



Rapport : Tables de routage

29 AVRIL 2024

Groupe 3 14h45 TD1
VALVERDE Victorien
YAPO Ebeguy
ASSIS Hugo

Préambule

Termes Techniques

Il convient d'expliquer les termes techniques de ce projet avant d'en expliquer son code et son contenu :

1. Graphe

- a. Il s'agit d'un réseau où des entités, comme des nœuds, communiquent au travers de liens. Ces liens représentent par où l'information peut circuler et avec quel coût. Le coût peut être en termes de temps, de distance, de monnaie etc.

2. Matrice d'adjacence

- a. Pour représenter un graphe il existe de nombreuses manières, mais la plus utile dans notre contexte est la matrice d'adjacence. Il s'agit d'une matrice, donc d'un tableau en deux dimensions de même taille, qui contient les informations nécessaires pour définir un graphe. Pour lire une matrice d'adjacence, on se réfère à la ligne et à la colonne.
- b. Il existe des types de matrice d'adjacence, telle qu'une matrice supérieure ou inférieure. On peut faire le choix d'exclure les diagonales.

3. Nœud

- a. Il s'agit d'un composant fondamental d'un graphe. En effet, c'est ce qui permet de représenter des entités qui interagissent entre elles, comme des ordinateurs, des commutateurs, des stations de métro etc.
- b. Dans notre contexte, les nœuds possèdent un nom, un tier, une liste de voisins et une table de routage vers d'autres nœuds.

4. Backbone

- a. Dans ce contexte, le graphe suit une structure particulière. En effet chaque nœud possède un tier, ou un niveau, qui le distingue par les coûts de ses liens. Plus le tier est proche de 0, plus le nœud est rapide.
- b. Le Backbone, 10 nœuds, représentent le squelette du réseau. Ce sont eux qui font transiter l'information le plus rapidement de tous les nœuds et qui relient les différentes parties du réseau.

5. Transit

- a. Les nœuds de transit sont moins rapides que les nœuds du backbone mais ils permettent de rendre le réseau plus connecté et de créer des chemins parfois plus courts.

6. Nœud Régulier

- a. Ces nœuds sont les plus nombreux. Ils ne servent pas d'intermédiaire contrairement aux nœuds de tier I et de tier II, ce qui explique pourquoi leurs liens sont les plus coûteux. C'est eux qu'il faut choisir quand on veut obtenir des routes pertinentes quand on teste les fonctions de plus court chemin.

7. Plus court chemin

- a. Problème classique de la théorie des graphes. Il s'agit de trouver le chemin le plus court reliant deux nœuds. Dans la vie courante, l'équivalent serait de trouver le chemin allant de notre position actuelle à une destination en prenant en compte des paramètres telle que le temps, le trafic etc.

Décisions

Structures de données

Nous avons choisi de représenter le graphe et les nœuds à l'aide de classes, ils ont ainsi des attributs et surtout des méthodes qui leurs sont propres. Cependant, il ne suffit pas d'une classe pour représenter un graphe composé d'autant de nœuds, c'est pourquoi nous le représentons à l'aide d'une matrice d'adjacence supérieure, qui est en fait un array Numpy en deux dimensions. Puisque le graphe est non-orienté, stocker les informations dans la partie inférieure serait redondant et complexe à gérer pour la génération des nœuds. Chaque cellule représente une connexion : Si un lien existe, la cellule contiendra sa valeur, sinon elle contiendra 0 si le lien n'existe pas, et l'infini si le lien est interdit (la diagonale).

Pour l'affichage de ce graphe, nous avons été confrontés à un problème : Il y'a beaucoup trop de nœuds pour que le terminal puisse l'afficher. Il existe deux solutions. La première consiste à exporter dans le dossier ./spreadsheets le graphe dans un tableur. Cela a ses avantages comme le fait de pouvoir tout visualiser et d'appliquer des couleurs. Nous appliquons les couleurs aux lignes et aux colonnes qui correspondent à la fin d'une plage de nœuds, basé sur leur niveau. Il s'agit de la méthode préférée. La seconde consiste à utiliser le module Panda et afficher une partie de la matrice. Il est déconseillé d'utiliser cette méthode car elle dépend de la police du terminal et elle n'est pas configurée dans le code pour être utilisée clairement. Néanmoins elle existe.

