ISEP

PROJECT
Advanced Algorithmic and Programming

Cyprien Bariant 10558
Tanguy Berthoud 60989
Thibault du Buisson de Courson 10496

January 13, 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`.

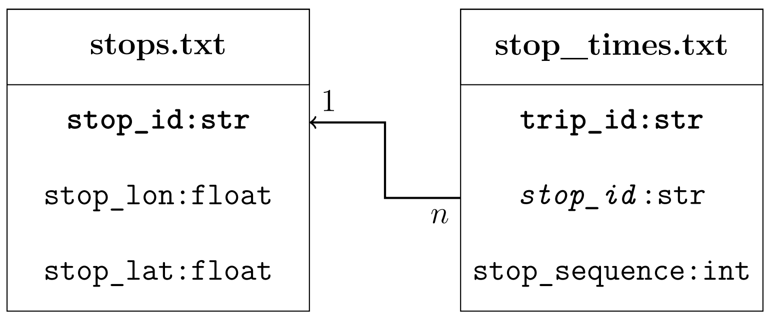

Figure 1 – Database representation of the relevant GTFS files

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_stops` 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 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.

After testing the pathfinding methods (see subsection 2.2), 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 our graph was just too big at that time to be able to work efficiently on it.
So we restricted the graph to a district of the city, so the graph's order and size have been divided by 8.

1.3 Results

- 7982 lines from `stops.txt` were imported to 891 nodes (7863 before restricting the graph) in the graph.
- 1720661 lines from `stop_times.txt` were imported to 975 edges (8319 before the restriction) in the graph.

	`import_stops`	**`import_edges`**	**`Graph`**
Before optimizations	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 neighbors 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

2.1 Common interface

To ease the usage of pathfinding for future computations, we created a common interface which is the `Pathfinder` class of `pathfinding.py`.

We used the *Strategy design pattern* here with the `method` property, which is assigned in the class constructor to a function with the following signature:

$$\text{method} : (\text{pathfinder} : \text{Pathfinder}, \text{start} : \text{int}) \to \text{None}$$

This function is to compute all shortest paths from `start` and store them in `pathfinder`, using its `previous` and `distance` dictionnaries.

2.2 Pathfinding methods

We implemented two methods for the common interface: `bfs` and `dijkstra`, defined in `pathfinding.py`.

Both use a queue to define the next nodes to be traversed, and proceed until this queue is empty. For `dijkstra`, this queue is actually a priority queue, sorted by ascending distance to `start`.

For each outward neighbor n of the current traversed node c, we initialize the relevant sections of `pathfinder` if needed, and we update them with the current neighbor traversed:

- `bfs` if $d(\text{start}, n) = d(\text{start}, c) + 1$, then we add c to the backtrack list of n.
- `dijkstra` if $d(\text{start}, n) \leq d(\text{start}, c) + w$ with w the weight of the edge (c, n), then we add c to the backtrack list of n. And if $d(\text{start}, n) < d(\text{start}, c) + w$, then we reset the backtrack list and distance.

To optimize these methods, at the time when our graph was not restricted, we replaced the queues (which were simple lists) by heap queues using the `heapq` module of Python.
This also improved the correctness of our methods, since it allows to pop nodes from the queue in a consistent order.

2.3 Results

	`bfs`	`dijkstra`
Before optimizations	1.54	2.05
	1.48	2.26
	2.72	3.51
	2.37	2.88
	2.15	2.33
After heaps optimization	1.92	4.09
	3.06	5.40
	2.97	5.77
	3.36	8.08
	2.71	4.38

Table 2 – Execution time (ms) for the pathfinding methods over several tries

We see in this Table 2 that after the heaps optimization, the execution times have increased from an average of 2.05 ms to 2.80 ms for `bfs` and from 2.61 ms to 5.54 ms for `dijkstra`. Even though this optimization helped a lot when our graph was not restricted to a single district, it seems that using heap queues for little graphs does not help.

3 Finding clusters

Clustering described in this section is implemented in the `clustering` function, defined in `clustering.py`.

3.1 Edge betweenness

Edge betweenness is a good indicator of clusters. Indeed, the edges with the highest betweenness can be seen as the bridges linking the clusters. Thus, destroying a number of these edges can separate these clusters, highlighting them.
Thus, the main component of the clustering algorithm is a loop (lines 13 to 117) that, at each iteration, identifies the clusters in the graph, computes the betweenness of each edge, spots the highest ones and deletes them. The loop repeats that operation until the good number of clusters is obtained (usually 5 to 15 iterations).

The betweenness of an edge is the number of shortest paths connecting nodes that use this edge. So, in order to obtain it, the algorithm must find every shortest path connecting nodes in the graph. To do so, the algorithm first create an empty betweenness list. Then, it uses two nested loops:

- The first one (lines 24 to 96) iterates over every node contained in the graph. Each retreived node is used as a starting point.

