This software-defined networking (SDN) controller allows the user to view a topology graph of a simulated network. Each OpenFlow switch is represented as a circular node labeled with its identifier. Links between nodes are represented with lines between nodes, with a number labeling each link's utilization as the number of flows travelling through the link. Priority flows will always take the shortest path using Dijkstra's algorithm as implemented in the NetworkX library. For non-priority flows, the controller computes all possible paths to the destination and chooses the one with the least utilization on each link as a load-balancing mechanism. Flows specified as critical will also have a backup path computed which avoids using links taken by the primary path. In the event of a link failure, the backup path will be activated within the flow table. Issuing a command to the controller returns the user a forwarding table generated for the passed node representing an OpenFlow switch based on the controller's established flow table. The controller script contains a SHA256 watermark hash as specified inserted into the main function of the script.

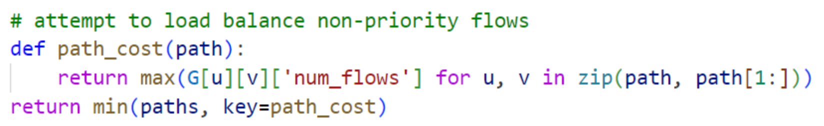
The commands and their usage are as follows:

- `insert_node <node>` : Adds a node with identifier "<node>" to the network
- `insert_link <node1> <node2>` : Adds a link
- `delete_node <node>` : Removes the node with identifier "<node>" from the network.
- `delete_link <node1> <node2>` : Removes a link from the network as if it went offline.
- `inject <node1> <node2> <0/1> <0/1>` : injects a traffic flow from <node1> into the network. If 3rd argument is set to 1, flow is marked as priority. If 3rd argument is set to true, flow is marked as critical.
- `disable <node1> <node2>` : simulates link failure between <node1> and <node2>. This performs similar functionality to removing the link from the network.
- `query <node1> <node2>` : displays information about the flow between <node1> and <node2>
- `quit` : terminates the program.

A specific challenge I faced while implementing the controller was determining how to implement load-balancing into my solution. I initially thought of using a random load-balancing algorithm but then decided to use a more explicit algorithm to compute the utilization of each potential path in the network. The controller would then set the added flow to use the least utilized one. My original algorithm was to take each link element and add up the number of flows to get a utilization count across the entire pathway. This can be seen as follows:

```
# attempt to load balance non-priority flows
def path_cost(path):
    return sum(G[u][v]['num_flows'] for u, v in zip(path, path[1:]))
return min(paths, key=path_cost)
```

The problem with this solution is that longer paths tended to report higher utilization because they simply had more links with flows on them. This meant the shorter paths within the network were overloaded with flows and the pathing algorithm failed to properly balance the network. Instead, I opted to find the link with the highest utilization in a path and compare this with the link with the highest utilization in other paths. The fix was implemented as follows:



```python
# attempt to load balance non-priority flows
def path_cost(path):
    return max(G[u][v]['num_flows'] for u, v in zip(path, path[1:]))
return min(paths, key=path_cost)
```

This simple line change ultimately leads to a more balanced network, easing congestion while enabling priority paths to always take the shortest route.