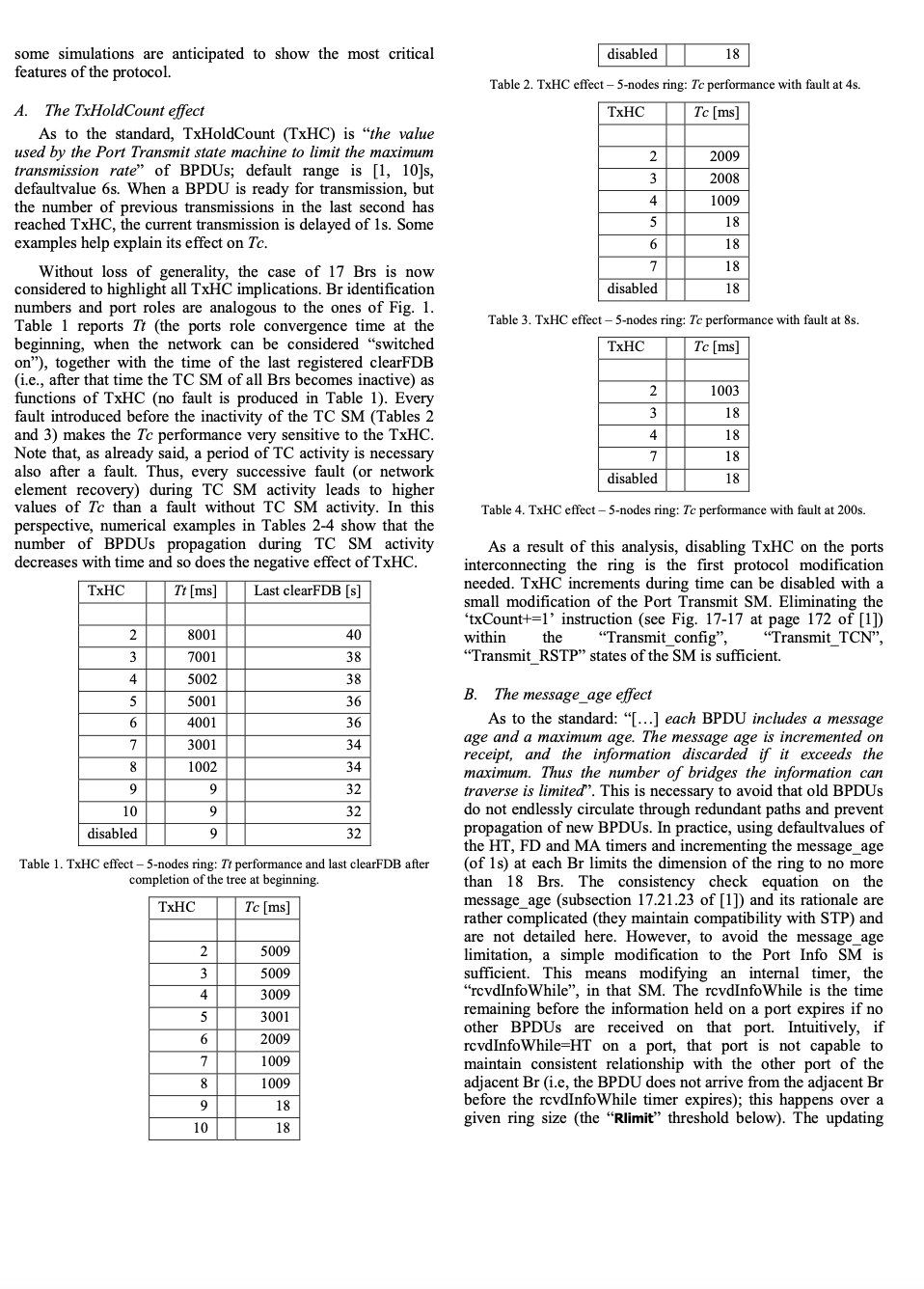
# 参考文献

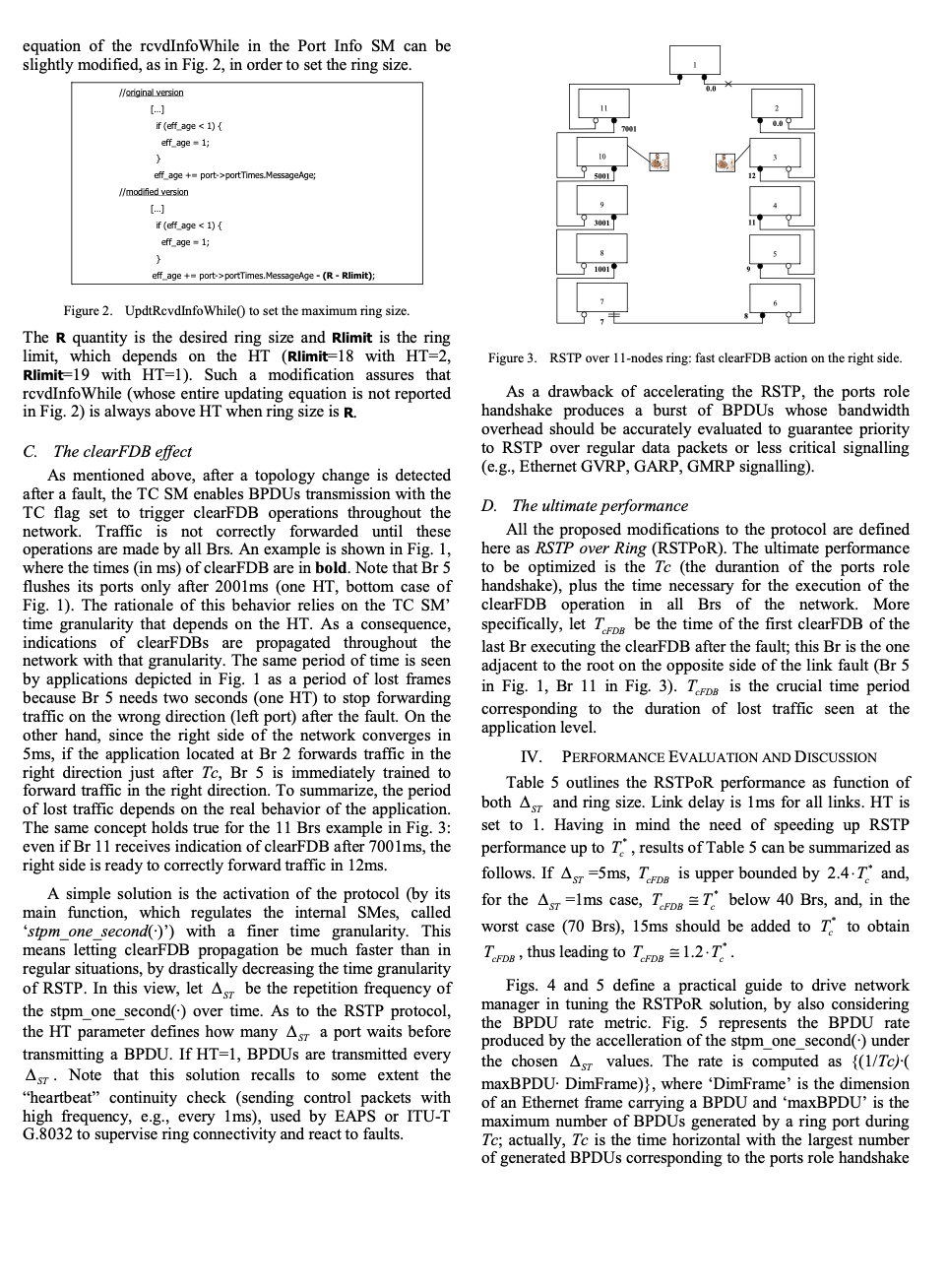
1. 古新文．计算机网络故障的归类分析［J］．科技信息(学术研究)，2007，No.237(25):197+199．
2. Beckett R, Gupta A, Mahajan R, et al. A general approach to network configuration verification[C]//Proceedings of the Conference of the ACM Special Interest Group on Data Communication. 2017: 155-168.
3. Prabhu S, Chou K Y, Kheradmand A, et al. Plankton: Scalable network configuration verification through model checking[C]//17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)．2020：953-967.
4. El-Hassany A, Tsankov P, Vanbever L, et al. Netcomplete: Practical network-wide configuration synthesis with autocompletion[C]//15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18). 2018: 579-594.
5. Birkner R, Drachsler-Cohen D, Vanbever L, et al. Config2Spec: Mining Network Specifications from Network Configurations[C]//NSDI. 2020: 969-984.
6. Perlman R. An algorithm for distributed computation of a spanningtree in an extended lan[J]. ACM SIGCOMM computer communication review, 1985, 15(4): 44-53.
7. 徐瀚，李汶青．双环网网络结构在田湾河流域梯级水电站计算机监控系统中的应用［J］．四川水力发电，2008，(03)：100-103+107．
8. 杨帆. 工业级冗余环网在大型机场中的应用[C]//.Proceedings of 2011 AASRI Conference on Artificial Intelligence and Industry Application(AASRI-AIIA 2011 V1), 2011:158-161.
9. 乌仁其梅．路由器链路故障分析［J］．中小企业管理与科技（下旬刊），2015(06)：209-210．
10. 刘伟杰．广州地铁五号线SDH以太网生成树协议的探讨［J］．计算机光盘软件与应. 用，2014，17(10)：279-280.
11. 张占国，刘淑芬，包铁等．基于ＳＴＰ协议的物理网络拓扑发现算法［J］．计算机工程，2008(06)：98-100．
12. 李雪梅，孙旭，鲁长江．以太网交换机中STP算法的优化设计［J］．微电子学与计算机，2006，23(5)：53-55．
13. 陈增吉． 基于 VRRP+MSTP 协议的可靠性网络设计［J］．计算机应用，2009，26(04)：208-21．
14. Longo E, Redondi A E C, Cesana M, et al. Mqtt-st: a spanning tree protocol for distributed mqtt brokers[C]//ICC 2020-2020 IEEE International Conference on Communications (ICC). IEEE, 2020: 1-6.
15. 陈金木．关于生成树协议（STP）在局域网中的研究应用［J］．网络安全和信息化，2021(03)：66-68．
16. Luo M, Hou X, Yang J. Surface optimal path planning using an extended Dijkstra algorithm[J]. IEEE access, 2020, 8: 147827-147838.
17. De Sousa A F. Improving load balance and resilience of ethernet carrier networks with ieee 802.1 s multiple spanning tree protocol[C]//International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06). IEEE, 2006: 95-95.
18. Fortz B, Gouveia L, Joyce-Moniz M. Optimal design of switched Ethernet networks implementing the Multiple Spanning Tree Protocol[J]. Discrete Applied Mathematics, 2018, 234: 114-130.
19. Ibáñez G, García A, Azcorra A. Alternative multiple spanning tree protocol (AMSTP) for optical Ethernet backbones[C]//29th Annual IEEE International Conference on Local Computer Networks. IEEE, 2004: 744-751.
20. 叶涛,王思齐,杨建彪.基于eNSP的大规模路由综合设计与仿真实验［J］．实验室研究与探索，2019，38(04)：109-114．
21. Willman J M. Beginning PyQt[M]. Berkeley, CA: Apress, 2020.
22. Anagreh M, Vainikko E, Laud P. Parallel Privacy-preserving Computation of Minimum Spanning Trees[C]//ICISSP. 2021: 181-190.
23. Mirjalily G, Sigari F A, Saadat R. Best multiple spanning tree in metro ethernet networks[C]//2009 Second International Conference on Computer and Electrical Engineering. IEEE, 2009, 2: 117-121.

