Simulation du routage de paquets sur un réseau

Emre UCAR

Table des matières

1 Introduction

La simulation du routage de paquets sur un réseau est un exercice éminemment important pour de nombreuses industries qui nécessitent une transmission efficace de leurs données sur différents réseaux, Internet étant le principal.

En particulier, les enjeux de modélisation et de routage sont primordiaux pour de nombreuses installations de surêté publique pour les réseaux d'alerte aux catastrophes naturelles ou bien, dans le futur, pour effectuer des opérations chirurgicales à distance, notamment dans les zones sinistrées.

L'objet de ce TIPE est de déterminer une approche intéressante pour le développement de simulations de réseaux ainsi que l'étude des avantages de quelques algorithmes de routage classiques.

Pour ce faire, on se donne un graphe $G = (V, E, f)$ avec V un ensemble de noeuds, $E \subset \mathcal{P}_2(V)$ et $f : E \to \mathbb{R}_+$ une fonction qui à chaque arête associe un poids positif.

On suppose par la suite que le graphe G est connexe.

Un chemin entre deux éléments u et v de V est une suite finie $p = (u_0, \dots, u_n)$ de noeuds tel que $u_0 = u$, $u_n = v$ et pour tout $i \in [1, n]$, $\{u_{i-1}, u_i\} \in E$. Pour un tel chemin, on définit le poids $w(p) = \sum_{i=1}^{n} f(u_{i-1}, u_i)$. Un chemin le plus court entre u et v est un chemin qui minimise ce poids $w(p)$. Ce chemin n'est pas nécessairement unique.

Dans un premier temps, on construit un réseau Internet avec une topologie inspirée du réseau Internet actuel, puis nous déployons sur ce réseau simulé différents protocoles de routage dont on évalue les performances respectives.

2 Génération de réseaux

2.1 Modélisation d'un réseau local « LAN » (`StarNetwork`)

Dans un premier temps, on peut s'intéresser à la topologie d'un réseau local (petite entreprise ou domicile d'un particulier).

Étant donnée la taille relativement faible de ces réseaux, leur topologie est souvent en étoile. [1] Ainsi, ce réseau possède un noeud central r auquel sont connectés tous les appareils du LAN.

Ce type de réseau est peu résilient car la panne du routeur entraîne la déconnexion du réseau dans son ensemble. En pratique, cela n'est pas vraiment problématique pour les réseaux résidentiels.

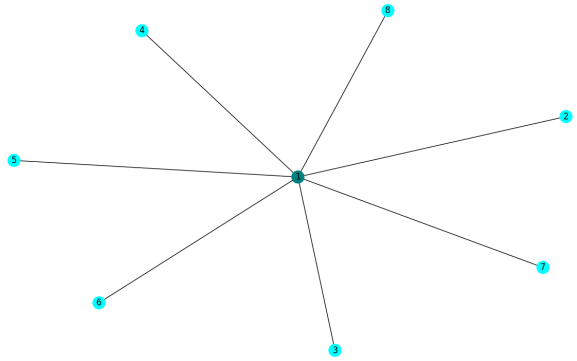

FIGURE 1 – Exemple de réseau en étoile de $n = 10$ noeuds dont le noeud central

2.2 Modélisation d'un réseau « backbone » (`MeshNetwork`)

Dans un second temps, on cherche à modéliser efficacement le réseau interne d'un fournisseur d'accès à Internet. Cela a été historiquement plutôt difficile. [3]

Ici, étant donné un ensemble V de noeuds et un entier k. Dans un premier temps, on connecte aléatoirement les noeuds avec un nombre minimal $|V| - 1$ arêtes afin de rendre le réseau connexe. Ensuite, on tente de connecter chaque noeud $x \in V$ à k autres noeuds. Si, lors de l'exécution de l'algorithme, un lien est établi plusieurs fois, on ne le compte qu'une seule fois. Cette algorithme permet d'obtenir des réseaux connexes et relativement bien distribués.

