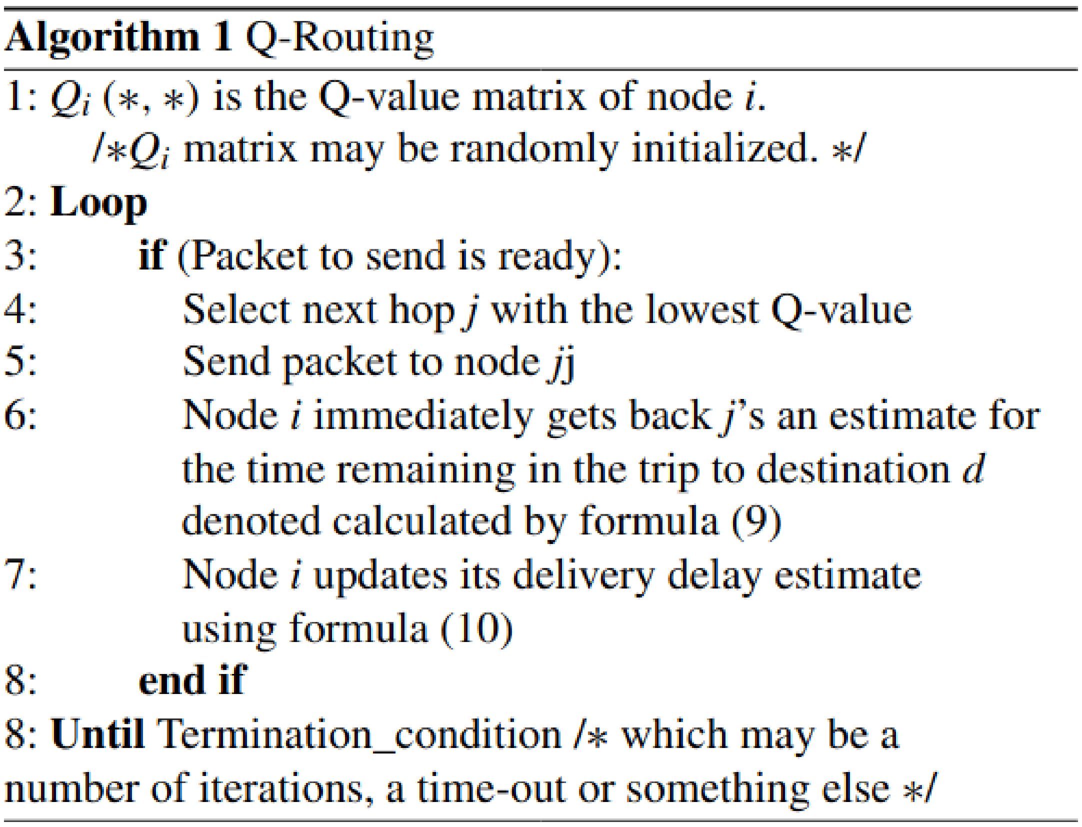
Q-Reinforcement Learning for Routing from Scratch

ELEN6772 - Machine Learning for Networks

Kristian Nikolov (kdn2117)

I initially set out to report on the impacts of transformers in reinforcement learning, but as I began my initial implementation, I found myself significantly more interested in the actual details of the implementation of Reinforcement Learning. Thus I set out to implement a simple reinforcement learning model for routing. This project was done in python 3.12 but should work in any version >=3.11, and only the only packages imported are those used for testing, linting and display.