# 附 录

# 外文原文



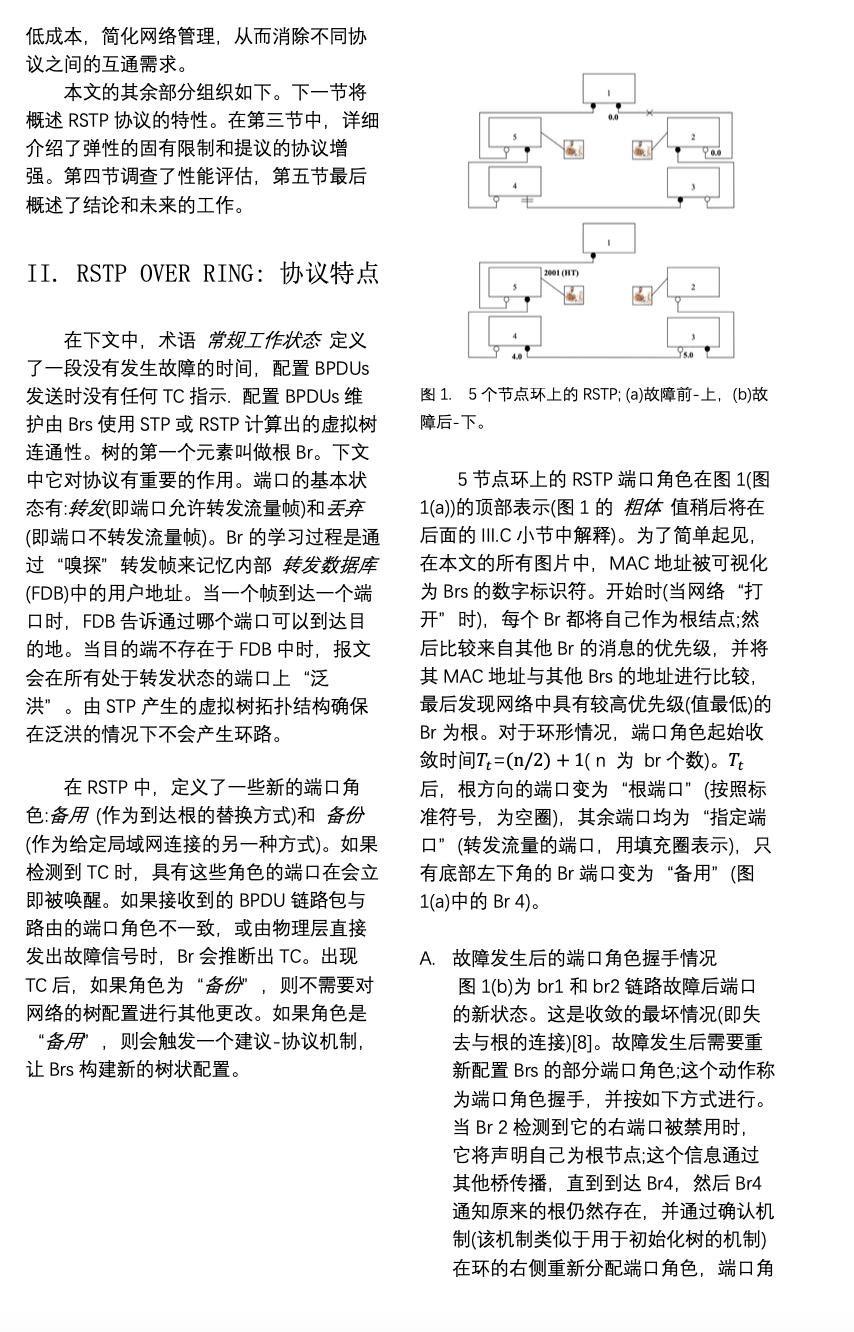


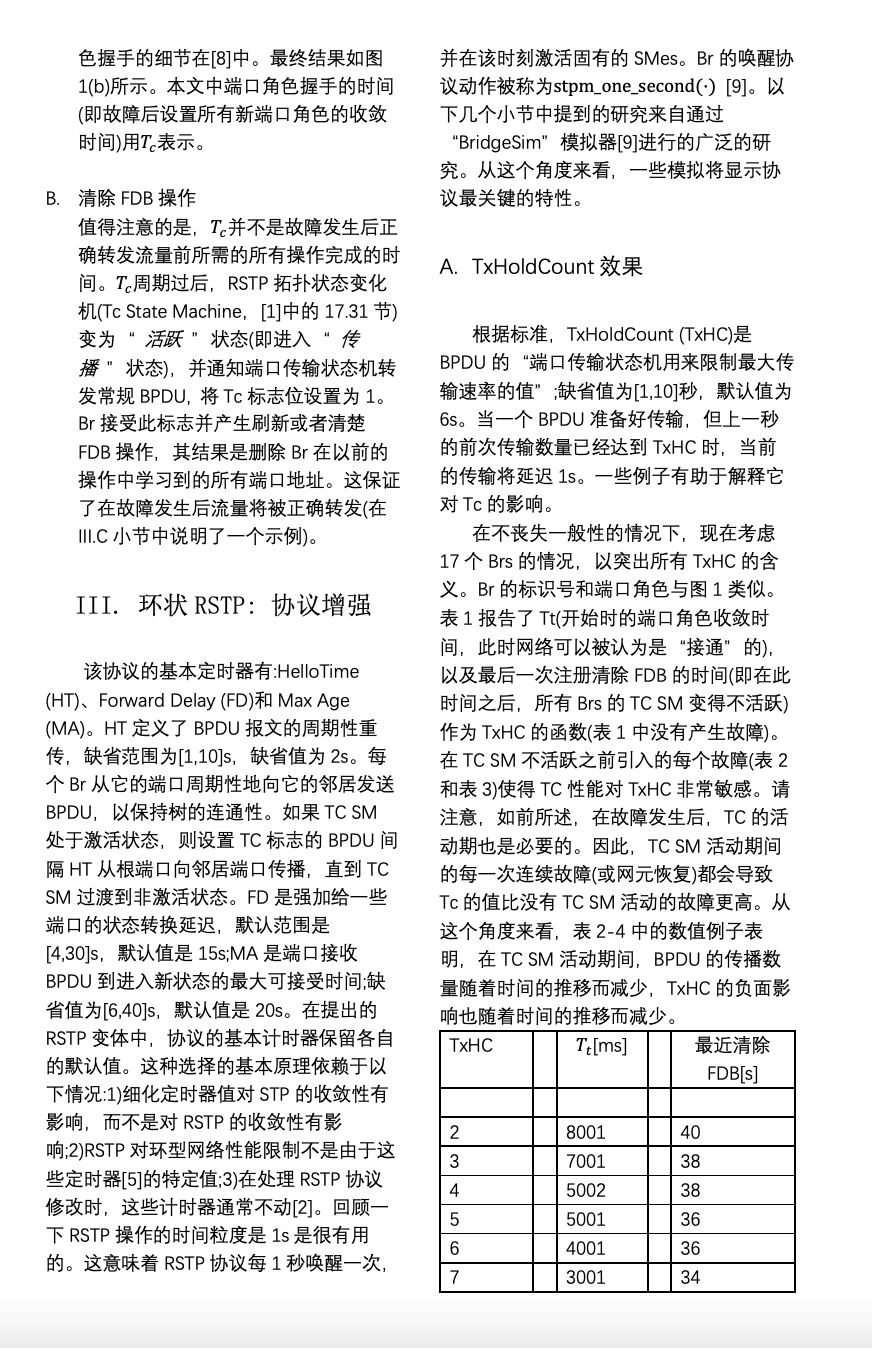


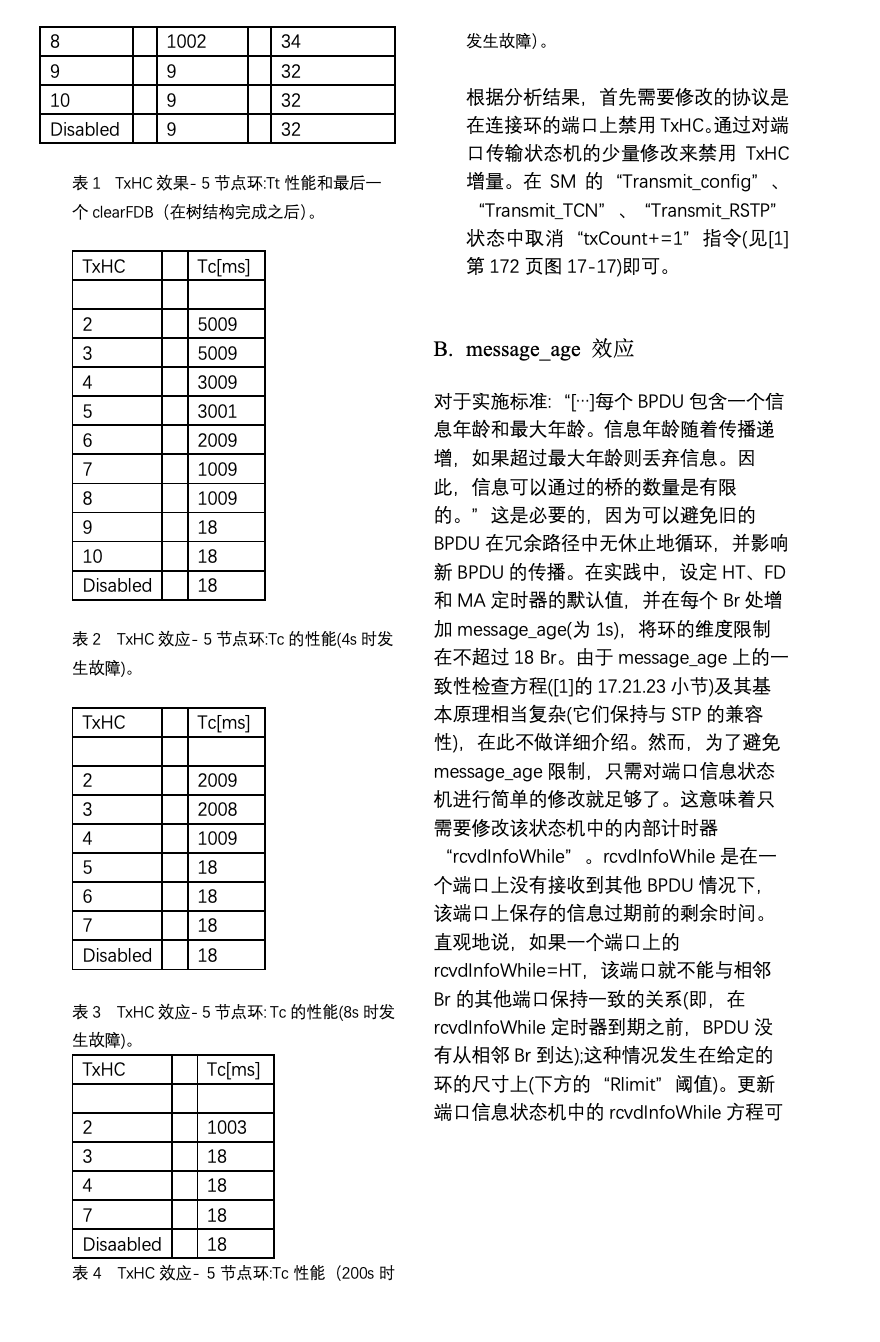


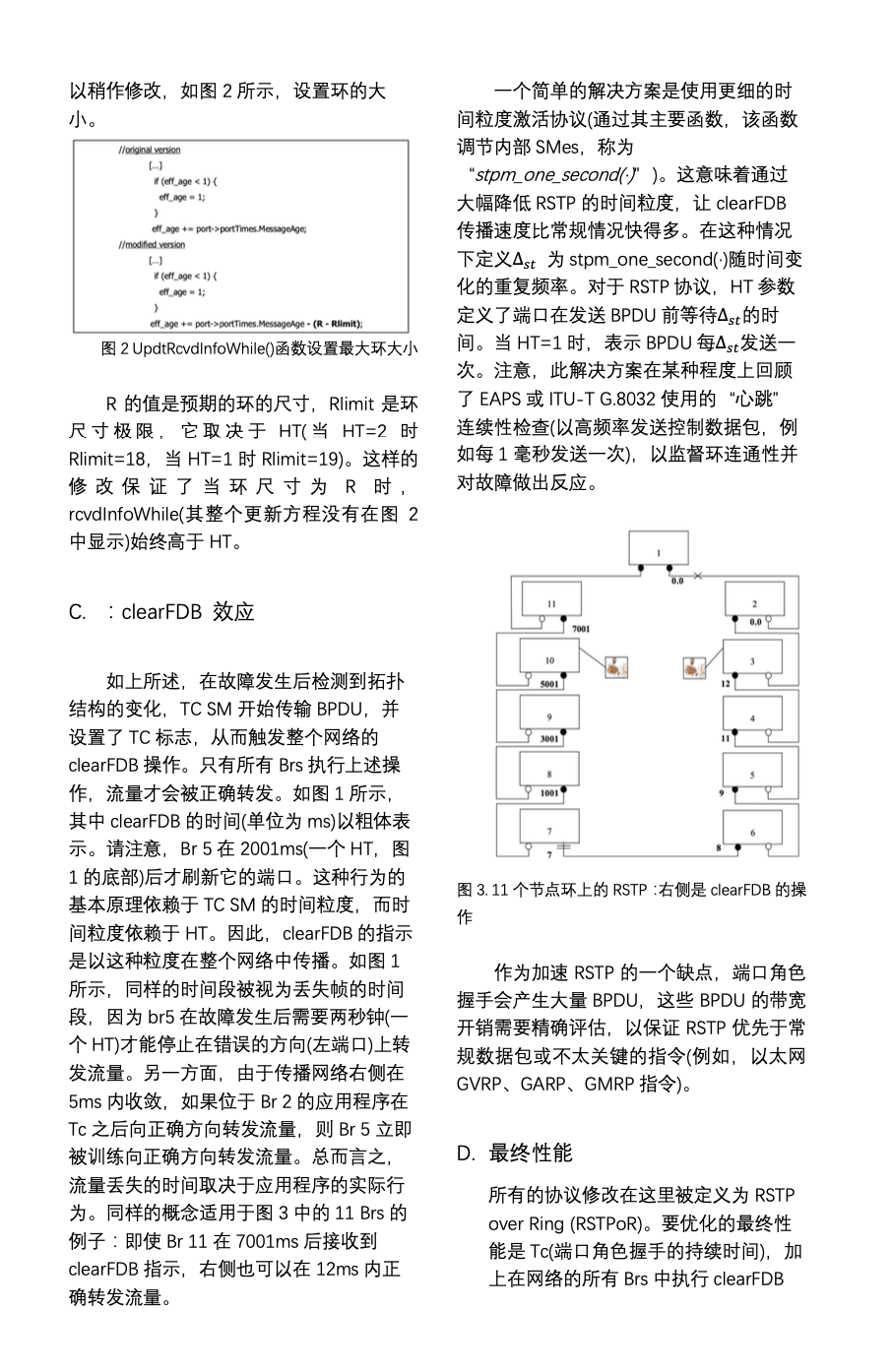
# 附录 B

# 外文译文

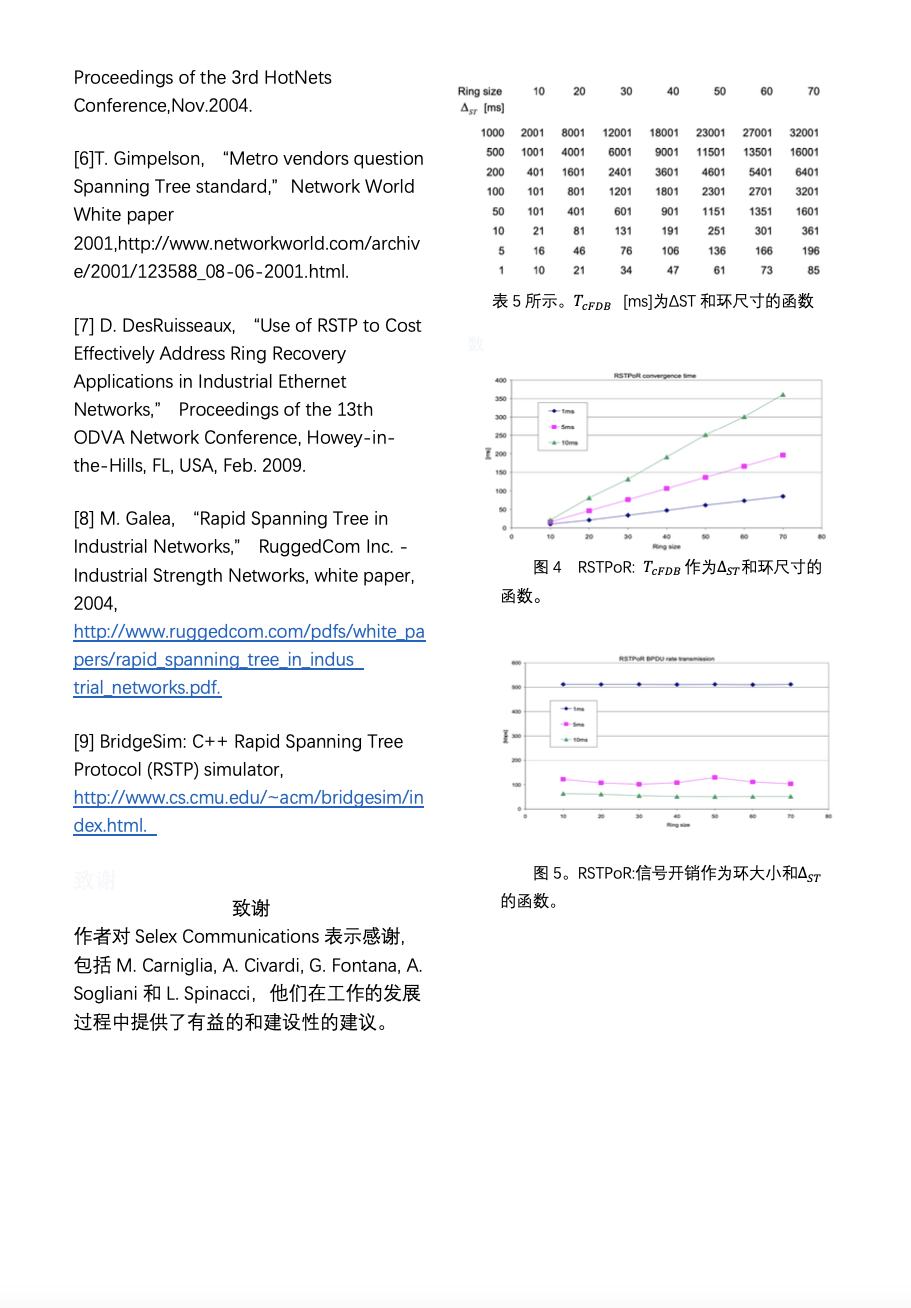












# 附 录 C

# 源程序

STP仿真算法：

import pandas as pd

import time

import networkx as nx

import matplotlib.pyplot as plt

import PIL

fp = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/print.txt',"w")

import matplotlib

g = nx.Graph() # 创建拓扑图

G = nx.Graph()