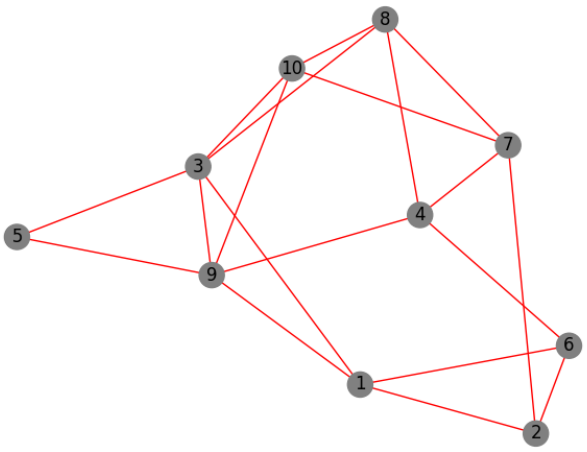

FIGURE 2 – Exemple de réseau « backbone » avec $|V| = 10$ et $k = 2$

2.3 Modèle par tiers

En pratique, le réseau Internet est divisé en sous-réseaux.[2] Ceux-ci sont les réseaux étendus formant la colonne vertébrale d'Internet, les « WAN » (Wide Area Network), les réseaux urbains de FAI « MAN » (Metropolitan Area Network) et enfin les réseaux d'entreprises ou de particuliers

« LAN »(Local Area Network). Les MAN et les WAN sont des instances de `MeshNetwork` et les LAN des instances de `StarNetwork`.

Chaque LAN est connecté à un MAN, chaque MAN à un WAN et les WAN sont connectés entre eux de sorte à rendre le réseaux connexe.

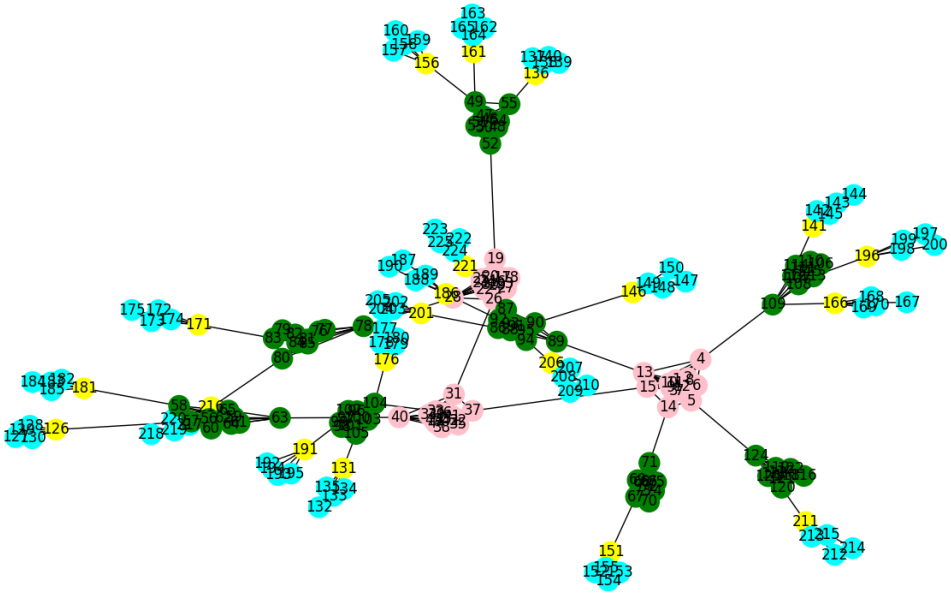

FIGURE 3 – Exemple de réseau par tiers.
Légende : Noeud WAN, Noeud MAN, Noeud central LAN, Noeud périphérique LAN

3 Algorithmes de routage de paquets

Maintenant que nous avons généré des réseaux, il s'agit de router efficacement des paquets entre deux noeuds quelconques du réseau.

3.1 Principe d'optimalité et propriété

Dans un premier temps, il est nécessaire de définir correctement l'optimalité. En effet, cette définition fluctue fortement selon les systèmes autonomes considérés et les objectifs du FAI.[1] Afin de simplifier la simulation, on suppose que la fonction f détermine entièrement cette optimalité et tous les routeurs partagent cette même définition.

Dans cette simulation, le poids est borné et définit aléatoirement.

