Things of the Internet (TOI)

LAB 3: Routing

Please note the submission deadline for this report is Wednesday 12:00 PM, March 25th, 2021.

Your name:

Introduction

In this practical, we will implement and compare two important types of routing protocols, the distance-vector (DV) protocol and the link-state (LS) protocol. In this lab, the Python kernel will researt everytime you run an experiment, so the program will intentionally shut down the kernel everytime. If you see the warning "The kernel appears to have died. It will restart automatically.", it's completely normal and you do not need to do anything about it.

1 The NetSim simulator

In this practical, we will primarily use the NetSim simulator, which is similar with the WSim simulator used in the previous practical. NetSim executes a set of steps every time slot, where time increases by 1 each slot. During each time slot a node can deliver one packet to its one-hop neighbour nodes via direct wireless links. You will have the full access to the source code p3_netsim.py, and you can try to read it and find out how the simulator works.

Run the code in Section 3 (Distance-vector routing). You may see something like this (Just ignore the blue dot):

There are several parameters you can modify:

Parameter Name	Description	Default Value
gui	Show GUI	True
numnodes	Number of nodes	12
simtime	Simulation time	2000
rand	Use randomly generated topology	False

You can click on the nodes to see the current neighbour and routing information in the terminal window. Note that now it should be pretty empty, and after you implement the routing protocols you should be able to see the neighbour list and routing table/costs.

If you set *gui* to False, the simulator will run itself against several test cases.

2 Test cases

Your protocols will be tested against the following test cases (see p3_tests.py):

Euclidean network: the cost on the edges of the network is Euclidean, i.e. for three nodes A, B and C, we have (cost(AB) + cost(BC) > cost(AC), and the links between nodes are stable.

Non-Euclidean network: the links between nodes are stable, but the cost can be non-Euclidean.

Network with cost changes: the cost of the edges may change during simulation.

Network with cost change and broken links: the links between nodes are not stable and can break during simulation, while the cost may also change

Disconnected Network: the network can become disconnected during simulation.

Passing all the test doesn't mean your code is correct, while the correct implementation could also fail in some test cases.

3 Distance-vector routing

In the distance-vector routing protocol, each node periodically broadcasts to its neighbours about its current routing table, i.e. a vector [(dst1, cost1), (dst2, cost2), ...], which describes the current cost of reaching each destination from itself. When receiving such an advertisement from a neighbour, the node updates its current routing table accordingly. In this practical, we assume the links in the network are bi-directional with symmetric cost, i.e. the cost of a link from A to B is the same as that of B to A.

Your task is to implement both the advertisement and integration steps. We provide the code skeleton below. Here is some key information you might need when implementing your functions:

self.routes, which is the routing table of the node. It is a python dictionary [dst1, link1; dst2, link2, ...], which maps a destination node to a *Link*. A *Link* object e.g. link1, represents the edge that connects the current node to its next hop neighbour towards the destination dst1. The cost associated with the link can be obtained by calling *link1.cost* (a variable defined in class *Link*). For instance, suppose node A connects to B, which then connects to C. Then in the routing table of A, you would see an entry [C, link$_{AB}$]. link$_{AB}$ can be obtained by calling *self.getlink(B)* (assuming your code is running on node A). Your code should update this routing table when you receive information from other nodes.

self.spcost, which is the cost table of the node. It is a python dictionary [dst1, cost1; dst2, cost2, ...], which maps a destination node to the currently estimated cost of getting there. Your code should set this cost table as well.