The primary algorithm I used was the Q-Routing protocol as described in the paper "Reinforcement Learning Based Routing in Networks: Review and Classification of Approaches" [1].



```
Algorithm 1 Q-Routing
1: $Q_i(*, *)$ is the Q-value matrix of node $i$.
      /*$Q_i$ matrix may be randomly initialized. */
2: Loop
3:       if (Packet to send is ready):
4:           Select next hop $j$ with the lowest Q-value
5:           Send packet to node $j$j
6:           Node $i$ immediately gets back $j$'s an estimate for
             the time remaining in the trip to destination $d$
             denoted calculated by formula (9)
7:           Node $i$ updates its delivery delay estimate
             using formula (10)
8:       end if
8: Until Termination_condition /* which may be a
number of iterations, a time-out or something else */
```

Figure 1: Q-Routing Algorithm [1]

The formula used to update the delivery delay estimate is provided below in figure 2. Qi (d, j) represents the estimated delivery delay from node i through node j to node d, the destination. α represents the learning rate, TxT ij represents the transmission time from node i to node j, qti is the time spent by the packet in node i's queue, and θj(d) represents the initial lowest delay estimate that node j has among all of it's neighbors to the destination node d. This formula is also provided below in figure 3. Although I have implemented the full formula in my code, queue time is always set to 0 for every node for now.

$$Q_i(d,j) = (1-\alpha) * Q_i(d,j) + \alpha * \left(qt_i + TxT_{i,j} + \theta_j(d)\right) \qquad (10)$$

Figure 2: Updated delivery delay estimate formula [1]

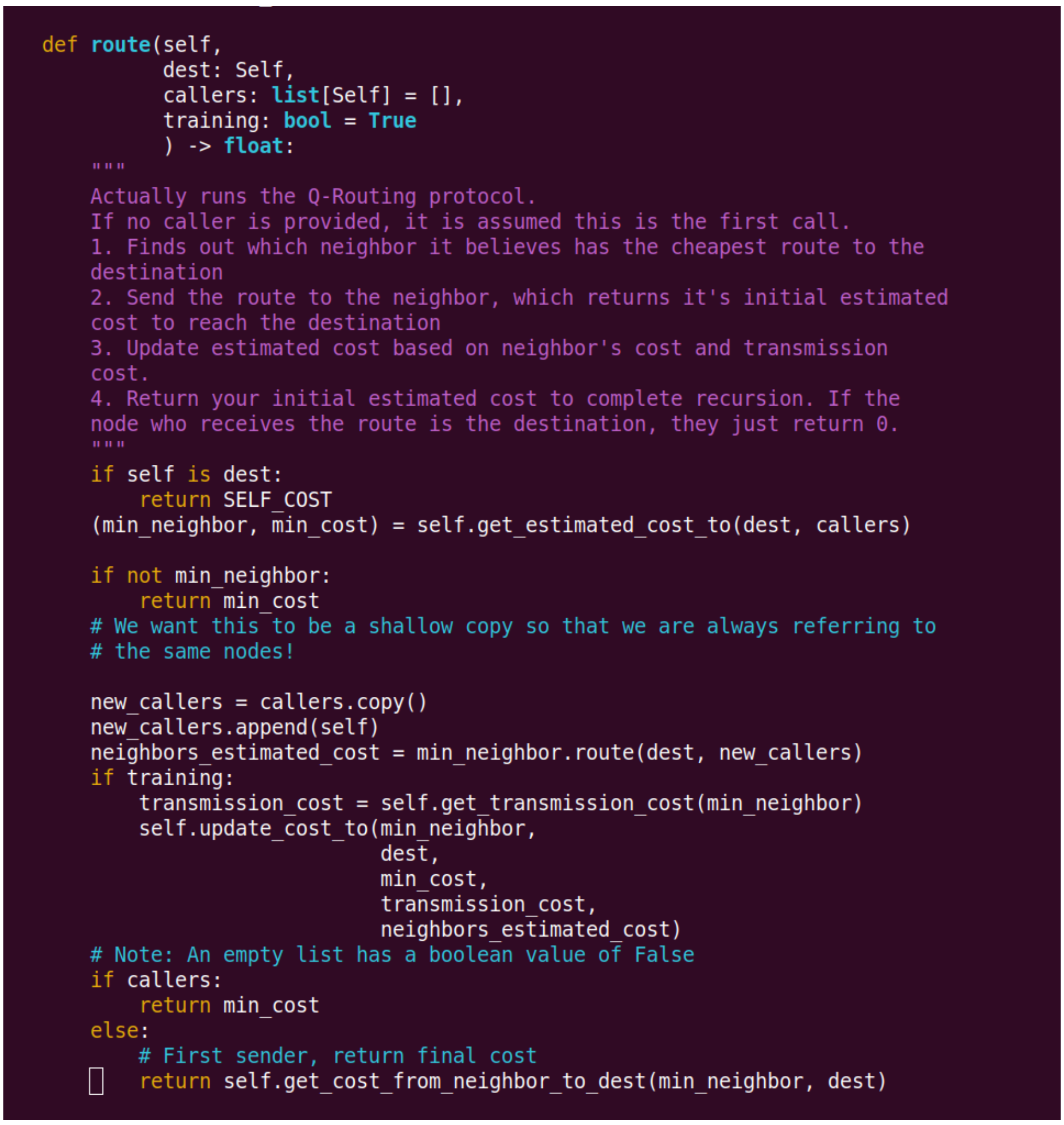
$$\theta_j(d) = \min_{k \in Ng(j)} Q_j(d,k) \qquad (9)$$

