Network Topology Discovery

Yan Jiaqi
A20321362
CS542 Project
Lectured by Prof. Edward Chlebus

December 6, 2015

Abstract

In this project, we first propose the content of topology update message for correct topology discovery. The *portion of the network seen by a router* is used to quantify the efficiency of the network update process. A small scale network built by 6 routers is simulated to demonstrate the process of populating topology information. The simulation, as well as the distributed discovery algorithm, should terminate when every router's *topology database* is stable. Simulation results are shown for both router R_1 and router R_6.

Contents

1 Problem Statement

In this project, we explore the problem of letting distributed routers know the entire network topology. As the 'God' of the network, we network operator have the global view of the network. The routers, however, only knows its direct neighbors. Since we are considering unidirectional links, **neighbors** of a particular router r can fall into one of two cases: the one that can be reached from r and the one that can reach r.

The problem can be stated more formally as follows. The network topology as the adjacent matrix AM is given to us. An example which we will simulate is shown below:

	R_1	R_2	R_3	R_4	R_5	R_6
R_1		3				
R_2	4		6			
R_3				7		
R_4					11	
R_5						9
R_6		8	5			

The cost from source router s to destination d is given as $AM[s][d]$. Empty cell means the cost is ∞, meaning no link exist between router s and d. Every router r maintain its **topology database**. Initially, this database in a given router r contains only row r and column r in matrix AM because

- r is aware the routers it can directly go to, corresponding to row $AM[r][0..N-1]$, all the outgoing links.
- r knows the routers that can directly go to itself, corresponding to column $AM[0..N-1][r]$, all the incoming links.

The problem is: how to let each router has the entire topology matrix or database? The motivation is that if a router has the complete topology matrix, it can further run Dijkstra's algorithm to obtain a shortest path tree, whose root is the router r itself. Next hop for any network or any other router can then be obtained from this shortest path tree. In other words, it can automatically and dynamically create/update its routing table[1].

[1] Just out of my interest, I implemented the procedure of routing table creation based on Dijkstra's shortest path algorithm. Not described in this report but can be found in the source code for router

2 Discovery Process

2.1 Message Format

The solution to topology discovery itself is simple. Every router advertises its current known links (column and row in its incomplete database) to its downstream neighbors. Each advertising message consists of 2 parts.

The first field k_m is called the key of the message. It is a unique ID for a router in the network, which, playing as a *key*, indicates the row index and column index in the imaginary topology matrix.

Then goes the links encapsulated inside the message. It is a 2-element tuple containing associated links of the router identified by the key k_m. In terms of topology matrix, this tuple contains the row and the column corresponding to this keyed router. Both elements in this tuple are key-value map structure: the keys are still a router ID and its values are the cost of links. However, the interpretation of the cost is different for the 1st and 2nd element in this tuple. The 1st elements specifies the cost of outing links while the 2nd element specifies the incoming links. For example, the cost value associated with a key k_1 in the first element represents the cost from router k_m to router k_1; for the cost value associated with k_2 in the second element, it represents the cost from router k_2 to router k_m.

An advertisement in one iteration from any router may contains a sequence of such messages if the source router has more than 1 record in its database. Once a router receives advertisements from other routers, it checks if there is anything new to its own database. If yes, it inserts a record, the (outgoing link cost, incoming link cost) tuple to its database with the key of the message as table index.

2.2 Measuring the Discovery Process

An obvious metric is the percentage of the network topology matrix a router currently has. For the example mentioned above, at the initial stage, every router only knows 1 row and 1 column of the matrix. So everyone starts from percentage $1/N$ where N is total number of routers in the network. At iteration $n = 1$, R_2 received messages from router R_1 and R_6; then it has 3 rows and 3 colums, which means 50% of the network is known to R_2.

3 Simulation and Termination

We can simulate the distributed discovery process. Simulation's basic flow is shown in Listing 1. Notice that this simulation assume that all routers are **synchronized** after each iteration.