#增加节点 节点是端口

#建立交换机类

print("-------添加节点中--------", file = fp)

class Switch:

def \_\_init\_\_(self, port\_list, number, switch\_name, pc\_mac = 0):

self.port = []

self.switch\_name = switch\_name

for i in port\_list:

self.port.append(i)

self.number = number ## switch bid

self.pc\_mac = pc\_mac

class fwd\_table:

def \_\_init\_\_(self, switch\_name):

self.name = switch\_name

self.fwd\_index = list()

file\_path = '/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/stp.xls'

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='switch')

data = raw\_data.values

switch\_list = list()

## 读取execl switch信息

for i in range(data.shape[0]) :

switch\_data = data[i]

str1 = str(switch\_data[0])

list1 = str1.split(',')

list1 = [int(i) for i in list1]

print(list1, file = fp)

switch\_list.append(Switch(list1, switch\_data[1], switch\_data[2],switch\_data[3]))

##输入节点 之后进行修改

# 对于4节点的拓扑图 8 个节点 初始化BPDU 端口初始化

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='port')

data = raw\_data.values

for i in range(data.shape[0]) :

port\_data = data[i]

g.add\_node(port\_data[0],RPC = 0,SBID = 0, BID = 0, SPID = port\_data[1],Port\_status = 'None')

##数据维修

for switch in switch\_list:

port\_list = switch.port

for port in port\_list:

g.nodes[port]['SBID'] = switch.number

g.nodes[port]['BID'] = switch.number

g.add\_node(switch.switch\_name,RPC = -1, SBID = -1, BID = switch.number, SPID = -1, Port\_status = 'Switch')

for node in g.nodes(data = True):

print(node, file = fp)

#初始化边

for switch in switch\_list :

#根据路由器来添加节点

port\_list = switch.port

switchname = switch.switch\_name

for port in port\_list :

g.add\_weighted\_edges\_from([(port,switchname,0)])

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='edge')

data = raw\_data.values

for i in range(data.shape[0]) :

list1 = list(data[i])

g.add\_weighted\_edges\_from([(list1[0],list1[1],list1[2])])

for edge in g.edges(data = True):

if edge[2]['weight'] != 0:

print(edge, file = fp)

print('----节点添加完成', file = fp)

#选举根桥

# 选择根节点找到对应的根可能对应多个端口

print('----选举根桥-----', file = fp)

time.sleep(0.5)

rootNode = 9999 #max

for node in g.nodes(data = True):

if rootNode > node[1]['BID'] :

rootNode = node[1]['BID']

print("THE rootNode BID is {}".format(rootNode), file = fp)

#此时直接根据这个来选择BID

#更新配置的BPDU

print("------更新配置的BPDU------", file = fp)

time.sleep(0.5)

#更改BPDU状态

for node in g.nodes(data = True):

if node[1]['BID'] == rootNode and node[1]['Port\_status'] != 'Switch':

g.nodes[node[0]]['Port\_status'] = "Designated"

node[1]['BID'] = rootNode

print(node, file = fp)

else:

print("the alter BID action is over", file = fp)

print("---------寻找root Port------计算RPC-----", file = fp)

time.sleep(0.5)

#寻找根端口计算RPC

#内置算法

def rpc(node1, node2):

#断开对应的端口计算

p = nx.shortest\_path\_length(g,source = node1, target = node2, weight = 'weight' )

return p

def path(node1,node2):

p = nx.shortest\_path(g, source = node1, target = node2, weight = 'weight')

return p

def find\_bid(port1): #参数为端口

for switch in switch\_list:

if port1 in switch.port:

return switch.number

#更新rpc 与配置 BPDU

#根节点为BID

target\_switch = Switch([],-1,'-1')

for switch in switch\_list:

#找到对应的switch

if switch.number == rootNode:

target\_switch = switch

break

for switch in switch\_list: #更改 定义端口数量

if switch == target\_switch:

continue;

#交换机 存储着BID 对应端口的PID 我们需要对每个端口计算到BP的距离

port\_list = switch.port

switchname = switch.switch\_name

#去掉端口的边

for i in port\_list:

if g.has\_edge(i,switchname):

g.remove\_edge(i,switchname)

#更新完成计算rpc

#对port\_list 计算

for i in port\_list:

min = 999999

for j in target\_switch.port:

p1 = rpc(i,j)

if min > p1:

min = p1

#更优秀

short\_path = path(i,j)

sbid = find\_bid(short\_path[1])

#计算出来后需要修改

g.nodes[i]['RPC'] = min

g.nodes[i]['SPID'] = short\_path[1]

g.nodes[i]['SBID'] = sbid

#恢复线路

for i in port\_list:

if not g.has\_edge(i,switchname) :

g.add\_weighted\_edges\_from([(i,switchname,0)])

for edge in g.edges(data=True):

print(edge, file = fp)

print("##########", file = fp)

## 已经算出了所有port 对应的rpc 接下来我们需要判断 RP

print('-------更新结果如下---------', file = fp)

for node in g.nodes(data = True):

print(node, file = fp)

#########################此时节点的BPDU更新完毕

## r RP 每个跟端口有且只有唯一一个

print('------选举RootPort-----', file = fp)

time.sleep(0.5)

for swith in switch\_list:

if swith == target\_switch:

continue

##开始选举

rootport = swith.port[0] ## 指明第一个端口

for i in swith.port:

if g.nodes[i]['RPC'] < g.nodes[rootport]['RPC']:

rootport = i

elif g.nodes[i]['RPC'] == g.nodes[rootport]['RPC'] :

if g.nodes[i]['SBID'] < g.nodes[rootport]['SBID']:

rootport = i

g.nodes[rootport]['Port\_status'] = 'Root Port'

for node in g.nodes(data = True):

print(node, file = fp)

## 此时已经交换完成了 我们可以用来选举DP 指定端口的选举

## DP 的选择是根据 路由器之间 为一个网段，每个网段选举出一个指定端口 比较SBID SPID 来进行决定

## 先比较 RPC

print('-----重新计算RPC------', file = fp)

time.sleep(0.2)

#######################################DP

### 此时不进行切边，我们需要对链路的RPC重新计算

for switch in switch\_list:

port\_list = switch.port

min = 99999

for port in port\_list:

if g.nodes[port]['RPC'] < min:

min = g.nodes[port]['RPC']

for port in port\_list:

g.nodes[port]['RPC'] = min

target\_port = target\_switch.port

for switch in switch\_list:

if switch == target\_switch:

continue

else:

port\_list =switch.port

for port in port\_list:

g.nodes[port]['SPID'] = port

g.nodes[port]['SBID'] = switch.number

print('-----选举指定端口DP------', file = fp)

#对边进行判断

port\_list = target\_switch.port #这些端口已经为DP

for edge in g.edges.data():

if edge[0] in port\_list or edge[1] in port\_list :

# 指定端口已经选举

continue

if(edge[2]['weight'] == 0):

continue

#此时未选举

if g.nodes[edge[0]]['RPC'] < g.nodes[edge[1]]['RPC']:

g.nodes[edge[0]]['Port\_status'] = 'Designated'

elif g.nodes[edge[0]]['RPC'] > g.nodes[edge[1]]['RPC']:

g.nodes[edge[1]]['Port\_status'] = 'Designated'

else:

#对比SBID

if g.nodes[edge[0]]['SBID'] < g.nodes[edge[1]]['SBID']:

g.nodes[edge[0]]['Port\_status'] = 'Designated'

elif g.nodes[edge[0]]['SBID'] > g.nodes[edge[1]]['SBID']:

g.nodes[edge[1]]['Port\_status'] = 'Designated'

else:

if g.nodes[edge[0]]['SPID'] < g.nodes[edge[1]]['SPID']:

g.nodes[edge[0]]['Port\_status'] = 'Designated'

elif g.nodes[edge[0]]['SPID'] > g.nodes[edge[1]]['SPID']:

g.nodes[edge[1]]['Port\_status'] = 'Designated'

for node in g.nodes(data = True):

print(node, file = fp)

## 端口更新成功

print("端口状态更新成功", file = fp)

print("-------选举堵塞AP端口------", file = fp)

time.sleep(0.5)

apPort\_list = list()

for node in g.nodes.data():

if node[1]['Port\_status'] == 'None' :

##此时为AP

node[1]['Port\_status'] = 'AP'

apPort\_list.append(node[0])

print('-------打印最终结果-------', file = fp)

for node in g.nodes(data = True):

print(node, file = fp)

print("##3种端口状态及根桥更新完毕##", file = fp)

for node in g.nodes(data = True):

print(node)

fp2 = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/role.txt', "w")

for node in g.nodes(data = True):

print(str(node[0])+ " " + "role: " + node[1]['Port\_status'], file = fp2)

fp2.close()

for edge in g.edges(data = True):

print(edge)

edge\_remove = list()

for edge in g.edges(data = True):

port1 = edge[0]

port2 = edge[1]

if edge[2]['weight'] == 0:

continue

else:

if port1 in apPort\_list or port2 in apPort\_list:

edge\_remove.append(edge)

for edge in edge\_remove:

g.remove\_edge(edge[0], edge[1])

print('开始计算MAC转发表')

fwdTable\_list = list()

pc\_list = dict()

allPc = list()

mac\_node\_map = dict()

for switch in switch\_list:

fwdTable\_list.append(fwd\_table(switch.switch\_name))

str1 = str(switch.pc\_mac)

list1 = str1.split(',')

list1 = [str(i) for i in list1]

## 现在添加PC节点

pc\_list[switch.switch\_name] = list1

for pc in list1:

allPc.append(pc)

mac = pc

pc = pc[3:]

mac\_node\_map[mac] = 'pc' + str(pc)

g.add\_node('pc' + str(pc), PRC = -1, SBID = -1, SPID = -1, Port\_status = 'PC' )

g.add\_edge(switch.switch\_name, 'pc' + str(pc), weight = 0)

print("----------增加PC节点后------------")

for edge in g.edges(data = True):

print(edge)

for node in g.nodes(data = True):

print(node)

#index MAC-地址————VLANID----------PORT

for switch in fwdTable\_list:

print(switch.name + "路由表项")

switch\_name = switch.name

##添加自己的路由表项

port\_list = pc\_list[switch\_name]

for pc in port\_list:

#添加自己的路由表项

mac = pc

node = mac\_node\_map[mac]

index = [mac, 0, node]

switch.fwd\_index.append(index)

print(index)

for pc\_des in allPc:

if pc\_des in port\_list:

continue

else:

destination = mac\_node\_map[pc\_des]

path1 = path(switch\_name, destination)

index = [pc\_des, 0, path1[1]]

switch.fwd\_index.append(index)

print(index)

for edge in edge\_remove:

g.add\_edge(edge[0], edge[1], weight = edge[2]['weight'])

print('---------MAC转发表添加完毕-------------')

#开始画图：

print('-----拓扑图更新-----', file = fp)

icons = {

"switch": '/Users/yewen/Desktop/2023毕业设计/code/switch\_node.png'

}

# Load images

images = {k: PIL.Image.open(fname) for k, fname in icons.items()}

port\_ap = list()

macALL = list()

for switch in switch\_list:

port\_list = switch.port

str1 = str(switch.pc\_mac)

list1 = str1.split(',')

list1 = [str(i) for i in list1]

for i in list1:

macALL.append(i)

G.add\_node(switch.switch\_name, BID=switch.number, Port\_status='Switch', image=images['switch'], fwd\_mac = list1, fwding = list())

for port in port\_list:

if g.nodes[port]['Port\_status'] =='AP':

G.add\_node(switch.switch\_name, BID=switch.number, Port\_status='Switch\_AP', image=images['switch'], fwd\_mac= list1, fwding = list())

port\_ap.append(port)

for edge in g.edges(data = True):

if edge[2]['weight'] == 0 :

continue

weight = edge[2]['weight']

flag = 0

if g.nodes[edge[0]]['Port\_status'] == 'AP' or g.nodes[edge[1]]['Port\_status'] == 'AP':

flag = 1

pair = list()

pair\_name = list()

pair.append(edge[0])

pair.append(edge[1])

##寻找

for port in pair:

for switch in switch\_list:

if port in switch.port:

pair\_name.append(switch.switch\_name)

##找到了

G.add\_edge(pair\_name[0], pair\_name[1], weight = weight, AP = 'false', color = 'blue')

if flag == 1:

G[pair\_name[0]][pair\_name[1]]['AP'] = 'true'

for node in G.nodes(data = True):

print(node)

for edge in G.edges(data = True):

print(edge)

#开始学习MAC表

# 未每个switch分配mac学习表

print("########", file = fp)

edge\_list\_red = list()

edge\_list\_blue = list()

## blue 为存储AP的端口

for edge in G.edges(data = True):

if edge[2]['AP'] == 'false' :

edge\_list\_red.append((edge[0],edge[1]))

if edge[2]['AP'] == 'true' :

edge\_list\_blue.append((edge[0], edge[1]))

print(edge, file = fp)

# G 中存储着对应的switch节点

# switch\_list 存储对应接口的具体消息

print("-------节点分类-----")

fp1 = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/mac\_table.txt', "w")

for switch in fwdTable\_list:

list\_fwd = switch.fwd\_index

n = len(list\_fwd)

print(switch.name + " MAC表如下：", file = fp1)

for i in range(n):

print(list\_fwd[i], file = fp1)

print(" ", file = fp1)

fp1.close()

###开始画图

pos = nx.spring\_layout(G, seed=1734289230)

fig, ax = plt.subplots()

