Routage de paquets sur le réseau Internet

Travail d'Initiative Personnelle Encadré

Emre UCAR - 12135 - Juin 2022 - https://github.com/Emrio/tipe

Plan

A. Introduction
    1. Enjeux généraux du routage sur Internet
    2. Applications dans le monde médical
    3. Problématique
B. Étude et génération d’un réseau
    1. Définitions
    2. Topologies particulières
    3. Étude de génération procédurale
C. Protocole de routage de paquets
    1. Définitions, propriétés et principe d’optimalité
    2. Implémentation du routage au sein d’un réseau
D. Conclusions
E. Annexes

A. Introduction

1. Enjeux généraux du routage sur Internet

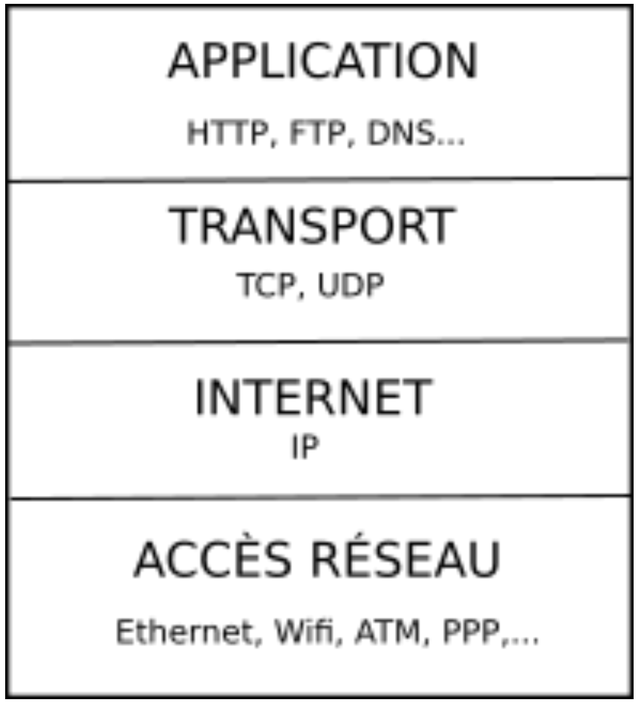
- Aspect critique de nombreuses industries (intrinsèques ou non) : transfert de données, divertissement, échanges financiers, télécommunications …
- Exigences strictes et variées des utilisations (particuliers et entreprises):
  - Performance: Haute disponibilité, faible latence, haut débit
  - Sécurité: Chiffrement des données en transit, respect de la vie privée
  - Coût: Services abordables

2. Applications dans le monde médical

- Communication entre les bases de données de santé
- Appels (VoIP) et communication entre les acteurs du domaine (ambulanciers, urgentistes, …)
- Réservation de vaccins en ligne
- Dans le futur: Opérations chirurgicales à distance (Chine)
- Sur d’autres réseaux: panne des numéros d’urgence sur le réseau téléphonique Orange (Juin 2021)

3. Problématique

- Comment générer et simuler des réseaux réalistes ?
- Comment router un paquet sur un tel réseau ?
- Comment faire face et prévenir les phénomènes de congestion ?