3.1 Termination

The discovery step is repeated until all of the routers' topology database is stable. That is, we jump out of the while-loop when the boolean variable *updated* is never set to $True$ when any router r invokes `update_topo()`. The reason is simple: if every router's topology database is identical, no new information will be transferred between each other. At this moment, all routers must have the same and correct network view, except for that something happened after the population ends.

In fact, if assuming that no change happens during the topology discovery/update process, it will always terminate in L iterations, where L is the length of the longest path in the network graph. Therefore, the worst case time complexity of this distributed algorithm is bounded by $O(N)$, where N is the number of routers in the graph. In the case that network is changed, just restart the discovery process at each router and wait for them to converge again. This topology update process can also be done incrementally.

3.2 Code Structure

Inside the while-loop of simulation, we have a scatter-gather pattern. In the scatter phase, router r will receive messages from its each neighbors n who can reach r directly. In the gathering phase, each router update its topology databased by handling the received messages, which is stored inside its buffer at the scatter phase. The processing is very simple: if router r have seen a message with unknown key, it inserts the links inside this message to its database with this unseen key.

Listing 1: Simulation Discovery Process

```
while updated:
    for r in routers:
        for n in r.neighbors:
            router_n = get_router_by_name(n, routers)
            # send items in topology database one by one
            for n_name, n_row_col in router_n.topo.items():
                n_msg = (n_name, n_row_col)
                r.recv_msg(n_msg)
    updated = False
    for r in routers:
        if r.update_topo_database():
            # something new is added to a router
            updated = True
```

4 Experiment Results

In this section, the discovery progresses for router R_1 and R_6 are shown.

In the log that shows each iteration result, we use "()" to group elements in tuple structure; we use "{}" to group a sequence of key-value items and ":" to separate key with value. For example, (e_1, e_2, e_3) is a tuple with 3 elements; $\{k_1 : v_1, k_2 : v_2\}$ is a dictionary contains 2 key-value items; $(\{k_{11} : v_{11}, k_{12} : v_{12}\}, \{k_{21} : k_{22}\})$ is a tuple of two dictionaries. This syntax is entirely adopted from Python's programming syntax.

The topology database of a router at a particular iteration is printed out as a list of tuples, surrounded by "[]" symbols. Each tuple represent a record of the database, first element as index and second element as content. The content is a two-element tuple depict link information.

4.1 Results for R_1

At each iteration, the snapshot of R_1's topology database is shown in Listing 2. Each entry in the database is indexed by router name or id. Inside the entry stores the links, both outgoing and incoming type, of the index router. For example, at the 3rd iteration, R_1 found out 3 other routers exist in the network. One of them is called $R6$, who can reach R_2 and R_3 with cost 8 and 5 respectively; also this R_6 can be reached from another router R_5 with cost 9. At 5th iteration and 6th iteration, R_1 has the same topology database. In fact all routers at the 6th iteration have the same topology database and meet the termination requirement. This stable database is also the complete representation of the network.

Listing 2: R_1's Topology at Each Iteration

```
0   [   ('r1', ({'r2': 3}, {'r2': 4}))   ]
1   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3}))   ]
2   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
3   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
4   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
5   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
```

```
6   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
```

4.2 Results for R_6

At each iteration, the snapshot of R_6's topology database is shown in Listing 5. For example, at the 3rd iteration, R_1 found out 3 other routers exist in the network. One of them is called R_3, who can reach R_4 with cost 7; also this R_3 can be reached from either router R_6 with cost 5 or router R_2 with cost 6. At 5th iteration and 6th iteration, R_6 has the same topology database. In fact all routers at the 6th iteration have the same topology database and meet the termination requirement. So the number of iterations is 7 for our simulation.