##对线进行处理

nx.draw\_networkx\_edges(

G,

pos=pos,

edgelist = edge\_list\_red,

ax=ax,

arrows=True,

arrowstyle="-",

min\_source\_margin=15,

min\_target\_margin=15,

edge\_color="tab:red",

)

nx.draw\_networkx\_edges(

G,

pos=pos,

edgelist = edge\_list\_blue,

ax=ax,

arrows=True,

arrowstyle="]-[",

min\_source\_margin=15,

min\_target\_margin=15,

edge\_color="tab:blue",

)

#设置字体为楷体

plt.title("STP result")

tr\_figure = ax.transData.transform

tr\_axes = fig.transFigure.inverted().transform

icon\_size = (ax.get\_xlim()[1] - ax.get\_xlim()[0]) \* 0.025

icon\_center = icon\_size / 2.0

for n in G.nodes:

xf, yf = tr\_figure(pos[n])

xa, ya = tr\_axes((xf, yf))

# get overlapped axes and plot icon

a = plt.axes([xa - icon\_center, ya - icon\_center, icon\_size, icon\_size])

a.imshow(G.nodes[n]["image"],)

a.axis("off")

plt.title(n)

fp.close()

plt.savefig('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/stp.png')

plt.show()

MSTP 算法：

import pandas as pd

import time

import networkx as nx

import matplotlib.pyplot as plt

import PIL

g = nx.Graph() # 创建无相图

fp = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/print.txt',"w")

fp2 = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/role.txt', "w")

#节点添加

print("-------添加节点中--------", file = fp)

# 定义switch类

class Switch:

#内容： port端口，name，mac，priority，bid

def \_\_init\_\_(self, port\_list, mac, switch\_name, priority, region\_name, mac\_list):

self.port = []

self.switch\_name = switch\_name

for i in port\_list:

self.port.append(i)

self.mac = mac ## switch bid

self.priority = priority

self.bid = int(str(self.priority) + str(self.mac)) # bid 代表优先级

self.region\_name = region\_name

self.mac\_list = mac\_list

class fwd\_table:

def \_\_init\_\_(self, switch\_name):

self.name = switch\_name

self.fwd\_index = list()

file\_path = '/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/mstp.xls'

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='switch')

data = raw\_data.values

#添加switch 信息

# 初始化switch 信息

switch\_list = list()

print('-----switch信息添加----', file = fp)

print(" ")

for i in range(data.shape[0]):

## 每行数据

switch\_data = data[i]

print(switch\_data)

str1 = str(switch\_data[0])

list1 = str1.split(',')

list1 = [int(i) for i in list1]

print(list1, file = fp)

switch\_list.append(Switch(list1, switch\_data[2], switch\_data[3], switch\_data[1], switch\_data[4],switch\_data[5]))

print('--------switch 添加完毕--------', file = fp)

print('--------添加端口port信息--------', file = fp)

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='port')

data = raw\_data.values

for i in range(data.shape[0]):

port\_data = data[i]

print(port\_data, file = fp)

pid = int(str(port\_data[2]) + str(port\_data[1]))

g.add\_node(port\_data[0], PID=pid, Root\_bridge\_ID='0', ERPC=0, \

Region\_bridge\_ID=0, IRPC=0, Designated\_BID=0, Designated\_PID=0, received\_PID=0, Port\_status='None',

switch\_bid=-1, portID=port\_data[1], Port\_priority=port\_data[2],VLAN = port\_data[3] )

##对端口对应 的switch进行数据维修

for switch in switch\_list:

name = switch.switch\_name

for port in switch.port:

g.nodes[name + '-' + str(port)]['switch\_bid'] = switch.bid

g.nodes[name + '-' + str(port)]['Root\_bridge\_ID'] = switch.bid

g.nodes[name + '-' + str(port)]['Region\_bridge\_ID'] = switch.bid

g.add\_node(switch.switch\_name, PID=-1, Root\_bridge\_ID=switch.bid, ERPC=-1, \

Region\_bridge\_ID=switch.bid, IRPC=-1, Designated\_BID=-1, Designated\_PID=-1, received\_PID=-1,

Port\_status='switch', switch\_bid=switch.bid, portID=-1, Port\_priority=-1, VLAN = 'none')

for node in g.nodes(data=True):

print(node, file = fp)

print('------node 信息添加完毕--------', file = fp)

print('------添加edge信息------------', file = fp)

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='edge')

data = raw\_data.values

for i in range(data.shape[0]):

data\_edge = list(data[i])

g.add\_edge(data\_edge[0], data\_edge[1], cost=data\_edge[2], region\_name='none')

print('--------创建edge region 映射关系------', file = fp)

switch\_region = {}

for switch in switch\_list:

switch\_region[switch.switch\_name] = switch.region\_name

for edge in g.edges.data():

str1 = (edge[0][0:4])

str2 = (edge[1][0:4])

if switch\_region[str1] == switch\_region[str2]:

g[edge[0]][edge[1]]['region\_name'] = switch\_region[edge[1][0:4]]

else:

g[edge[0]][edge[1]]['region\_name'] = 'mid\_region'

for switch in switch\_list:

# 根据路由器来添加节点

port\_list = switch.port

switchname = switch.switch\_name

for port in port\_list:

g.add\_edge(switchname + '-' + str(port) \

, switchname, \

cost=0, region\_name=switch.region\_name)

for edge in g.edges(data=True):

print(edge, file = fp)

print('------edge添加完毕-------', file = fp)

# ---------现在开始加载端口的设置

print('------开始选举总根', file = fp)

root\_bridge = switch\_list[0].bid

root\_bridge\_name = switch\_list[0].switch\_name

for switch in switch\_list:

if root\_bridge > switch.bid:

root\_bridge = switch.bid

root\_bridge\_name = switch.switch\_name

##选举出最小的bid

print("全局根桥为{}".format(root\_bridge\_name), file = fp)

##更新端口信息

for node in g.nodes(data=True):

##更新Root\_bridhe\_ID:

g.nodes[node[0]]['Root\_bridge\_ID'] = root\_bridge

print('--------选举备份端口Backup------', file = fp)

##创建端口映射关系

def judge\_from\_the\_same\_switch(list1):

# 1 表示不重复 0表示重复

list2 = list1[:]

n = len(list1)

for i in range(n):

list2[i] = list2[i][0:4]

list2 = list(set(list2))

n1 = len(list2)

if n1 == n:

return 1

else:

return 0

def count\_num(list1):

list2 = list1[:]

n = len(list1)

for i in range(n):

list2[i] = list2[i][0:4]

dict1 = dict()

for item in list2:

if item in dict1: # 直接判断key在不在字典中

dict1[item] += 1

else:

dict1[item] = 1

list2 = list()

for i in dict1:

if dict1[i] > 1:

for j in list1:

if i in j:

list2.append(j)

return list2

port\_edge = dict()

for edge in g.edges(data=True):

key = edge[0]

port\_edge.setdefault(key, []).append(edge[1])

key = edge[1]

port\_edge.setdefault(key, []).append(edge[0])

print(port\_edge, file = fp)

# 逐个访问 字典，判断有无来自两个的

backup\_del = list()

backup = []

backup\_name = []

for key in port\_edge:

list1 = port\_edge[key]

# key 为汇聚的东西

if len(key) <= 4:

continue

if (judge\_from\_the\_same\_switch(list1) == 1):

continue

if (judge\_from\_the\_same\_switch(list1) == 0):

# 重复了 堵塞其中一个为backup list1中存在重复的端口

# 挨个检查对应的

repeat = count\_num(list1)

print(repeat, file = fp)

print(key, file = fp)

# 选择一个进行阻塞

max\_cost = -1

max1 = repeat[0]

for i in repeat:

if max\_cost < g[key][i]['cost']:

max\_cost = g[key][i]['cost']

max1 = i

print('-----{}-{}由于重复阻塞成为backup-----'.format(key, max1), file = fp)

g.nodes[max1]['Port\_status'] = 'backup'

print(g[max1][key], file = fp)

backup\_del.append([max1, key, g[max1][key]])

backup.append([max1, g.nodes[max1]])

backup\_name.append(max1)

print(g.nodes[max1], file = fp)

g.remove\_edge(key, max1)

g.remove\_node(max1)

print("backup删除链路为：", backup\_del, file = fp)

print("backup删除节点为", backup, file = fp)

for node in g.nodes(data=True):

print(node[0], node[1]['Port\_status'], file = fp)

##去掉backup来计算其他的节点

for node in g.nodes(data=True):

print(node, file = fp)

##更新Root\_bridgr\_ID成功

##分别为每个域分开计算

## region:

print('----------域映射关系---------', file = fp)

region\_dict = {}

for switch in switch\_list:

key = switch.region\_name

region\_dict.setdefault(key, []).append(switch.switch\_name)

print(region\_dict, file = fp)

def find\_bid(switch\_name):

for switch in switch\_list:

if switch.switch\_name == switch\_name:

return switch.bid

def find\_name(switch\_bid):

for switch in switch\_list:

if switch.bid == switch\_bid:

return switch.switch\_name

print("-----开始生成IST------", file = fp)

#对每个域分开生成

#断开edge

edge\_list = []

for edge in g.edges(data=True):

print(list(edge), file = fp)

if edge[2]['region\_name'] == 'mid\_region':

edge = list(edge)

print(edge)

edge\_list.append(list(edge))

print("------已经选拔出域间edge并且存放在edge\_list 中-------", file = fp)

print(edge\_list, file = fp)

#断开edge

for i in range(len(edge\_list)):

g.remove\_edge(edge\_list[i][0], edge\_list[i][1])

print("-----------域间edge删除成功----------------", file = fp)

for edge in g.edges(data=True):

print(edge, file = fp)

print('-------选拔域边缘端口 domain\_edge--------', file = fp)

domain\_edge\_list = list()

for edge in edge\_list:

domain\_edge\_list.append(edge[0])

domain\_edge\_list.append(edge[1])

print(domain\_edge\_list, file = fp)

domain\_edge\_list = list(set(domain\_edge\_list))

for node in g.nodes(data=True):

if node[0] in domain\_edge\_list and g.nodes[node[0]]['Port\_status'] != 'backup':

g.nodes[node[0]]['Port\_status'] = 'domain\_edge\_port'

for node in g.nodes(data=True):

print(node[0], node[1]['Port\_status'], file = fp)

print('-------域边缘端口选举完毕-------', file = fp)

def rpc(node1, node2,g):

# 断开对应的端口计算

p = nx.shortest\_path\_length(g, source=node1, target=node2, weight='cost')

return p

def path(node1, node2,g):

p = nx.shortest\_path(g, source=node1, target=node2, weight='cost')

return

def find\_switch(switch\_name):

for switch in switch\_list:

if switch.switch\_name == switch\_name:

return switch

def find\_designated\_bridge(Designated\_port\_name):

for switch in switch\_list:

port\_list = switch.port

for port in port\_list:

name = switch.switch\_name + '-' + str(port)

if name == Designated\_port\_name:

return switch.bid

def find\_designated\_pid(Designated\_port\_name):

for node in g.nodes(data=True):

if node[0] == Designated\_port\_name:

return node[1]['PID']

def find\_self\_pid(name):

for node in g.nodes(data=True):

if node[0] == name:

return node[1]['PID']

print("--------开始在域内选举MSTI--------", file = fp)

print("---------域根桥选举-----------", file = fp)

for region in region\_dict:

switch\_region\_list = region\_dict[region]

print(region, switch\_region\_list, file = fp)

#bid

#对域分开操作

#寻找region root

switch\_bid = list()

for switch\_name in switch\_region\_list:

switch\_bid.append(find\_bid(switch\_name))

region\_root\_bid = min(switch\_bid)

目前可以找到每一个域内对应switch的bid 和 域的名字

region\_root\_name = find\_name(region\_root\_bid)

print("{}域根桥bid为{},名字为{}".format(region, region\_root\_bid, region\_root\_name), file = fp)

print('-------{}域根桥选举完毕'.format(region) + '----------', file = fp)

#更新每一个域node节点的

#更新

#已断开e域间edge

#now 我们断开域内的edge

del\_switch = list()

for switch in switch\_list:

if switch.bid not in switch\_bid:

continue

if switch.bid in switch\_bid:

del\_switch.append(switch)

# del\_switch存储着域的switch

for node in g.nodes(data=True):

if node[1]['switch\_bid'] in switch\_bid:

g.nodes[node[0]]['Region\_bridge\_ID'] = region\_root\_bid

if node[1]['switch\_bid'] == region\_root\_bid and node[1]['Port\_status'] != 'switch' and node[1]['Port\_status'] != 'domain\_edge\_port':

g.nodes[node[0]]['Port\_status'] = 'DP'

print("{}域内node，BPDU中Region\_bridge\_ID已更新".format(region), file = fp)

print("------开始更新IRPC 为了选举域内根端口-------", file = fp)

region\_switch\_list = list() # 域中对应的switch

for switch\_name in switch\_region\_list:

region\_switch\_list.append(find\_switch(switch\_name))

target\_switch = region\_switch\_list[0]

for switch in region\_switch\_list:

# 依次对每个switch进行更新

if switch.bid == region\_root\_bid:

target\_switch = switch ##域内根桥

# target port 更逊

for port in target\_switch.port:

g.nodes[switch.switch\_name + '-' + str(port)]['Designated\_BID'] = switch.bid

g.nodes[switch.switch\_name + '-' + str(port)]['Designated\_PID'] = \

g.nodes[switch.switch\_name + '-' + str(port)]['PID']

break

for switch in region\_switch\_list:

if switch.bid == region\_root\_bid:

continue

port\_list = switch.port # 域内桥上的port

for port in port\_list:

if switch.switch\_name + '-' + str(port) not in backup\_name:

g.remove\_edge(switch.switch\_name, switch.switch\_name + '-' + str(port))

##已经删除了switch对应的节点

## domain\_edge\_list 为域边缘的端口

# 域边缘端口不参与计算，不用管

# 此switch 对应switch-port edge已经删除

##对应交换机

##计算对应的rpc

for port in port\_list:

port\_name = switch.switch\_name + '-' + str(port)

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

##此时不为边缘端口且不为备份端口需要进行计算

min\_num = 9999999

Designated\_port\_name = str()

for j in target\_switch.port:

port\_name2 = target\_switch.switch\_name + '-' + str(j)

p1 = rpc(port\_name, port\_name2,g)

if min\_num > p1:

min\_num = p1

print('{}{}为{}'.format(port\_name, port\_name2, min\_num), file = fp)

# 更优秀

short\_path = path(port\_name, port\_name2,g)

Designated\_port\_name = short\_path[1

##此时我们需要进行更新BPDU

g.nodes[port\_name]['IRPC'] = min\_num

g.nodes[port\_name]['Designated\_BID'] = find\_designated\_bridge(Designated\_port\_name)

g.nodes[port\_name]['Designated\_PID'] = find\_designated\_pid(Designated\_port\_name)

g.nodes[port\_name]['received\_PID'] = find\_self\_pid(port\_name)

for port in port\_list:

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

g.add\_edge(switch.switch\_name, switch.switch\_name + '-' + str(port), cost=0,

region\_name=switch.region\_name)

print('add {} {}'.format(switch.switch\_name, switch.switch\_name + '-' + str(port)), file = fp)

print('--------域BPDU更新完毕--------开始选举根节点', file = fp)

# region\_switch\_list 域内switch列表 域内选举

for switch in region\_switch\_list:

# 每个switch均有

if switch.bid == region\_root\_bid:

target\_switch = switch ##域内根桥

continue

##开始选举

rootport = ''

for i in switch.port:

port\_name = switch.switch\_name + '-' + str(i)

if port\_name in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

rootport= switch.switch\_name + '-' + str(i)

break

for i in switch.port:

port\_name = switch.switch\_name + '-' + str(i)

if port\_name in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

if g.nodes[port\_name]['IRPC'] < g.nodes[rootport]['IRPC']:

rootport = port\_name

elif g.nodes[port\_name]['IRPC'] > g.nodes[rootport]['IRPC']:

pass

elif g.nodes[port\_name]['IRPC'] == g.nodes[rootport]['IRPC']:

if g.nodes[port\_name]['Designated\_BID'] < g.nodes[rootport]['Designated\_BID']:

rootport = port\_name

elif g.nodes[port\_name]['Designated\_BID'] == g.nodes[rootport]['Designated\_BID']:

if g.nodes[port\_name]['Designated\_PID'] < g.nodes[rootport]['Designated\_PID']:

rootport = port\_name

elif g.nodes[port\_name]['Designated\_PID'] == g.nodes[rootport]['Designated\_PID']:

if g.nodes[port\_name]['received\_PID'] < g.nodes[rootport]['received\_PID']:

rootport = port\_name

g.nodes[rootport]['Port\_status'] = 'Root Port'

#选举指定端口

print("--------域内ROOT PORT选拔完毕---------", file = fp)

for node in g.nodes(data=True):

print(node, file = fp)

print("-------根据选拔跟端口信息进行更新配置BPDU信息选拔DP-------", file = fp)

for switch in region\_switch\_list:

if switch.bid == region\_root\_bid:

target\_switch = switch ##域内根桥

continue

#开始选举DP端口

#switch 不是域根

port\_list = switch.port

min\_cost = 999999

for port in port\_list:

port\_name = switch.switch\_name + '-' + str(port)

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

if g.nodes[port\_name]['Port\_status'] == 'domain\_edge\_port':

# 域间的不进行更新

continue

if min\_cost > g.nodes[port\_name]['IRPC']:

min\_cost = g.nodes[port\_name]['IRPC']

# 更新出switch中最小的

for port in port\_list:

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

port\_name = switch.switch\_name + '-' + str(port)

if g.nodes[port\_name]['Port\_status'] == 'domain\_edge\_port':

# 域间的不进行更新

continue

g.nodes[port\_name]['IRPC'] = min\_cost

g.nodes[port\_name]['Designated\_BID'] = switch.bid

g.nodes[port\_name]['Designated\_PID'] = g.nodes[port\_name]['PID']

for node in g.nodes(data=True):

print(node, file = fp)

## 对于每个edge选拔出一个指定端口

DP\_list = list()

for port in target\_switch.port:

portName = target\_switch.switch\_name + '-' + str(port)

if g.nodes[portName]['Port\_status'] == 'domain\_edge\_port':

# 不更新

continue

DP\_list.append(portName)

# DP\_list存储着域内为DP的端口

for edge in g.edges(data=True):

if (edge[2]['cost'] == 0):

continue

print(edge, file = fp)

for edge in g.edges(data=True):

if edge[0] in DP\_list or edge[1] in DP\_list or edge[2]['cost'] == 0:

# 指定端口已经选举

continue

if g.nodes[edge[0]]['IRPC'] < g.nodes[edge[1]]['IRPC']:

g.nodes[edge[0]]['Port\_status'] = 'DP'

elif g.nodes[edge[0]]['IRPC'] > g.nodes[edge[1]]['IRPC']:

g.nodes[edge[1]]['Port\_status'] = 'DP'

elif g.nodes[edge[0]]['IRPC'] == g.nodes[edge[1]]['IRPC']:

if g.nodes[edge[0]]['Designated\_BID'] < g.nodes[edge[1]]['Designated\_BID']:

g.nodes[edge[0]]['Port\_status'] = 'DP'

elif g.nodes[edge[0]]['Designated\_BID'] > g.nodes[edge[1]]['Designated\_BID']:

g.nodes[edge[1]]['Port\_status'] = 'DP'

else:

if g.nodes[edge[0]]['Designated\_PID'] < g.nodes[edge[1]]['Designated\_PID']:

g.nodes[edge[0]]['Port\_status'] = 'DP'

elif g.nodes[edge[0]]['Designated\_PID'] > g.nodes[edge[1]]['Designated\_PID']:

g.nodes[edge[1]]['Port\_status'] = 'DP'

print("--------DP端口选择完毕--------", file = fp)

region\_port = list()

for switch in region\_switch\_list:

for port in switch.port:

portName = switch.switch\_name + '-' + str(port)

region\_port.append((portName))

print(region\_port, file = fp)

for edge in g.edges(data=True):

if edge[0] in region\_port and edge[1] in region\_port and edge[2]['cost'] != 0:

if g.nodes[edge[0]]['Port\_status'] == 'None':

g.nodes[edge[0]]['Port\_status'] = 'AP'

if g.nodes[edge[1]]['Port\_status'] == 'None':

g.nodes[edge[1]]['Port\_status'] = 'AP'

print("--------AP端口选择完毕--------", file = fp)

#恢复

def recover\_backup\_edge(backup\_del):

for i in backup\_del:

print(list1, file = fp)

g.add\_edge(i[0], i[1], cost=i[2]['cost'], region\_name=i[2]['region\_name'])

def recover\_backup\_node(backup):

for port\_data in backup:

g.add\_node(port\_data[0], PID=port\_data[1]['PID'], Root\_bridge\_ID=port\_data[1]['Root\_bridge\_ID'], ERPC=0, \

Region\_bridge\_ID=port\_data[1]['Region\_bridge\_ID'], IRPC=0, Designated\_BID=-1, Designated\_PID=-1,

received\_PID=-1, Port\_status=port\_data[1]['Port\_status'], \

switch\_bid=port\_data[1]['switch\_bid'], portID=port\_data[1]['portID'],

Port\_priority=port\_data[1]['Port\_priority'],VLAN = 'backup')

g.nodes[port\_data[0]]['Region\_bridge\_ID'] = g.nodes[port\_data[0][0:4]]['Region\_bridge\_ID']

# 恢复 backup堵塞端口

recover\_backup\_edge(backup\_del)

recover\_backup\_node(backup)

#恢复 域间线路 为后面的计算做好准备

for edge in edge\_list:

g.add\_edges\_from([ (edge[0],edge[1],edge[2]) ])

print("-------开始计算ERPC---------", file = fp)

for region in region\_dict:

switch\_region\_list = region\_dict[region]

print(region, switch\_region\_list, file = fp)

switch\_bid = list()

for switch\_name in switch\_region\_list:

switch\_bid.append(find\_bid(switch\_name))

region\_root\_bid = min(switch\_bid)

##目前可以找到每一个域内对应switch的bid 和 域的名字

region\_root\_name = find\_name(region\_root\_bid)

region\_switch = find\_switch(region\_root\_name)

#后期需要进行函数修改

print(region\_switch.switch\_name, file = fp)

region\_switch\_list = list()

for switch in switch\_list:

if switch.bid not in switch\_bid:

continue

if switch.bid in switch\_bid:

region\_switch\_list.append(switch)

root\_bridge = find\_switch(root\_bridge\_name #后期需要进行函数修改

print(root\_bridge.switch\_name, file = fp)

#如果总根桥位于域中则为0

if root\_bridge == region\_switch :

for switch\_name in switch\_region\_list:

g.nodes[switch\_name]['ERPC'] = 0

switch = find\_switch(switch\_name)

for port in switch.port:

g.nodes[switch.switch\_name + '-' + str(port)]['ERPC'] = 0

else:

#计算ERPC

target\_port = root\_bridge.switch\_name + '-' + str(root\_bridge.port[0])

source\_port = region\_switch.switch\_name + '-' + str(region\_switch.port[0])

r = rpc(target\_port,source\_port,g)

for switch\_name in switch\_region\_list:

g.nodes[switch\_name]['ERPC'] = r

switch = find\_switch(switch\_name)

for port in switch.port:

g.nodes[switch.switch\_name + '-' + str(port)]['ERPC'] = r

#目前ERPC生成完毕

print('----------开始选拔master端口-------------', file = fp)

for edge in g.edges(data = True):

print(edge, file = fp)

for node in g.nodes(data=True):

print(node, file = fp)

for node in g.nodes(data=True):

print(node[0], node[1]['Port\_status'], file = fp)

print('-----IST及 MSTI0 生成完毕---------', file = fp)

print("---------开始生成CST--------", file = fp)

#CIST 的计算只计算机只计算域间的cost不计算域内的cost edgeport 可能成为AP端口被堵塞

#master 端口位于域到总根最近的端口,每个域有一个

#将每个域看作一个switch 域间连接为switch连接

# EPRC为 域根到达总根的长度

G = nx.Graph()

for region in region\_dict:

switch\_region\_list = region\_dict[region]

print(region, switch\_region\_list, file = fp)

switch\_bid = list()

for switch\_name in switch\_region\_list:

switch\_bid.append(find\_bid(switch\_name))

region\_root\_bid = min(switch\_bid)

##目前可以找到每一个域内对应switch的bid 和 域的名字

region\_root\_name = find\_name(region\_root\_bid)

region\_switch\_list = list()

for switch in switch\_list:

if switch.bid not in switch\_bid:

continue

if switch.bid in switch\_bid:

region\_switch\_list.append(switch)

region\_port = list()

for switch in region\_switch\_list:

for port in switch.port:

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

region\_port.append(switch.switch\_name + '-' + str(port))

#查找其中对应的edge——port

del\_list = list()

for port\_name in region\_port:

if g.nodes[port\_name]['Port\_status'] != 'domain\_edge\_port':

del\_list.append(port\_name)

for port in del\_list:

region\_port.remove(port)

print(region, file = fp)

print(region\_port, file = fp)

print("-----", file = fp)

G.add\_node(region,BID = region\_root\_bid,RPC = 0, Port\_status = 'region')

for node in region\_port:

G.add\_node(node, BID = g.nodes[node]['switch\_bid'],RPC = 0, Port\_status = g.nodes[node]['Port\_status'])

G.add\_edge(region,node,cost = 0)

#此时 region——port中只有对应的edge'端口了

#对于除了第一个域每一个域均需要选出一个master端口，我们应该怎么选取呢

#添加域间连线

for edge in edge\_list:

G.add\_edge(edge[0],edge[1],cost = edge[2]['cost'])

print('---------开始计算域间MASTER端口-------', file = fp)

print(root\_bridge.region\_name, file = fp)

for node in G.nodes(data = True):

if node[0] == root\_bridge.region\_name:

continue

print(node, file = fp)

G.nodes[node[0]]['RPC'] = rpc(root\_bridge.region\_name,node[0],G)

short\_path = path(node[0],root\_bridge.region\_name,G)

if node[0] in region\_dict:

master\_name = short\_path[1]

region\_name = short\_path[0]

g.nodes[master\_name]['Port\_status'] = g.nodes[master\_name]['Port\_status'] + ' and ' + 'master'

# 开始在域间进行选举了啊对于域间的edge进行选举DP端口 d\_edge and xx

print("------------", file = fp)

for edge in G.edges(data=True):

print(edge, file = fp)

print("------", file = fp)

for node in g.nodes(data = True):

print(node[0], node[1]['Port\_status'], file = fp)

print("------------------------------------", file = fp)

def find\_switch\_via\_id(bid,switch\_list):

for switch in switch\_list:

if switch.bid == bid:

return switch

for edge in edge\_list:

node1 = edge[0]

node2 = edge[1]

port\_switch1 = find\_switch\_via\_id(g.nodes[node1]['switch\_bid'],switch\_list)

port\_switch2 = find\_switch\_via\_id(g.nodes[node2]['switch\_bid'],switch\_list)

print(port\_switch1.switch\_name, file = fp)

print(port\_switch2.switch\_name, file = fp)

if port\_switch1.region\_name == root\_bridge.region\_name or port\_switch2.region\_name == root\_bridge.region\_name:

#此时对应的边为主域的某一边

if port\_switch1.region\_name == root\_bridge.region\_name:

g.nodes[node1]['Port\_status'] = g.nodes[node1]['Port\_status'] + ' and ' +'DP'

if port\_switch2.region\_name == root\_bridge.region\_name:

g.nodes[node2]['Port\_status'] = g.nodes[node2]['Port\_status'] + ' and ' + 'DP'

##此时 master端口就是对应的根端口，我们不用从新选择了

#对于不是的这种情况我们的判断

#由于域根ID肯定不同

else:

if g.nodes[node1]['ERPC'] < g.nodes[node2]['ERPC']:

g.nodes[node1]['Port\_status'] = g.nodes[node1]['Port\_status'] + ' and ' + 'DP'

elif g.nodes[node1]['ERPC'] > g.nodes[node2]['ERPC']:

g.nodes[node2]['Port\_status'] = g.nodes[node2]['Port\_status'] + ' and ' + 'DP'

else:

#相当

if g.nodes[node1]['Region\_bridge\_ID'] < g.nodes[node2]['Region\_bridge\_ID']:

g.nodes[node1]['Port\_status'] = g.nodes[node1]['Port\_status'] + ' and ' + 'DP'

elif g.nodes[node1]['Region\_bridge\_ID'] >= g.nodes[node2]['Region\_bridge\_ID']:

g.nodes[node2]['Port\_status'] = g.nodes[node2]['Port\_status'] + ' and ' + 'DP'

#DP端口选择完毕

for edge in edge\_list:

node1 = edge[0]

node2 = edge[1]

if g.nodes[node1]['Port\_status'] == 'domain\_edge\_port':

g.nodes[node1]['Port\_status'] = g.nodes[node1]['Port\_status'] + ' and' + ' AP'

if g.nodes[node2]['Port\_status'] == 'domain\_edge\_port':

g.nodes[node2]['Port\_status'] = g.nodes[node2]['Port\_status'] + ' and' + ' AP'

for node in g.nodes(data = True):

print(node[0],node[1]['Port\_status'], file = fp)

print('--------------完整的一棵CIST生成完毕-----------------', file = fp)

print('--------------域内的多生成树实例选取------------------', file = fp)

for node in g.nodes(data = True):

print(node, file = fp)