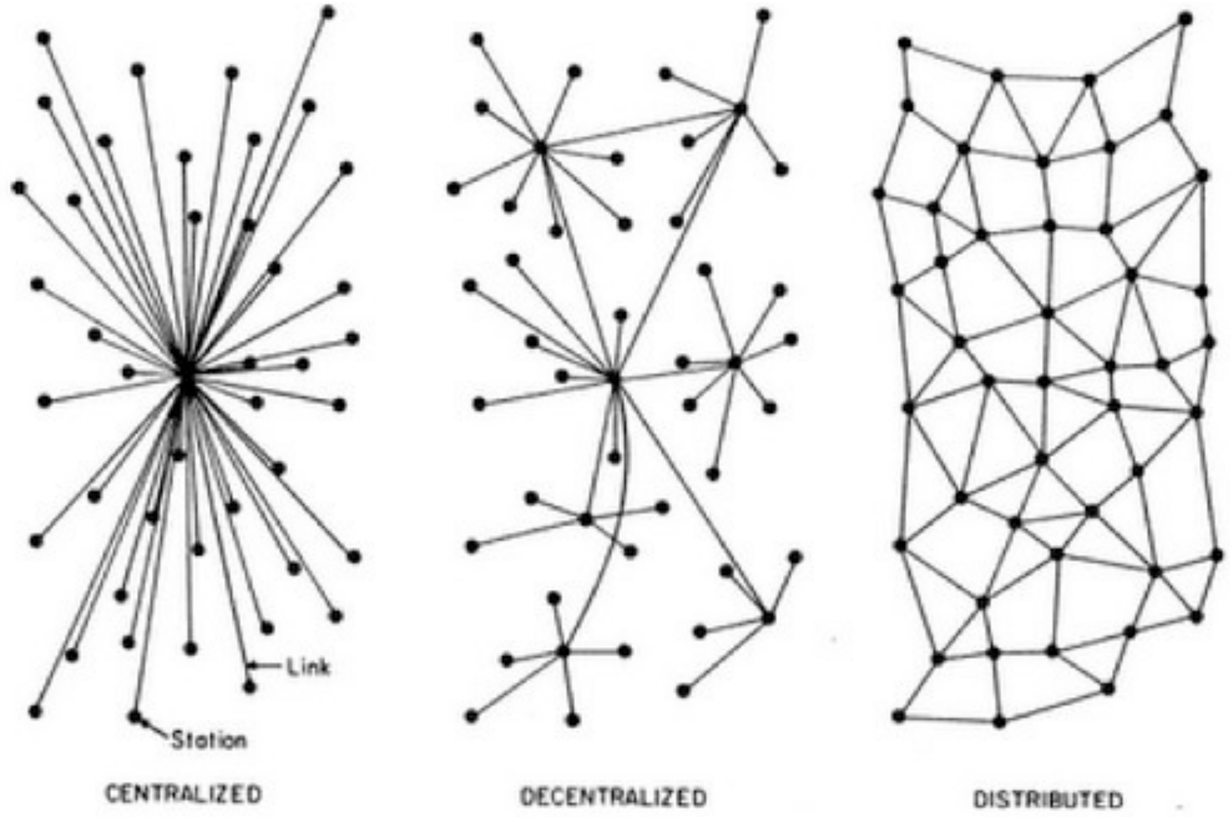
B. Étude et génération d’un réseau

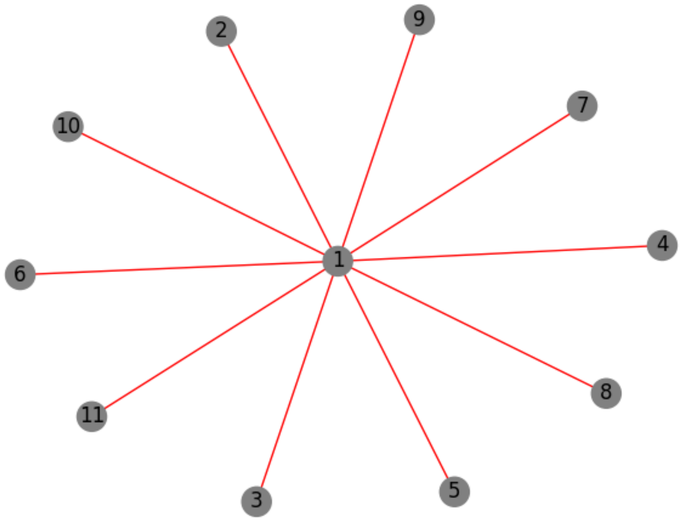
1. Définitions

- V ensembles des noeuds (routeurs et serveurs sur le réseau)
- $E \subset \left\{ \{u, v\} \,|\, (u, v) \in V^2 \right\}$ ensembles d'arêtes
- $G = (V, E)$. Sauf mention contraire, on suppose G connexe.
- Chemin entre u et v : $(u_1, \ldots, u_n) \in V^n$ avec $n \in \mathbb{N}$, $u_1 = u$, $u_n = v$ et $\forall i \leq n - 1 \; \{u_i, u_{i+1}\} \in E$. On note $P_{u,v}$ l'ensemble de ces chemins
- $\forall u, v \in V, \; \rho(u, v) = \min \left\{ n \in \mathbb{N} \,|\, \exists u_1, \ldots, u_n \in V, (u_1, \ldots, u_n) \in P_{u,v} \right\}$

