Assignment 3 - Intra-Domain Routing Algorithms

Network Centric Computing - Spring '25

Lead TAs: Ayesha Mirza, Hamza, Muhammad Abdullah, Talha, Wajiha, Yahya, Zaeem

Due Date: 11:59pm, 11th April, 2025

Contents

Disclaimer on Plagiarism and AI Tools

Academic integrity is paramount. Any form of plagiarism, including but not limited to copying code from online sources, unauthorized collaboration, or using AI-assisted tools such as GitHub Copilot, ChatGPT, or similar, is strictly prohibited.

All submissions will be analyzed using `MOSS` and other plagiarism detection tools. If plagiarism or unauthorized AI-assisted work is detected, the case will be forwarded to the **Disciplinary Committee (DC)** for review and appropriate action, which may include academic penalties or further disciplinary measures.

"AI wrote this for me" is not a valid justification. Submissions should reflect **your own** understanding, problem-solving ability, and effort. While AI tools can be useful for learning, relying on them to generate solutions undermines the purpose of the assignment and will be treated as academic misconduct.

Ensure that all work submitted is your **own** and adheres to academic integrity policies. If you have any doubts about what constitutes plagiarism or unauthorized AI use, seek clarification before submission.

1 Assignment Overview

The goal of this assignment is to introduce you to two commonly used intra-domain routing algorithms: Link-state Routing and Distance-Vector Routing. The assignment must be done individually, and you are required to use Python.

1.1 Queries

You should post any assignment related queries ONLY on Slack. Please do not email the course staff regarding any assignment-related queries. **Refrain from DMing the TAs unless it's a very personal issue**. This will ensure that everyone benefits from the answers to your queries.

1.2 Grade Distribution

This assignment is worth **12.5%** of your final grade. It will be graded out of 50 points. The distribution of points is as follows:

Test 1: Small Networks	**(3 points)**
Test 2: Small networks with link changes	
i) All correct routes	(5 points)
ii) 1 incorrect route	(3 points)
iii) 2 incorrect routes	(2 points)
iv) 3 incorrect routes	(1 points)
Test 3: Medium Network	10 points
Test 4: Large Network	(12 points)
Test 5: Large network with link changes	
i) All correct routes	(15 points)
ii) 1 incorrect route	(12 points)
iii) 2 incorrect routes	(10 points)
iv) 3 incorrect routes	(8 points)
v) 4 incorrect routes	(5 points)
vi) 5 incorrect routes	(3 points)
Code Quality: Following good code practices as described in **section 1.4**	**(5 points)**

Table 1: Points distribution for tests and code quality.

Note: All test cases will be run three times, and the final score for each test will be determined by *averaging* the scores of these runs. The workflow set up for you already runs the test cases three times.

1.3 Submission Instructions

You should submit your assignment on Github (details below).

Bonus: If you submit both `DVrouter.py` and `LSrouter.py` and they pass **ALL** the tests, you can earn a bonus of **10 points**. This means that only if you score **FULL** on both `LSRouter.py` AND `DVrouter.py`, will you be awarded bonus marks. In case any one of your routes is incorrect in either of the files, the bonus will not be awarded as the bonus is a binary grade; either you get a full bonus of 10 marks or no bonus at all.

These bonus points can be used to make up for lost points in any of the assignments, but **NO** other graded components.

1. Ensure all your code is pushed to GitHub before the deadline. Pushing your work to GitHub correctly is your responsibility.
   - If you do not push your final submission to GitHub, your assignment will **NOT** be graded.
2. Only the submission on the `main` branch will be graded.
3. No separate submission on LMS is required; your latest pushed commit will be considered for grading.
4. The staff will evaluate the code using the test suite and automated scripts.
5. Ensure that your latest work is visible on GitHub before the deadline.
6. Navigate to your repository URL and verify that your latest commit is present in the `main` branch before submission.
7. The time of your last commit will determine whether your grace days will be used or not, depending on whether your last commit was before or after the deadline.
8. Late days will only count if you push a new commit to GitHub after the deadline. Running a workflow does **not** count as using your grace days, as it is just like running test cases.
9. The staff will fetch the latest workflow output from each repository to extract the marks.
10. When you push new code to your GitHub repository, you should **not** rerun an existing job. That will use an older version of your code. Instead, you should run the workflow from scratch using the `Run Workflow` button in the Actions tab.
11. You do not need to commit something to run the workflow. You can manually trigger the workflow run without making a new commit, as demonstrated in the Git tutorial, granted that the workflow will then run on your last committed code that is available in your remote GitHub repository.