- Then, the algorithm uses the second loop (lines 49 to 96) to iterate over every other node contained in the graph.

At each iteration the algorithm use a `Pathfinder` instance (described earlier in subsection 2.1) to check if there is a path between the starting and target nodes (line 52). If there is at least one path, the program iterates on the list of paths retreived (lines 81 to 96). For each path the algorithm browses the list of nodes and, for each pair of neighboring nodes, computes increments the betweenness of the edge in the betweenness list.

3.2 Isolating clusters

A cluster is a list of linked nodes. Identifying them can be done during the computing of the betweenness.

Each time a starting node is chosen the algorithm checks if it has already been used as a target node. If it has not, a potential new cluster has been found, and a new cluster list is created (lines 43 to 47). If it has, the algorithm search the number of the cluster in which the node is (lines 34 to 41). In both cases the algorithm saves the number of the cluster in which is the starting node.

Each time a linked node is found the algorithm can determine that it is in the same cluster as the first. Before doing so the algorithm checks if the node has already been found. If it has, it is the sign that a connected cluster has been found, it is merged into the current one (lines 57 to 72). If it has not, the target node is added to the cluster being explored (lines 73 to 78).

Appendix

This program is meant to be run by invoking Python on the gtfs.py script and adding the path to the GTFS data as argument.
For example, you may execute `python src/gtfs.py ./data/` from the workspace root.

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 __repr__(self) -> str:
        return "{0} {1}".format(self.__id, self.__position)
    def __lt__(self, other: ’Stop’) -> bool:
        return self.id < other.id
    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:
        return self.__position

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

    # Arguments
    - ‘path‘ - Path to the file

    # Return value
```

```python
    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(path, "rt") as file:
        for i, line in enumerate(file):
            if i > 0:
                data = line.strip().split(",")
                if data[10] == "TE" and int(data[8]) < 2 and len(data[9]) == 0: # Filter stops
                    stop = Stop(data[0], float(data[4]), float(data[5]))
                    stops.append(stop)
                    id_map[stop.id] = len(stops) - 1
    return stops, id_map
def import_edges(path: str, id_map: Dict[str, int]) -> Set[Tuple[int, int]]:
    """Import edges from GTFS `stop_times.txt`

    # Arguments
    - `path` - Path to the file
    - `id_map` - `id_map` returned from `import_stops`

    # Return value
    Set of ordered tuples of stop IDs
    """
    trips: Dict[str, List[Tuple[int, int]]] = dict()
    # Import raw data
    with open(path, "rt") as file:
        for i, line in enumerate(file):
            if i > 0:
                data = line.strip().split(",")
                if data[3] in id_map:
                    if data[0] not in trips:
                        trips[data[0]] = []
                    heappush(trips[data[0]], (int(data[4]), id_map[data[3]]))
    # Transform data
    edges: Set[Tuple[int, int]] = set()
    for trip in trips.values():
        while len(trip) > 1:
            edges.add((heappop(trip)[1], trip[0][1]))
    return edges

if __name__ == "__main__":
    # Get data path
    """The path to the data files can be set using a script argument.

    For example, if you execute Python from the workspace root, you can enter: `python src/gtfs.py ./data/`.
    Or, if you execute Python from the `src/` directory: `python gtfs.py ../data/`.
    """
    DATAPATH = argv[1] if len(argv) > 1 else "../data/"
    # 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"), id_map)
    print("Imported {0} edges in {1}ms".format(len(edges), exetime * 1e3))
    # 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 start, end in edges:
        GRAPH.add_edge(start, end)
    print("Constructed graph in {0}ms".format((perf_counter() - exetime) * 1e3))
    # Construct pathfinders
    BFS = Pathfinder(GRAPH, bfs)
    DIJKSTRA = Pathfinder(GRAPH, dijkstra)
    # Create clustering
    clustering(DIJKSTRA, set(id_map.values()), 147)
```

B graph.py

```
from typing import Callable, Dict, Generic, Iterable, Iterator, List, TypeVar

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

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

    # Properties
    - `value` - Value of the node
    - `neighbors_out` - Dictionnary `node => weight` of outward neighbors
    - `neighbors_in` - Dictionnary `node => weight` of inward neighbors
    """
    def __init__(self, value: T):
        self.__value = value
        self._neighbors_out: Dict[int, float] = dict()
        self._neighbors_in: Dict[int, float] = dict()
    def __repr__(self) -> str:
        return repr(self.__value)
    def __lt__(self, other: 'Node') -> bool:
        return self.value < other.value
    def __eq__(self, other: 'Node') -> bool:
        return self.value == other.value
    def __hash__(self) -> int:
        return hash(self.__value)
    @property
    def value(self) -> T:
        return self.__value
    @property
    def neighbors_out(self) -> Dict[int, float]:
        return self._neighbors_out
    @property
    def neighbors_in(self) -> Dict[int, float]:
        return self._neighbors_in
```

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

    # Generic
    - ‘T‘ - Type of the nodes

    # Properties
    - ‘nodes‘ - List of nodes
    - ‘size‘ - Size of the graph
    - ‘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):
        """Add a node to the graph

        # Arguments
        - ‘node‘ - Node value to add
        """
        self.__nodes.append(Node(node))
    def add_edge(self, start: int, end: 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
        - ‘start‘ - Key of the first node
        - ‘end‘ - Key of the second node

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

C pathfinding.py