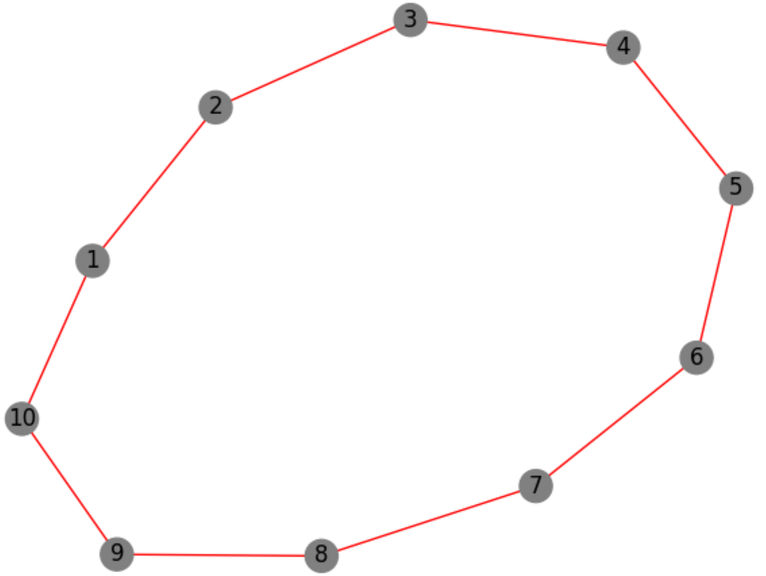
2. Topologies particulières

a. Exemples particuliers

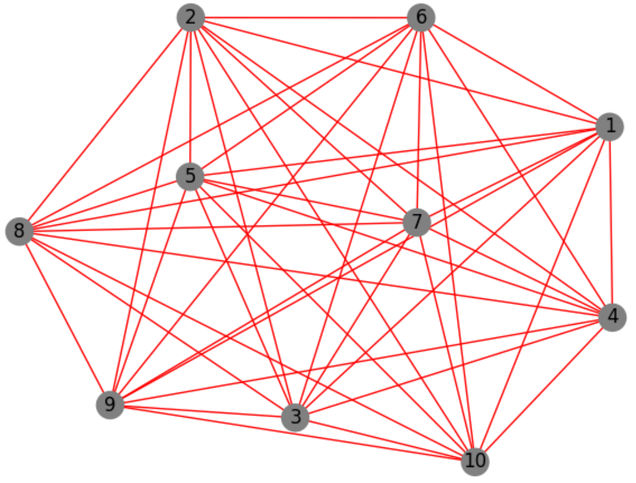




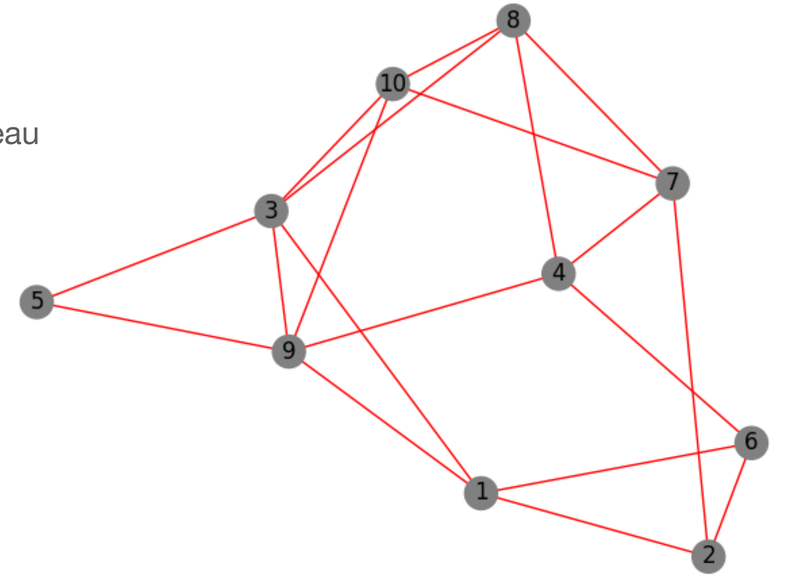
(a) Topologie en étoile



(b) Topologie en anneau



(c) Topologie de graphe complet



(d) Topologie mesh

Différentes topologies de réseaux

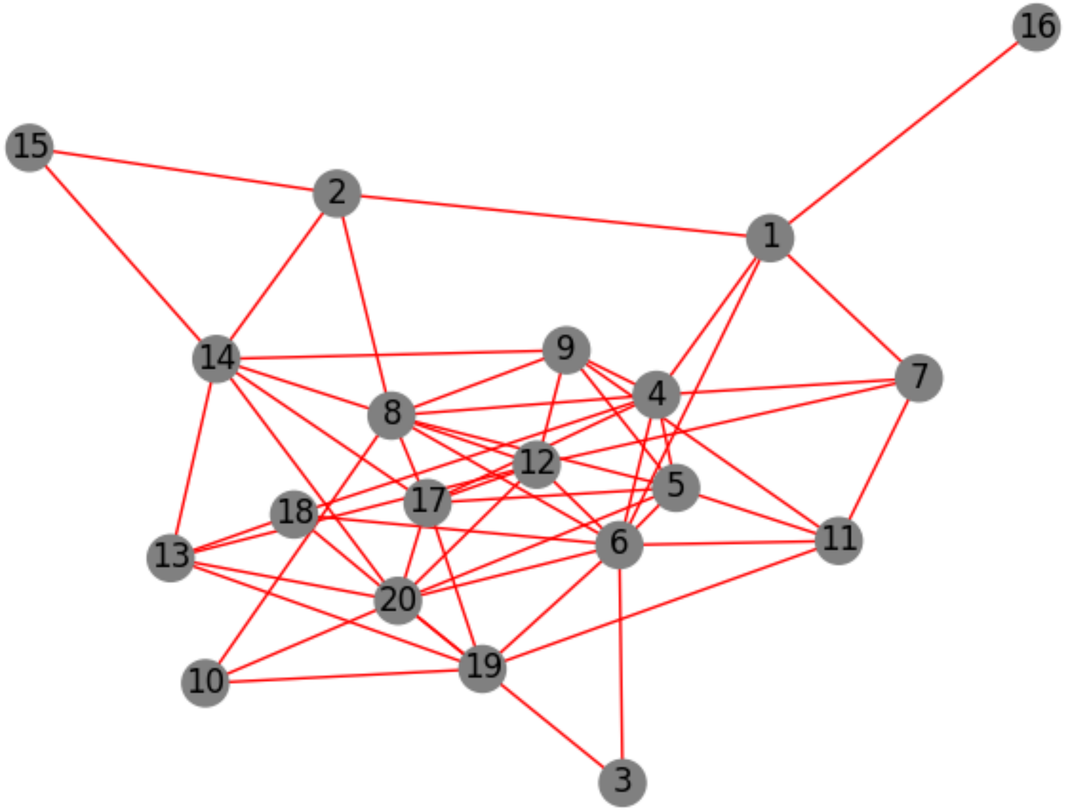
b. Détails d’implémentation

- `NetworkGraph`
  - Classe de base pour tous les réseaux, méthodes utilitaires générales
  - Utilise `networkx` et `matplotlib` pour représenter le graphe
- `Node`
  - Paramètre : adresse

3. Étude de génération procédurale

a. Une première approche : Modèle de Erdös-Rényi

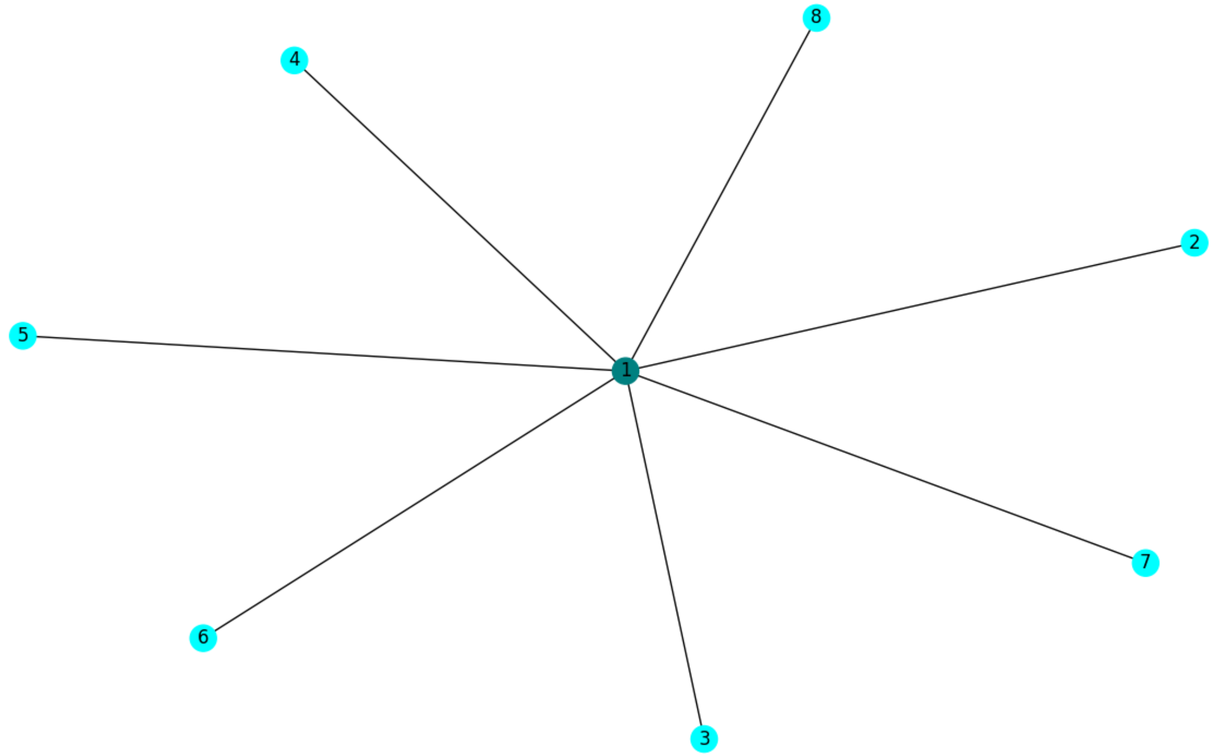
- Paramètres
  - V liste des noeuds (de taille n)
  - p probabilité qu'une arête $\{u, v\}$ appartienne à G
- Problème: pas de phénomène de regroupement (« clustering ») comme dans les réseaux réels



Réseau avec $n = 20$ et $p = 42\,\%$

b. Modélisation d'un réseau local

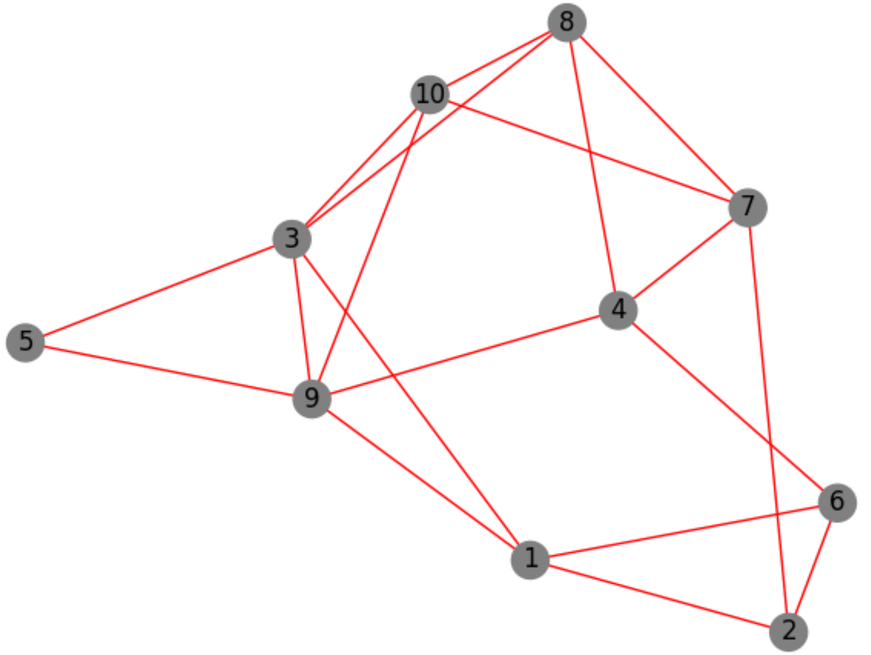
- `StarNetwork`
  - c noeud central (routeur)
  - V noeuds périphériques
  - Topologie en étoile