Pour les tables de routages, nous utilisons l'algorithme de Dijkstra pour trouver le plus court chemin. En effet, nous ne traitons pas de liens négatifs entre les nœuds et le graphe est non-orienté, sinon nous aurions recours à d'autres algorithmes, plus complexes et adaptés.

Une fois les tables de routage calculées, nous reconstituons le chemin à l'aide d'une fonction récursive qui s'arrêtent une fois la destination atteinte (Ou si l'utilisateur à rentrer un nœud qui n'existe pas).

Code

Comme demandé dans le sujet, voici le code en annexe :

```
from subprocess import run
installation = input("Would you like to install the required modules? Y/N:")
match installation:
    case "Y" | "Yes" | "1":
        run(["pip", "install", "numpy"])
        run(["pip", "install", "pandas"])
        run(["pip", "install", "openpyxl"])
        run(["pip", "install", "xlsxwriter"])
    case "N" | "No" | "0":
        print("The user cancelled the installation.")
    case _:
        print(f'The user input "{installation}" is wrong.')

from random import choice, randint, random, sample
from numpy import array, zeros, fill_diagonal, inf
from pandas import DataFrame, set_option, ExcelWriter
from datetime import datetime
from os import path, chdir, getcwd
from heapq import heappop, heappush

chdir(path.dirname(path.abspath(__file__)))
# Can use NetworkX for visualization

"""
Projet Graphes WIP
"""

set_option('float_format', '{:.0f}'.format)
set_option('display.max_columns', 100)

def random_event(probability: int):
    """The probability is an int indicating a percentage [probability%]"""
    return (random() < (probability / 100))

def format_infinity(x):
    """Used for readability when printing dataframes"""
    if x == inf:
        return '∞'
    elif x == -inf:
        return '-∞'
    return x
```

```

class Node:
    """Used to represent a Node which is used for Graphs."""

    global_nodes = {}

    def __init__(self, tier: int = 0, name: str = None):
        self.id = id(self)
        self.tier = tier
        self.name = self.set_name(name, self.id)
        self.neighbors = []
        self.global_nodes[self.id] = self.name
        self.routing_table = {}

    def __repr__(self) -> str:
        return (f'{self.infos()}')

    def __str__(self) -> str:
        return (f'{self.name}')

    def set_name(self, name, id):
        if (self.global_nodes).get(name, None):
            return (name + str(id))
        else:
            return (name)

    def infos(self=None):
        """Used to get all attributes of an object"""
        return (vars(self))

class Graph:
    """
    This class represent a non-oriented graph, made of Nodes.
    The matrix is upper half only, excluding the diagonal.
    \nRules are the distribution of nodes per tier
    \nOptional arguments (kwargs):
    \n - no_generation: bool = False -> Use to generate a blank Graph.
    \n - connected: bool = True -> Generate a graph until it's connected or
    not.
    """

    def __init__(self, name: str = None, size=100, rules=(10, 20, 70),
**kwargs):
        self.name = name if name else
f"Graph_{datetime.now().strftime("%Y_%m_%d_%H_%M_%S")}"
        self.distribution = rules
        self.size = size

        self.nodes = self._generate_nodes(rules)
        self.matrix = self._generate_matrix(self.size)