```python
from typing import Callable, Dict, Generic, List, Set, Tuple, TypeVar
from math import inf
from heapq import heapify, heappop, heappush
from graph import Graph

T = TypeVar("T")
class Pathfinder(Generic[T]):
    """Wrapping class to allow pathfinding in graphs

    # Generic
    - `T` - Type of the graph nodes

    # Properties
    - `graph` - Graph used to compute pathfinding
    - `previous` - Dictionnary `from => to => previous` where `previous` is a list of the previous nodes 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[T], method: Callable[['Pathfinder[T]', int], None]):
        self.graph = graph
        self._previous: Dict[int, Dict[int, Set[int]]] = dict()
        self._distance: Dict[int, Dict[int, float]] = dict()
        self.__method = method
    def reset(self):
        """Reset the pathfinding results"""
        Pathfinder.__init__(self, self.graph, self.__method)
    def compute(self, start: int):
        """Execute the pathfinding method from a certain node

        Save the newly computed data to the save file

        # Arguments
        - `start` - Key of the starting node
        """
        if start not in self._previous:
            self.__method(self, start)
    def has_path(self, start: int, end: int) -> bool:
        """Check if a path exist between two nodes

        # Arguments
        - `start` - Key of the starting node
        - `end` - Key of the ending node

        # Errors thrown
        - `ValueError` if both keys are equal

        # Return value
        `True` if a path exists; `False` otherwise
        """
        if start == end:
            raise ValueError("start={0} and end={0} are equal".format(start, end))
        else:
```

```python
            if start not in self._previous:
                self.compute(start)
            return end in self._previous[start]
    def get_paths(self, start: int, end: int) -> List[List[int]]:
        """Get the shortest path between two nodes

        # Arguments
        - `start` - Key of the starting node
        - `end` - Key of the ending node

        # Errors thrown
        - `ValueError` if both keys are equal

        # Return value
        List of paths, with a path being a list of node indexes
        """
        if start == end:
            raise ValueError("start={0} and end={0} are equal".format(start, end))
        else:
            if start not in self._previous:
                self.compute(start)
            paths: List[List[int]] = []
            def __recurse(path: List[int], pos: int):
                if path[pos] == start:
                    paths.append(path[:pos + 1][::-1])
                else:
                    for previous in self._previous[start][path[pos]]:
                        if len(path) < pos + 2:
                            path.append(previous)
                        else:
                            path[pos + 1] = previous
                        __recurse(path, pos + 1)
            if self.has_path(start, end):
                __recurse([end], 0)
            return paths
    def get_distance(self, start: int, end: int) -> float:
        """Get the distance between two nodes

        # Arguments
        - `start` - Key of the starting node
        - `end` - Key of the ending node

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

def bfs(self: Pathfinder[T], start: int):
    """Breadth-First Search method for `Pathfinder`"""
    if start not in self._previous:
        self._previous[start] = dict()
        self._distance[start] = dict()
        self._distance[start][start] = 0
        queue = [start]
```

```python
        heapify(queue)
        while len(queue) > 0:
            current = heappop(queue)
            for u in self.graph[current].neighbors_out:
                if u not in self._distance[start]:
                    self._previous[start][u] = set()
                    self._distance[start][u] = self._distance[start][current] + 1
                    heappush(queue, u)
                if self._distance[start][u] == self._distance[start][current] + 1:
                    self._previous[start][u].add(current)
def dijkstra(self: Pathfinder[T], start: int):
    """Dijkstra method for `Pathfinder`"""
    if start not in self._previous:
        self._previous[start] = dict()
        self._distance[start] = dict()
        self._distance[start][start] = 0
        marked: Set[int] = set()
        queue: List[Tuple[float, int]] = [(0, start)]
        heapify(queue)
        while len(queue) > 0:
            _, current = heappop(queue)
            marked.add(current)
            for (u, weight) in self.graph[current].neighbors_out.items():
                tentative_distance = self._distance[start][current] + weight
                if u not in self._distance[start] or tentative_distance < self._distance[start][u]:
                    self._previous[start][u] = set()
                    self._distance[start][u] = tentative_distance
                if self._distance[start][u] == tentative_distance:
                    self._previous[start][u].add(current)
                if u not in marked:
                    heappush(queue, (self._distance[start][u], u))
```

D clustering.py