Réseau en étoile à $n = 9$ noeuds dont le noeud central

c. Modélisation d'un réseau « backbone »

- `MeshNetwork`
  - V liste des noeuds (de taille n)
  - k nombre d'arêtes établies par noeud



Réseau avec $n = 10$ et $k = 3$

d. Nouvelle approche : modèle par tiers

- Sous-réseaux
  - WAN (« Wide Area Network », réseau « backbone »)
  - MAN (« Metropolitan Area Network », réseau d’un FAI)
  - LAN (« Local Area Network », réseau résidentiel ou réseau d’une entreprise)
- Connections entre les sous-réseaux
  - LAN ↔ MAN : un lien par LAN (pas de multidomiciliation)
  - MAN ↔ WAN : au moins un lien par MAN à un seul WAN
  - WAN ↔ WAN : plusieurs liens

e. Paramètres requis

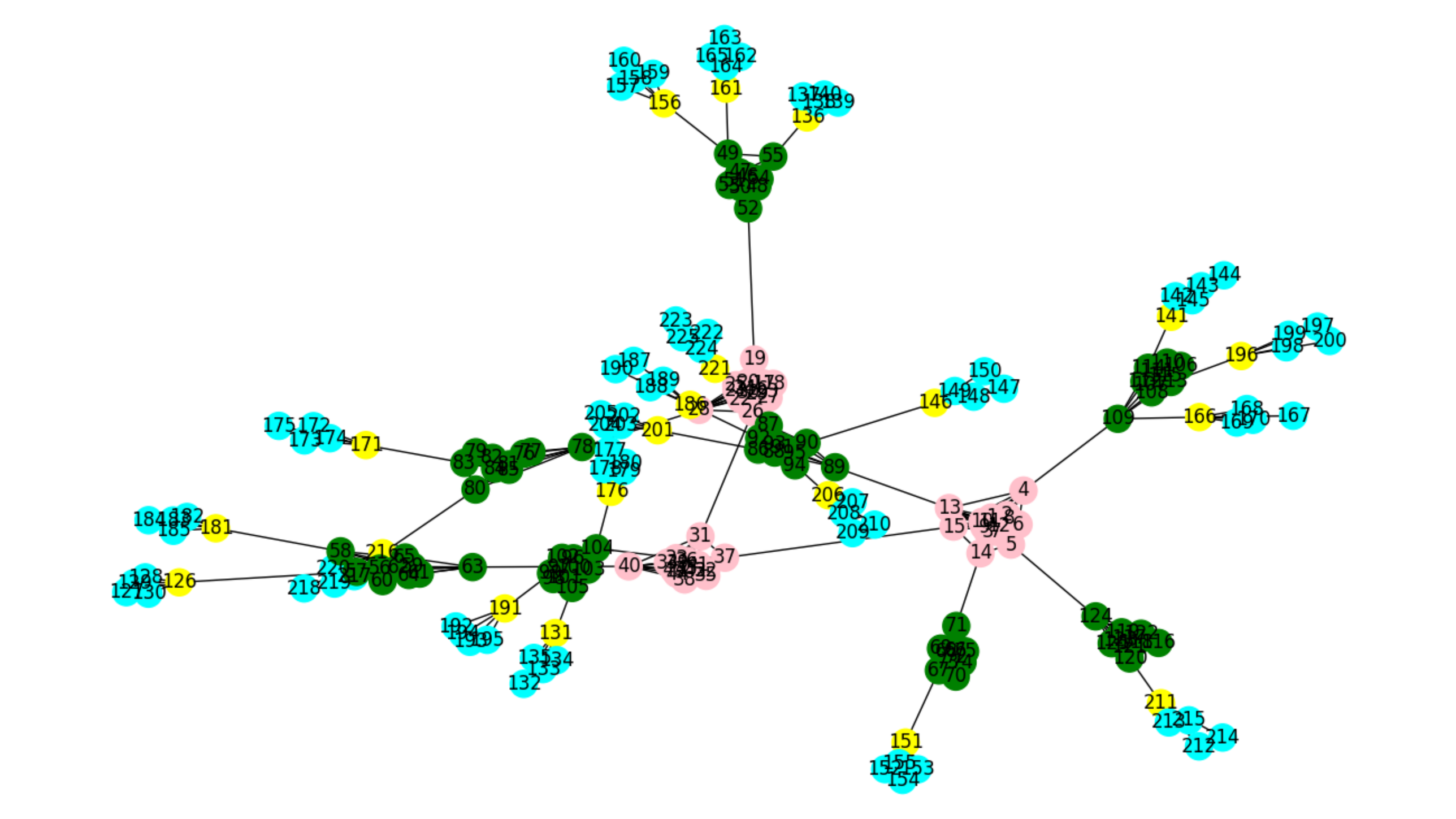
- T nombre de WANs
- N_T nombre de noeuds par WAN
- E_T nombre de liens au sein d'un WAN
- E_{TT} nombre moyens de liens entre les WANs ($E_{TT} \geq T$)
- S nombre de MANs
- N_S nombre de noeuds par MAN
- E_S nombre de liens au sein d'un MAN
- L nombre de LANs

f. Details d’implémentation

- `WideAreaNetwork ← MeshNetwork`
- `MetropolitanAreaNetwork ← MeshNetwork`
- `LocalAreaNetwork ← StarNetwork`

- `TiersNodeFactory`

- `TiersNetwork ← NetworkGraph`



C. Protocole de routage de paquets

1. Définitions, propriétés et principe d'optimalité

a. Définitions

- $f\colon E \to \mathbb{R}_+$ fonction poids
- On se donne $G = (V, E, f)$ connexe
- Poids d'un chemin $(u_0, \ldots, u_n) \in P_{u,v} : \mu_{(u_0,\ldots,u_n)} = \sum_{i=1}^{n-1} f(u_i, u_{i+1})$
- Chemin le plus court: Élément de $P_{u,v}$ minimisant μ. Ce chemin n'est pas unique. On note $\pi(u, v)$ le poids de ce(s) chemin(s)

b. Un théorème

- Théorème : Pour tout $u, v \in V$, si $(u_1, \ldots, u_n)$ est un chemin le plus court, alors pour tout $i \in [2,n-1]$, $(u_1, \ldots, u_i)$ et $(u_i, \ldots, u_n)$ sont des chemins les plus courts pour, respectivement, (u, u_i) et (u_i, v)

c. Principe d’optimalité

- Minimiser le coût du transfert du paquet ?
- Minimiser le nombre de sauts ?
- Minimiser le temps de transport ?
- Privilégier les SA occidentaux et éviter les noeuds russes et chinois ?
- Éviter les noeuds des FAI concurrents et privilégier les AS partenaires ?