#g 存储着对应的node结构

print('-------------------------------------------------', file = fp)

#每一个MSTI的实例相当于一副新的图片，因此我们可以采用新的拓扑结构

for node in g.nodes(data = True):

print(node)

for edge in g.edges(data = True):

print(edge)

##CIST 写入文件

print("CIST 端口角色：", file = fp2)

for node in g.nodes(data = True):

print(node[0] + " " + "role:" + " " + node[1]['Port\_status'], file = fp2)

## 我们需要删除其中的链路 当然由于链路最优·，因此我们也可以不加删除对应的链路来进行处理

print('-----------')

## pc 与 switch 的映射

fwd\_list = list() ## 存储路由表项的类

pc\_list = dict() # 存储switch\_name : pc\_mac —— 存储每个switch——name 对应的PC\_mac地址

allPc = list() #存储所有的路由表项 存储图结构所有的PC值

mac\_node\_map = dict() #存储 mac ： port\_ID mac1 mac2 mac3 -------对应的PC1 PC2 PC3

mac\_switch\_map = dict()

for switch in switch\_list:

fwd\_list.append(fwd\_table(switch.switch\_name)) #根据name创建对应的fwd类：对应每个switch均有一个

str1 = str(switch.mac\_list)

list1 = str1.split(',') ## switch连接的对应PC的mac地址列表

list1 = [str(i) for i in list1] # list1 存储对应的mac地址列表

# mac列表相当于为 连接PC的列表

# 现在添加PC节点

pc\_list[switch.switch\_name] = list1 #存储所有的switch\_name 对应的 PC（mac地址）

for pc in list1:

allPc.append(pc)#名字是mac1//mac2这种的mac地址

mac\_switch\_map[pc] = switch.switch\_name

mac = pc

pc = pc[3:]

mac\_node\_map[mac] = 'pc' + str(pc)

g.add\_node('pc' + str(pc), PRC = -1, Root\_bridge\_ID = -1, ERPC = -1, Region\_bridge\_ID = -1,IRPC = -1,Designated\_BID = -1, Designated\_PID = -1, received\_PID = -1, Port\_status = 'PC', switch\_bid = -1,portID = -1, Port\_priority = -1, VLAN = 'none' )

g.add\_edge(switch.switch\_name, 'pc' + str(pc), weight = 0, region\_name = "pc\_edge")

# pc1 pc2 pc3 pc4 对应的就是 对应的PC节点 根据 mac地址 可以找到对应的节点那么

print('------已经添加了所有的节点及边---------')

print("----------增加PC节点后------------")

for edge in g.edges(data = True):

print(edge

for node in g.nodes(data = True):

print(node)

# 首先计算自己的路由表项

#每个switch拥有自己的路由表项

for switch in fwd\_list:

print(switch.name + "路由表项")

switch\_name = switch.name

##添加自己的路由表项

port\_list = pc\_list[switch\_name]

#每一步存储的都是自己的路由表项

for pc in port\_list:

#添加自己的路由表项

mac = pc

node = mac\_node\_map[mac] # 存储mac地址对应的 PC端口的名字

index = [mac, 0, node] ## 0 表示的是CIST

switch.fwd\_index.append(index)

print(index)

for pc\_des in allPc:

#需要对所有的PC均进行一遍更新

if pc\_des in port\_list:

continue

#自己已经更新过了

else:

destination = mac\_node\_map[pc\_des] ##端口的名字

#自己的destination 到达刚刚产生的节点的最短路径

path1 = path(switch\_name, destination,g)

index = [pc\_des, 0, path1[1]]

switch.fwd\_index.append(index)

print(index)

##添加其他的所有的mac地址

print("CIST 域内的路由表项添加完毕")

#开始添加域内的路由表项

print("done")

#删除域内switch与端口互联的直线

def del\_internal\_edge(G,switch):

remove\_edge\_list = list()

for edge in G.edges(data = True):

if edge[2]['cost'] ==0 and ( edge[1] == switch.switch\_name or edge[2] == switch.switch\_name):

remove\_edge\_list.append(edge)

G.remove\_edges\_from(remove\_edge\_list)

return remove\_edge\_list

#恢复边

def recover\_internal\_edge(G,remove\_edge\_list):

G.add\_edges\_from(remove\_edge\_list)

#删除VLAN不能传播的链路

def remove\_VLAN\_edge(G, edge, vlan\_range):

vlan\_range = vlan\_range.split('-')

#存储对应的上限和下限

vlan\_range = list(map(int, vlan\_range))

port1 = edge[0]

port2 = edge[1]

# vlan1 2 为edge 端口两端对应的vlan通过情况

vlan1 = G.nodes[port1]['VLAN']

vlan2 = G.nodes[port2]['VLAN']

if vlan1 == 'none' or vlan2 == 'none' :

return 'true'

vlan1 = vlan1.split('-')

vlan2 = vlan2.split('-')

vlan1 = list(map(int, vlan1))

vlan2 = list(map(int, vlan2))

#vlan 为edge所能通过的vlan

#取可以通过的子集

vlan = list()

vlan.append(max(vlan1[0], vlan2[0]))

vlan.append(min(vlan1[1], vlan2[1]))

if vlan[0]<= vlan\_range[0] and vlan[1] >= vlan\_range[1]:

return 'true'

else:

return 'false'

#存储可以通过的vlan

#如果可以通过则返回对应的vlan，否则返回不能通过的vlan格式

print(region\_dict, file = fp)

print('----------------', file = fp)

raw\_data = pd.read\_excel(file\_path, header=0, sheet\_name='VLAN')

data = raw\_data.values

for i in range(data.shape[0]):

port\_data = data[i]

print(port\_data, file = fp)

region\_name = port\_data[0]

MSTI\_name = port\_data[1]

vlan\_range = port\_data[2]

MSTI\_bridge = port\_data[3]

root\_bridge = find\_switch(MSTI\_bridge)

target\_switch = root\_bridge

# 优先级置为0

MSTI\_BID = root\_bridge.mac

switch\_name\_list = region\_dict[region\_name]

print(switch\_name\_list, file = fp)

switch\_list1 = list()

switch\_portlist = list()

for switch\_name in switch\_name\_list:

switch\_list1.append(find\_switch(switch\_name))

for switch in switch\_list1:

for port in switch.port:

port\_name = switch.switch\_name + '-' + str(port)

switch\_portlist.append(port\_name)

#在新的图形中添加消息 每次会重新添加一次

region\_g = nx.Graph()

root\_switch\_bid = 0

list\_node = list()

for node in g.nodes(data = True):

if node[0] in switch\_name\_list:

list\_node.append(node)

if node[0] in switch\_portlist:

list\_node.append(node)

region\_g.add\_nodes\_from(list\_node)

backup\_node = list()

for node in region\_g.nodes(data = True):

#修改端口角色的状态：

region\_g.nodes[node[0]]['Root\_bridge\_ID'] = MSTI\_BID

region\_g.nodes[node[0]]['Region\_bridge\_ID'] = MSTI\_BID

region\_g.nodes[node[0]]['IRPC'] = 0

region\_g.nodes[node[0]]['ERPC'] = 0

region\_g.nodes[node[0]]['Designated\_BID'] = node[1]['switch\_bid']

region\_g.nodes[node[0]]['Designated\_PID'] = node[1]['PID']

region\_g.nodes[node[0]]['received\_PID'] = 0

if region\_g.nodes[node[0]]['Port\_status'] == 'backup':

backup\_node.append(node[0])

continue

else:

if region\_g.nodes[node[0]]['Port\_status'][0:6] == 'domain':

region\_g.nodes[node[0]]['Port\_status'] = 'domain\_edge\_port'

elif region\_g.nodes[node[0]]['Port\_status'] == 'switch':

continue

else :

region\_g.nodes[node[0]]['Port\_status'] = 'none'

# 添加边

#这里删除不能转播此VLAN的边

edge\_list = list()

for edge in g.edges(data=True):

if edge[2]['region\_name'] == region\_name and edge[0] not in backup\_name and edge[1] not in backup\_name and edge[2]['region\_name'] != 'pc\_edge':

edge\_list.append(edge)

region\_g.add\_edges\_from(edge\_list)

for edge in region\_g.edges(data = True):

res = remove\_VLAN\_edge(region\_g, edge, vlan\_range)

if res == 'false':

region\_g.remove\_edge(edge[0],edge[1])

else:

continue

#开始选拔端口

#阻塞备份端口的节点

print('{}域 MSTI{} 根桥为 {}'.format( region\_name ,MSTI\_name, MSTI\_bridge), file = fp)

#根桥的所有非域间端口变成 DP

for node in region\_g.nodes(data = True):

if root\_bridge.bid == node[1]['switch\_bid'] and node[1]['Port\_status'] != 'domain\_edge\_port' and node[1]['Port\_status'] != 'backup' and node[1]['Port\_status'] != 'switch':

region\_g.nodes[node[0]]['Port\_status'] = 'DP'

#计算根端口

for switch in switch\_list1:

if switch == root\_bridge:

continue

else:

#断开端口间的连线

remove\_edge\_list = del\_internal\_edge(region\_g, switch)

port\_list = switch.port

for port in port\_list:

port\_name = switch.switch\_name + '-' + str(port)

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

#此时不为边缘端口且不为备份端口需要进行计算

min\_num = 9999999

Designated\_port\_name = str()

for j in target\_switch.port:

port\_name2 = target\_switch.switch\_name + '-' + str(j)

if port\_name2 in backup\_name :

continue

if port\_name2 in domain\_edge\_list:

continue

p1 = rpc(port\_name, port\_name2, region\_g)

if min\_num > p1:

min\_num = p1

print('{}{}为{}'.format(port\_name, port\_name2, min\_num), file = fp)

# 更优秀

short\_path = path(port\_name, port\_name2, region\_g)

print(short\_path, file = fp)

Designated\_port\_name = short\_path[1]

#此时我们需要进行更新BPDU

region\_g.nodes[port\_name]['IRPC'] = min\_num

region\_g.nodes[port\_name]['Designated\_BID'] = find\_designated\_bridge(Designated\_port\_name)

region\_g.nodes[port\_name]['Designated\_PID'] = find\_designated\_pid(Designated\_port\_name)

region\_g.nodes[port\_name]['received\_PID'] = find\_self\_pid(port\_name)

recover\_internal\_edge(region\_g,remove\_edge\_list)

#此时内部的IRPC更新完毕，为非根桥依次选取根端口：

print('--------------------域内MSTI BPDU更新完毕开始选取根端口-----------------', file = fp)

for switch in switch\_list1:

# 对于域内根桥已经选取结束

if switch == root\_bridge:

continue

#开始选举

rootport = ''

#找到第一个优先级最高的节点

for i in switch.port:

port\_name = switch.switch\_name + '-' + str(i)

if port\_name in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

rootport = port\_name

break

for i in switch.port:

port\_name = switch.switch\_name + '-' + str(i)

if port\_name in backup\_name:

continue

if port\_name in domain\_edge\_list:

continue

if region\_g.nodes[port\_name]['IRPC'] < region\_g.nodes[rootport]['IRPC']:

rootport = port\_name

elif region\_g.nodes[port\_name]['IRPC'] > region\_g.nodes[rootport]['IRPC']:

pass

elif region\_g.nodes[port\_name]['IRPC'] == region\_g.nodes[rootport]['IRPC']:

if region\_g.nodes[port\_name]['Designated\_BID'] < region\_g.nodes[rootport]['Designated\_BID']:

rootport = port\_name

elif region\_g.nodes[port\_name]['Designated\_BID'] == region\_g.nodes[rootport]['Designated\_BID']:

if region\_g.nodes[port\_name]['Designated\_PID'] < region\_g.nodes[rootport]['Designated\_PID']:

rootport = port\_name

elif region\_g.nodes[port\_name]['Designated\_PID'] == region\_g.nodes[rootport]['Designated\_PID']:

if region\_g.nodes[port\_name]['received\_PID'] < region\_g.nodes[rootport]['received\_PID']:

rootport = port\_name

region\_g.nodes[rootport]['Port\_status'] = 'Root Port'

print("--------域内ROOT PORT选拔完毕---------", file = fp)

print('--------开始选取DP端口----------------', file = fp)

print("-------根据选拔跟端口信息进行更新配置BPDU信息选拔DP-------", file = fp)

for switch in switch\_list1:

if switch == root\_bridge:

continue

#开始选举DP端口

#switch 不是域根

port\_list = switch.port

min\_cost = 999999

#更新switch到域根最小的IRPC

for port in port\_list:

port\_name = switch.switch\_name + '-' + str(port)

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

if region\_g.nodes[port\_name]['Port\_status'] == 'domain\_edge\_port':

# 域间的不进行更新

continue

if min\_cost > region\_g.nodes[port\_name]['IRPC']:

min\_cost = region\_g.nodes[port\_name]['IRPC']

# 更新出switch中最小的

for port in port\_list:

if switch.switch\_name + '-' + str(port) in backup\_name:

continue

port\_name = switch.switch\_name + '-' + str(port)

if region\_g.nodes[port\_name]['Port\_status'] == 'domain\_edge\_port':

# 域间的不进行更新

continue

region\_g.nodes[port\_name]['IRPC'] = min\_cost

region\_g.nodes[port\_name]['Designated\_BID'] = switch.bid

region\_g.nodes[port\_name]['Designated\_PID'] = region\_g.nodes[port\_name]['PID']

print('------------域内BPDU更新完毕，开始选举DP端口------------', file = fp)

DP\_list = list()

for port in root\_bridge.port:

portName = root\_bridge.switch\_name + '-' + str(port)

if region\_g.nodes[portName]['Port\_status'] == 'domain\_edge\_port':

# 不更新

continue

if region\_g.nodes[portName]['Port\_status'] == 'backup':

continue

DP\_list.append(portName)

# DP\_list存储着域内为DP的端口

for edge in region\_g.edges(data=True):

if edge[0] in DP\_list or edge[1] in DP\_list or edge[2]['cost'] == 0:

# 指定端口已经选举

continue

if region\_g.nodes[edge[0]]['IRPC'] < region\_g.nodes[edge[1]]['IRPC']:

region\_g.nodes[edge[0]]['Port\_status'] = 'DP'

elif region\_g.nodes[edge[0]]['IRPC'] > region\_g.nodes[edge[1]]['IRPC']:

region\_g.nodes[edge[1]]['Port\_status'] = 'DP'

elif region\_g.nodes[edge[0]]['IRPC'] == region\_g.nodes[edge[1]]['IRPC']:

if region\_g.nodes[edge[0]]['Designated\_BID'] < region\_g.nodes[edge[1]]['Designated\_BID']:

region\_g.nodes[edge[0]]['Port\_status'] = 'DP'

elif region\_g.nodes[edge[0]]['Designated\_BID'] > region\_g.nodes[edge[1]]['Designated\_BID']:

region\_g.nodes[edge[1]]['Port\_status'] = 'DP'

else:

if region\_g.nodes[edge[0]]['Designated\_PID'] < region\_g.nodes[edge[1]]['Designated\_PID']:

region\_g.nodes[edge[0]]['Port\_status'] = 'DP'

elif region\_g.nodes[edge[0]]['Designated\_PID'] > region\_g.nodes[edge[1]]['Designated\_PID']:

region\_g.nodes[edge[1]]['Port\_status'] = 'DP'

for node in region\_g.nodes(data = True):

if node[1]['Port\_status'] == 'none':

region\_g.nodes[node[0]]['Port\_status'] = 'AP'

print('------AP端口选择完毕----------', file = fp)

print('-------下方为所有node对应的状态---------', file = fp)

print(' ',file = fp2)

print('{}域 MSTI{} VLAN端口角色划分如下'.format(region\_name, MSTI\_name, MSTI\_bridge), file = fp)

print('---------------------------------------------------------', file = fp2)

print('{}域 MSTI{} VLAN端口角色划分如下'.format(region\_name, MSTI\_name, MSTI\_bridge), file=fp2)

for edge in region\_g.edges(data=True):

print(edge, file = fp)

print('-----------------------------', file = fp)

for node in region\_g.nodes(data = True):

print(node, file = fp)

print(node[0] + " " + "role:" + " " + node[1]['Port\_status'], file=fp2)

print('-----------------------------', file = fp)

#仅仅包含其bid

print('------', file = fp)

for edge in region\_g.edges(data=True):

print(edge)

print('-----------------------------')

for node in region\_g.nodes(data = True):

print(node)

print('-----------------------------')

#vlan\_range 表示对应的vlan号为多少

#现在需要给每一个switch添加MSTI的域内表项

#需要找到域内的PC mac地址

region\_mac = list()

## 删除对应的线路：

del\_edge = list()

for edge in region\_g.edges(data = True):

if region\_g.nodes[edge[0]]['Port\_status'] == 'AP' or region\_g.nodes[edge[1]]['Port\_status'] == 'AP' :

del\_edge.append(edge)

for edge in del\_edge:

region\_g.remove\_edge(edge[0], edge[1])

#记录域内所有的mac地址

all\_mac = list()

for switch in switch\_list1:

str1 = str(switch.mac\_list)

list1 = str1.split(',') ## switch连接的对应PC的mac地址列表

list1 = [str(i) for i in list1] # list1 存储对应的mac地址列表

for mac in list1:

all\_mac.append(mac)

port\_list = list()

for switch in fwd\_list:

# 只添加自己域内的 index

if switch.name not in switch\_name\_list:

continue

print(switch.name + "路由表项" + "更新 :")

switch\_name = switch.name

##添加自己的路由表项

port\_list = pc\_list[switch\_name]

## index = [mac, VLAN\_ID, port\_ex]

# 每一步存储的都是自己的路由表项

# 只用更新域内的MSTI

# 首先添加自己的路由表项

for pc in port\_list:

mac = pc

node = mac\_node\_map[mac] # 存储mac地址对应的 PC端口的名字

index = [mac, vlan\_range, node] ## 0 表示的是CIST

switch.fwd\_index.append(index)

print(index)

# 自己的路由项添加完毕

for pc\_des in all\_mac:

# 需要对所有的PC均进行一遍更新

if pc\_des in port\_list:

continue

# 自己已经更新过了

else:

destination = mac\_switch\_map[pc\_des] ##端口的名字

# 自己的destination 到达刚刚产生的节点的最短路径

path1 = path(switch\_name, destination, region\_g)

index = [pc\_des, vlan\_range, path1[1]]

switch.fwd\_index.append(index)

print(index)

##添加其他的所有的mac地址

#region\_g 中的线路

# 恢复对应的edge

region\_g.add\_edges\_from(del\_edge)

print("CIST 域内的路由表项添加完毕")

fp1 = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/mac\_table.txt', "w")

for switch in fwd\_list:

list\_fwd = switch.fwd\_index

n = len(list\_fwd)

print(switch.name + " MAC表如下：", file = fp1)

for i in range(n):

print(list\_fwd[i], file = fp1)

print(" ", file = fp1)

fp1.close()

fp.close()

fp2.close()

交互界面设计：

主界面：

from PyQt5 import QtCore, QtGui, QtWidgets

class Ui\_Form(object):

def setupUi(self, Form):

Form.setObjectName("Form")

Form.resize(647, 357)

self.tabWidget = QtWidgets.QTabWidget(Form)

self.tabWidget.setGeometry(QtCore.QRect(0, 50, 641, 261))

self.tabWidget.setToolTip("")

self.tabWidget.setObjectName("tabWidget")

self.tab = QtWidgets.QWidget()

self.tab.setObjectName("tab")

self.lineEdit = QtWidgets.QLineEdit(self.tab)

self.lineEdit.setGeometry(QtCore.QRect(60, 80, 541, 31))

self.lineEdit.setObjectName("lineEdit")

self.horizontalLayoutWidget = QtWidgets.QWidget(self.tab)

self.horizontalLayoutWidget.setGeometry(QtCore.QRect(60, 130, 541, 81))

self.horizontalLayoutWidget.setObjectName("horizontalLayoutWidget")

self.horizontalLayout = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget)

self.horizontalLayout.setContentsMargins(0, 0, 0, 0)

self.horizontalLayout.setObjectName("horizontalLayout")

self.pushButton\_2 = QtWidgets.QPushButton(self.horizontalLayoutWidget)

self.pushButton\_2.setObjectName("pushButton\_2")

self.horizontalLayout.addWidget(self.pushButton\_2)

self.pushButton = QtWidgets.QPushButton(self.horizontalLayoutWidget)

self.pushButton.setObjectName("pushButton")

self.horizontalLayout.addWidget(self.pushButton)

self.label = QtWidgets.QLabel(self.tab)

self.label.setGeometry(QtCore.QRect(30, 10, 171, 61))

self.label.setStyleSheet("font: 25pt \".AppleSystemUIFont\";")

self.label.setObjectName("label")

self.tabWidget.addTab(self.tab, "")

self.tab\_2 = QtWidgets.QWidget()

self.tab\_2.setObjectName("tab\_2")

self.tabWidget\_2 = QtWidgets.QTabWidget(self.tab\_2)

self.tabWidget\_2.setGeometry(QtCore.QRect(10, 0, 631, 231))

self.tabWidget\_2.setObjectName("tabWidget\_2")

self.tab\_4 = QtWidgets.QWidget()

self.tab\_4.setToolTip("")

self.tab\_4.setObjectName("tab\_4")

self.tableWidget = QtWidgets.QTableWidget(self.tab\_4)

self.tableWidget.setGeometry(QtCore.QRect(140, 0, 311, 201))

self.tableWidget.setRowCount(20)

self.tableWidget.setObjectName("tableWidget")

self.tableWidget.setColumnCount(3)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget.setHorizontalHeaderItem(2, item)

self.tableWidget.verticalHeader().setVisible(False)

self.tabWidget\_2.addTab(self.tab\_4, "")

self.tab\_5 = QtWidgets.QWidget()

self.tab\_5.setObjectName("tab\_5")

self.tableWidget\_2 = QtWidgets.QTableWidget(self.tab\_5)

self.tableWidget\_2.setGeometry(QtCore.QRect(130, 10, 341, 201))

self.tableWidget\_2.setRowCount(40)

self.tableWidget\_2.setObjectName("tableWidget\_2")

self.tableWidget\_2.setColumnCount(2)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_2.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_2.setHorizontalHeaderItem(1, item)

self.tableWidget\_2.verticalHeader().setVisible(False)

self.tabWidget\_2.addTab(self.tab\_5, "")

self.tab\_6 = QtWidgets.QWidget()

self.tab\_6.setObjectName("tab\_6")

self.tableWidget\_3 = QtWidgets.QTableWidget(self.tab\_6)

self.tableWidget\_3.setGeometry(QtCore.QRect(160, 0, 311, 201))

self.tableWidget\_3.setRowCount(20)

self.tableWidget\_3.setObjectName("tableWidget\_3")

self.tableWidget\_3.setColumnCount(3)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_3.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_3.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_3.setHorizontalHeaderItem(2, item)

self.tableWidget\_3.verticalHeader().setVisible(False)

self.pushButton\_3 = QtWidgets.QPushButton(self.tab\_6)

self.pushButton\_3.setGeometry(QtCore.QRect(490, 135, 101, 41))

self.pushButton\_3.setObjectName("pushButton\_3")

self.tabWidget\_2.addTab(self.tab\_6, "")

self.tabWidget.addTab(self.tab\_2, "")

self.tab\_3 = QtWidgets.QWidget()

self.tab\_3.setObjectName("tab\_3")

self.tabWidget\_3 = QtWidgets.QTabWidget(self.tab\_3)

self.tabWidget\_3.setGeometry(QtCore.QRect(4, 0, 631, 241))

self.tabWidget\_3.setObjectName("tabWidget\_3")

self.tab\_7 = QtWidgets.QWidget()

self.tab\_7.setObjectName("tab\_7")

self.tableWidget\_4 = QtWidgets.QTableWidget(self.tab\_7)

self.tableWidget\_4.setGeometry(QtCore.QRect(60, 0, 511, 201))

self.tableWidget\_4.setRowCount(20)

self.tableWidget\_4.setObjectName("tableWidget\_4")

self.tableWidget\_4.setColumnCount(5)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_4.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_4.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_4.setHorizontalHeaderItem(2, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_4.setHorizontalHeaderItem(3, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_4.setHorizontalHeaderItem(4, item)

self.tableWidget\_4.verticalHeader().setVisible(False)

self.tabWidget\_3.addTab(self.tab\_7, "")

self.tab\_8 = QtWidgets.QWidget()

self.tab\_8.setObjectName("tab\_8")

self.tableWidget\_5 = QtWidgets.QTableWidget(self.tab\_8)

self.tableWidget\_5.setGeometry(QtCore.QRect(100, 0, 411, 211))

self.tableWidget\_5.setRowCount(40)

self.tableWidget\_5.setObjectName("tableWidget\_5")

self.tableWidget\_5.setColumnCount(4)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_5.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_5.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_5.setHorizontalHeaderItem(2, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_5.setHorizontalHeaderItem(3, item)

self.tableWidget\_5.verticalHeader().setVisible(False)

self.tabWidget\_3.addTab(self.tab\_8, "")

self.tab\_10 = QtWidgets.QWidget()

self.tab\_10.setObjectName("tab\_10")

self.tableWidget\_6 = QtWidgets.QTableWidget(self.tab\_10)

self.tableWidget\_6.setGeometry(QtCore.QRect(160, 0, 311, 211))

self.tableWidget\_6.setRowCount(30)

self.tableWidget\_6.setObjectName("tableWidget\_6")

self.tableWidget\_6.setColumnCount(3)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_6.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_6.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_6.setHorizontalHeaderItem(2, item)

self.tableWidget\_6.verticalHeader().setVisible(False)

self.tabWidget\_3.addTab(self.tab\_10, "")

self.tab\_9 = QtWidgets.QWidget()

self.tab\_9.setObjectName("tab\_9")

self.tableWidget\_7 = QtWidgets.QTableWidget(self.tab\_9)

self.tableWidget\_7.setGeometry(QtCore.QRect(110, 0, 411, 211))

self.tableWidget\_7.setRowCount(20)

self.tableWidget\_7.setObjectName("tableWidget\_7")

self.tableWidget\_7.setColumnCount(4)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_7.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_7.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_7.setHorizontalHeaderItem(2, item)

item = QtWidgets.QTableWidgetItem()

font = QtGui.QFont()

font.setPointSize(15)

item.setFont(font)

self.tableWidget\_7.setHorizontalHeaderItem(3, item)

self.tableWidget\_7.verticalHeader().setVisible(False)

self.pushButton\_4 = QtWidgets.QPushButton(self.tab\_9)

self.pushButton\_4.setGeometry(QtCore.QRect(520, 160, 101, 41))

self.pushButton\_4.setObjectName("pushButton\_4")

self.tabWidget\_3.addTab(self.tab\_9, "")

self.tabWidget.addTab(self.tab\_3, "")

self.label\_2 = QtWidgets.QLabel(Form)

self.label\_2.setGeometry(QtCore.QRect(70, 15, 531, 21))

self.label\_2.setText("")

self.label\_2.setObjectName("label\_2")

self.label\_3 = QtWidgets.QLabel(Form)

self.label\_3.setGeometry(QtCore.QRect(30, 10, 301, 21))

self.label\_3.setStyleSheet("font: 20pt \".AppleSystemUIFont\";")

self.label\_3.setObjectName("label\_3")

self.pushButton\_5 = QtWidgets.QPushButton(Form)

self.pushButton\_5.setGeometry(QtCore.QRect(540, 320, 101, 31))

self.pushButton\_5.setObjectName("pushButton\_5")

self.retranslateUi(Form)

self.tabWidget.setCurrentIndex(1)

self.tabWidget\_2.setCurrentIndex(2)

self.tabWidget\_3.setCurrentIndex(1)

QtCore.QMetaObject.connectSlotsByName(Form)

def retranslateUi(self, Form):

