ISEP

PROJECT
Advanced Algorithmic and Programming

Cyprien Bariant
Tanguy Berthoud 60989
Thibault du Buisson de Courson

January 6, 2021

Table of Contents

List of Figures

List of Tables

1 Creating a graph from GTFS data

1.1 Importing relevant GTFS data

Our city of choice for the project is Phoenix, in Arizona.
We use the public data available here: https://transitfeeds.com/p/valley-metro/68/latest

The difficulty here was to import the relevant data and convert it to a graph. Some Python libraries seem to exist but none was convenient for the project, so we needed to transform the data manually. After reading documentation on GTFS, we needed only two files from the data feed: `stops.txt` and `stop_times.txt`.

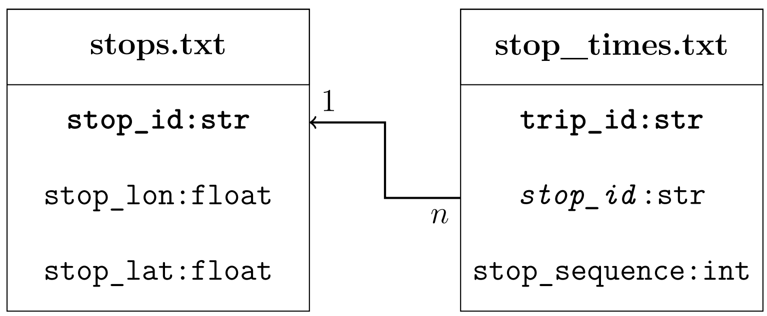

Figure 1 – Database representation of the relevant GTFS files

The nodes of the graph are directly given by the file `stops.txt`, so we can just parse the file to import them. This is done in the `import_nodes` function of `gtfs.py`.

Whereas the edges need some operations:

1. Parse the file
2. Regroup in order the stops in the same trips
3. Create edges between consecutive stops in a trip

This is done in the `import_edges` function of `gtfs.py`.

1.2 Creating a graph

The `Graph` class is in `graph.py`.

The class supports weighted directed graphs, but unweighted or undirected graphs can also be created.

The class constructor accepts an optional parameter which is a callback to compute weight from two given nodes. In our case, this function takes two `Stop` (defined in `gtfs.py`) instances and returns the Euclidian distance between them:

$$\text{compute\_weight} : (s, s') \mapsto \sqrt{(s'_{\text{lat}} - s_{\text{lat}})^2 + (s'_{\text{lon}} - s_{\text{lon}})^2}$$

If this parameter is not passed, all edge weights will be set to 0.
The support for directions comes from the `neighbors_in` and `neighbors_out` methods. The method `neighbors` can be used instead for undirected graphs.

After testing the pathfinding methods (see section 2), we realized that our way of storing neighbors was not optimized.
Indeed we stored the neighbors in a huge adjacency list. So when we needed to compute some neighbors, we had to search in the whole list for neighbors.
So we optimized this aspect. We are now storing the neighbors of a node inside the `Node` object itself. This implementation has redundancy in memory, but fetching the neighbors of a node is now $\mathcal{O}(1)$ for the `neighbors_out` and `neighbors_in` methods, but not for `neighbors` since it computes the union of two sets.

1.3 Results

- 7982 lines from `stops.txt` were imported to 7982 nodes in the graph.
- 1720661 lines from `stop_times.txt` were imported to 8462 edges in the graph.
  This is not surprising, because one line is used for a single stop in a single trip at certain hours: there are a lot of redundancies.

	`import_stops`	`import_edges`	`Graph`
Before neighbors optimization	38.61	4211.02	78.64
	31.75	5036.29	49.05
	44.50	5085.64	68.83
	28.56	3786.03	43.43
	26.48	4070.86	43.62
After neighbors optimization	27.24	4931.54	155.09
	29.34	3610.81	74.15
	49.93	4096.88	115.51
	28.70	3279.42	86.15
	26.39	3685.92	82.94

Table 1 – Execution time (ms) for the graph creation over several tries

We see in this Table 1 that after the optimization described in subsection 1.2, the execution time for the `Graph` instanciation has increased from an average of 56.71 ms to 102.77 ms. This can be explained by the new storage method of neighbors: we need to add two different tuples into two different sets.