1.4 Code Quality

Your code in `LSrouter.py` and `DVrouter.py` will be evaluated using automated tools as well as a manual review based on the following criteria:

- **Maintainability, Modularity, and Complexity:** Your code will be analyzed using tools like `Radon` and `Pylint` to assess maintainability, modularity, and cyclomatic complexity. The grading will follow the detailed breakdown provided in Section 7.
- **Naming Conventions:** You are expected to follow `PEP 8` naming conventions. You can find the standards here. We will specifically look at:
  - **Variables, Functions, and Methods:** Use lowercase letters and separate words with underscores. For example, `my_variable`, `calculate_sum()`.
  - **Constants:** Use uppercase letters and separate words with underscores. For example, `MAX_SIZE`, `PI`.
  - **Classes:** Use CamelCase (capitalize the first letter of each word without underscores). For example, `MyClass`, `CarModel`.
- **Code Documentation and Comments:** Your code should be well-documented, following the comment ratio guidelines. The comment percentage will be evaluated as described in Section 7, ensuring that your code is understandable to others.

- **Code Modularity and Reusability:** Your implementation should not be monolithic; instead, break down your code into smaller reusable functions. This will be assessed as part of the modularity score in the automated quality check.
- **Readability and Formatting:** Your code should be readable with meaningful variable and function names. Additionally, proper indentation and spacing should be maintained to improve readability.

The final quality score will be calculated using automated scripts. Ensure that your code adheres to these best practices, as they are essential for writing maintainable and efficient software.

2 GitHub Setup and Assignment Submission

The assignment will be managed and graded via GitHub. Follow the steps below to set up your repository and submit your work.

Important Warning

The `.github` folder is a protected path. Modifying any files within this folder, including workflows and configuration files, is strictly prohibited, and GitHub classrooms will flag such repositories for us. Any unauthorized changes to this folder will result in your assignment being discarded. Ensure that you do not tamper with the contents of the `.github` directory.

2.1 Working on the Assignment

1. Implement the required functionality by modifying the provided files (**LSRouter.py or DVRouter or Both!** should be modified only).
2. Regularly commit your changes using git with the following commands or by Github Desktop:

```
git add .
git commit -m "Implemented feature XYZ"
git push origin main
```

Ensure that your latest work is always pushed to the repository before the deadline.

Important Guidelines

You are only allowed to use the imports explicitly mentioned in the starter code or in the `runAll.py` file under the `ALLOWED_IMPORTS` constant. **Any additional imports will result in an automatic score of `0`.** Ensure that your submission strictly follows the allowed imports list.

2.2 Running Tests on GitHub Workflows

GitHub Actions is set up to test your code. To check your test results:

1. Navigate to your repository on GitHub.
2. Click on the **Actions** tab.
3. Pick the **PA3 Autograding Runner** on the left side.
4. Press the **Run Workflow** button on the right side.

5. Refresh your page (You may need to do this a few times)
6. Look for the latest workflow run.
7. Click on it to view detailed logs of the test results, including your marks, by toggling open the step **Run TestRouter**.
8. You can also view your marks in the workflow artifact `run_output.txt`, which you can find at the bottom of **Summary** tab on the left after clicking on a workflow run.

If your tests fail, fix the issues locally and push your changes again.

3 Introduction

The Internet is composed of many independent networks (called autonomous systems) that must cooperate in order for packets to reach their destinations. This necessitates different protocols and algorithms for routing packets within autonomous systems, where all routers are operated by the same network's administrators, versus between autonomous systems, where business agreements and other policy considerations affect routing decisions.