Figure 3: Formula for θj(d) [1]

The bulk of the project is in the Node class of the node.py module. The Node.create_nodes_from_table class method takes in a table of floats, where the value in row i and column j is a float between 0 and 1 and represents the transmission cost from node i to node j. Following this, all one has to do to begin the reinforcement learner training is to continuously provide a start node and destination node via the Node.get_node() class method, and then use the route method to run a full routing with the related transmission delay estimates being updated.

Each Node class object has only two attributes, the first being it’s index, which is an integer instance attribute. It’s uniqueness is maintained by the Node class which has a dictionary of all the current Node objects, with their indexes being the keys. The second class object attribute is the node’s neighbors, which is stored as a dictionary of dictionaries of Node objects. The outer dictionary’s keys are the node’s neighbors, and the inner dictionary associated with the neighbor has destination nodes as keys, and the original node’s transmission delay estimate to a destination from the neighbor as the value. As an example, if I have a node, node_0, which has neighbors node_1 and node_2, it’s neighbors attribute would look something like this: {node_1: {node_1: 0.5, node_2: 0.3}, node_2: {node_1: 0.7, node_2: 0.3}}. It should be noted, when the inner and outer dictionary node is the same, (EX: neighbors[node_1][node_1]) the value is the transmission time from the original node, in this case node_0, to that neighboring node, node_1.

The route method is passed the destination, a list of which nodes have been previously called in this route to prevent infinite recursion, as well as a training boolean to determine or not costs should be updated. It first checks if the destination is actually itself, if so it returns a SELF_COST constant, which is 0. It then gets the node with the minimum estimated cost to the destination. If there is no neighbor that can reach the destination, the get_estimated_cost returns a NOT_ACCESSIBLE_COST value, which is set to 1 above the MAX_COST, 1, for a total of 2. This was done so that in the future I can implement a system where after every iteration of routing, all estimated values are slightly decreased, in order to allow the system to adapt to new connections being formed, and eventually re-check neighbors it originally believed to be unable to reach the destination. The next step is to then add itself to the list of callers, and ask for the selected node to return it's estimated cost to the destination by recursively calling route. The penultimate step is to update the node's expected transmission delay from node j to the destination node. Finally the recursion is completed by returning the original estimated delay if there are any nodes in it's caller list. If there are none, that means that this node was the first to send out the packet and it returns its updated estimate value.



```python
def route(self,
          dest: Self,
          callers: list[Self] = [],
          training: bool = True
          ) -> float:
    """
    Actually runs the Q-Routing protocol.
    If no caller is provided, it is assumed this is the first call.
    1. Finds out which neighbor it believes has the cheapest route to the
    destination
    2. Send the route to the neighbor, which returns it's initial estimated
    cost to reach the destination
    3. Update estimated cost based on neighbor's cost and transmission
    cost.
    4. Return your initial estimated cost to complete recursion. If the
    node who receives the route is the destination, they just return 0.
    """
    if self is dest:
        return SELF_COST
    (min_neighbor, min_cost) = self.get_estimated_cost_to(dest, callers)

    if not min_neighbor:
        return min_cost
    # We want this to be a shallow copy so that we are always referring to
    # the same nodes!

    new_callers = callers.copy()
    new_callers.append(self)
    neighbors_estimated_cost = min_neighbor.route(dest, new_callers)
    if training:
        transmission_cost = self.get_transmission_cost(min_neighbor)
        self.update_cost_to(min_neighbor,
                            dest,
                            min_cost,
                            transmission_cost,
                            neighbors_estimated_cost)
    # Note: An empty list has a boolean value of False
    if callers:
        return min_cost
    else:
        # First sender, return final cost
        return self.get_cost_from_neighbor_to_dest(min_neighbor, dest)
```