\_translate = QtCore.QCoreApplication.translate

Form.setWindowTitle(\_translate("Form", "Form"))

self.lineEdit.setPlaceholderText(\_translate("Form", "请输入文件路径"))

self.pushButton\_2.setText(\_translate("Form", "STP"))

self.pushButton.setText(\_translate("Form", "MSTP"))

self.label.setText(\_translate("Form", "文件路径："))

self.tabWidget.setTabText(self.tabWidget.indexOf(self.tab), \_translate("Form", "文件输入"))

item = self.tableWidget.horizontalHeaderItem(0)

item.setText(\_translate("Form", "Port"))

item = self.tableWidget.horizontalHeaderItem(1)

item.setText(\_translate("Form", "BID"))

item = self.tableWidget.horizontalHeaderItem(2)

item.setText(\_translate("Form", "Name"))

self.tabWidget\_2.setTabText(self.tabWidget\_2.indexOf(self.tab\_4), \_translate("Form", "Switch子表"))

item = self.tableWidget\_2.horizontalHeaderItem(0)

item.setText(\_translate("Form", "PID"))

item = self.tableWidget\_2.horizontalHeaderItem(1)

item.setText(\_translate("Form", "SPID"))

self.tabWidget\_2.setTabText(self.tabWidget\_2.indexOf(self.tab\_5), \_translate("Form", "Port\_子表"))

item = self.tableWidget\_3.horizontalHeaderItem(0)

item.setText(\_translate("Form", "Port1"))

item = self.tableWidget\_3.horizontalHeaderItem(1)

item.setText(\_translate("Form", "Port2"))

item = self.tableWidget\_3.horizontalHeaderItem(2)

item.setText(\_translate("Form", "Cost"))

self.pushButton\_3.setText(\_translate("Form", "输入完成"))

self.tabWidget\_2.setTabText(self.tabWidget\_2.indexOf(self.tab\_6), \_translate("Form", "Edge子表"))

self.tabWidget.setTabText(self.tabWidget.indexOf(self.tab\_2), \_translate("Form", "STP\_algorithm"))

item = self.tableWidget\_4.horizontalHeaderItem(0)

item.setText(\_translate("Form", "Port"))

item = self.tableWidget\_4.horizontalHeaderItem(1)

item.setText(\_translate("Form", "Priority"))

item = self.tableWidget\_4.horizontalHeaderItem(2)

item.setText(\_translate("Form", "mac"))

item = self.tableWidget\_4.horizontalHeaderItem(3)

item.setText(\_translate("Form", "name"))

item = self.tableWidget\_4.horizontalHeaderItem(4)

item.setText(\_translate("Form", "region\_name"))

self.tabWidget\_3.setTabText(self.tabWidget\_3.indexOf(self.tab\_7), \_translate("Form", "Switch子表"))

item = self.tableWidget\_5.horizontalHeaderItem(0)

item.setText(\_translate("Form", "PortID"))

item = self.tableWidget\_5.horizontalHeaderItem(1)

item.setText(\_translate("Form", "Port"))

item = self.tableWidget\_5.horizontalHeaderItem(2)

item.setText(\_translate("Form", "Priority"))

item = self.tableWidget\_5.horizontalHeaderItem(3)

item.setText(\_translate("Form", "VLAN"))

self.tabWidget\_3.setTabText(self.tabWidget\_3.indexOf(self.tab\_8), \_translate("Form", "Port子表"))

item = self.tableWidget\_6.horizontalHeaderItem(0)

item.setText(\_translate("Form", "Port1"))

item = self.tableWidget\_6.horizontalHeaderItem(1)

item.setText(\_translate("Form", "Port2"))

item = self.tableWidget\_6.horizontalHeaderItem(2)

item.setText(\_translate("Form", "Cost"))

self.tabWidget\_3.setTabText(self.tabWidget\_3.indexOf(self.tab\_10), \_translate("Form", "Edge子表"))

item = self.tableWidget\_7.horizontalHeaderItem(0)

item.setText(\_translate("Form", "Region\_name"))

item = self.tableWidget\_7.horizontalHeaderItem(1)

item.setText(\_translate("Form", "MSTI\_ID"))

item = self.tableWidget\_7.horizontalHeaderItem(2)

item.setText(\_translate("Form", "VLAN\_ID"))

item = self.tableWidget\_7.horizontalHeaderItem(3)

item.setText(\_translate("Form", "RootBridge"))

self.pushButton\_4.setText(\_translate("Form", "输入完成"))

self.tabWidget\_3.setTabText(self.tabWidget\_3.indexOf(self.tab\_9), \_translate("Form", "MSTI\_VLAN映射表"))

self.tabWidget.setTabText(self.tabWidget.indexOf(self.tab\_3), \_translate("Form", "MSTP/RSTP\_algorithm"))

self.label\_3.setText(\_translate("Form", "网络二层协议仿真工具"))

self.pushButton\_5.setText(\_translate("Form", "详细信息"))

二级页面代码：

from PyQt5 import QtCore, QtGui, QtWidgets

class Ui\_Form\_2(object):

def setupUi(self, Form):

Form.setObjectName("Form")

Form.resize(753, 527)

self.textBrowser = QtWidgets.QTextBrowser(Form)

self.textBrowser.setGeometry(QtCore.QRect(10, 380, 741, 141))

self.textBrowser.setObjectName("textBrowser")

self.textBrowser\_2 = QtWidgets.QTextBrowser(Form)

self.textBrowser\_2.setGeometry(QtCore.QRect(10, 40, 331, 311))

self.textBrowser\_2.setObjectName("textBrowser\_2")

self.label\_2 = QtWidgets.QLabel(Form)

self.label\_2.setGeometry(QtCore.QRect(0, 10, 181, 21))

self.label\_2.setStyleSheet("font: 15pt \".AppleSystemUIFont\";")

self.label\_2.setObjectName("label\_2")

self.label\_4 = QtWidgets.QLabel(Form)

self.label\_4.setGeometry(QtCore.QRect(10, 350, 181, 21))

self.label\_4.setStyleSheet("font: 15pt \".AppleSystemUIFont\";")

self.label\_4.setObjectName("label\_4")

self.label\_6 = QtWidgets.QLabel(Form)

self.label\_6.setGeometry(QtCore.QRect(390, 10, 181, 21))

self.label\_6.setStyleSheet("font: 15pt \".AppleSystemUIFont\";")

self.label\_6.setObjectName("label\_6")

self.textBrowser\_3 = QtWidgets.QTextBrowser(Form)

self.textBrowser\_3.setGeometry(QtCore.QRect(400, 40, 331, 311))

self.textBrowser\_3.setObjectName("textBrowser\_3")

self.retranslateUi(Form)

QtCore.QMetaObject.connectSlotsByName(Form)

def retranslateUi(self, Form):

\_translate = QtCore.QCoreApplication.translate

Form.setWindowTitle(\_translate("Form", "Form"))

self.label\_2.setText(\_translate("Form", "链路状态及端口角色："))

self.label\_4.setText(\_translate("Form", "打印日志："))

self.label\_6.setText(\_translate("Form", "MAC转发表："))

主页面业务流程代码：

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QMessageBox

from untitled import Ui\_Form

import second\_main

import xlwt # 负责写excel

import numpy as np

import os

def generateExcel\_mstp( list\_switch, list\_port, list\_edge, list\_msti):

list\_switch = np.array(list\_switch)

list\_port = np.array(list\_port)

list\_edge = np.array(list\_edge)

list\_msti = np.array(list\_msti)

filename = xlwt.Workbook(encoding = "utf-8") # 创建工作簿

sheet\_switch = filename.add\_sheet(u'switch', cell\_overwrite\_ok=True) # 创建sheet

sheet\_port = filename.add\_sheet(u'port', cell\_overwrite\_ok=True) # 创建sheet

sheet\_edge = filename.add\_sheet(u'edge', cell\_overwrite\_ok=True) # 创建sheet

sheet\_VLAN = filename.add\_sheet(u'VLAN', cell\_overwrite\_ok=True) # 创建sheet

[h, l] = list\_switch.shape # h为行数，l为列数

for i in range(l):

sheet\_switch.write(0,i," header")

for i in range(h):

for j in range(l):

sheet\_switch.write(i + 1, j, list\_switch[i, j])

[h, l] = list\_port.shape # h为行数，l为列数

for i in range(l):

sheet\_port.write(0, j, " header")

for i in range(h):

for j in range(l):

sheet\_port.write(i + 1, j, list\_port[i, j])

[h, l] = list\_edge.shape # h为行数，l为列数

for i in range(l):

sheet\_edge.write(0, j, " header")

for i in range(h):

for j in range(l):

sheet\_edge.write(i + 1, j, list\_edge[i, j])

[h, l] = list\_msti.shape # h为行数，l为列数

for i in range(l):

sheet\_VLAN.write(0, j, " header")

for i in range(h):

for j in range(l):

sheet\_VLAN.write(i + 1, j, list\_msti[i, j])

path = os.path.dirname(os.path.dirname(\_\_file\_\_)) + '/data\_test' + '/mstp.xls'

filename.save(path) # 保存到当前工作目录

print(path)

def generateExcel\_stp( list\_switch, list\_port, list\_edge):

list\_switch = np.array(list\_switch)

list\_port = np.array(list\_port)

list\_edge = np.array(list\_edge)

filename = xlwt.Workbook(encoding = "utf-8") # 创建工作簿

sheet\_switch = filename.add\_sheet(u'switch', cell\_overwrite\_ok=True) # 创建sheet

sheet\_port = filename.add\_sheet(u'port', cell\_overwrite\_ok=True) # 创建sheet

sheet\_edge = filename.add\_sheet(u'edge', cell\_overwrite\_ok=True) # 创建sheet

[h, l] = list\_switch.shape # h为行数，l为列数

for i in range(h):

for j in range(l):

sheet\_switch.write(i + 1, j, list\_switch[i, j])

[h, l] = list\_port.shape # h为行数，l为列数

for i in range(h):

for j in range(l):

sheet\_port.write(i + 1, j, list\_port[i, j])

[h, l] = list\_edge.shape # h为行数，l为列数

for i in range(h ):

for j in range(l):

sheet\_edge.write(i + 1, j, list\_edge[i, j])

path = os.path.dirname(os.path.dirname(\_\_file\_\_)) + '/data\_test' + '/stp.xls'

filename.save(path) # 保存到当前工作目录

print(path)

class Demo(QWidget, Ui\_Form):

def \_\_init\_\_(self):

super(Demo, self).\_\_init\_\_()

self.setupUi(self)

#STP按钮和MSTP按钮, 按下按钮会获得当前的输入文件

self.pushButton.clicked.connect(self.mstp\_start)

self.pushButton\_2.clicked.connect(self.stp\_start)

self.pushButton\_3.clicked.connect(self.stp\_data)

self.pushButton\_4.clicked.connect(self.mstp\_data)

self.pushButton\_5.clicked.connect(self.open\_btn\_clicked)

def open\_btn\_clicked(self):

"""点击相应按钮，跳转到第二个界面"""

# 实例化第二个界面的后端类，并对第二个界面进行显示

# 通过派生新类去访问类

self.another\_window = second\_main.AnotherWindowActions()

second\_main.show1(self.another\_window)

second\_main.show2(self.another\_window)

second\_main.show3(self.another\_window)

self.another\_window.show()

def mstp\_start(self):

fp = self.lineEdit.text()

print('文件输入 mstp开始')

os.system(r'python3 /Users/yewen/Desktop/2023毕业设计/code/STP/MSTP/mstp.py')

print(fp)

def stp\_start(self):

print('文件输入 stp开始')

fp = self.lineEdit.text()

os.system(r'python3 /Users/yewen/Desktop/2023毕业设计/code/STP/STP/stp.py')

print(fp)

def stp\_data(self):

list\_switch = list()

list\_port = list()

list\_edge = list()

#对于Switch子表来说

rows = self.tableWidget.rowCount()

cols = self.tableWidget.columnCount()

for i in range(rows) :

if self.tableWidget.item(i, 0) is None:

break

list\_switch.append(list())

for j in range(cols):

if self.tableWidget.item(i, j) is None:

break

data = self.tableWidget.item(i, j).text()

if data is None :

continue

else:

list\_switch[i].append(data)

rows = self.tableWidget\_2.rowCount()

cols = self.tableWidget\_2.columnCount()

for i in range(rows):

if self.tableWidget\_2.item(i, 0) is None:

break

list\_port.append(list())

for j in range(cols):

if self.tableWidget\_2.item(i, j) is None:

break

data = self.tableWidget\_2.item(i, j).text()

if data is None:

continue

else:

list\_port[i].append(data)

rows = self.tableWidget\_3.rowCount()

cols = self.tableWidget\_3.columnCount()

for i in range(rows):

if self.tableWidget\_3.item(i, 0) is None:

break

list\_edge.append(list())

for j in range(cols):

data = self.tableWidget\_3.item(i, j)

if data is None:

continue

data = self.tableWidget\_3.item(i, j).text()

if data is None:

continue

else:

list\_edge[i].append(data)

print(list\_switch)

print(list\_port)

print(list\_edge)

print('------运行程序----')

generateExcel\_stp(list\_switch, list\_port, list\_edge)

os.system(r'python3 /Users/yewen/Desktop/2023毕业设计/code/STP/STP/stp.py')

print('done')

def mstp\_data(self):

list\_switch = list()

list\_port = list()

list\_edge = list()

list\_msti = list()

# 对于Switch子表来说

rows = self.tableWidget\_4.rowCount()

cols = self.tableWidget\_4.columnCount()

for i in range(rows):

if self.tableWidget\_4.item(i, 0) is None:

break

list\_switch.append(list())

for j in range(cols):

data = self.tableWidget\_4.item(i, j).text()

if data is None:

continue

else:

list\_switch[i].append(data)

rows = self.tableWidget\_5.rowCount()

cols = self.tableWidget\_5.columnCount()

for i in range(rows):

if self.tableWidget\_5.item(i, 0) is None:

break

list\_port.append(list())

for j in range(cols):

data = self.tableWidget\_5.item(i, j).text()

if data == 'none':

continue

else:

list\_port[i].append(data)

rows = self.tableWidget\_6.rowCount()

cols = self.tableWidget\_6.columnCount()

for i in range(rows):

if self.tableWidget\_6.item(i, 0) is None:

break

list\_edge.append(list())

for j in range(cols):

data = self.tableWidget\_6.item(i, j).text()

if data == 'none':

continue

else:

list\_edge[i].append(data)

rows = self.tableWidget\_7.rowCount()

cols = self.tableWidget\_7.columnCount()

for i in range(rows):

if self.tableWidget\_7.item(i, 0) is None:

break

list\_msti.append(list())

for j in range(cols):

data = self.tableWidget\_7.item(i, j).text()

if data == 'none':

continue

else:

list\_msti[i].append(data)

print(list\_switch)

print(list\_port)

print(list\_edge)

print(list\_msti)

print('--------运行程序-------')

generateExcel\_mstp(list\_switch, list\_port, list\_edge, list\_msti)

#调用程序

os.system(r'python /Users/yewen/Desktop/2023毕业设计/code/STP/STP/stp.py')

print('done')

print('运行结束')

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

demo = Demo()

demo.show()

sys.exit(app.exec())

二级页面业务代码：

import sys

import second

from PyQt5.QtWidgets import QApplication, QMainWindow, QWidget, QFileDialog, QMessageBox,QLabel

from PyQt5 import QtWidgets

import numpy as np

from PyQt5.QtGui import QPixmap

def show1(x):

file = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/print.txt', 'r', encoding='utf-8')

data = file.readlines()

for i in range(len(data)):

x.textBrowser.append(data[i])

file.close()

def show2(x):

file = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/mac\_table.txt', 'r', encoding='utf-8')

data = file.readlines()

for i in range(len(data)):

x.textBrowser\_3.append(data[i])

file.close()

def show3(x):

file = open('/Users/yewen/Desktop/2023毕业设计/code/STP/data\_test/role.txt', 'r', encoding='utf-8')

data = file.readlines()

for i in range(len(data)):

x.textBrowser\_2.append(data[i])

file.close()

# 业务类需要继承两个类，一个设计的主界面，另一个是QMainWindow

class AnotherWindowActions(second.Ui\_Form\_2, QMainWindow):

def \_\_init\_\_(self):

super(second.Ui\_Form\_2, self).\_\_init\_\_()

self.setupUi(self)