This assignment focuses on intra-domain routing algorithms used by routers within a single autonomous system (AS). The goal of intra-domain routing is typically to forward packets along the shortest or lowest-cost path through the network.

The need to rapidly handle unexpected router or link failures, changing link costs (usually depending on traffic volume), and connections from new routers motivates the use of distributed algorithms for intra-domain routing. In these distributed algorithms, routers start with only their local state and must communicate with each other to learn the lowest-cost paths.

Many intra-domain routing algorithms used in real-world networks fall into one of two categories: distance-vector or link-state. In this assignment, you will implement distributed distance-vector and link-state routing algorithms in Python and test them with a provided network simulator.

4 Background

Begin by reviewing relevant lecture slides on LMS. Read Chapters 4.5.1 and 4.5.2 from the course textbook. An extract from Computer Networks: A Systems Approach has also been given to you. It provides enough information to design both the link-state and distance-vector routing algorithms. At a high level, they work as follows. Your goal in this assignment is to turn this high-level description into the actual working code.

4.1 Link-State Routing

- Each router keeps its own link-state and other nodes' link states it receives. The link state of a router contains the links and their weights between the router and its neighbors
- When a router receives a link-state from its neighbor, it updates the stored link state and the forwarding table. **Then it broadcasts the link state to other neighbors.**
- Each router broadcasts its own link state to all neighbors when the link state changes. The broadcast is also done periodically if no detected change has occurred.
- A sequence number is added to each link-state message to distinguish between old and new link-state messages. Each router stores the sequence number together with the link state. If a router receives a link-state message with a smaller sequence number (*i.e.*,, an old link-state message), the link-state message is simply disregarded.

4.2 Distance-Vector Routing

- Each router keeps its own distance-vector, which contains its distance to all destinations.
- When a router receives a distance vector from a neighbor, it updates its own distance vector and the forwarding table.
- Each router broadcasts its own distance vector to all neighbors when the distance vector changes. The broadcast is also done periodically if no detected change has occurred.
- Each router **does not** broadcast the received distance vector to its neighbors. It only broadcasts its own distance vector to its neighbors.

5 Provided Code

The provided code implements a network simulator that abstracts away many details of a real network, allowing you to focus on intra-domain routing algorithms. Each `.json` file in the `assignment3` directory is the specification for a different network simulation with different numbers of routers, links, and link costs. Some of these simulations also contain link additions and/or failures that will occur at pre-specified times. The network simulator can be run with or without a graphical interface. For example, the command `python3 visualize_network.py test1.json` will run the simulator on a simple network with two routers and three clients. The default router implementation returns all traffic back out of the link on which it arrives. This is obviously a terrible routing algorithm, which your implementations will fix.

Note: `visualize_network.py` is only for your use during development and debugging. It will not be used for grading. The final grading will be done using the script: `runAll.py`, which internally uses `network.py` (not the visualizer).

Also note that `visualize_network.py` requires Tkinter. If you're running into issues with the GUI not launching, make sure Tkinter is properly installed on your system. Refer to the `README` in our GitHub repository for more information about installing `Tkinter` or refer to its official documentation.

The network architecture is shown on the left side of the visualization. Routers are colored red; clients are colored blue. Each client periodically sends gray traceroute-like packets addressed to every other client in the network. These packets remember the sequence of routers they traverse, and the most recent route taken to each client is printed in the text box on the top right. This is an important debugging tool.

The cost of each link is printed on the connections.

Clicking on a client hides all packets except those addressed to that client, so you can see the path chosen by the routers. Clicking on the client again will go back to showing all packets.

Clicking on a router causes a string about that router to print in the text box on the lower right. You will be able to set the contents of this string for debugging your router implementations.

The same network simulation can be run without the graphical interface by the command `python3 network.py test1.json`. The simulation will run faster without having to go at a visualizable speed. It will stop after a predetermined amount of time, print the final routes taken by the traceroute packets to and from all clients, and indicate whether these routes are correct based on the known lowest-cost paths through the network.