Figure 4: route method

As an example of the entire system functioning, I created a simple example, with only 4 nodes. Node 0 has Node 1 as it’s neighbor, Node 1 has Node 0, Node 2 and Node 3 as neighbors, while Nodes 2 and 3 both only have Node 1 as a neighbor. The initial values are all 1 and shown below in figure 5. The numbers above each graph represent what the current destination node is. Note that the only values are 1 for the nodes that are neighbors with the destination, and 0 for nodes that aren’t.

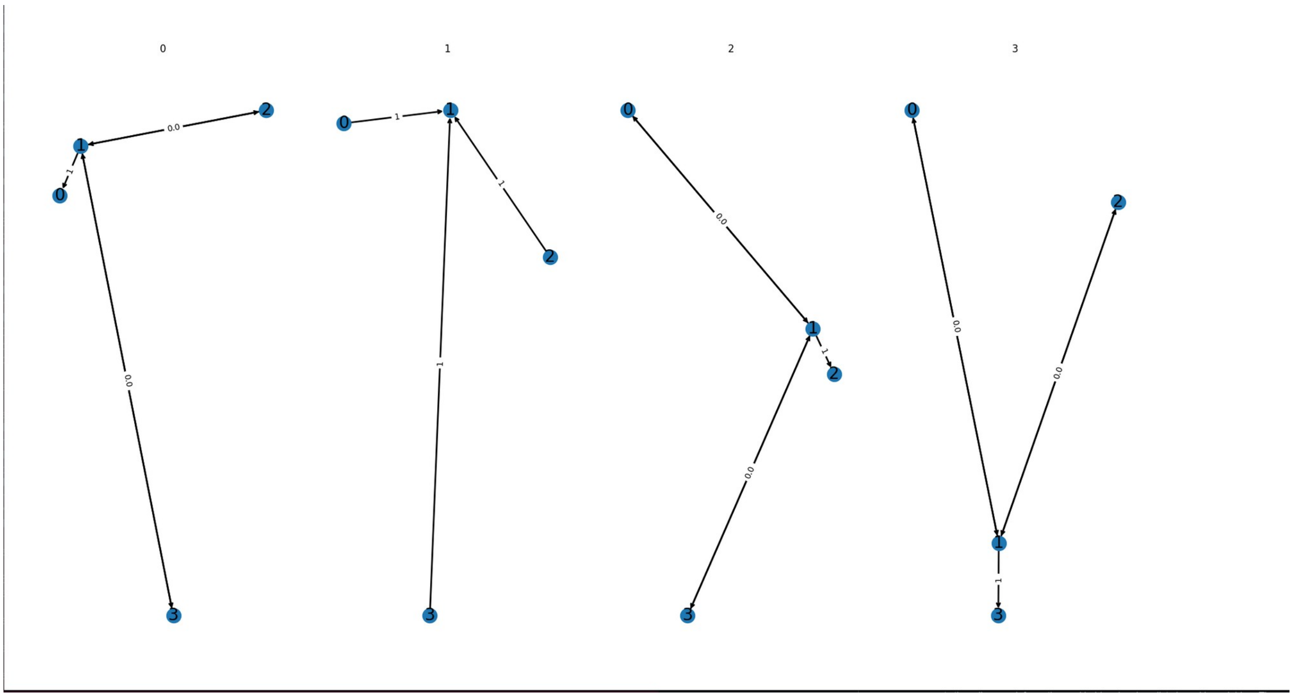


Figure 5: Initial values

After 100 iterations of random starting and destination nodes the values are shown below in figure 6. As we can see, Nodes that are neighbors with the destination still have values of 1, while nodes that are 1 node away have begun to converge to the correct value of 2.0. After another 100 iterations, as seen in figure 7, they are even closer.