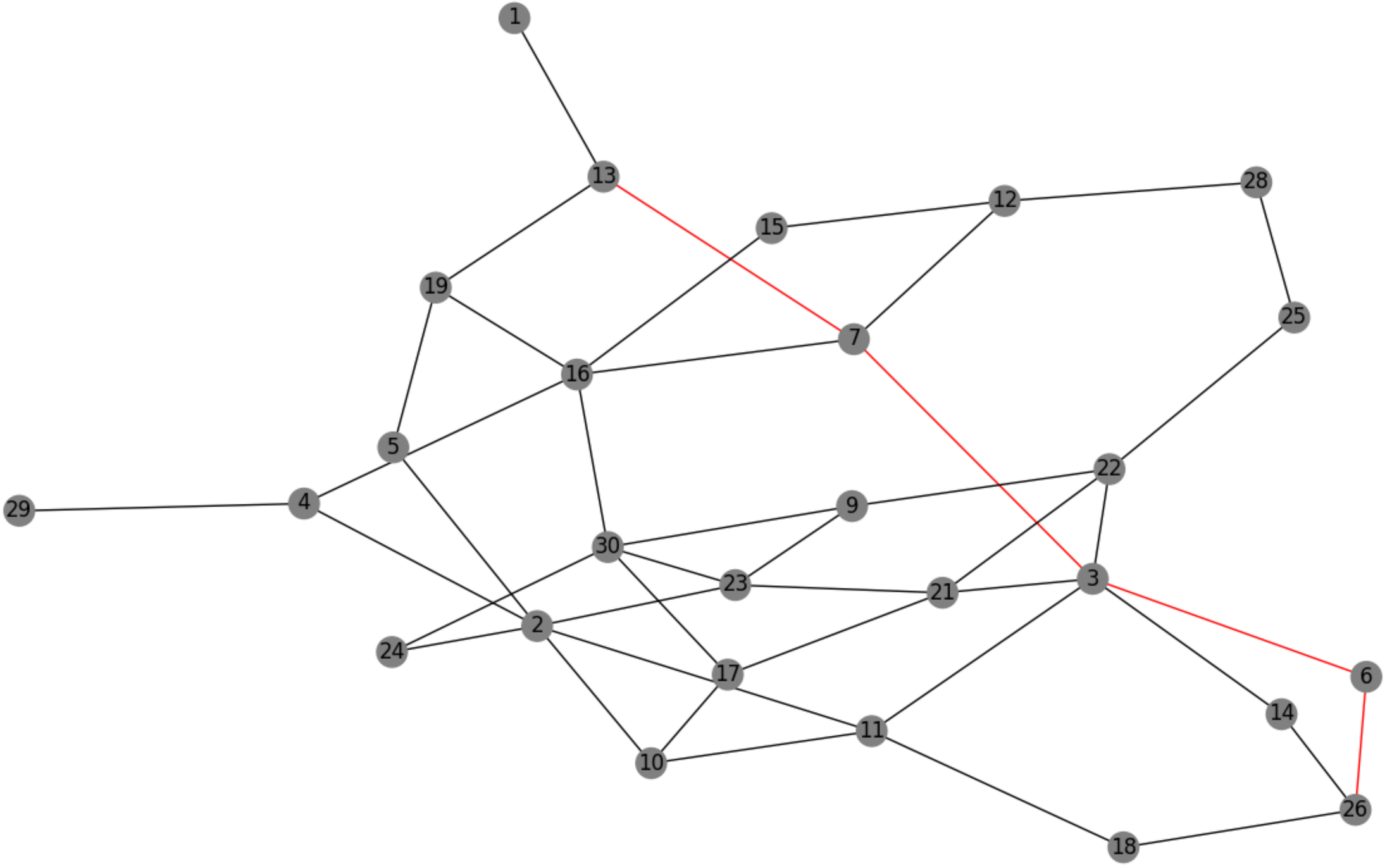
2. Implémentation du routage au sein d’un réseau

a. Hypothèses préliminaires

- On définit le poids de manière aléatoire sur notre simulation
- Routage dans le réseau d’un FAI (réseau en mesh ou avec Erdös-Rényi)
- Même définition de l’optimalité partagée par tous les routeur

b. Première approche : Routage par inondation

- Nombre de paquets émis théoriquement infini
- Utilisation d'un paramètre « Time To Live »
- Packet reçu plusieurs fois
- Temps de calcul en $O\left(N^M\right)$ avec N nombre de noeuds et M distance entre la source et la destination


1
13
28
12
15
19
25
7
16
5
22
29
4
9
30
23
21
3
2
24
17
6
14
11
10
26
18

c. Deuxième approche : Routage par vecteur de distance

- « Distance Vector Routing » ou « Routage Bellman-Ford »
- Historiquement, premier algorithme de routage d'ARPANET
- Initialement : échanges successifs d'informations
  - Ainsi chaque noeud connaît la meilleure « direction » (noeud adjacent) vers la destination
- Pas de multiplication des paquets
- Lors des pannes : propagation lente de l'information

Destination	Direction	Poids
12	12	2
18	18	1
1	12	3
19	12	3
15	12	4
10	18	3
11	18	3
5	18	4
9	18	5
7	18	9

Exemple de table de routage

d. Approche actuelle : Routage par état de lien

- État de lien d’un noeud : ses voisins et le poids des arêtes sortantes
- Initialement : partage de l’état de lien de chaque noeud par inondation
  - Chaque noeud connaît la topologie du réseau
- Lors d’un routage : calcul de plus court chemin
- Espace mémoire utilisé potentiellement important
- Dijkstra utilisé à chaque saut
- Utilisé dans les protocoles OSPF et IS-IS

e. Analyse des performances

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

Statistiques sur les algorithmes de routage (`MeshNetwork`, $n = 50$, $m = 2$, $k = 50$)

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

Statistiques sur les algorithmes de routage (`MeshNetwork`, $n = 100$, $m = 2$, $k = 50$)