Dans la suite, les algorithmes présentés vont souvent utiliser un résultat important sur les graphes. Celui-ci stipule que si $(u_0, \ldots, u_n)$ est un chemin le plus court entre u_0 et u_n, alors tous ses sous-chemins (ie les suites $(u_i, \ldots, u_j)$ pour $0 \leq i \leq j \leq n$) sont également des chemins les plus courts.

En effet, si tel n'était pas le cas, cela contredirait immédiatement le caractère minimal du chemin initial.

3.2 Routage par inondation

Le routage par inondation consiste à envoyer le paquet reçu au niveau de chaque routeur à tous ses voisins immédiats sauf le voisin d'où vient le paquet.

Cette méthode semble *a priori* particulièrement inefficace. Sans dispositif visant à limiter le temps de vie des paquets, leur nombre diverge d'autant plus rapidement que le degré moyen des noeuds augmente.

Toutefois, l'algorithme utilisé est extrêmement simple, peu gourmand en mémoire et assure de router le paquet vers sa destination en empruntant le chemin le plus court, étant donné qu'on emprunte tous les chemins possibles.

3.3 Routage par vecteur distance

Afin de réduire l'encombrement du réseau tout en garantissant un routage efficace, une deuxième approche consiste à garder en mémoire pour chaque destination le noeud auquel il faut envoyer le paquet afin d'atteindre ladite destination ainsi que le poids du chemin à emprunter.

Initialement, chaque table ne contient que les plus proches voisins. Chaque routeur annonce ensuite régulièrement sa propre table à ses voisins. Lorsqu'un routeur reçoit une table d'un voisin, il met à jour sa propre table s'il remarque que celui-ci annonce un meilleur itinéraire.

À l'aide d'une structure de donnée adéquate, on peut obtenir un routage au niveau d'un routeur en temps constant $O(1)$ et une taille en mémoire en $O(|V|)$ soit $O(|V|^2)$ pour l'ensemble du réseau.

Les tables de routage convergent en $O(d|V|)$ avec d diamètre du graphe en supposant les poids des arêtes bornés et une transmission en temps linéaire en la taille du paquet. En effet, c'est le temps asymptotique nécessaire pour que les annonces envoyées soient reçues par tous les noeuds du réseau.

3.4 Routage par état de lien

Une dernière méthode consiste à diffuser la topologie complète du réseau à l'ensemble des noeuds afin que chaque noeud applique un algorithme de plus court chemin, tel l'algorithme de Dijkstra, afin de router le paquet.

Dans un premier temps, chaque noeud a uniquement connaissance de ses voisins directs. Graduellement, les voisins s'échangent leurs connaissances de la topologie du graphe du réseau jusqu'à ce que tous les noeuds puissent reconstituer ledit graphe dans leurs mémoires respectives.

Cette méthode est la plus lourde en terme de temps de calcul ($O(|E| \log |V|)$ au niveau de chaque noeud) et de mémoire utilisée ($O(|E| + |V|)$ au niveau de chaque noeud) mais c'est également la plus robuste car les changements de topologie sont rapidement communiqués à l'ensemble du réseau.

La vitesse de convergence est également en $O(d(|E| + |V|))$. À ce moment, chaque noeud a reçu les états de lien de tous les autres noeuds du réseau.

4 Mesures de performance

On cherche maintenant à évaluer les performances des différents algorithmes sur des réseaux de taille variable. Pour cela, on génère aléatoirement des instances de `MeshNetwork` de paramètres $n \in \{50,\ 100,\ 500\}$ et $k = 2$ et on route un paquet entre deux noeuds choisis aléatoirement sur le graphe.

Pour chaque itération ($m = 50$), on mesure la taille du chemin, le nombre de noeuds ayant reçu le paquet, le temps réel d'initialisation ainsi que le temps réel de routage.