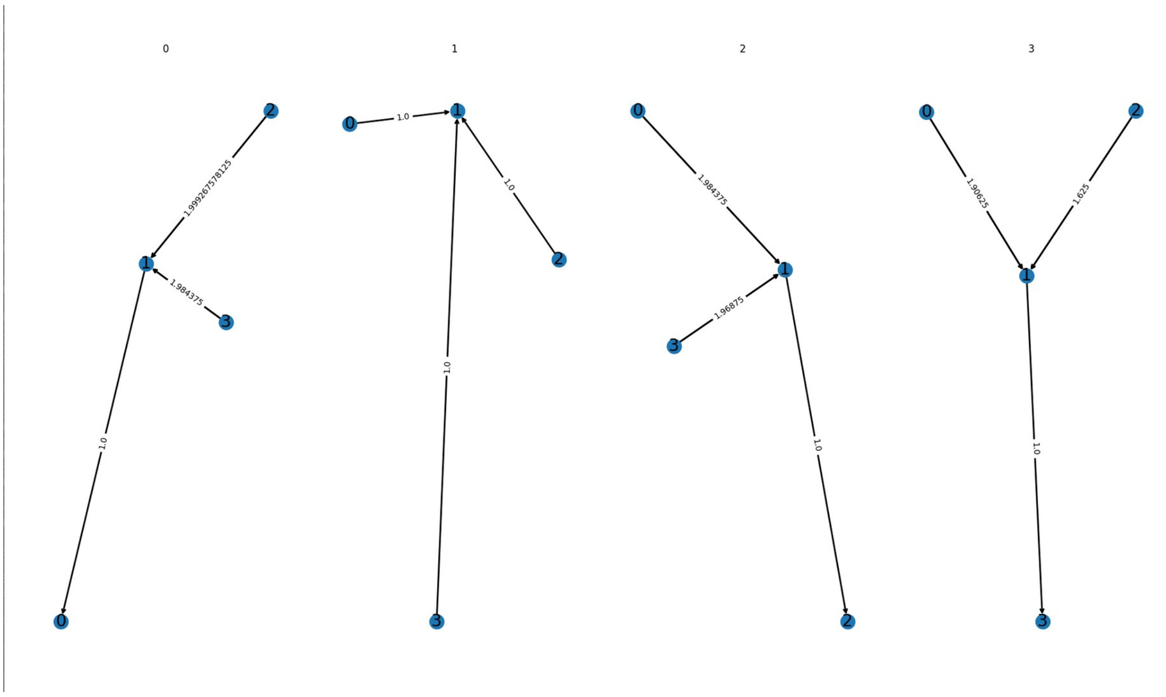


Figure 6: Values after 100 iterations

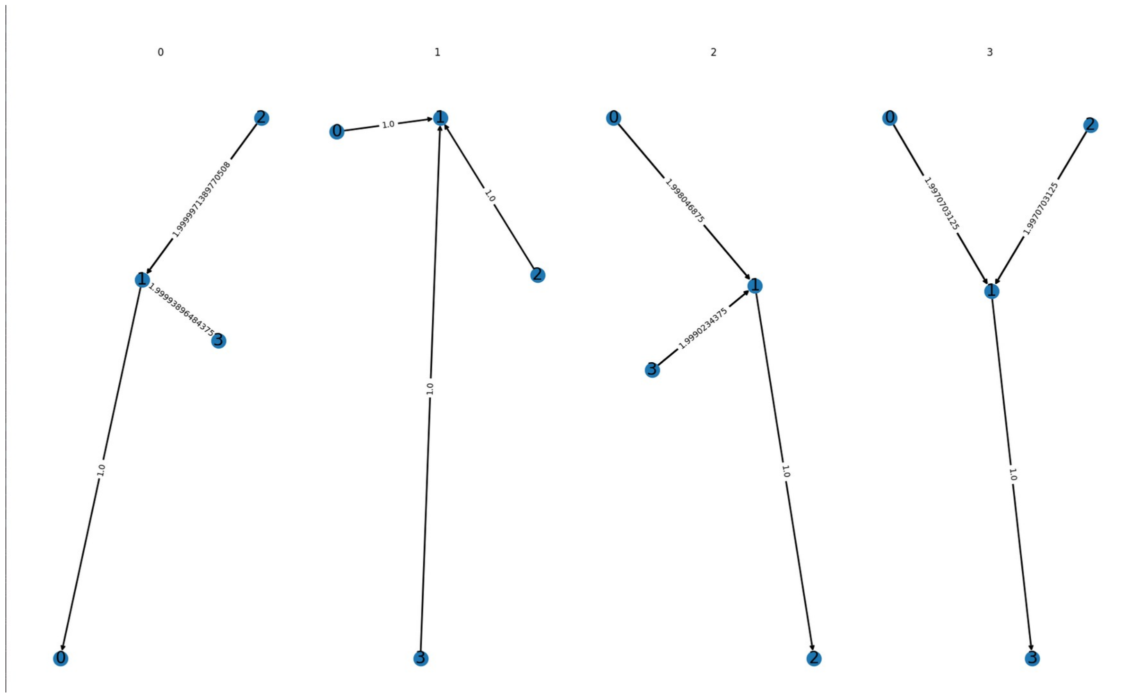


Figure 7: Values after 200 iterations

In conclusion, the project works in a basic form, although there is significant room for further work. Firstly, I was barely able to get the display to properly work as I do not have much experience with the matplotlib library. Although the algorithm works with significantly more complex network structures, it's very difficult to display them in a way where the user can gain any useful information. Beyond display, I would like to do further testing with removing and adding nodes during the simulation, and seeing how the network responds. With the way it is currently set up, node's should be able to respond to nodes disappearing, but not with new nodes appearing. This could be solved with fully implementing the gradual lessening of higher estimated costs as mentioned previously. Additionally, it would be interesting to implement both a way for transmission times to evolve during the simulation, as well as adding queue times.

All the code written has been provided as a zip of the git repository, as well as a link to the current github page. In order to run the project, you first need to create a virtual environment, then run 'make dev_env' to install the necessary packages, which are all in requirements.txt. Running 'make tests' ensures that everything is set up correctly and works. Then running 'python3 main.py' actually runs a simulation and displays the values at 3 different stages.

Ctations:

[1] Z. Mammeri, "Reinforcement Learning Based Routing in Networks: Review and Classification of Approaches," in *IEEE Access*, vol. 7, pp. 55916-55950, 2019, doi: 10.1109/ACCESS.2019.2913776. keywords: {Routing;Routing protocols;Computational modeling;Quality of service;Reinforcement learning;Optimization;Reinforcement learning;communication networks;routing protocols;path optimization;quality of service},