```

```

self.last_route = [] # [(node_name, weight), ]
self._generate_links(rules, connected=kwargs.get("connected", True))
self.not_connected = set()
self._generate_routing_table()

def __str__(self) -> str:
    return ('\n'.join(str(node.infos()) for node in self.nodes))

def _generate_nodes(self, rules: tuple):
    """Creates the list of Nodes and their distribution"""
    temp = []
    # Backbone
    for iteration in range(1, rules[0] + 1):
        temp += [Node(1, f"B{iteration}")]
    # Transit
    for iteration in range(1, rules[1] + 1):
        temp += [Node(2, f"T{iteration}")]
    # Regular
    for iteration in range(1, rules[2] + 1):
        temp += [Node(3, f"R{iteration}")]
    return (temp)

def _generate_matrix(self, size) -> array:
    temp = zeros((size, size))
    fill_diagonal(temp, inf)
    return (temp)

def _generate_links(self, rules, **kwargs):
    """
    The matrix follow these rules: A link exists if it's >1, a line show
a Node's neighbors,
    a column show what a Node is connected to, the value represents the
speed of the link
    \nOptionnal Arguments:
    \n - connected: bool -> Generate a graph until it's connected or not.
    """

    first_iter = True
    cpt = 0
    while (self.is_connected() is not kwargs.get("connected", True)) or
(first_iter is True):
        cpt += 1
        self.matrix = self._generate_matrix(self.size) # Reset the
matrix if failure
        # Tier I
        for node_A in range(0, rules[0]):
            for node_B in range(0, rules[0]):
                if (self.get_link(self.matrix, node_A, node_B) == 0) and
(random_event(75)):
                    link_value = randint(5, 10)

```

```

        self.set_link(self.matrix, node_A, node_B,
link_value)

        self.nodes[node_A].neighbors += [node_B]
        self.nodes[node_B].neighbors += [node_A]

    # Tier II
    for node_A in range(rules[0], rules[0] + rules[1]):
        # Part 2
        if len(self.nodes[node_A].neighbors) < 2:
            candidates = self.filter_nodes(output="index", tier=2,
neighbors_limit=(3, "<"), exclude=node_A)

            if len(candidates) > 3: # Sample doesn't work if the
population is less than the picked amount
                selection = sample(list(candidates), randint(2, 3))
            else:
                selection = candidates
            for node_B in selection:
                if (self.get_link(self.matrix, node_A, node_B) ==
0): # Do not override an existing link or a diagonal
                    link_value = randint(10, 20)
                    self.set_link(self.matrix, node_A, node_B,
link_value)

                    self.nodes[node_A].neighbors += [node_B]
                    self.nodes[node_B].neighbors += [node_A]
        for node_A in range(rules[0], rules[0] + rules[1]):
            # Part 1
            selection = sample(list(range(0, rules[0])), randint(1, 2))
            for node_B in selection:
                if (self.get_link(self.matrix, node_A, node_B) == 0): #
Do not override an existing link or a diagonal
                    link_value = randint(10, 20)
                    self.set_link(self.matrix, node_A, node_B,
link_value)

                    self.nodes[node_A].neighbors += [node_B]
                    self.nodes[node_B].neighbors += [node_A]

    # Tier III
    for node_A in range(rules[0] + rules[1], sum(rules)):
        # Part 1
        selection = sample(list(range(rules[0], rules[0] +
rules[1])), 2)
        for node_B in selection:
            if (self.get_link(self.matrix, node_A, node_B) == 0): #
Do not override an existing link or a diagonal
                link_value = randint(20, 50)
                self.set_link(self.matrix, node_A, node_B,
link_value)

                self.nodes[node_A].neighbors += [node_B]
                self.nodes[node_B].neighbors += [node_A]

    first_iter = False