In [ ]:
```
### Distance vector routing
import random,sys,math
```

```python
from optparse import OptionParser
from p3_netsim import *
import p3_tests
import os
import time

class DVRouter(Router):
    INFINITY = 32

    def send_advertisement(self, time):
        adv = self.make_dv_advertisement()
        for link in self.links:
            p = self.network.make_packet(self.address, self.peer(link),
                                         'ADVERT', time,
                                         color='red', ad=adv)
            link.send(self, p)

    # Make a distance vector protocol advertisement, which will be sent
    # by the caller along all the links
    def make_dv_advertisement(self):
        ## Task 3.1
        ## Your code here
        adv = []
        for dst, cost in self.spcost.items():
            adv.append((dst,cost))
        return adv

    def link_failed(self, link):
        # If a link is broken, remove it from my routing/cost table
        self.clear_routes(self)

    def process_advertisement(self, p, link, time):
        self.integrate(link, p.properties['ad'])

    # Integrate new routing advertisement to update routing
    # table and costs
    def integrate(self,link,adv):
        ## Task 3.2
        ## Your code here
        for dst, dst_cost in adv:
            # If I don't know dst yet, or the cost to dst thru link is smaller
            if ((not dst in self.spcost) or (link.cost + dst_cost < self.spcost[dst])):
                # Update the new cost to my cost table
                self.spcost[dst] = link.cost + dst_cost
                # Update the new neighbour link to my routing table
                self.routes[dst] = link
            # Handle the cases when cost changes
            if (self.routes[dst] == link and self.spcost[dst] != link.cost + dst_cost):
                self.spcost[dst] = link.cost + dst_cost
                self.routes[dst] = link

        pass

# A network with nodes of type DVRouter.
class DVRouterNetwork(RouterNetwork):
    # nodes should be an instance of DVNode (defined above)
    def make_node(self,loc,address=None):
        return DVRouter(loc,address=address)

########################################################################

if __name__ == '__main__':

    gui = False
    numnodes = 12
    simtime = 2000
    rand = False

    if rand == True:
        rg = RandomGraph(numnodes)
        (NODES, LINKS) = rg.genGraph()
    else:
        # build the deterministic test network
        #   A---B   C---D
        #   |   | / | / |
        #   E   F---G---H
        # format: (name of node, x coord, y coord)

        NODES =(('A',0,0), ('B',1,0), ('C',2,0), ('D',3,0),
                ('E',0,1), ('F',1,1), ('G',2,1), ('H',3,1))

        # format: (link start, link end)
        LINKS = (('A','B'),('A','E'),('B','F'),('E','F'),
                 ('C','D'),('C','F'),('C','G'),
                 ('D','G'),('D','H'),('F','G'),('G','H'))

    # setup graphical simulation interface
    if gui == True:
        net = DVRouterNetwork(simtime, NODES, LINKS, 0)
        sim = NetSim()
        sim.SetNetwork(net)
        sim.MainLoop()
    else:
        p3_tests.verify_routes(DVRouterNetwork)
    time.sleep(3)
    os._exit(0)
```

```
Test case #1: Euclidean network with no broken links...
        A---B   C---D
        |   | / | / |
        E   F---G---H
        link costs = 1 on straight links, sqrt(2) on diagonal links
Test case #1: Pass with convergence time: 82

Test case #2: Non-Euclidean network with no broken links...
         A-7-B-1-E
         |     / |
         1  2/   9
         | /     |
         C-4-D-1-F
Test case #2: Pass with convergence time: 42

Test case #3: Network with cost changes...
          Y
         / \
        X---Z
        Costs: XY=4 YZ=1 ZX=12
Now change cost ZX to 2
Your convergence time after changing cost ZX to 2: 22
...Good, now change back
Now change cost XY to 14
Your convergence time after changing cost XY to 14: 150

Test case #4: Network with cost changes and broken links...
        A---B-4-C---D
        |   | /2X /2|
        E---F---G---H
        Costs: BC=4 DG=2 CF=2 CG broken; all other costs are 1
Before any cost change or link breaks, your convergence time: 62

Now breaking links CF, CG, DG; changing costs BF to 15, CD to 13
Test case #4: Pass with convergence time: 990

Test case #5: Disconnected network...
        A---B   C---D
        |   | / | / |
        E   F---G---H
        link costs = 1 on straight links, sqrt(2) on diagonal links
Now breaking links F-C and F-G ('X' marks the broken links)
        A---B   C---D
        |   | X | / |
        E   F-X-G---H
Test case #5: Failed!
```

Task 3.1: Implement *make_dv_advertisement(self)* function, which constructs the distance vector advertisement based on the current cost table self.spcost of the node. Your code should return a list, containing the constructed distance vector advertisement.

