Project 2 - Spanning Tree Protocol

In the lectures, you learned about Spanning Trees, which can be used to prevent forwarding loops on a layer 2 network. In this project, you will develop a simplified **distributed** version of the Spanning Tree Protocol that can be run on an arbitrary layer 2 topology (a topology of arbitrary nodes and links consisting of connected, uniquely-numbered switches with non-redundant links – see "Key assumptions and clarifications" below). This project is different from the previous project in that we're not running the simulation using the Mininet environment (Don't worry, Mininet will be back in later projects!). Rather, we will be simulating the communications between switches until they converge on a single solution, and then output the final spanning tree to a file. For this project, please use the class VM that you set up and used for Project 1.

We will provide a Project 2 walk-through and answer questions during Week 2 and Week 3 TA Office Hours, and please post questions to the dedicated Project 2 Piazza Posts:

- Project 2 Posted @81
- Project 2 Questions about Project Code @80
- Project 2 Test Topologies, Spanning Tree Outputs/Log Files, and Test Suites @79
- Project 2 Infinite Loops/Message Processing/Queues @78
- Project 2 What is pathThrough? @77
- Project 2 Assumptions and Clarifications in the Project Description @76
- Project 2 Log File Formatting @75

Project Layout

In the `Project2` directory, you can see quite a few files. You only need to (and should only) modify one of them, `Switch.py`. It represents a layer 2 switch that implements our simple Spanning Tree Protocol. You will have to implement the functionality described in the lectures and at the links above in the code sections marked with `TODO` comments to generate a Spanning Tree for each Switch.

The other files that form the project skeleton are:

- `Topology.py` - Represents a network topology of layer 2 switches. This class reads in the specified topology and arranges it into a data structure that your switch code can access.
- `StpSwitch.py` - A superclass of the class you will edit in `Switch.py`. It abstracts certain implementation details to simplify your tasks.

- `Message.py` - This class represents a simple message format you will use to communicate between switches. The format is similar to what was described in the course lectures. Specifically, you will create and send messages in Switch.py by declaring a message as msg
  = Message(claimedRoot, distanceToRoot, originID, destinationID, pathThrough)
  and assigning the correct data to each input. See the comments in Message.py for more information on the data in these variables.
- `run_spanning_tree.py`- A simple "main" file that loads a topology file (see `XXXTopo.py` below), uses that data to create a Topology object containing Switches, and starts the simulation.
- `XXXTopo.py`, etc - These are topology files that you will pass as input to the `run_spanning_tree.py` file.
- `sample_output.txt` - Example of a valid output file for `Sample.py` as described in the comments in `Switch.py`.

Here is an outline of the few sections of code that you will have to complete in Switch.py with suggestions for implementing the code, but you are free to implement your distributed spanning tree algorithm in other ways as long as it adheres to the spirit of the project and completes the labeled TODO sections in the project code per the comments in the code:

1. Decide on the data structure that you will use to keep track of the spanning tree. The collection of active links across all switches is the resultant spanning tree. The data structure may be any variables needed to track each switch's own view of the tree. Keep in mind that in a distributed algorithm, the switch can only communicate with its neighbors. It does not have an overall view of the tree as a whole. An example data structure would include, at a minimum, a variable to store the switchID that this switch currently sees as the root, a variable to store the distance to the switch's root, and a *list* or other datatype that stores the "active links" (i.e., the links to neighbors that should be drawn in the spanning tree). More variables may be helpful to track data needed to build the spanning tree.

   The switches are trying to learn the root, or the switch with the lowest id and the path to the switch. To track the path to the root, each switch may need to know which neighbor it goes through to get there and the distance of the path to the root. To output the spanning tree, the switch also needs to know whether it is on the path its neighbor takes to get to the root.

2. Implement the Spanning Tree Protocol by writing code that sends the initial messages to neighbors of the switch, and processes a message from an immediate neighbor. The comments describe how to send messages to immediate neighbors using the provided superclass. The messages are processed as a FIFO queue. As each switch processes messages, it compares the received message data to the data in its data structure to build the spanning tree. The switches do not need to push or pop on the FIFO queue since Topology.py does this for the switch as each switch calls send_msg.