6 Implementation Instructions

Your job is to complete the DVrouter and LSrouter classes in the DVrouter.py and LSrouter.py files so they implement distance-vector or link-state routing algorithms, respectively. The simulator will run independent instances of your completed DVrouter or LSrouter classes in separate threads, simulating independent routers in a network. You will notice that the DVrouter and LSrouter classes contain several unfinished methods marked with docstrings (including handle_packet, debug_string, etc.). These methods override those in

the Router superclass (in router.py) and are called by the simulator when a corresponding event occurs (e.g. handle_packet() will be called when a router instance receives a packet). The arguments to these methods contain all the information you need to implement the routing algorithms. Each of these methods is described in greater detail below. In addition to completing each of these methods, you are free to add additional fields (instance variables) or helper methods to the DVrouter and LSrouter classes. You will be graded on whether your solutions find the lowest-cost paths in the face of link failures and additions. Here are a few further simplifications:

- Each client and router in the network simulation has a single static address. Do not worry about address prefixes, families, or masks.
- You do not need to worry about packet authentication and checksums. Assume that a lower layer protocol handles corruption checking.
- As long as your routers behave correctly when notified of link additions and failures, you do not need to worry about time-to-live (TTL) fields. The network simulations are short and routers/links will not fail silently
- The slides discuss the "count-to-infinity" problem for distance-vector routing. You will need to handle this problem. You can use the heuristic discussed in the slides. Setting infinity = 16 is fine for the networks in this assignment.
- Link-state routing involves reliably flooding link-state updates. You will need to use **sequence numbers** to distinguish new updates from old updates, but you will not need to check (via acknowledgments and retransmissions) that LSPs send successfully between adjacent routers. Assume that a lower-level protocol makes single-hop sends reliable.
- Link-state routing involves computing the shortest paths. You can choose to implement Dijkstra's algorithm, and the pseudo-code is in the slides. Since this is a networking class instead of a data structures and algorithms class, you can also use a Python package like NetworkX or Dijkstar, **we recommend using Dijkstar.**
- Finally, LS and DV routing involve periodically sending routing information even if no detected change has occurred. This allows changes occurring far away in the network to propagate even if some routers do not change their routing tables in response to these changes (important for this assignment). It also allows the detection of silent router failures (not tested in this assignment). Your implementations should send periodic routing packets every heartbeat_time milliseconds, where heartbeat_time is an argument to the DVrouter or LSrouter constructor. You will regularly get the current time in milliseconds as an argument to the handle_time method (see section 5.2).

6.1 Restrictions

There are limitations on what your `DVrouter` and `LSrouter` classes are allowed to access from the other provided Python files. Unlike C and Java, Python does not support private variables and classes. Instead, these limitations will be checked when grading. Violating any of these requirements will result in serious grade penalties.

- Your solution must not require modification to any files other than `DVrouter.py` and `LSrouter.py`. The grading tests will be performed with unchanged versions of the other files.
- You are not allowed any imports other than the ones mentioned in starter files or in the `runAll.py` file under the `ALLOWED_IMPORTS` constant.
- Your code may not call any functions or methods, instantiate any classes, or access any variables defined in any of the other provided Python files, with the following exceptions:
  - `LSrouter` and `DVrouter` can call the inherited `send` function of the `Router` superclass (e.g., `self.send(packet)`).

  - `LSrouter` and `DVrouter` can access the `addr` field of the `Router` superclass (e.g., `self.addr`) to get their own address.
  - `LSrouter` and `DVrouter` can create new `Packet` objects and call any of the methods defined in `packet.py` except for `getRoute()`, `addToRoute()`, and `animateSend()`. You can access and change any of the fields of a `Packet` object except for `route`.