4.1 Données

Les données récupérées sont compilés dans les tableaux qui suivent :

Algorithme	Longueur du chemin	Noeuds touchés	Temps initialisation	Temps routage
Inondation	3.68 ± 0.29	39.6 ± 11.7	1.52 ± 1.07 ms	0.405 ± 0.105 ms
Vecteur distance	4.54 ± 0.39	3.54 ± 0.39	20.0 ± 1.72 ms	0.240 ± 0.055 ms
État de lien	4.14 ± 0.36	3.14 ± 0.37	184 ± 8 ms	0.375 ± 0.071 ms

TABLE 1 – Performance de différents algorithmes de routage (`MeshNetwork`, $n = 50$, $k = 2$)

Algorithme	Longueur du chemin	Noeuds touchés	Temps initialisation	Temps routage
Inondation	4.36 ± 0.37	107 ± 37	2.49 ± 0.09 ms	0.919 ± 0.279 ms
Vecteur distance	5.22 ± 0.41	4.22 ± 0.41	86.8 ± 7.2 ms	0.303 ± 0.028 ms
État de lien	5.28 ± 0.42	4.28 ± 0.42	963 ± 31 ms	1.17 ± 0.46 ms

TABLE 2 – Performance de différents algorithmes de routage (`MeshNetwork`, $n = 100$, $k = 2$)

Algorithme	Longueur du chemin	Noeuds touchés	Temps initialisation	Temps routage
Inondation	6.28 ± 0.40	743 ± 280	86.8 ± 6.3 ms	13.0 ± 3.9 ms
Vecteur distance	6.58 ± 0.48	5.58 ± 0.48	2.99 ± 0.08 s	1.48 ± 0.12 ms
État de lien	6.54 ± 0.40	5.54 ± 0.40	38.6 ± 4.9 s	6.54 ± 2.48 ms

TABLE 3 – Performance de différents algorithmes de routage (`MeshNetwork`, $n = 500$, $k = 2$)

4.2 Analyse

On constate dans un premier temps que la longueur du chemin est quasiment la même quelque soit le protocole utilisé et la taille du réseau. Cela donne une bonne indication que les algorithmes ont été implémentés correctement.

Toutefois, les autres métriques sont extrêmement contrastés selon l'algorithme et la taille du réseau considéré. Ainsi, comme prévu par la théorie, le nombre de noeuds ayant reçu le paquet est très supérieur pour le routage par inondation, dépassant même la taille du réseau pour $n = 100$

et $n = 500$ alors qu'il est égal au nombre de noeuds du chemin privé de 1 pour les deux autres algorithmes car le paquet n'est envoyé qu'une fois à chaque saut.

Cela signifie qu'en moyenne chaque routeur reçoit le paquet plusieurs fois lors d'une inondation. Mais en pratique, l'effet sur le temps de routage ne devient significatif que pour les grands réseaux.

Enfin, on constate que le temps d'initialisation est très faible lors de l'inondation mais qu'il croît exponentiellement avec les routages par vecteur distance et par état de lien car le partage initiale des informations se fait d'autant plus lentement que le réseau grandit. Cependant, en contre-partie pour les grands réseaux, le temps de routage est extrêmement faible pour le routage par vecteur distance et relativement faible pour le routage par état de lien comparé au routage par inondation.

5 Bibliographie

[1] Andrew Tanenbaum, David Wetherall, « Réseaux », Pearson, 5e édition, 2011

[2] K. L. Calvert, M. B. Doar and E. W. Zegura, "Modeling Internet topology", IEEE Communications Magazine, vol. 35, no. 6, pp. 160-163, June 1997, DOI 10.1109/35.587723, https://ieeexplore.ieee.org/document/587723/

[3] S. Floyd and V. Paxson, "Difficulties in simulating the Internet", IEEE/ACM Transactions on Networking, vol. 9, no. 4, pp. 392-403, Aug. 2001, DOI : 10.1109/90.944338, https://ieeexplore.ieee.org/document/944338

