**SDN Controller Design Document**

This project focuses on building a Python-based **Software-Defined Networking (SDN) controller** that simulates OpenFlow switch behavior. The controller is designed to handle real-time network topology changes, compute routing paths, manage flow tables, and adapt to network failures. It also provides a user-friendly command-line interface (CLI) and a visual display of the network.

The main goal is to demonstrate core SDN concepts like dynamic flow installation, priority-based routing, and resilience through backup paths, while keeping the system easy to test and interact with.

## 2. Controller Architecture

The SDN controller is organized into three core components:

### A. Topology Management

- Uses networkx.Graph() to maintain the live network topology.
- Supports adding and removing nodes and links at any time.
- Keeps track of how many flows are using each link through a link_utilization dictionary.

### B. Routing & Flow Control

- Finds the shortest path between any two nodes using Dijkstra's algorithm.
- Uses networkx.all_shortest_paths() to identify multiple equal-cost paths to help balance traffic across them.
- Installs flow entries in each switch along a path, including match rules (based on destination), actions (where to forward), priority levels, and backup flags.
- For flows marked as critical, it calculates and installs a backup path that is disjoint from the main path.

### C. Visualization & CLI

- The show_topology() function draws the network, labeling each link with the number of active flows.
- The CLI (built with Python's cmd module) provides simple commands to manage the network: adding/removing nodes and links, injecting flows, simulating failures, and querying flow tables.

## 3. Routing Logic and Policies

### Shortest Path Computation

When a flow is injected, the controller calculates the shortest path between the source and destination using Dijkstra's algorithm. It also identifies all equal-cost shortest paths to enable load balancing.

Example: A flow from node A to node D could take both A-B-C-D and A-C-D if both paths are equally short.

**Flow Table Setup**

For every switch along a path, the controller installs a flow that:

- Matches the destination.
- Forwards the packet to the next hop.
- Sets the priority as specified by the user.
- Flags the flow as a backup if it's part of a critical path.

**Backup Path Installation**

For critical flows, a backup path is calculated by:

1. Copying the graph.
2. Temporarily removing the edges used in the primary path.
3. Finding an alternative shortest path.
4. Installing backup flows with slightly lower priority.

This ensures the network has a failover option if the main path fails.

**Handling Link Failures**

When a link failure is simulated, the controller:

- Removes the link from the graph.
- Prints confirmation of the simulated failure.
- Allow backup paths (if installed) to take over traffic forwarding.

**4. Visualization**

The show_topology() command pops up a live diagram of the network showing:

- Nodes and links.
- Edge labels that display how many flows are using each link.

This gives a real-time look at the network and helps track changes after flow injection or link failures.

**5. Cryptographic Watermark**

To guarantee the uniqueness and authenticity of this submission, a cryptographic watermark was added using a SHA-256 hash.

- **Input:**
  897263492NeoDDaBRgX5a9
- **SHA-256 Hash:**
  1a968120d8c0389bbfa55b9a83af1dd6f4c71240912a9c3921fbe17c208aa935
- **Implementation:**
  This hash is embedded at the top of sdn_controller.py:

WATERMARK = '1a968120d8c0389bbfa55b9a83af1dd6f4c71240912a9c3921fbe17c208aa935'

**Why It Matters:**

This watermark acts as proof of originality and ties the submission uniquely to my student ID. It ensures Including the watermark ties the **authorship of the code** directly to my student ID, ensuring authenticity. I chose to add the hash as a **constant in the main controller file** because this is the **core of the system**, all critical logic like topology management, flow installation, and routing decisions live there. By placing the watermark here, it ensures that any future modifications to the SDN controller can be cross-checked against the hash to confirm whether the original logic has been altered.

**6. Challenge Encountered**

**Problem:**
Initially, I had difficulty ensuring that backup paths for critical flows were fully disjoint from the primary path. Using the same graph for both primary and backup paths caused overlapping, which would defeat the purpose of having a true failover route.

**Solution:**
To solve this, I copied the graph using:

alt_graph = self.graph.copy()

alt_graph.remove_edges_from(zip(path, path[1:]))

This allowed me to remove the edges of the primary path before computing the backup path, ensuring that the backup was a true disjoint path whenever possible.

**Outcome:**
This approach made backup flows reliable and prevented routing loops or overlaps during failures.

**7. Conclusion**

This SDN controller effectively simulates OpenFlow switch behavior. It supports dynamic flow installation, prioritization, backup paths, and real-time visualization. The CLI makes it simple to experiment with SDN concepts like routing changes and network resilience.

The cryptographic watermark was added to secure the originality of the project, and the system has been tested to confirm that all requirements are met.