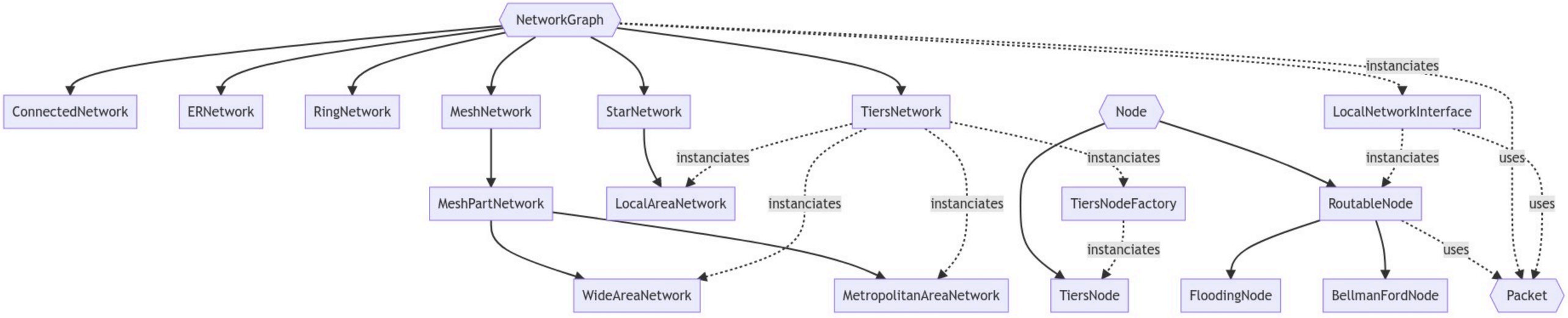
Algorithme	Longueur du chemin	Nombre de noeuds touchés	Temps d'initalisation (ms)	Temps de routage (ms)
Inondation	6.28 ± 0.40	743 ± 280	86.8 ± 6.3	13.0 ± 3.9
Vecteur distance	6.58 ± 0.48	5.58 ± 0.48	2.99 ± 0.08 s	1.48 ± 0.12
État de lien	6.54 ± 0.40	5.54 ± 0.40	38.6 ± 4.9 s	6.54 ± 2.48

Statistiques sur les algorithmes de routage (`MeshNetwork`, $n = 500$, $m = 2$, $k = 50$)

f. Détails d’implémentation

- `Network`
  - `routing_iterator`
  - `routing_step()`
- `LocalNetworkInterface`
  - `node`, `network`
  - `get_neighbors()`, `get_neighbors_and_weights()`
  - `send_neighbor(dest_node, packet)`

- Packet
  - data, src_addr, dest_addr, ttl, route
  - is_dead()
  - copy(), add_node(node)
- RoutableNode(Node)
  - set_interface(interface),
  - _on_receive_packet(packet, sender),
  - _update_buffer()
  - _route_packet()
  - send_packet()


NetworkGraph
ConnectedNetwork
ERNetwork
RingNetwork
MeshNetwork
StarNetwork
TiersNetwork
Node
LocalNetworkInterface
instanciates
MeshPartNetwork
LocalAreaNetwork
TiersNodeFactory
RoutableNode
uses
WideAreaNetwork
MetropolitanAreaNetwork
TiersNode
FloodingNode
BellmanFordNode
Packet

D. Conclusion

D. Conclusion

- Approche essentielle pour l’industrie des télécommunications
- Routage inter-réseaux
- Gestion de la congestion
- Considérations de sécurité
  - Noeuds / AS malicieux
  - Chiffrement des paquets

E. Annexes

```python
from network import LocalNetworkInterface

# base class for all nodes
class Node:
  def __init__ (self, addr):
    self.addr = addr
    self.interface = None

  def __repr__ (self):
    return 'Node #' + str(self.addr)

  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'
```

nodes/bellman_ford.py

```python
from nodes.routable import RoutableNode

class BellmanFordNode(RoutableNode):
  def __init__ (self, addr):
    super().__init__(addr)
    self.weights = {}
    self.routing_table = {}

  # retrieve weights of edges and updates internal routing table with this
  # new information
  def update_weights (self):
    self.weights = dict(self.interface.get_neighbors_and_weights())

    for node, weight in self.weights.items():
      self.routing_table[node.addr] = (node, weight)

  # sends routing table to adjacent nodes
  def notify_neighbors (self):
    data = {
      node_addr: weight for node_addr, (_, weight) in self.routing_table.items()
    }

    for neighbor in self.interface.get_neighbors():
      neighbor._notify(data, self)

  # internal method, called by adjacent nodes which send their routing table
  # our routing table will be updated accordingly when better routes are
  # discovered
  def _notify (self, data, sender):
    edge_weight = self.weights[sender]
    table = self.routing_table

    for node_addr, weight in data.items():
      w = edge_weight + weight

      # address unregistered or better route
      if node_addr not in table or table[node_addr][1] > w:
        table[node_addr] = (sender, w)
```

```python
def _route_packet (self, packet, sender):
  dst = packet.dest_addr

  if dst not in self.routing_table:
    print(self, ':: No route to host')
    return

  direction, _ = self.routing_table[dst]
  self.interface.send_neighbor(direction, packet)
```

nodes/flooding.py

```
from nodes.routable import RoutableNode

class FloodingNode(RoutableNode):
  def __init__ (self, addr):
    super().__init__(addr)

  def _route_packet (self, packet, sender):
    neighbors = self.interface.get_neighbors()
    recipients = filter(lambda n: n is not sender, neighbors)

    for recipient in recipients:
      self.interface.send_neighbor(recipient, packet)
```

nodes/link_state.py

```
import networkx as nx
from nodes.routable import RoutableNode

class LinkStateNode(RoutableNode):
  def __init__ (self, addr):
    super().__init__(addr)

    self.graph = nx.Graph()
    self.neighbors = {}

  def update_graph_with_neighbors (self):
    for neighbor, weight in
self.interface.get_neighbors_and_weights():
      self.graph.add_edge(self.addr, neighbor.addr, weight=weight)
      self.neighbors[neighbor.addr] = neighbor

  # sends routing table to adjacent nodes
  def notify_neighbors (self):
    for neighbor in self.interface.get_neighbors():
      neighbor._notify(self.graph)

  # internal method, called by adjacent nodes which send their
graph
  # our graph table will be updated accordingly with new topology
information
  def _notify (self, graph):
    for node1, node2, weight in graph.edges.data('weight',
default=1):
      self.graph.add_edge(node1, node2, weight=weight)
```

```
  def _route_packet (self, packet, sender):
    try:
      dst = packet.dest_addr
      route = nx.dijkstra_path(self.graph, self.addr, dst,
weight='weight')
      next_addr = route[1]
      next_node = self.neighbors[next_addr]
      self.interface.send_neighbor(next_node, packet)
    except nx.NetworkXNoPath as e:
      print(self, ':: No route to host')
```

```python
from nodes.base import Node
from network import LocalNetworkInterface

class RoutableNode(Node):
  def __init__ (self, addr):
    super().__init__(addr)
    self.packet_buffer = []
    self.packets_received = []

  # can be called by the network or the node factory to change the
interface
  # used especially during network merge operations
  def set_interface (self, interface):
    self.interface = interface

  def _route_packet (self, packet, sender):
    # INFO: routing logic goes here
    # INFO: override this method to implement a routing strategy
    raise Exception('No routing algorithm')

  # this method is called by the network to indicate the router that it
can
  # process its packet buffer
  # at that point, all buffer times are decremented and suitable packets
  # are routed
  def _update_buffer (self):
    def filter_buffer (b):
      packet, sender, buffer_time = b
      if buffer_time > 0:
        return True
      self._route_packet(packet, sender)
      return False

    self.packet_buffer = list(filter(filter_buffer, self.packet_buffer))
    for b in self.packet_buffer:
      b[2] -= 1

  # internal method called when receiving a packet
  # this method is called before the _route_packet method
  # it implements TTL and reception of packets intended to this node
  def _on_receive_packet (self, packet, sender):
    packet.add_node(self)

    if packet.dest_addr == self.addr:
      self.packets_received.append(packet)
      print(self, ':: received packet:', packet.data, packet.route)
      return

    packet.ttl -= 1

    if packet.is_dead():
      return

    if sender is None:
      buffer_time = 0
    else:
      buffer_time = self.interface.network.weight(sender, self)

    self.packet_buffer.append([packet, sender, buffer_time])

  # public method to send a packet through the network beginning at this
router
  def send_packet (self, packet):
    self._on_receive_packet(packet, sender=None)
```