6 Annexe : Codes

6.1 Classes de base `base.py`

```python
from random import choice, shuffle, randint
import networkx as nx
import matplotlib.pyplot as plt

# base class for all networks
class NetworkGraph:
  def __init__ (self):
    self.graph = nx.Graph()
    self.total_packet_count = 0
    self.routing_iterator = iter([])
  def connect (self, node1, node2, weight=None):
    weight = weight if weight is not None else randint(1, 5)
    self.graph.add_edge(node1, node2, weight=weight)
  def show (self, get_edge_color=None, show_weight=False, file=None):
    get_edge_color = get_edge_color or lambda _, __, ___: 'black'
    pos = nx.spring_layout(self.graph, iterations=100)
    color_map = [node._get_color() for node in self.graph]
    edge_color = [get_edge_color(node1, node2, self.weight(node1, node2)) for
    node1, node2 in self.graph.edges()]
    nx.draw(self.graph, pos, with_labels=True, node_color=color_map, edge_color=
    edge_color)
    fig = plt.gcf()
    fig.set_size_inches(13, 8)
    if file:
      fig.savefig(file)
    plt.show()
```

```
25    # get the weight of a link given the two nodes
26    def weight (self, node1, node2):
27      if self.graph.has_edge(node1, node2):
28        return self.graph[node1][node2]['weight']
29      return None
30    # merge a network into this network
31    def add (self, net):
32      for node1, node2 in net.graph.edges():
33        self.connect(node1, node2, weight=net.weight(node1, node2))
34      # updating local network interfaces
35      for node in net.graph.nodes():
36        if node.interface is not None:
37          node.interface.network = self
38    # execute one step of global packet routing
39    def routing_step (self):
40      node = next(self.routing_iterator, None)
41      if node is None:
42        nodes = filter(lambda n: len(n.packet_buffer) != 0, self.graph.nodes())
43        nodes_list = list(nodes)
44        if nodes_list == []:
45          raise Exception('No route to host')
46        self.routing_iterator = iter(nodes_list)
47        return self.routing_step()
48      node._update_buffer()
49
50  # python does not provide private properties and methods
51  # this class acts as a network abstraction for a node
52  class LocalNetworkInterface:
53    def __init__ (self, node, network):
54      self.node = node
55      self.network = network
56    def get_neighbors (self):
57      if self.node not in self.network.graph:
58        return []
59      return self.network.graph.neighbors(self.node)
60    def get_neighbors_and_weights (self):
61      neighbors = self.get_neighbors()
62      return [(node, self.network.weight(self.node, node)) for node in neighbors]
63    def send_neighbor (self, node, packet):
64      if node not in self.get_neighbors():
65        raise Exception('Cannot send to this node : not my neighbor')
66      # copying to avoid routers manipulating the same object (ttl, ...)
67      # in some routing protocols like flooding
68      new_packet = packet.copy()
69      self.network.total_packet_count += 1
70      node._on_receive_packet(new_packet, self.node)
71
72  class Packet:
73    def __init__(self, data, src_addr, dest_addr, ttl=10, route=None):
74      self.data = data
75      self.ttl = ttl
76      self.src_addr = src_addr
77      self.dest_addr = dest_addr
78      self.route = route if route is not None else []
79    def is_dead (self):
80      return self.ttl == 0
81    def copy (self):
82      route = self.route.copy()
83      return Packet(self.data, self.src_addr, self.dest_addr, self.ttl, route)
84    def add_node (self, node):
85      self.route.append(node)
86
```

```
# base class for all nodes
class Node:
  def __init__ (self, addr):
    self.addr = addr
    self.interface = None
  def __str__ (self):
    return str(self.addr)
  def __hash__ (self):
    return self.addr
  # used by NetworkGraph to give a color to the node
  def _get_color (self):
    return 'gray'

class MeshNetwork(NetworkGraph):
  def __init__(self, nodes, deg_target):
    super().__init__()
    for node1 in nodes:
      for _ in range(deg_target):
        node2 = choice(list(filter(lambda x: x is not node1, nodes)))
        self.connect(node1, node2)

class StarNetwork(NetworkGraph):
  def __init__(self, center_node, orbiting_nodes):
    super().__init__()
    self.center_node = center_node
    self.orbiting_nodes = orbiting_nodes
    for node in orbiting_nodes:
      self.connect(node, center_node)

if __name__ == '__main__':
  nodes = [Node(i + 1) for i in range(10)]
  net = MeshNetwork(nodes, 3)
  net.show()
```