Task 3.2: Implement the *integrate(self,link,adv)* function, which updates the routing table and cost table based on the advertisement. The input parameter link is the link through which the actual advertisement adv has been delivered, i.e. the link that connects that node and the neighbour who advertised adv. You may want to use the distributed Bellman-Ford algorithm discussed in the lecture. Run the code to check if your code passes the first two test cases.

Your answer: Please see the code.

Task 3.3: Improve your implementation of *integrate(self,link,adv)* function to handle cost changes. Consider the following scenario: node A connects to B, which connects to C. Assume that the costs are AB = 1, BC = 1. Suppose the current routing table at A contains this entry (C, link$_{AB}$), and the cost table has entry (C, 2). This means A knows that to go to C, it has to follow the link to B, and the cost is 2. Now let's say the cost BC is changed to 10. How should you make your code robust to such a situation?

Your answer: Please see the code.

Task 3.4: We define the convergence time of a routing protocol as the total amount of timestamps it needs to make every node in the network has the correct routing table. In the final test case where the network becomes disconnected, the distance-vector routing protocol is expected to fail. Can you explain why?

Your answer: please refer to https://www.geeksforgeeks.org/route-poisoning-and-count-to-infinity-problem-in-routing/

4 Link-state routing

Recall from the lecture that in link-state routing, each node advertises its current link state (cost to its neighbours) to all the neighbours, and each recipient re-sends this information on all of its links. In this way, the link state is flooded through the network, and eventually all nodes know about all the links and nodes in the network topology. Then each node integrates the received information to compute the minimum cost path to every other node in the network.

Your task is to implement both the advertisement and integration steps. We have provided the code skeleton in below.