```



```

        print(f"Generated graph in {cpt} attempts")

    def display_links(self, shape=(10, 20), slice: tuple = (0, 999)):
        """
        Will display all the Nodes and their connections.
        However, since a terminal cannot display 100 Nodes
        , this function divide all the nodes by a specified shape.
        \nArguments:
        \n\tshape: tuple (rows, columns)
        \n\tslice: tuple (rows, columns) -> based on the shape, the slice
will be multiplied
        """

        set_option('display.width', 9 * shape[1]) # Adjusts the number of
columns/rows to display

        slice_size = array(slice) * shape[0]
        slice_diff = (slice_size[1] - slice_size[0])

        matrix = self.matrix[slice_size[0]:slice_size[1]]
        labels = [node.name for node in self.nodes]

        chunks = [matrix[row:row + slice_diff] for row in range(0,
len(matrix), slice_diff)]
        dataframes = [DataFrame(chunk) for chunk in chunks]
        for dataframe in dataframes:
            dataframe.index = labels[slice_size[0]:slice_size[1]]
            dataframe.columns = labels
            print(dataframe.map(format_infinity))

    def export(self):
        """
        Will export the current Graph's matrix in an excel spreadsheet. The
file is in the folder "spreadsheets"
        """

        temp_dataframe = DataFrame(self.matrix).map(format_infinity)
        temp_dataframe.index = [node.name for node in self.nodes]
        temp_dataframe.columns = [node.name for node in self.nodes]

        print(f"Exporting to \033[96m{getcwd()}\033[0m as
\033[96m{self.name}_spreadsheet.xlsx\033[0m")

        with ExcelWriter(f'spreadsheets/{self.name}_spreadsheet.xlsx',
engine='xlsxwriter') as writer:
            temp_dataframe.to_excel(writer, sheet_name=self.name, startrow=0,
startcol=0, index=True)

            # Access the workbook and worksheet objects
            workbook = writer.book
            worksheet = writer.sheets[self.name]

```

```

        # Cell Formats
        centered_format = workbook.add_format({'align': 'center'})
        backbone_format = workbook.add_format({'align': 'center',
'bg_color': '#963634'})
        transit_format = workbook.add_format({'align': 'center',
'bg_color': '#007BA7'})
        ignore_format = workbook.add_format({'align': 'center',
'bg_color': '#222222'})

        for index in range(1, len(temp_dataframe.columns) + 1):
            worksheet.set_column(index, index, 3, centered_format)
            if index == self.distribution[0]: # Backbone
                worksheet.set_row(index, None, backbone_format)
                worksheet.set_column(index, index, 3, backbone_format)
            if index == sum(self.distribution[:-1]): # Transit
                worksheet.set_row(index, None, transit_format)
                worksheet.set_column(index, index, 3, transit_format)

        # All bottom matrix, including diagonal
        for column in range(1, len(temp_dataframe) + 1):
            for row in range(column, len(temp_dataframe) + 1):
                worksheet.write(row, column, temp_dataframe.iloc[row - 1,
column - 1], ignore_format)

    def get_invert_id(self, node_id: int | str):
        """
        Get the identifier if you provide an index or a name.
        \nindex -> name
        \nname -> index
        """

        if not isinstance(node_id, int | str):
            raise TypeError(f"The provided node type({type(node_id)}) is
incorrect")
        elif isinstance(node_id, int):
            return (self.nodes[node_id].name)
        else:
            for index, node in enumerate(self.nodes):
                if node.name == node_id:
                    return (index)
            raise NameError(f"The provided Node name ({node_id}) does not
exist")

    def get_neighbors(self, matrix, node: any, **kwargs):
        """
        Returns the neighbors of a Node, indicated by it's index
        \nOptional Arguments:
        \n\toutput: str {index, name, amount} -> The output format
        """