   The switch will update the *root* stored in the data structure if the switch receives a message with a root of a lower ID. The switch will update the *distance* stored in the data structure if the switch updates the root or if there is a tiebreaker situation that indicates a shorter path to the root. The switch will change the *activeLinks* stored in the data structure in two situations: First, if the switch finds a new path to the root, meaning the switch goes through a new neighbor to get to the root, so the switch will ensure the new neighbor is in its activeLinks. Second, if the switch receives a message with pathThrough = True but the switch did not have that originID in its activeLinks list, so the switch must add originID to the activeLinks list or if the switch receives a message with paththrough = False but the switch has that originID in its activeLinks so it needs to remove originID from the activeLinks list.

   You may find other variables helpful for determining when to update the data structure, so those should be added to your data structure and updated as needed. You may also choose to update your data structure variables in a different algorithm. When to send messages is a significant decision in designing your algorithm, so think carefully about this. There are many correct algorithms that send messages at different times. When sending out messages, pathThrough should only be True if the destinationID switch is the neighbor that the switch with ID of originID goes through to get to the root. For example, if switchID 6 goes through neighbor with switchID 9 to get to the root, then the message from switch 6 to switch 9 would be originID= 6, destinationID = 9, pathThrough = True. And the messages to the other neighbors of switchID 6 would have pathThrough = False. Each switch only goes through one neighbor to get to the root.

3. Write a logging function that is specific to your particular data structure. The format is simple, and should output only the links active in spanning tree.

To run your code on a specific topology (SimpleLoopTopo in this case) and output the results

to a text file (out.txt in this case), execute the following command:

```
python run_spanning_tree.py SimpleLoopTopo out.txt
```

For this project, you will be able to create as many topologies as you wish and share them on Piazza. **We encourage you to share new topologies, and your output files to confirm correctness of your algorithm**. We have included several topologies for you to test your code against.

Key assumptions and clarifications

To avoid confusion, there are some assumptions we will make about behaviors of switches in this project:

1. You should assume that all switch IDs are positive integers, and distinct. These integers do not have to be consecutive and they will not always start at 1.
2. Tie breakers: All ties will be broken by lowest switch ID, meaning that if a switch has multiple paths to the root at the same length, it will select the path through the lowest id neighbor. For example, assume switch 5 has two paths to the root, through switch 3 and switch 2. Assume further each path is 2 hops in length, then switch 5 will select switch 2 as the path to the root and disable forwarding on the link to switch 3.
3. Combining points one and two above, there is a single distinct solution spanning tree for each topology.
4. You can assume all switches in the network will be connected to at least one other switch, and all switches are able to reach every other switch.
5. You can assume that there will be no redundant links and there will be only 1 link between each pair of connected switches.
6. You can assume that the topology given at the start will be the final topology and there won't be any changes as your algorithm runs (i.e adding a new switch).
7. Note that when a switch deactivates/blocks a port, this port is not completely discarded. While the switch treats it as inactive, it will still be communicated with during the simulation.

What to turn in

You only need to turn in the file you modify, `Switch.py` to Canvas, but you need to zip the file with your GTID you use to log into Canvas and the project number, such as `mmckinzie5_p2.zip`.

Be sure Switch.py is at the top directory in the zip file such that when it is unzipped.

There are some very important guidelines for this file you must follow:

1. **Your submission must terminate!** If your submission runs indefinitely (i.e. contains an infinite loop, or logic errors prevent solution convergence) it will not receive full credit. Termination here is defined as self-termination by the process. Manually killing your submission via console commands or interrupts is not an acceptable means of termination. The simulation stops when there are no messages left to send, at which time the logging function is called for each switch.
2. **Pay close attention to the logging format!** Our autograder expects your logs to be in a very specific format, specified in the comments. We provide you an example of a valid output log as well. Also pay close attention to how the links in the spanning tree should be ordered, separated in your log file, and what information belongs on its own line. Don't log anything other than what we ask you to! We provide example logs in the `Logs` directory.
3. **Remove any print statements from your code before turning it in!** Print statements left in the simulation, particularly for inefficient but logically sound implementations, have drastic effects on run-time. Your submission should take well less than 10 seconds to process a topology. If your leave print statements in your code and they adversely affect the grading process, your work will not receive full credit. Feel free to use print statements during the project for debugging, but please remove them before you submit to Canvas.