6.2 Method Descriptions

These are the methods you need to complete in `DVrouter` and `LSrouter`:

- `__init__(self, addr, heartbeat_time)`:
  This is the address of this router. Add your own class fields and initialization code (e.g., to create forwarding table data structures). Routing information should be sent by this router at least once every `heartbeatTime` milliseconds.
- `handle_packet(self, packet)`:
  Process incoming packets. This method is called whenever a packet arrives at the router on any port. You need to check whether the packet is a traceroute packet or a routing packet and handle it appropriately. Methods and fields of the `Packet` class are defined in `packet.py`.
- `handle_new_link(self, port, endpoint, cost)`:
  This method is called whenever a new link is added to the router on port number `port`, connecting to a router with address `endpoint` and link cost `cost`. You should store these values in a data structure to use for routing. If you want to send packets along this link, call `self.send(port, packet)`.
- `handle_remove_link(self, port)`:
  This method is called when the existing link on the port number `port` is disconnected. You should update your data structures appropriately.
- `handle_timer(self, timeMillisecs)`:
  This method is called regularly and provides you with the current time in milliseconds for sending routing packets at regular intervals.
- `debug_string(self)`:
  This method is called by the network visualization to print current details about the router. It should return any string that will be helpful for debugging. This method is for your own use and will not be graded.

6.3 Dijkstra Package

These are the methods you need to be familiar with from the `Dijkstra` package:

- `add_edge(u, v, cost)`: Adds an edge from `u` to `v`. If the graph is undirected (as in this assignment), the edge will also be added from `v` to `u`.
- `remove_edge(u, v)`: Removes the edge from `u` to `v`.
- `find_path(graph, u, v)`: Given a graph, it finds a path from `u` to `v`. It returns a `PathInfo` object containing:
  - The total cost of the path
  - A list of edges on the path
  - A list of nodes on the path
  - The total path cost

For Dijkstra usage examples, please see the official project. If you choose to use `NetworkX`, refer to its official documentation for usage details.

6.4 Creating and Sending Packets

You will need to create packets to send information between routers using the `Packet` class defined in `packet.py`. For instance, if you want to send routing information for a distance-vector algorithm, you might set `packet.kind = ROUTING`. You will also need to decide how to store additional data in the packet. A reasonable approach is to store the data in `packet.data`.

We cannot parse your entire data structure, but we can check the fields. For example, you can store a text string or a dictionary. In the skeleton code, you may make a function (e.g., `make_packet()`) that returns a new `Packet` object with the fields `srcAddr`, `dstAddr`, `kind`, and `data` appropriately set. To send a packet, call `self.send(port, packet)`.

6.5 Link Cost Changes

In this assignment, link costs can change after the initial network topology is set. When a link's cost changes, the routers connected to that link may effectively receive both a `handle_remove_link()` event and a `handle_new_link()` event (with the new cost). You should update your data structures accordingly in those methods.

6.6 The Simulator

This assignment abstracts away many low-level details, focusing on distance-vector or link-state algorithms. This high-level approach allows you to concentrate on the essence of the algorithms without worrying about more intricate protocols (e.g., ARP) or meticulous system programming issues. If you have questions about these real-world details, please ask on Slack or during office hours.

7 Testing

You should test your `DVRouter` and `LSrouter` using the provided network simulator. There are multiple JSON files defining different network architectures and link failures and additions. `test3.json` and `test4.json` files define the networks on pages 242 and 244 of the provided reading. The JSON files without "events" in their file name do not have link failures or additions and are good for initial testing.

If you use `visualize_network.py`, understanding the JSON files is not necessary. However, it may still be useful to understand the **changes** and **links** sections.

- **The links section** in the JSON contains a list of all the edges in the network graph. The structure is as follows: `[node1, node2, port1, port2, cost1, cost2]`. You don't have to worry about the ports, and since the graph is undirected, `cost1` and `cost2` are always equal.

  Example: `["A", "B", 1, 1, 3, 3]` – An edge from A to B with cost 3.

- **The changes section** contains a list of all link changes that will occur in the network at different times. There are two types of changes:
  - **Up:** can be a new link or an update to an existing link.
    Example: `[12, ["G", "F", 2, 2, 1, 1], "up"]` – An edge from node G to F is added at time 12.
  - **Down:** removal of a link
    Example: `[32, ["E", "G"], "down"]` – An edge from node E to G is removed at time 32.