In [ ]:

```
# Link-state routing protocal
import random,sys,math
from optparse import OptionParser
from p3_netsim import *
import p3_tests
import os, time

import numpy as np


class LSRouter(Router):
    INFINITY = sys.maxsize

    def __init__(self,location,address=None):
        Router.__init__(self, location, address=address)
        # address -> (seqnum,(nbr1,cost1),(nbr2,cost2),(nbr3,cost3),...)
        self.LSA = {}
        self.LSA_seqnum = 0     # uniquely identify each LSA broadcast

    def make_ls_advertisement(self):
        # return a list of all neighbors to send out in an LSA
        ## Your code here
        neighbour_links = []
        for value in self.neighbors.values():
            neighbour_links.append((value[1], value[2]))
        return neighbour_links

    def send_lsa(self, time):
        self.LSA_seqnum += 1
        lsa_info = self.make_ls_advertisement()
        for link in self.links:
            p = self.network.make_packet(self.address, self.peer(link),
                                         'ADVERT', time, color='red',
                                         seqnum=self.LSA_seqnum,
                                         neighbors=lsa_info)
            link.send(self, p)

    def send_advertisement(self, time):
        self.send_lsa(time)
        self.clear_stale_lsa(time)

    def clear_stale_lsa(self, time):
        # After sending out LSA packets, clear out older LSA entries
        for key in list(self.LSA):
            if self.LSA[key][0] < self.LSA_seqnum - 1:
                del self.LSA[key]
    def process_advertisement(self, p, link, time):
        # Process incoming LSA advertisement.
        # First get sequence number from packet, then see if we have a
        # EX entry in LSA from the same node
        seq = p.properties['seqnum']
        saved = self.LSA.get(p.source, (-1,))
        if seq > saved[0]:
            # update only if incoming seqnum is larger than saved seqnum
            if p.properties['neighbors'] is not None:
                self.LSA[p.source] = [seq] + p.properties['neighbors']
            else:
                print(p.properties)
                print('Malformed LSA: No LSA neighbor information in packet.  Exiting...')
                sys.exit(1)
            # Rebroadcast packet to our neighbors.  We don't _have_ to
            # rebroadcast to the neighbor we just got the LSA from,
            # but we're going to do it anyway...
            for link in self.links:
                link.send(self, self.network.duplicate_packet(p))

    # get_all_nodes scans each node's LSA to visit all the other
    # non-neighbor nodes emulating a breadth first search (BFS).  The
    # reason we do a BFS traversal rather than simply use self.LSA is
    # because we want to have a route to every node that is currently
    # reachable from us.
    def get_all_nodes(self):
        nodes = [self.address]
        for u in nodes:
            if self.LSA.get(u) != None:
                lsa_info = self.LSA[u][1:]
                for i in range(len(lsa_info)):
                    v = lsa_info[i][0]
                    if not v in nodes:
                        nodes.append(v)
        return nodes

    # Each node's spcost and routes[] table should be set correctly
    # in this function by processing the information in self.LSA.
    # "nodes" is the list of nodes we know about.
    def run_dijkstra(self, nodes):
        ## Your code here\
        # Myself is the source
        src_node = self.address
        # The temporary cost table: [(dst1, cost1), (dst2, cost2), ...]
        cost_table = {}
        # A temporary set, storing the nodes that is on the shortest path
        sp_nodes = {}
        # Initialize with infinite cost, where the cost to myself is 0
        for node in nodes:
            cost_table[node] = self.INFINITY
        cost_table[src_node] = 0
        # For every node we know about
        while len(nodes) > 0:
            # Find the node that has the smallest cost to myself. Obviously, in the first iteration it should be my self
            nodeInd = nodes.index(min(nodes, key = lambda k:cost_table[k]))
            node = nodes.pop(nodeInd)
            # Trivial case: jump out early
            if cost_table[node] >= self.INFINITY:
                break
            # Now use the advertised LSA data from this node (with smallest cost) to update the cost table
            # We don't need the first element since it is seqnum
            for node_cost_pair in self.LSA[node][1:]:
                # Try a neighbour of this node
                candidate_node = node_cost_pair[0]
                #Compute the cost if we goes through this candidate
                candidate_cost = node_cost_pair[1] + cost_table[node]
                # The relaxation step
                if (candidate_node != node) and ((candidate_cost < cost_table[candidate_node]) or (candidate_node not in cost_table)):
                    cost_table[candidate_node] = candidate_cost
                    # This ndoe is now on a short path, save it
                    sp_nodes[candidate_node] = node
        self.spcost = cost_table
        # now reconstruct the routes table
        for dst_node in cost_table.keys():
            # Look into the set of nodes that are on the shortest path
            node = dst_node
            while node != src_node and sp_nodes[node] != src_node:
                # Search back the shortest path towards dst_node
                node = sp_nodes[node]
            # Reach the source node, fill the routing table with the link towards dst_node
            self.routes[dst_node] = self.getlink(node)

    # Let's clear the current routing table and rebuild it.  The hard
    # work is done by run_dijkstra().
    def integrate(self, time):
        self.routes.clear()
        self.routes[self.address] = 'Self'
        #initialize our own LSA
        self.LSA[self.address] = [self.LSA_seqnum] + \
                                 self.make_ls_advertisement()
        nodes = self.get_all_nodes()
        self.spcost = {}
        for u in nodes:
            self.spcost[u] = self.INFINITY
        self.spcost[self.address] = 0 # path cost to myself is 0

        self.run_dijkstra(nodes)

    def transmit(self, time):
        Router.transmit(self, time)
        if (time % self.ADVERT_INTERVAL) == self.ADVERT_INTERVAL/2:
            self.integrate(time)

    def OnClick(self,which):
        if which == 'left':
            print(self)
            print('  LSA:')
            for (key,value) in self.LSA.items():
                print('    ',key,': ',value)
        Router.OnClick(self,which)

# A network with nodes of type LSRouter
class LSRouterNetwork(RouterNetwork):
    def make_node(self,loc,address=None):
        return LSRouter(loc,address=address)

########################################################################

if __name__ == '__main__':

    gui = False
    numnodes = 12
    simtime = 2000
    rand = False

    if rand == True:
        rg = RandomGraph(opt.numnodes)
        (NODES, LINKS) = rg.genGraph()
    else:
        # build the deterministic test network
        #   A---B   C---D
        #   |   | / | / |
        #   E   F---G---H
        # format: (name of node, x coord, y coord)

        NODES =(('A',0,0), ('B',1,0), ('C',2,0), ('D',3,0),
                ('E',0,1), ('F',1,1), ('G',2,1), ('H',3,1))

        # format: (link start, link end)
        LINKS = (('A','B'),('A','E'),('B','F'),('E','F'),
                 ('C','D'),('C','F'),('C','G'),
                 ('D','G'),('D','H'),('F','G'),('G','H'))

    # setup graphical simulation interface
    if gui == True:
        net = LSRouterNetwork(simtime, NODES, LINKS, 0)
        sim = NetSim()
        sim.SetNetwork(net)
        sim.MainLoop()
    else:
        p3_tests.verify_routes(LSRouterNetwork)
    time.sleep(3)
    os._exit(0)
```

```
Test case #1: Euclidean network with no broken links...
        A---B   C---D
        |   | / | / |
        E   F---G---H
        link costs = 1 on straight links, sqrt(2) on diagonal links
Test case #1: Pass with convergence time: 30

Test case #2: Non-Euclidean network with no broken links...
         A-7-B-1-E
         |     / |
         1  2/   9
         | /     |
         C-4-D-1-F
Test case #2: Pass with convergence time: 30

Test case #3: Network with cost changes...
          Y
         / \
        X---Z
        Costs: XY=4 YZ=1 ZX=12
Now change cost ZX to 2
Your convergence time after changing cost ZX to 2: 30
...Good, now change back
Now change cost XY to 14
Your convergence time after changing cost XY to 14: 30

Test case #4: Network with cost changes and broken links...
        A---B-4-C---D
        |   | /2X /2|
        E---F---G---H
        Costs: BC=4 DG=2 CF=2 CG broken; all other costs are 1
Before any cost change or link breaks, your convergence time: 30

Now breaking links CF, CG, DG; changing costs BF to 15, CD to 13
Test case #4: Pass with convergence time: 30

Test case #5: Disconnected network...
        A---B   C---D
        |   | / | / |
        E   F---G---H
        link costs = 1 on straight links, sqrt(2) on diagonal links
Now breaking links F-C and F-G ('X' marks the broken links)
        A---B   C---D
        |   | X | / |
        E   F-X-G---H
Test case #5: Pass with convergence time: 30
```