Spirit of the Project

The goal of this project is to implement a simplified version of a network protocol using a **distributed** algorithm. This means that your algorithm should be implemented at the network switch level. Each switch only knows its internal state, and the information passed to it via messages from its direct neighbors - the algorithm **must** be based on these messages.

The skeleton code we provide you runs a simulation of the larger network topology, and for the sake of simplicity, the `StpSwitch` class defines a link to the overall topology. This means it is possible using the provided code for one Switch to access another's internal state. This goes against the spirit of the project, and is not permitted. Additional detail is available in the comments of the skeleton code.

Additionally, you are not permitted to change the message passing format. We will not accept

modified versions of `Message.py`, nor are you permitted to subclass the `Message` class.

When we grade your code, we will use a special version of the skeleton code that will have a randomly generated variable name for the topology object. If you access it directly in order to generate your spanning tree, your code will throw a runtime error, and receive no credit. **In short - do not use *topolink* or *self.topology* objects in your code**. Pay careful attention to the code comments. If you have questions about whether your code is accessing data it should not, please ask on Piazza or during office hours!

What you can (and cannot) share

Do **not** share code from `Switch.py` with your fellow students, on Piazza, or publicly in any form. You **may** share log files for any topology, and you may also share any code you write that will *not be turned in*, such as new topologies or other testing code. (It may be a good idea to share a "correct" logs for a particular topology, if you have one, when you share the code for that topology.)

Grading

10 pts	Correct Submission	for turning in all the correct files with the correct names, and significant effort has been made in each file towards completing the project.
60 pts	Provided Topologies	for correct Spanning Tree results (log file) on the provided topologies.
80 pts	Unannounced Topologies	for correct Spanning Tree results (log file) on three topologies that you will not see in advance. They are slightly more complex than the provided ones, and may test for corner cases.

GRADING NOTE: Partial credit is not available for individual topology spanning tree output files. The output spanning tree must be correct to receive credit for that input topology. Thus, missing a single link will result in losing credit for that topology. Additionally, we will be using many topologies to test your project, including but not limited to the topologies we provide.

Additional Tips and Resources

Creating the Data Structure:

It is important to create a data structure in the correct place in Python (and most object oriented programming languages). If you create it inside a method, every time method is called it will be created as new. You should create a class object in the class constructor so that the data stored in the object exists for the life of the class instance that is created by Topology.py. For example self.mylist = [ ] in the constructor should create an empty list data structure and act as instance variable. But if mylist were instantiated in, say, process_messages, then it will be created every time the method is called. This could be useful in how you track which links are active to certain neighbors for any given switch.

Logging the Output

The output must be sorted. For example, your output should be:

1 - 2, 1 - 3
2 - 1, 2 - 4
3 - 1
4 – 2

However, this is not sorted:

1 – 3, 1 - 2,
2 – 4, 2 - 1,
4 - 2
3 - 1

Use of pathThrough

The pathThrough object passed in the messages is a boolean value representing whether the source of the message sees the destination as its path to the root. It is used so the destination keeps the source as an ActiveLink. Do not confuse this as a switch ID for the path to the root. You

may define a variable to keep track of the switch's path to the root, storing the switchID. Keep in mind that in the message() function you are sending the boolean value, not the switchID.

Questions to ask yourself

1. How does an infinite loop happen? When does each switch know it has a complete view of the tree or it should stop processing? How do the queues empty?
2. What is that pathThrough variable in the project code? Why is it required? Why did the project creators force that structure on the project?
3. Why did we have to give those assumptions? What would be unclear if they were not included?
4. How should the log files be formatted? Commas, semicolons, line endings? What is meant by sorted?