```

```

        res = []
        for index in range(len(self.nodes)):
            if self.get_link(matrix, node if isinstance(node, int) else
self.get_invert_id(node), index) not in [inf, 0]:
                res += [self.nodes[index].name if kwargs.get("output") ==
"name" else index]

        return (len(res) if kwargs.get("output") == "amount" else res)

    def filter_nodes(self, **kwargs) -> list | Node:
        """
        Return all nodes based on the provided filters. If the provided
        filter is wrong, it will not know.
        \nOptional Arguments:
        \n\ttier: int -> Filter based on the tier
        \n\tname: str -> Filter based on the name
        \n\tneighbors_limit: tuple (int, mode: str {==, <, >, <=, >=}) ->
        Filter based on the amount of neighbors and the mode
        \n\toutput: str {None, index, name, amount} -> Returns a list of
        indexes or names
        \n\texclude: [int] -> Will not include the specified nodes
        """

        excluded = [kwargs.get("exclude", [])] if
isinstance(kwargs.get("exclude", []), int) else kwargs.get("exclude", [])
        res = [node for index, node in enumerate(self.nodes) if index not in
excluded]

        if tier := kwargs.get("tier"):
            res = [node for node in res if node.tier == tier]
        if name := kwargs.get("name"):
            res = [node for node in res if node.name == name]
        if neighbors_limit := kwargs.get("neighbors_limit"):
            match neighbors_limit[1]:
                case "==":
                    res = [node for node in res if len(node.neighbors) ==
neighbors_limit[0]]
                case "<":
                    res = [node for node in res if len(node.neighbors) <
neighbors_limit[0]]
                case "<=":
                    res = [node for node in res if len(node.neighbors) <=
neighbors_limit[0]]
                case ">":
                    res = [node for node in res if len(node.neighbors) >
neighbors_limit[0]]
                case ">=":
                    res = [node for node in res if len(node.neighbors) >=
neighbors_limit[0]]

```

```

        if (kwargs.get("output", False) == "name"):
            return [node.name for node in res]
        if (kwargs.get("output", False) == "amount"):
            return (len(res))
        if (kwargs.get("output", False) in [False, "index"]):
            return (list(map(self.get_invert_id, [node.name for node in
res])))

    def get_node(self, node: int | str):
        if not isinstance(node, int | str):
            raise TypeError(f"The provided node type({type(node)}) is
incorrect")
        elif isinstance(node, int):
            return (self.nodes[node])
        else:
            return (self.nodes[self.get_invert_id(node)])

    def get_link(self, matrix, node1: any, node2: any):
        if isinstance(node1, str) and isinstance(node2, str): # If given by
name
            a, b = self.get_invert_id(node1), self.get_invert_id(node2)
            return (matrix[(a, b) if a < b else (b, a)])
        elif isinstance(node1, int) and isinstance(node2, int): # If given
by index
            return (matrix[(node1, node2) if node1 < node2 else (node2,
node1)])
        else:
            print(f"The node's type provided is invalid ({type(node1)},
{type(node2)})")
            return (None)

    def set_link(self, matrix, node1: any, node2: any, value: float):
        if isinstance(node1, str) and isinstance(node2, str): # If given by
name
            a, b = self.get_invert_id(node1), self.get_invert_id(node2)
            if a != b:
                matrix[(a, b) if a < b else (b, a)] = value
                print(f"\033[92mCreated link ({node1},
{node2})={value}\033[0m")
            elif isinstance(node1, int) and isinstance(node2, int): # If given
by index
                if node1 != node2:
                    matrix[(node1, node2) if node1 < node2 else (node2, node1)] =
value
                    print(f"\033[92mCreated link ({self.nodes[node1].name},
{self.nodes[node2].name})={value}\033[0m")
                else:
                    print(f"The node's type provided is invalid ({type(node1)},
{type(node2)})")
                    return (None)

