# Assignment 2:
# Intra-Domain Routing Algorithms

**Deadline: Friday,April 2, 2021 at 11:55 PM**

The goal of this assignment is to introduce you to two commonly used intra-domain routing algorithms: Link-state Routing and Distance-Vector Routing.

**This assignment is to be done individually. You are only required to implement either distance vector OR link-state.** If you implement both, you can get a bonus of 10 points.

Note: You should post any assignment related query ONLY on Piazza. Do not directly email the course staff.

Note: The assignment is worth 10% of your total grade.

Note: Course policy about **plagiarism** is as follows:

- This assignment should be done individually.
- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection. We will run MOSS on your assignment and compare with both online solutions as well as solutions from previous years.
- Students cannot copy the code from the Internet. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

## 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 entity, and 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 and clients, 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.

# Background

Begin by reviewing relevant lecture slides on LMS. Read the Chapter 4.5.1 and 4.5.2 from the course textbook.
An extract from the *Computer Networks: A Systems Approach* is also given to you (Assignment2_reading.pdf). 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 to the actual working code.

## 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.

## 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.

# Provided code

## Download & Setup

- Download *Assignment2_code.zip* from LMS and extract it to a folder of your choice.
- In a terminal, execute:
  - **sudo apt-get update**
  - **sudo apt-get install python3 python3-pip python3-tk**
  - **pip3 install Dijkstar**
  - **pip3 install networkx**
- Restart your computer

## Familiarize yourself with the network simulator

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 assignment2 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 2 routers and 3 clients. The default router implementation returns all traffic back out the link on which it arrives. This is obviously a terrible routing algorithm, which your implementations will fix.

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 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 whether these routes are correct given the known lowest-cost paths through the network.

# 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 TODO (including handlePacket, debugString, 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. handlePacket() 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 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 heartbeatTime milliseconds where heartbeatTime is an argument to the DVrouter or LSrouter constructor. You will regularly get the current time in milliseconds as an argument to the handleTime method (see below).

## Restrictions

There are limitations on what information 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, the list of limitations here 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.
- 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(port, 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.

## Method descriptions

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

- __init__(self, addr, heartbeatTime)
    - Class constructor.addr is 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.

- handlePacket(self, port, packet)
    - Process incoming packet: This method is called whenever a packet arrives at the router on port number port. You should 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.

- handleNewLink(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 or client with address endpoint and link cost cost. You should store the argument values in a data structure to use for routing. If you want to send packets along with this link, call self.send(port, packet).

- handleRemoveLink(self, port)
  - This method is called when the existing link on the port number port is disconnected. You should update data structures appropriately.

- handleTime(self, timeMillisecs)

  This method is called regularly and provides you with the current time in millisecs for sending routing packets at regular intervals.

- debugString(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.

These are the methods you need to be familiar with from the *Dijkstar* 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 be added from v to u too.

- 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 following:
    - A list of nodes on the path
    - A list of edges on the path
    - The costs of edges on the path
    - Total path cost

For *Dijkstar* usage examples, please see the official project site here.

If you choose to use *NetworkX*, please see its official documentation here.

## Creating and sending packets

You will need to create packets to send information between routers using the `Packet` class defined in `packet.py`. Any packet `p` you create to send routing information should have `p.kind == ROUTING`.

You will have to decide what to include in the `content` field of these packets. The content should be reasonable for the algorithm you are implementing (e.g. don't send an entire routing table for linkstate routing).

Packet content must be a string. This is checked by an assert statement when the packet is sent. `DVrouter` and `LSrouter` import the `dumps()` and `loads()` functions that return a string (in JSON format) when given a python object. Using these functions is an easy way to stringify and destringify.

You can access and set/modify any of the fields of a packet object (including `content`, `srcAddr`, `dstAdddr` , and `kind`) except for `route` (see "Restrictions" above).

## Link reliability

If a link between two routers fails or is added, the appropriate `handle` function will *always* be called on both routers after the failure or addition.

Links have varying latencies (usually proportional to their costs). Packets may not arrive in the global order that they are sent.

## This is not a network…

The simulated network in this assignment abstracts away many details you would need to consider when implementing distance-vector or link-state algorithms on real routers. This should allow you to focus on the core ideas of the algorithms without worrying about other protocols (e.g. ARP) or meticulous systems programming issues. If you are curious about these real-world details, please ask on Piazza or in office hours.

# Running and 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 *visual_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.

Run the simulation with the graphical interface using the command:

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

The argument DV or LS indicates whether to run DVrouter or LSrouter, respectively.

Run the simulation without the graphical interface with the command:

`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).

# Submission

- You must submit the source code for `DVrouter.py` or `LSrouter.py`. Submit the assignment by compressing either *LSrouter.py* or *DVrouter.py* (or both) in a zipped file named **<roll_number>.zip** and upload the compressed file to LMS.

- Bonus: If you submit both `DVrouter.py` and `LSrouter.py` and they pass all the tests, you can get a bonus of 10 points. The bonus points can be used to make up for lost points in any of the assignments.

# Grading

We will run the network simulation using the provided JSON files. Your grade will be based on whether your algorithm finds the lowest cost paths and whether you have violated any of the restrictions listed above. We will also check that `DVrouter` actually runs a distance-vector algorithm and that `LSrouter` actually runs a link-state algorithm.

**You can earn up to 50 points from this assignment.**

The following table describes each test case along with its maximum points. You can get partial points based on the following criteria:

| Test Description | Maximum Points |
|---|---|
| **Test 1: Small network** | 3 |
| **Test 2: Small network with link changes**<br>a) All correct routes<br>b) 1 incorrect route<br>c) 2 incorrect routes<br>d) 3 incorrect routes | <br>5<br>3<br>2<br>1 |
| **Test 3: Medium network** | 10 |
| **Test 4: Large network** | 12 |

| | |
|---|---|
| **Test 5: Large network with link changes**<br>a) All correct routes | 15 |
| b) 1 incorrect route<br>c) 3 incorrect routes<br>d) 5 incorrect routes<br>e) 7 incorrect routes | 12<br>10<br>8<br>5 |
| **Code quality marks**<br>• Your submission should have a good coding style that adheres to the formatting and name conventions of Python.<br>• Your code should be modular, e.g., you may lose points if your code consists of a single large function that encapsulates all the functionality.<br>• Your code should be well commented. | 5 |

If the number of incorrect routes is more than the maximum number specified in each test case, then you will receive no credit.

As always, start early and feel free to ask questions on Piazza and in office hours.

# Acknowledgements

- The provided reading is an excerpt from the *Computer Networks: A Systems Approach* textbook.