6.2 Génération de réseau `tiers.py`

```
from random import choice, shuffle, randint
from base import Node, NetworkGraph, MeshNetwork, StarNetwork

class TiersNode(Node):
  def __init__ (self, addr, type):
    super().__init__(addr)
    self.type = type
  def _get_color (self):
    if self.type == 'wan':
      return 'pink'
    if self.type == 'man':
      return 'green'
    if self.type == 'lan-center':
      return 'yellow'
    return 'cyan'

# fast list.remove
def pop2 (lst, i):
  lst[i], lst[-1] = lst[-1], lst[i]
  return lst.pop()

# control address attribution
class TiersNodeFactory:
  def __init__ (self, network, addr_start=0):
    self.addr_count = addr_start
    self.network = network
```

```
27    def make_node (self, type):
28      self.addr_count += 1
29      return TiersNode(self.addr_count, type)
30
31  class TiersNetwork(NetworkGraph):
32    def __init__ (self, T=1, NT=1, ET=2, ETT=1, S=1, NS=1, ES=2, L=1, NL=1):
33      super().__init__()
34      self.T = T # nbr of transit domains
35      self.NT = NT # avg nbr of nodes per transit domain
36      self.ET = ET # nbr of edges within transit domains
37      self.ETT = ETT # nbr of edges between transit domains
38      self.S = S # nbr of stub domains
39      self.NS = NS # avg nbr of nodes per stub domains
40      self.ES = ES # nbr of edges within stub domains
41      self.L = L # nbr of LANs
42      self.NL = NL # avg nbr of nodes per LAN
43      self.factory = TiersNodeFactory(self)
44      self._generate()
45    # strategy: generate individual WANs, MANs, and LANs
46    # afterwards, connect and merge networks in order to
47    # obtain a single connex network
48    def _generate (self):
49      make_node = self.factory.make_node
50      self.wans = [WideAreaNetwork(self.NT, self.ET, make_node) for _ in range(self.
        T)]
51      self.mans = [MetropolitanAreaNetwork(self.NS, self.ES, make_node) for _ in
        range(self.S)]
52      self.lans = [LocalAreaNetwork(self.NL, make_node) for _ in range(self.L)]
53      for lan in self.lans:
54        man = choice(self.mans)
55        man_node = choice(man.nodes)
56        man.add(lan)
57        man.connect(lan.center, man_node)
58      for man in self.mans:
59        wan = choice(self.wans)
60        wan_node = choice(wan.nodes)
61        wan.add(man)
62        wan.connect(choice(man.nodes), wan_node)
63      # making a connex component
64      subnets = self.wans.copy()
65      while len(subnets) > 1:
66        wan1 = pop2(subnets, randint(0, len(subnets) - 1))
67        wan2 = pop2(subnets, randint(0, len(subnets) - 1))
68        node1 = choice(wan1.nodes)
69        node2 = choice(wan2.nodes)
70        wan1.add(wan2)
71        wan1.connect(node1, node2)
72        subnets.append(wan1)
73      self.add(subnets[0])
74      # satisfying ETT
75      wans = self.wans.copy()
76      for _ in range(self.ETT - len(wans)):
77        wan1 = choice(wans)
78        wan2 = choice(wans)
79        if wan1 is not wan2:
80          node1 = choice(wan1.nodes)
81          node2 = choice(wan2.nodes)
82          self.connect(node1, node2)
83
84  # WAN and MAN base
85  class MeshPartNetwork(MeshNetwork):
86    def __init__ (self, type, n, m, make_node):
```

```
87      self.nodes = [make_node(type=type) for _ in range(n)]
88      self.type = type
89      super().__init__(self.nodes, m)
90
91  # WAN
92  class WideAreaNetwork(MeshPartNetwork):
93    def __init__ (self, n, m, make_node):
94      super().__init__('wan', n, m, make_node)
95
96  # MAN
97  class MetropolitanAreaNetwork(MeshPartNetwork):
98    def __init__ (self, n, m, make_node):
99      super().__init__('man', n, m, make_node)
100
101 # LAN
102 class LocalAreaNetwork(StarNetwork):
103   def __init__ (self, node_count, make_node):
104     self.center = make_node(type='lan-center')
105     nodes = [make_node(type='lan') for _ in range(node_count - 1)]
106     super().__init__(self.center, nodes)
107
108 if __name__ == '__main__':
109   net = TiersNetwork(T=3, NT=15, ET=8, ETT=5, S=8, NS=10, ES=5, L=20, NL=5)
110   net.show()
```