To run the simulation with the graphical interface, use the following command (The argument `DV` or `LS` indicates whether to run `DVrouter` or `LSrouter`, respectively):

```
$ python3 visualize_network.py <networkSimulationFile.json> [DV|LS]
```

IMPORTANT: The above visualization is only there for your own sake and to make debugging easier for you. Final grading and all testing will be done using the `runAll.py` file.

To run the simulation without the graphical interface, use the following command (The argument `DV` or `LS` indicates whether to run `DVrouter` or `LSrouter`, respectively):

```
$ python3 network.py <networkSimulationFile.json> [DV|LS]
```

The routes to and from each client at the end of the simulation will print, along with whether they match the reference lowest-cost routes. If the routes match, your implementation has passed for that simulation. If they do not, continue debugging (using print statements and the `debugString()` method in your router classes).

To simply run test all the test cases at once, you can simply use the command

```
$ python3 runAll.py
```

For more details on testing and troubleshooting setup issues please refer to the **README** file in your PA3 GitHub repository.

8 Starter Code

```
PA3/
|-- Code/
|   |-- client.py
|   |-- DVrouter.py
|   |-- link.py
|   |-- LSrouter.py
|   |-- network.py
|   |-- packet.py
|   |-- requirements.txt
|   |-- ReviewCodeQuality.py
|   |-- router.py
|   |-- runAll.py
|   |-- test1.json
|   |-- test2.json
|   |-- test3.json
|   |-- test4.json
|   |-- test5.json
|   |-- visualize_network.py
|-- docker-compose.yml
|-- Dockerfile
|-- Manual/
|   |-- PA3_Manual.pdf
|-- README.md
|-- templates/
|   |-- DVrouter.py
|   |-- LSrouter.py
```

9 Running the Code on Docker

A **Dockerfile** is included for consistent testing. Use the following steps:

9.1 Build and Run Container

To build and run the container, use the following command:

```
docker compose run --rm netcen-spring-2025
# or
docker-compose run --rm netcen-spring-2025
```

What this command does:

- `run`: Executes the `netcen-spring-2025` service defined in the `docker-compose.yml` file as a temporary, one-off container.
- `-rm`: Automatically removes the container after the task completes, ensuring no leftover containers clutter the system.
- This command starts the container and runs the service or command specified for `netcen-spring-2025` in `docker-compose.yml`.

9.2 Access Code

- The code is mounted at `/home/netcen_pa3` inside the container.
- This allows you to access and work with your files directly within the container.

10 Tips on Getting Started

Before you begin coding, consider the following tips to help you tackle the assignment effectively:

- **Visualize the Problem:** Take a moment to sketch the network topology and the flow of each routing algorithm (both Distance-Vector and Link-State) on paper. This can help clarify how the algorithms compute shortest paths and react to network changes before you start coding.
- **Understand the Algorithms:** Make sure you have a solid understanding of the underlying concepts:
  - **Distance-Vector Routing:** Focus on how routers exchange their distance vectors, update their routes, and address issues like the count-to-infinity problem.
  - **Link-State Routing:** Visualize how routers flood the network with link-state packets and use Dijkstra's algorithm (or a similar method) to compute shortest paths.
- **Start Early:** Read the handout carefully and begin working on the assignment as soon as possible. Early testing and debugging can prevent last-minute issues.
- **Leverage Online Resources:** Use search engines to quickly resolve common problems—whether it's a "Bad File Descriptor" error, string parsing issues, or list iteration challenges. A few targeted keywords can often lead to a solution faster than waiting for TA responses.
- **Follow Best Coding Practices:** Adhere to good coding standards (e.g., PEP 8 for Python) by using meaningful variable names, modularizing your code, and including appropriate comments. Clean, well-structured code is not only easier to maintain but also easier to debug.
- **Ask for Help When Needed:** If you run into issues or have questions, don't hesitate to ask on Slack. Sharing your progress and specific questions can lead to quicker, more targeted assistance.

Good Luck, and Happy Coding!