```

```

def is_connected(self):
    """
    Check whether the graph is connected using a breadth-first path
    (BFS).
    """
    visited = set()
    queue = [0]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            neighbors = self.get_neighbors(self.matrix, node,
output="index") # Get the neighbors of the current node
            for neighbor in neighbors:
                if neighbor not in visited:
                    queue.append(neighbor)

    self.not_connected = visited ^ {node_index for node_index in
range(self.size)}
    return (len(visited) == len(self.nodes))

def _generate_routing_table(self):
    """
    Calculate the routing tables for each node using Dijkstra's
    algorithm.
    Function that can be adjusted or modified.
    """

    for node_index, node in enumerate(self.nodes):
        # Initialize distances and predecessors
        distances = {i: float('inf') for i in range(len(self.nodes))}
        predecessors = {i: None for i in range(len(self.nodes))}
        distances[node_index] = 0
        priority_queue = [(0, node_index)]

        while priority_queue:
            current_distance, current_node = heappop(priority_queue)

            # Use the get_neighbors method to get the neighbors of the
            current node
            neighbors = self.get_neighbors(self.matrix, current_node,
output="index")

            for neighbor in neighbors:
                link_value = self.get_link(self.matrix, current_node,
neighbor)

                if link_value != float('inf'):
                    new_distance = current_distance + link_value

```

```

        if new_distance < distances[neighbor]:
            distances[neighbor] = new_distance
            predecessors[neighbor] = current_node
            heappush(priority_queue, (new_distance,
neighbor))

    routing_table = {}
    for dest_index in distances:
        if dest_index != node_index:
            # Reconstruct the path
            path = []
            current = dest_index
            while current is not None:
                path.insert(0, current)
                current = predecessors[current]

            # Next jump is the next node in the path
            routing_table[self.nodes[dest_index].name] =
self.nodes[path[1]].name if len(path) > 1 else None

    # Link the routing table to the node
    node.routing_table = routing_table

def traceroute(self, node_A: int | str, node_B: int | str, **kwargs):
    """Recursive function that traces the route from node A to node B.
    \n Optionnal Arguments:
    \n - first_iteration: bool = True -> Used to reset the attribute
last_route. Do not modify unless needed.
    \n - display: bool = True -> Used to display the results when called.
Include
    """
    # Error and format handling
    if type(node_A) is not type(node_B):
        raise TypeError(f"{node_A}'s type is different from {node_B}'s
type.")

    if isinstance(node_A, str) and not node_A.isdigit():    # If not a
digit
        node_A = self.get_invert_id(node_A)
    else:
        node_A = int(node_A)
    if isinstance(node_B, str) and not node_B.isdigit():    # If not a
digit
        node_B = self.get_invert_id(node_B)
    else:
        node_B = int(node_B)

    if kwargs.get("first_iteration", True) is True:
        if isinstance(node_A, int):
            self.last_route = [(self.get_invert_id(node_A), 0), ]

```

```

        else:
            self.last_route = [(node_A, 0), ]

    if node_A == node_B:    # Stop condition
        if kwargs.get("display", True) is True:
            weights = [step[1] for step in self.last_route]
            route = [step[0] for step in self.last_route]
            return (f"The whole route takes \033[96m{sum(weights)} units
({str(weights)[1:-1]})\033[0m.\nThe route is \033[96m{'-
>'.join(route)}\033[0m")
        else:
            return sum([step[1] for step in self.last_route])

    next_node =
self.nodes[node_A].routing_table.get(self.get_invert_id(node_B))
    self.last_route += [(next_node, self.get_link(self.matrix, node_A,
self.get_invert_id(next_node)))]
    return self.traceroute(self.get_invert_id(next_node), node_B,
first_iteration=False)

G = Graph(connected=True)    # Can force a graph to be not connected (Very
difficult)
print(f"The graph is {"connected" if G.is_connected() else "not connected"}")

export = input("Would you like to export in an excel spreadsheet? Y/N:")
match export:
    case "Y" | "Yes" | "1":
        G.export()
    case "N" | "No" | "0":
        print("The user cancelled the export.")
    case _:
        print(f'The user input "{export}" is wrong. Cancelled the export.')

node_tracing = input('Do you want to traceroute? Type "Stop" to stop.')
while node_tracing not in ["Stop", "stop"]:
    node_A = input("What is the node A?")
    node_B = input("What is the node B?")
    print(G.traceroute(node_A, node_B, display=True))
    node_tracing = input('Do you want to traceroute? Type "Stop" to stop.')

```