6.3 Routage sur le réseau `routing.py`

```
1   from random import choices
2   from timeit import timeit
3   from math import sqrt
4   from time import time
5   import networkx as nx
6   from base import Node, LocalNetworkInterface, MeshNetwork, Packet
7
8   class RoutableNode(Node):
9     def __init__ (self, addr):
10      super().__init__(addr)
11      self.packet_buffer = []
12      self.packets_received = []
13    # can be called by the network or the node factory to change the interface
14    # used especially during network merge operations
15    def set_interface (self, interface):
16      self.interface = interface
17    def _route_packet (self, packet, sender):
18      # INFO: routing logic goes here
19      # INFO: override this method to implement a routing strategy
20      raise Exception('No routing algorithm')
21    # this method is called by the network to indicate the router that it can
22    # process its packet buffer
23    # at that point, all buffer times are decremented and suitable packets
24    # are routed
25    def _update_buffer (self):
26      def filter_buffer (b):
27        packet, sender, buffer_time = b
28        if buffer_time > 0:
29          return True
30        self._route_packet(packet, sender)
31        return False
32      self.packet_buffer = list(filter(filter_buffer, self.packet_buffer))
33      for b in self.packet_buffer:
34        b[2] -= 1
35    # internal method called when receiving a packet
```

```
36    # this method is called before the _route_packet method
37    # it implements TTL and reception of packets intended to this node
38    def _on_receive_packet (self, packet, sender):
39      packet.add_node(self)
40      if packet.dest_addr == self.addr:
41        self.packets_received.append(packet)
42        print(self, ':: received packet:', packet.data, packet.route)
43        return
44      packet.ttl -= 1
45      if packet.is_dead():
46        return
47      if sender is None:
48        buffer_time = 0
49      else:
50        buffer_time = self.interface.network.weight(sender, self)
51      self.packet_buffer.append([packet, sender, buffer_time])
52    # public method to send a packet through the network beginning at this router
53    def send_packet (self, packet):
54      self._on_receive_packet(packet, sender=None)
55
56  class FloodingNode(RoutableNode):
57    def __init__ (self, addr):
58      super().__init__(addr)
59    def _route_packet (self, packet, sender):
60      neighbors = self.interface.get_neighbors()
61      recipients = filter(lambda n: n is not sender, neighbors)
62      for recipient in recipients:
63        self.interface.send_neighbor(recipient, packet)
64
65  class BellmanFordNode(RoutableNode):
66    def __init__ (self, addr):
67      super().__init__(addr)
68      self.weights = {}
69      self.routing_table = {}
70    # retrieve weights of edges and updates internal routing table with this
71    # new information
72    def update_weights (self):
73      self.weights = dict(self.interface.get_neighbors_and_weights())
74      for node, weight in self.weights.items():
75        self.routing_table[node.addr] = (node, weight)
76    # sends routing table to adjacent nodes
77    def notify_neighbors (self):
78      data = {node_addr: weight for node_addr, (_, weight) in self.routing_table.
      items()}
79      for neighbor in self.interface.get_neighbors():
80        neighbor._notify(data, self)
81    # internal method, called by adjacent nodes which send their routing table
82    # our routing table will be updated accordingly when better routes are
83    # discovered
84    def _notify (self, data, sender):
85      edge_weight = self.weights[sender]
86      table = self.routing_table
87      for node_addr, weight in data.items():
88        w = edge_weight + weight
89        # address unregistered or better route
90        if node_addr not in table or table[node_addr][1] > w:
91          table[node_addr] = (sender, w)