Task 4.1: Implement the *make_ls_advertisement* function, which constructs the link state advertisement based on the current knowledge of the node. The current link state is stored in a python dictionary (or equivalently a hashmap) self.neighbors, and can be retrieved by calling *self.neighbors.values()*, which returns a list of [(timestamp1, neighbour1, linkcost1), (timestamp2, neighbour2, linkcost2), …] (Hint: you don't need the timestamps). Your code should return a list, containing all the link state information possessed by this node.

Task 4.2: (optional) Implement the *run_dijkstra(self,nodes)* function, which computes the shortest path from this node to all other nodes in the network. The input parameter nodes is all the nodes we know about within the network. Here is some extra data structure you might need to reference.

self.LSA, which is the database of the received link state advertisement. It is again a python dictionary [advertiser, linkstate], which maps a node that sent this advertisement to the advertised link state. The linkstate is in a list [seqnum, (node1, cost1), (node2, cost2), …]. The first element seqnum denotes the "freshness" of this advertisement, and is increased by 1. The rest (node, cost) pairs represent the neighbours of the advertiser node, and the link cost.

Your code should be able to process the information encoded in *self.LSA*, and update the *self.routes* and *self.spcost* accordingly. Recap the Dijkstra algorithm from exercise 2, or Wiki: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. Run the code to check if your code passes the test cases.

Task 4.3: (optional) Run both the distance-vector and link-state routing protocols. Check the convergence time of each test case: what can you observe? Can you explain why? Note that in our simulation, each node checks whether its neighbours are "alive" every 5 timestamps, and advertises to them at an interval of 20.

Your answer: Link-state converge faster. In Distance-vector, each node has to run the routing algorithm (Bellman-Ford) before forwarding the routing information to its neighbors. As a result, the larger the network, the slower the convergence. However, in link-state, the router can directly forward the link-state information without running the routing algorithm, so it would coverge faster.