```python
from pathfinding import Pathfinder

def clustering(DIJKSTRA: Pathfinder, nodes, n, display_all=False):
    """
    Clustering method (first try)
    """

    print("\nCreating", n, "clusters...")
    clusters = []  # Content of the clusters
    n_nodes = len(nodes)  # Number of nodes
    iteration = 0

    while len(clusters) < n:  # As long as the right number of clusters has not been constituted
        iteration += 1
        print("\nIteration", iteration)
        edge_betweenness = {}  # Betweenness of each edge
        DIJKSTRA.reset()
        nodes_found = set()  # List of discovered nodes
```

```
        nodes_to_explore = nodes.copy()  # List of nodes to explore
        progress = 0
        clusters = []
        current_cluster = -1

        while nodes_to_explore:  # As long as there is still unexplored nodes

            # Displaying the progress
            new_progress = int(round(100 * (1 - len(nodes_to_explore) / n_nodes), -1))
            if new_progress > progress:
                progress = new_progress
                print(progress, "%", sep='')

            starting_node = nodes_to_explore.pop()  # Take a node to explore

            # Update the list of clusters
            if starting_node in nodes_found:  # If the node has already been discovered...

                # Search the node in every cluster list
                for i in range(len(clusters)):
                    if starting_node in clusters[i]:
                        current_cluster = i

            else:  # If the node hasn't been discovered report the discovery of a new cluster

                clusters.append({starting_node})  # Create a new entry for the new cluster containing its first node
                current_cluster = len(clusters) - 1  # Set the new cluster as the current one

            nodes_found.add(starting_node)  # Add the node to the discovered nodes list

            for target_node in nodes_to_explore:  # Search every other node
                # (paths), exetime = timing(DIJKSTRA.get_paths)(starting_node, target_node)
                # print("Dijkstra of {0} in {1}ms".format(starting_node, exetime * 1e3))
                paths = DIJKSTRA.get_paths(starting_node, target_node)  # Search a path between the two nodes

                if len(paths) > 0:  # If there is a path...

                    # Generating list of contents of each cluster
                    if target_node in nodes_found:  # If the node has already been found

                        # Search the node in the lists of the others clusters
                        for other_cluster in range(len(clusters)):
                            if current_cluster != other_cluster:
                                if target_node in clusters[other_cluster]:

                                    # Add the content of the other cluster to the ↩
```

```
current cluster
                                clusters[current_cluster].update(clusters[other_cluster])
                                del clusters[other_cluster]  # Delete the other cluster

                                # If the deleted cluster was older take into account the shift of keys in the list
                                if other_cluster < current_cluster:
                                    current_cluster -= 1
                                break

                else:  # If the node hasn't been found...

                    # Set the second node as discovered at add it to the same cluster as the first one
                    nodes_found.add(target_node)
                    clusters[current_cluster].add(target_node)

                n_paths = len(paths)  # Number of paths of equal length found

                for path in paths:  # For each path...
                    last_node = path.pop()  # Get the last node of the path between the two nodes

                    while path:  # For each edge of the path...
                        previous_node = path.pop()  # Get the previous node
                        edge_name = (previous_node, last_node)  # Compute the name of the edge between them

                        # Compute the new edge betweenness
                        if edge_name in edge_betweenness:
                            edge_betweenness[edge_name] += 1 / n_paths
                        else:
                            edge_betweenness[edge_name] = 1 / n_paths

                        last_node = previous_node

    n_clusters = len(clusters)  # Count the number of clusters
    print(n_clusters, "clusters found")

    # Delete as many edges as there are clusters to create
    for i in range(n - n_clusters):

        # Get the edge with the highest betweenness
        highest_betweenness = max(edge_betweenness, key=edge_betweenness.get)

        print("Deleting the edge between ", highest_betweenness[0], " and ", highest_betweenness[1],
              " (Betweenness: ", edge_betweenness[highest_betweenness], ")", sep='')

        del edge_betweenness[highest_betweenness]  # Delete it from the list of edges betweenness

        # Deleting the edge by deleting its name from the neighbors_out list
```

```
                of the start node
            del DIJKSTRA.graph[highest_betweenness[0]].neighbors_out[highest_betweenness[1]]
            """
            Doing so should delete de facto the edge from the graph.
            A ghost edge will still be listed in neighbors_in but it shouldn't be used by the program
            """

    # Displaying the clusters created
    for i in range(len(clusters)):
        size = len(clusters[i])
        if size > 1 or display_all:
            print("Cluster ", i + 1, ": size: ", size, " (", round(100 * len(clusters[i]) / n_nodes), "%), content: ",
                  sorted(clusters[i]), sep='')
```