2 Finding the shortest paths

Appendix

A gtfs.py

```
from typing import Tuple
from sys import argv
from os import getcwd
from os.path import join
from time import perf_counter
from math import sqrt
from timing import timing
from pathfinding import *
from clustering import clustering


Position = Tuple[float, float]

class Stop:
    """Stop representation

    # Properties
    - `id` - Unique identifier
    - `position` - Position of the stop
    """

    def __init__(self, id: str, lat: float, lon: float):
        self.__id: str = id
        self.__position: Position = (lat, lon)

    def from_csv(line: str) -> 'Stop':
        """Construct a Stop instance from CSV data

        # Arguments
        - `line` - CSV line containing data of the stop
        """
        data = line.split(",")
        return Stop(data[0], float(data[4]), float(data[5]))

    def __repr__(self) -> str:
        return "{0} {1}".format(self.__id, self.__position)

    def __eq__(self, other: 'Stop') -> bool:
        return self.id == other.id and self.position == other.position

    def __hash__(self) -> int:
        return hash((self.__id, self.__position))

    @property
    def id(self) -> str:
        return self.__id

    @property
    def position(self) -> Position:
```

```python
        return self.__position


def import_stops(file: str) -> Tuple[List[Stop], Dict[str, int]]:
    """Import stops from GTFS `stops.txt`

    # Arguments
    - `file` - Path to the file

    # Return value
    Tuple `(stops, id_map)` where `stops` is a list of `Stop` instances, and `id_map` is a dictionnary `stop.id => node_id` where `node_id` is the index of the node in `stops`
    """
    stops: List[Stop] = []
    id_map: Dict[str, int] = dict()
    with open(file, "rt") as data:
        for (i, line) in enumerate(data.readlines()[1:]):
            stop = Stop.from_csv(line)
            stops.append(stop)
            id_map[stop.id] = i
    return (stops, id_map)

def import_edges(file: str) -> Set[Tuple[str, str]]:
    """Import edges from GTFS `stop_times.txt`

    # Arguments
    - `file` - Path to the file

    # Return value
    Set of ordered tuples of stop IDs
    """
    trips: Dict[str, Dict[int, str]] = dict()
    # Import raw data
    with open(file, "rt") as data:
        for line in data.readlines()[1:]:
            cols = line.split(",")
            if cols[0] not in trips:
                trips[cols[0]] = dict()
            trips[cols[0]][int(cols[4])] = cols[3]
    # Transform data
    edges: Set[Tuple[str, str]] = set()
    for trip in trips.values():
        stop_seq = sorted(trip)
        for (i, stop) in enumerate(stop_seq[:-1]):
            edges.add((trip[stop], trip[stop_seq[i + 1]]))
    return edges


if __name__ == "__main__":
    DATAPATH = argv[1] if len(argv) > 1 else getcwd()
    # Import data
    ((stops, id_map), exetime) = timing(import_stops)(join(DATAPATH, "stops.txt"))
    print("Imported {0} stops in {1}ms".format(len(stops), exetime * 1e3))
    (edges, exetime) = timing(import_edges)(join(DATAPATH, "stop_times.txt"))
    print("Imported {0} edges in {1}ms".format(len(edges), exetime * 1e3))
```

```python

    # Construct graph
    exetime = perf_counter()
    GRAPH = Graph(stops, compute_weight=lambda u, v: sqrt(
        (v.position[0] - u.position[0]) ** 2 + (v.position[1] - u.position[1]) ** 2))
    for edge in edges:
        GRAPH.add_edge(id_map[edge[0]], id_map[edge[1]])
    print("Constructed graph in {0}ms".format((perf_counter() - exetime) * 1e3))

    # Construct pathfinders
    BFS = Pathfinder(GRAPH, bfs)
    DIJKSTRA = Pathfinder(GRAPH, dijkstra)

    # Detect clustering
    # clustering(DIJKSTRA, stops.keys(), 5)
```

B graph.py

```python
from typing import Callable, Generic, Iterable, Iterator, List, Set, Tuple, TypeVar


T = TypeVar("T")
Adjacency = Tuple[int, float]

class Node(Generic[T]):
    """Graph node representation

    # Generic
    - `T` - Type of the node value

    # Properties
    - `value` - Value of the node
    - `neighbors_out` - Outward neighbors of the node
    - `neighbors_in` - Inward neighbors of the node
    """

    def __init__(self, value: T):
        self.__value = value
        self.__neighbors_out: Set[Adjacency] = set()
        self.__neighbors_in: Set[Adjacency] = set()

    def __repr__(self) -> str:
        return repr(self.__value)

    def __eq__(self, other: 'Node') -> bool:
        return self.value == other.value and self.neighbors_out == other.neighbors_out and self.neighbors_in == other.neighbors_in

    def __hash__(self) -> int:
        return hash((self.__value, frozenset(self.__neighbors_out), frozenset(self.__neighbors_in)))