Listing 3: R_6's Topology at Each Iteration

```
0   [   ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
1   [   ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
2   [   ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
3   [   ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
4   [   ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
5   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
6   [   ('r1', ({'r2': 3}, {'r2': 4})),
        ('r2', ({'r1': 4, 'r3': 6}, {'r6': 8, 'r1': 3})),
        ('r3', ({'r4': 7}, {'r6': 5, 'r2': 6})),
        ('r4', ({'r5': 11}, {'r3': 7})),
        ('r5', ({'r6': 9}, {'r4': 11})),
        ('r6', ({'r2': 8, 'r3': 5}, {'r5': 9}))   ]
```

4.3 Discovery Percentage as a function of the iteration number

As a function of iteration, the percentage of network discovered by both router R_1 and R_6 is shown in Figure 1. The iteration number begins from 0 and ends at 6, an extra iteration to ensure that all routers in this network hold identical topology database. As we can see the progress grows linearly at each iteration: at each step, router R_1 and R_6 found a new router and its links.

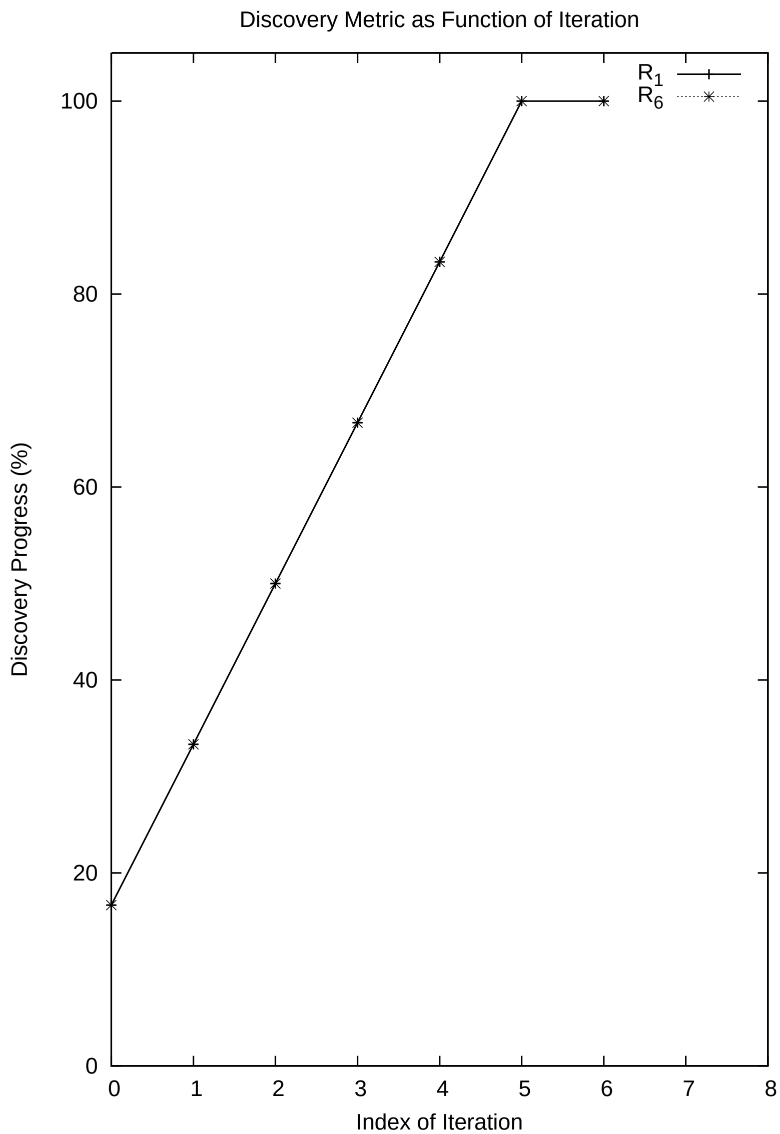

Figure 1: Discovery Percentage as Function of Iteration for R_1 and R_6

A Source Code Printout

The printout of all source code used in this project.