nodes/tiers.py

```
from nodes.base import Node

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'
```

topo/connected.py

```python
from network import NetworkGraph

class ConnectedNetwork(NetworkGraph):
  def __init__(self, nodes):
    super().__init__()
    for node1 in nodes:
      for node2 in nodes:
        if node1 is not node2:
          self.connect(node1, node2)
```

topo/er.py

```python
from random import random
from network import NetworkGraph

# WARN: The resulting graph is not garanteed to be connex
class ERNetwork(NetworkGraph):
  def __init__(self, nodes, p):
    super().__init__()
    for node1 in nodes:
      for node2 in nodes:
        if node1 is not node2 and random() < p:
          self.connect(node1, node2)
```

topo/mesh.py

```python
from random import choice
from network import NetworkGraph

class MeshNetwork(NetworkGraph):
  def __init__(self, nodes, deg_target):
    super().__init__()

    # TODO: Ensure the graph is connex
    for node1 in nodes:
      for _ in range(deg_target):
        node2 = choice(list(filter(lambda x: x is not node1, nodes)))
        self.connect(node1, node2)
```

topo/ring.py

```python
from network import NetworkGraph

class RingNetwork(NetworkGraph):
  def __init__(self, nodes):
    super().__init__()
    for node1, node2 in zip(nodes, nodes[1:] + [nodes[0]]):
      self.connect(node1, node2)
```

topo/star.py

```python
from network import NetworkGraph

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)
```

topo/tiers.py

```python
from random import choice, shuffle, randint
from numpy.random import normal
from network import NetworkGraph
from nodes.tiers import TiersNode
from topo.mesh import MeshNetwork
from topo.star import StarNetwork

# 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

  def make_node (self, type):
    self.addr_count += 1
    return TiersNode(self.addr_count, type)

class TiersNetwork(NetworkGraph):
  def __init__(self, T=1, NT=1, ET=2, ETT=1, S=1, NS=1, ES=2, L=1,
NL=1):
    super().__init__()
    self.T = T # nbr of transit domains
    self.NT = NT # avg nbr of nodes per transit domain
    self.ET = ET # nbr of edges within transit domains
    self.ETT = ETT # nbr of edges between transit domains
    self.S = S # nbr of stub domains
    self.NS = NS # avg nbr of nodes per stub domains
    self.ES = ES # nbr of edges within stub domains
    self.L = L # nbr of LANs
    self.NL = NL # avg nbr of nodes per LAN

    self.factory = TiersNodeFactory(self)

    self._generate()

  # strategy: generate individual WANs, MANs, and LANs
  # afterwards, connect and merge networks in order to
  # obtain a single connex network
  def _generate (self):
    make_node = self.factory.make_node

    T = self.T

    NT, ET = self.NT, self.ET
    self.wans = []
    for _ in range(T):
      self.wans.append(WideAreaNetwork('wan', NT, ET, make_node))

    S = self.S
    NS, ES = self.NS, self.ES
    self.mans = []
    for _ in range(S):
      self.mans.append(MetropolitanAreaNetwork('man', NS, ES,
make_node))

    L = self.L

    NL = self.NL
    self.lans = []
    for _ in range(L):
      self.lans.append(LocalAreaNetwork(NL, make_node))

    for lan in self.lans:
      lan_node = lan.center

      man = choice(self.mans)
      man_node = choice(man.nodes)

      man.add(lan)
      man.connect(lan_node, man_node)

    for man in self.mans:
      man_node = choice(man.nodes)

      wan = choice(self.wans)
      wan_node = choice(wan.nodes)

      wan.add(man)
      wan.connect(man_node, wan_node)

    # making a connex component
    subnets = self.wans.copy()
    while len(subnets) > 1:
      i1 = randint(0, len(subnets) - 1)
      wan1 = pop2(subnets, i1)

      i2 = randint(0, len(subnets) - 1)
      wan2 = pop2(subnets, i2)

      node1 = choice(wan1.nodes)
      node2 = choice(wan2.nodes)

      wan1.add(wan2)
      wan1.connect(node1, node2)

      subnets.append(wan1)

    self.add(subnets[0])

    # satisfying ETT
    wans = self.wans.copy()
    remaining = self.ETT - len(wans)
    for _ in range(remaining):
      wan1 = choice(wans)
      wan2 = choice(wans)

      if wan1 is wan2:
        continue

      node1 = choice(wan1.nodes)
      node2 = choice(wan2.nodes)

      self.connect(node1, node2)

# WAN and MAN base
class MeshPartNetwork(MeshNetwork):
  def __init__ (self, type, n, m, make_node):
    self.nodes = [make_node(type=type) for _ in range(n)]
    self.type = type
    super().__init__(self.nodes, m)


# WAN
class WideAreaNetwork(MeshPartNetwork):
  def __init__(self, n, m, make_node):
    super().__init__('wan', n, m, make_node)

# MAN
class MetropolitanAreaNetwork(MeshPartNetwork):
  def __init__(self, n, m, make_node):
    super().__init__('man', n, m, make_node)


# LAN
class LocalAreaNetwork(StarNetwork):
  def __init__(self, node_count, make_node):
    self.center = make_node(type='lan-center')
    nodes = [make_node(type='lan') for _ in range(node_count - 1)]
    super().__init__(self.center, nodes)
```