```

```
    @property
    def value(self) -> T:
        return self.__value

    @property
    def neighbors_out(self) -> Set[Adjacency]:
        return self.__neighbors_out

    @property
    def neighbors_in(self) -> Set[Adjacency]:
        return self.__neighbors_in

class Graph(Generic[T]):
    """Graph (weighted directed) representation

    # Generic
    - `T` - Type of the nodes

    # Properties
    - `nodes` - List of nodes
    - `compute_weight` - Function to compute edge weight from two nodes
    """

    def __init__(self, nodes: Iterable[T], compute_weight: Callable[[T, T], float]↩
        = None):
        self.__nodes: List[Node[T]] = []
        for node in nodes:
            self.add_node(node)
        self.__size: int = 0
        self.__compute_weight = compute_weight

    def __iter__(self) -> Iterator[Node[T]]:
        return iter(self.__nodes)

    def __getitem__(self, key: int) -> Node[T]:
        return self.__nodes[key]

    @property
    def order(self) -> int:
        return len(self.__nodes)

    @property
    def size(self) -> int:
        return self.__size

    def add_node(self, node: T) -> int:
        """Add a node to the graph

        # Arguments
        - `node` - Node value to add

        # Return value
        Key of the newly-added node
        """
        self.__nodes.append(Node(node))
        return len(self.__nodes) - 1
```

```

    def add_edge(self, u: int, v: int):
        """Add an edge `u-(weight)->v` to the graph

        Will compute the weight using the `compute_weight` property.
        If `compute_weight` is `None`, the weight will be 0.

        # Arguments
        - `u` - Key of the first node
        - `v` - Key of the second node

        # Errors thrown
        - `ValueError` if both keys are equal
        """
        if u == v:
            raise ValueError("u={0} and v={0} are equal".format(u, v))
        else:
            weight = float(0) if self.__compute_weight is None else self.__compute_weight(self.__nodes[u].value, self.__nodes[v].value)
            self.__nodes[u].neighbors_out.add((v, weight))
            self.__nodes[v].neighbors_in.add((u, weight))
            self.__size += 1
```

C pathfinding.py

```
from typing import Callable, Dict, List, Set
from math import inf
from graph import Graph


class Pathfinder:
    """Wrapping class to allow pathfinding in graphs

    # Properties
    - `graph` - Graph used to compute pathfinding
    - `previous` - Dictionnary `from => to => previous` where `previous` is the previous node of `to` when searching from `from`
    - `distance` - Dictionnary `from => to => distance`
    - `method` - Function to compute shortest paths from a node
    """

    def __init__(self, graph: Graph, method: Callable[['Pathfinder', int], None]):
        self.graph = graph
        self._previous: Dict[int, Dict[int, int]] = dict()
        self._distance: Dict[int, Dict[int, float]] = dict()
        self.__method = method

    def has_path(self, u: int, v: int) -> bool:
        """Check if a path exist between two nodes

        # Arguments
        - `u` - Key of the starting node
        - `v` - Key of the ending node

```

```
        # Return value
        `True` if a path exists; `False` otherwise
        """
        return u in self._previous and v in self._previous[u]

    def get_path(self, u: int, v: int):
        """Get the shortest path between two nodes

        # Arguments
        - `u` - Key of the starting node
        - `v` - Key of the ending node

        # Return value
        Ordered list of node keys; or `None` if a path does not exist
        """
        if u not in self._previous:
            self.__method(self, u)
        if not self.has_path(u, v):
            return None
        else:
            path: List[str] = [v]
            current = v
            while current != u:
                next = self._previous[u][current]
                path.append(next)
                current = next
            path.reverse()
            return path

    def get_distance(self, u: int, v: int) -> float:
        """Get the distance between two nodes

        # Arguments
        - `u` - Key of the starting node
        - `v` - Key of the ending node

        # Return value
        Distance from `u` to `v`, in edges
        """
        if u not in self._previous:
            self.__method(self, u)
        return self._distance[u][v] if v in self._distance[u] else inf


def bfs(self: Pathfinder, v: int):
    """Breadth-First Search method for `Pathfinder`"""
    if v not in self._previous:
        self._previous[v] = dict()
        self._distance[v] = dict()
        self._distance[v][v] = 0
        queue = [v]
        while len(queue) > 0:
            current = queue.pop(0)
            for (u, _) in self.graph.neighbors_out(current):
                if u not in self._previous[v]:
                    self._previous[v][u] = current
```

```python
                    self._distance[v][u] = self._distance[v][current] + 1
                    queue.append(u)


def dijkstra(self: Pathfinder, v: int):
    """Dijkstra method for `Pathfinder`"""
    if v not in self._previous:
        self._previous[v] = dict()
        self._distance[v] = dict()
        self._distance[v][v] = 0
        marked: Set[int] = set()
        queue = [v]
        while len(queue) > 0:
            current = queue.pop(0)
            marked.add(current)
            for (u, weight) in self.graph.neighbors_out(current):
                tentative_distance = self._distance[v][current] + weight
                if u not in self._distance[v] or tentative_distance < self._distance[v][u]:
                    self._previous[v][u] = current
                    self._distance[v][u] = tentative_distance
                if u not in marked:
                    queue.append(u)
```