Listing 4: Source Code for Router: drrouter.py

```
#!/usr/bin/python

class DRNode(object):
    "Extra info used by a router to compute shortest path"
    def __init__(self, name, adjacents):
        """Create DRNode object

        name(str) -- unique name of the router
        adjacents(list) -- all the names of its adjacents
        dist -- distance to this node from the running router
        pi -- previous hop to this node from running router
        """
        super(DRNode, self).__init__()
        self.name = name
        self.adjacents = adjacents
        # use an arbitrary large value as infinity
        self.dist = 999999999
        self.pi = None

class DRRouter(object):
    "Distributed routers"
    def __init__(self, name, topo=None):
        """Create router obect.

        name: unique string ID
        topo: Topology database: map router name to its
            corresponding row and column in the link matrix
            Row(a dictionary) stores outgoing link cost
            from this router; column(a dictionary) stores
            incoming link cost to this router
        recv_from: list of routers can be reached from
        recv_buffer: a dictionary storing the messages
            received from other routers
        converged: indicate if topology database is complete
        routing_table: next hop to certain destination
            based on Dijkstra's SP
        """
        super(DRRouter, self).__init__()
        self.name = name
        self.topo = topo
        self.recv_from = [] # send @self.topo to them
```

```
        self.discover_neighbors()
        self.recv_buffer = {}
        self.converged = False
        self.routing_table = {}

    def discover_neighbors(self):
        """Discovery routers that can reach self"""
        if self.name in self.topo.keys():
            # get column/incoming link of that tuple
            col = self.topo[self.name][1]
            for src in col.keys():
                if src not in self.recv_from:
                    self.recv_from.append(src)

    def recv_msg(self, msg):
        """Put messages into receiver's buffer

        msg -- a tuple/entry to be added to buffer
        msg[0] -- the router name
        msg[1] -- stores a tuple of (row, col)
        """
        if msg[0] not in self.recv_buffer.keys():
            self.recv_buffer[msg[0]] = msg[1]

    def update_topo(self):
        """Handle all the messages in the receive buffer

        updated -- return a flag to tell if topology
            database is updated
        """
        updated = False
        for router in self.recv_buffer.keys():
            if router not in self.topo.keys():
                self.topo[router] = self.recv_buffer[router]
                updated = True
        self.recv_buffer = {} # clear receive buffer
        return updated

    def discover_adjacents(self, router_name):
        """Discovery routers that a particular router
        can reach

        adjacents(list): router names
        """
        adjacents = []
        if self.converged:
```

```
            if router_name in self.topo.keys():
                row = self.topo[router_name][0]
                for dst in row.keys():
                    adjacents.append(dst)
        return adjacents

    def create_routing_table(self):
        """Dijkstra's shortest path algorithm"""
        def get_node_by_name(name, nodes):
            """Search DRNode in a list

            name -- provided search ID
            """
            for n in nodes:
                if n.name == name:
                    return n
            return None

        if not self.converged:
            return
        heap = []
        # create a node to each router in the network
        for router_name in self.topo.keys():
            adjacents = self.discover_adjacents(router_name)
            u = DRNode(router_name, adjacents)
            if u.name == self.name:
                u.dist = 0
            heap.append(u)

        while heap:
            heap.sort(key=lambda node: node.dist, reverse=False)
            # equivalent to heap.extract-min()
            u = heap.pop(0)
            self.routing_table[u.name] = {'dist': u.dist, \
                                          'pi': u.pi}
            # do relaxation for all u's neighbors
            for v_name in u.adjacents:
                v = get_node_by_name(v_name, heap)
                # v may be None if we have a loop in the topology
                if v and v.dist > u.dist + self.topo[u.name][0][v.name]:
                    v.dist = u.dist + self.topo[u.name][0][v.name]
                    v.pi = u
        # backtrack along the pi to get
        # the next hop for this router
        for u_name in self.topo.keys():
            if u_name != self.name:
```

```
                    next_name = u_name
                    prev = self.routing_table[u_name]['pi']
                    while prev.name != self.name:
                        next_name = prev.name
                        prev = prev.pi
                    self.routing_table[u_name]['next'] = next_name
                else: # next hop to itself is itself
                    self.routing_table[u_name]['next'] = u_name

    def print_routing_table(self):
        """Pretty print of router's routing table"""
        print "Routing table for %s" % self.name
        print '-' * 38
        for dst in self.routing_table.keys():
            print "%s\tcost = %d\tnext hop = %s" % \
                (dst, \
                self.routing_table[dst]['dist'], \
                self.routing_table[dst]['next'])
        print '-' * 38
        print
```