92    def _route_packet (self, packet, sender):
93      dst = packet.dest_addr
94      if dst not in self.routing_table:
95        print(self, ':: No route to host')
96        return
```

```
97      direction, _ = self.routing_table[dst]
98      self.interface.send_neighbor(direction, packet)
99
100 class LinkStateNode(RoutableNode):
101   def __init__ (self, addr):
102     super().__init__(addr)
103     self.graph = nx.Graph()
104     self.neighbors = {}
105   def update_graph_with_neighbors (self):
106     for neighbor, weight in self.interface.get_neighbors_and_weights():
107       self.graph.add_edge(self.addr, neighbor.addr, weight=weight)
108       self.neighbors[neighbor.addr] = neighbor
109   # sends routing table to adjacent nodes
110   def notify_neighbors (self):
111     for neighbor in self.interface.get_neighbors():
112       neighbor._notify(self.graph)
113   # internal method, called by adjacent nodes which send their graph
114   # our graph table will be updated accordingly with new topology information
115   def _notify (self, graph):
116     for node1, node2, weight in graph.edges.data('weight', default=1):
117       self.graph.add_edge(node1, node2, weight=weight)
118   def _route_packet (self, packet, sender):
119     try:
120       dst = packet.dest_addr
121       route = nx.dijkstra_path(self.graph, self.addr, dst, weight='weight')
122       next_addr = route[1]
123       next_node = self.neighbors[next_addr]
124       self.interface.send_neighbor(next_node, packet)
125     except nx.NetworkXNoPath as e:
126       print(self, ':: No route to host')
127
128 def _get_routable_net (Node, n, m):
129   nodes = [Node(i + 1) for i in range(n)]
130   net = MeshNetwork(nodes, m)
131   for node in nodes:
132     node.set_interface(LocalNetworkInterface(node, net))
133   return net, nodes
134
135 def routing_test (NodeClass, init_func=None, n=20, m=None):
136   t0 = time()
137   net, nodes = _get_routable_net(NodeClass, n=n, m=m)
138   t1 = time()
139   if init_func:
140     init_func(net, nodes)
141   t2 = time()
142   src, dst = choices(nodes, k=2)
143   src.send_packet(Packet('hello world 42', src.addr, dst.addr))
144   while len(dst.packets_received) == 0:
145     net.routing_step()
146   t3 = time()
147   route = dst.packets_received[0].route
148   return net, route, t0, t1, t2, t3
149
150 def routing_flooding (n=20):
151   return routing_test(FloodingNode, n=n, m=2)
152
153 def init_bellman_ford (net, nodes):
154   for node in nodes:
155     node.update_weights()
156   d = nx.diameter(net.graph)
157   for _ in range(d + 2):
158     for node in nodes:
```

```
      node.notify_neighbors()

def routing_bellman_ford (n=20):
  return routing_test(BellmanFordNode, init_bellman_ford, n=n, m=2)

def init_link_state (net, nodes):
  for node in nodes:
    node.update_graph_with_neighbors()
  d = nx.diameter(net.graph)
  for _ in range(d + 2):
    for node in nodes:
      node.notify_neighbors()

def routing_link_state (n=20):
  return routing_test(LinkStateNode, init_link_state, n=n, m=2)

def avgstd (data):
  n, s = len(data), sum(data)
  m = s / n
  e = sqrt(sum((x - m) ** 2 for x in data) / (n * (n - 1)))
  return m, e

def performace_test (func, n=20, k=50):
  network_size = n
  route_lengths, routers_hits, init_times, routing_times = [], [], [], []
  for i in range(k):
    net, route, t0, t1, t2, t3 = func(n)
    route_lengths.append(len(route))
    routers_hits.append(net.total_packet_count)
    init_times.append(1000 * (t2 - t0))
    routing_times.append(1000 * (t3 - t2))
  return network_size, avgstd(route_lengths), avgstd(routers_hits), avgstd(
    init_times), avgstd(routing_times)

def perf_on_size (n, k=50):
  return [
    performace_test(routing_flooding, n=n, k=k),
    performace_test(routing_bellman_ford, n=n, k=k),
    performace_test(routing_link_state, n=n, k=k)
  ]

if __name__ == '__main__':
  print(perf_on_size(50, k=50))
  print(perf_on_size(100, k=50))
  print(perf_on_size(500, k=50))
  print('Done')
```