network.py

```python
from random import 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):
    if get_edge_color is None:
      get_edge_color = 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
    )

    # # bandwidth / weight / labelling stuff
    # edge_labels = nx.get_edge_attributes(self.graph, 'weight')
    # nx.draw_networkx_edge_labels(self.graph, pos,
edge_labels=edge_labels)

    fig = plt.gcf()
    fig.set_size_inches(13, 8)

    if file:
      fig.savefig(file)

    plt.show()

  # test whether a node is linked to any other node
  def node_is_linked (self, node):
    return node in self.graph and len(self.graph.edges(node))

  # test wether two nodes are linked
  def nodes_are_linked (self, node1, node2):
    return self.graph.has_edge(node1, node2)


# get the weight of a link given the two nodes
  def weight (self, node1, node2):
    if self.nodes_are_linked(node1, node2):
      return self.graph[node1][node2]['weight']
    return None

  # merge a network into this network
  def add (self, net):
    for node1, node2 in net.graph.edges():
      self.connect(node1, node2, weight=net.weight(node1, node2))

    # updating local network interfaces
    for node in net.graph.nodes():
      if node.interface is not None:
        node.interface.update_network(self)

  # execute one step of global packet routing
  def routing_step (self):
    node = next(self.routing_iterator, None)

    if node is None:
      nodes = filter(lambda n: len(n.packet_buffer) != 0,
self.graph.nodes())
      nodes_list = list(nodes)
      if nodes_list == []:
        raise Exception('No route to host')
      self.routing_iterator = iter(nodes_list)
      return self.routing_step()

    node._update_buffer()


# python does not provide private properties and methods
# this class acts as a network abstraction for a node
class LocalNetworkInterface:
  def __init__ (self, node, network):
    self.node = node
    self.network = network

  def update_network (self, network):
    self.network = network

  def get_neighbors (self):
    if self.node not in self.network.graph:
      return []

    return self.network.graph.neighbors(self.node)

  def get_neighbors_and_weights (self):
    neighbors = self.get_neighbors()
    return [(node, self.network.weight(self.node, node)) for node
in neighbors]

  def send_neighbor (self, node, packet):
    if node not in self.get_neighbors():
      raise Exception('Cannot send to this node : not my
neighbor')

    # copying to avoid routers manipulating the same object
(ttl, ...)
    # in some routing protocols like flooding
    new_packet = packet.copy()

    self.network.total_packet_count += 1

    node._on_receive_packet(new_packet, self.node)


class Packet:
  def __init__(self, data, src_addr, dest_addr, ttl=10,
route=None):
    self.data = data
    self.ttl = ttl
    self.src_addr = src_addr
    self.dest_addr = dest_addr
    self.route = route if route is not None else []

  def is_dead (self):
    return self.ttl == 0

  def copy (self):
    route = self.route.copy()
    return Packet(self.data, self.src_addr, self.dest_addr,
self.ttl, route)

  def add_node (self, node):
    self.route.append(node)
```

main.py

```python
from time import time
import numpy as np
from network import LocalNetworkInterface, Packet
from routing_test import show_routing_graph, routing_test, init_bellman_ford,
init_link_state
from nodes.base import Node
from nodes.tiers import TiersNode
from nodes.flooding import FloodingNode
from nodes.bellman_ford import BellmanFordNode
from nodes.link_state import LinkStateNode
from topo.ring import RingNetwork
from topo.star import StarNetwork
from topo.connected import ConnectedNetwork
from topo.mesh import MeshNetwork
from topo.er import ERNetwork
from topo.tiers import TiersNetwork, LocalAreaNetwork

# utility : create an image filename for network images
image_name = lambda: str(int(time())) + '.png'

def tiers ():
  net = TiersNetwork(T=3, NT=15, ET=8, ETT=5, S=8, NS=10, ES=5, L=20, NL=5)
  net.show(file=image_name())

def star ():
  n = 5
  nodes = [Node(i + 2) for i in range(n)]
  net = StarNetwork(Node(1), nodes)
  net.show(file='star.png')

def ring ():
  n = 10
  nodes = [Node(i + 1) for i in range(n)]
  net = RingNetwork(nodes)
  net.show(file='ring.png')

def connected ():
  n = 10
  nodes = [Node(i + 1) for i in range(n)]
  net = ConnectedNetwork(nodes)
  net.show(file='connected.png')

def mesh ():
  n = 10
  m = 3
  nodes = [Node(i + 1) for i in range(n)]
  net = MeshNetwork(nodes, m)
  net.show(file='mesh.png')

def er ():
  n = 20
  m = 80
  p = m / n / (n - 1) # m / (n 2)
  nodes = [Node(i + 1) for i in range(n)]
  net = ERNetwork(nodes, p)
  net.show(file='er.png')

def lan ():
  net = LocalAreaNetwork(8)
  net.show(file='lan.png')

def routing_flooding ():
  net, route, *_ = routing_test(FloodingNode, n=20, m=2)
  show_routing_graph(net, route)

def routing_bellman_ford ():
  net, route, *_ = routing_test(BellmanFordNode, init_bellman_ford, n=20, m=2)
  show_routing_graph(net, route)

def routing_link_state ():
  net, route, *_ = routing_test(LinkStateNode, init_link_state, n=20, m=2)
  show_routing_graph(net, route)

# er()
# tiers()
# mesh()
# routing_flooding()
# routing_bellman_ford()
# routing_link_state()
```

routing_test.py

```python
from time import time
from timeit import timeit
from random import choices
from math import sqrt
import networkx as nx
from network import LocalNetworkInterface, Packet
from nodes.flooding import FloodingNode
from nodes.bellman_ford import BellmanFordNode
from nodes.link_state import LinkStateNode
from topo.mesh import MeshNetwork
from topo.er import ERNetwork

def _get_routable_net (Node, n, p=None, m=None):
  nodes = [Node(i + 1) for i in range(n)]

  if m is not None:
    net = MeshNetwork(nodes, m)
  elif p is not None:
    net = ERNetwork(nodes, p)
  else:
    raise Exception('Expected p or m to be provided')

  for node in nodes:
    interface = LocalNetworkInterface(node, net)
    node.set_interface(interface)

  return net, nodes

def show_routing_graph (net, route, filename=None):
  edges = set((node1, node2) for node1, node2 in zip(route[:-1],
route[1:]))

  def get_edge_color (node1, node2, weight):
    if (node1, node2) in edges or (node2, node1) in edges:
      return 'red'
    return 'black'

  net.show(file=filename, get_edge_color=get_edge_color)

def routing_test (NodeClass, init_func=None, n=20, p=None,
m=None):
  t0 = time()

  net, nodes = _get_routable_net(NodeClass, n=n, p=p, m=m)
  t1 = time()

  if init_func:
    init_func(net, nodes)
  t2 = time()

  src, dst = choices(nodes, k=2)

  src.send_packet(Packet('hello world', src.addr, dst.addr))

  while len(dst.packets_received) == 0:
    net.routing_step()
  t3 = time()

  route = dst.packets_received[0].route

  return net, route, t0, t1, t2, t3

def routing_flooding (n=20):
  return routing_test(FloodingNode, n=n, m=2)

def init_bellman_ford (net, nodes):
  for node in nodes:
    node.update_weights()

  d = nx.diameter(net.graph)

  for _ in range(d + 2):
    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 = len(data)
  s = 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')
```