Listing 5: Source Code for Simulation, topology.py

```
#!/usr/bin/python

from drrouter import DRRouter

def simulate_topology_discovery():
    """Simulate the topology discovery process"""
    def get_router_by_name(name, routers):
        """Search DRRouter object with provided router name

        name -- the name of the router
        routers -- all routers in the network
        """
        for r in routers:
            if r.name == name:
                return r
        return None

    # initialize router's topology database
    r1 = DRRouter('r1', {'r1' : ({'r2':3},
                                  {'r2':4})})
    r2 = DRRouter('r2', {'r2' : ({'r1':4, 'r3':6},
                                  {'r1':3, 'r6':8})})
    r3 = DRRouter('r3', {'r3' : ({'r4':7},
```

```
                                 {'r2':6, 'r6':5})})
    r4 = DRRouter('r4', {'r4' : ({'r5':11},
                                 {'r3':7})})
    r5 = DRRouter('r5', {'r5' : ({'r6':9},
                                 {'r4':11})})
    r6 = DRRouter('r6', {'r6' : ({'r2':8, 'r3':5},
                                 {'r5':9})})
    routers = [r1, r2, r3, r4, r5, r6]

    for router in routers:
        print router.name, "will recv from", router.recv_from

    iter_count = 1
    num_routers = float(len(routers))

    # initialize progress array at iteration 0
    saw = float(len(r1.topo.keys()))
    p0 = saw / num_routers
    r1_progress = [p0]
    saw = float(len(r6.topo.keys()))
    p0 = saw / num_routers
    r6_progress = [p0]

    topo_base_file1 = open('r1.topo', 'w+')
    topo_base_file6 = open('r6.topo', 'w+')

    # termination flag: if all routers converged?
    updated = True
    while updated:
        # scatter phase: receive from its neighbors
        for r in routers:
            for n in r.recv_from:
                router_n = get_router_by_name(n, routers)
                for n_name, n_row_col in router_n.topo.items():
                    n_msg = (n_name, n_row_col)
                    r.recv_msg(n_msg)

        # gather phase: update topology database
        updated = False
        for r in routers:
            if r.update_topo():
                updated = True
                saw = float(len(r.topo.keys()))
                if r.name == 'r1':
                    topo_base_file1.write("%d  %s\n" % \
                        (iter_count, sorted(r.topo.items())))
```

```
                    r1_progress.append(saw / num_routers)
                if r.name == 'r6':
                    topo_base_file6.write("%d  %s\n" % \
                        (iter_count, sorted(r.topo.items())))
                    r6_progress.append(saw / num_routers)

        iter_count += 1
    # extra progress after stable
    r1_progress.append(1)
    r6_progress.append(1)

    # write progress results to file
    i = 0
    progress_file = open('progress.dat', 'w+')
    for p1, p6 in zip(r1_progress, r6_progress):
        progress_file.write("%d\t%3.3f\t%3.3f" % (i, p1, p6))
        i += 1
    progress_file.close()

    # run Dijkstra's shortest path on every router
    for r in routers:
        r.converged = True
        r.create_routing_table()
        r.print_routing_table()

if __name__ == '__main__':
    simulate_topology_discovery()
```