

PROJECT Advanced Algorithmic and Programming

Cyprien BARIANT 10558
Tanguy BERTHOUD 60989
Thibault DU BUISSON DE COURSON 10496

January 12, 2021

Table of Contents

1	Creating a graph from GTFS data	2
1.1	Importing relevant GTFS data	2
1.2	Creating a graph	2
1.3	Results	3
2	Finding the shortest paths	4
2.1	Common interface	4
2.2	Pathfinding methods	4
2.3	Results	5
3	Finding clusters	5
	Appendix	6
A	gtfs.py	6
B	graph.py	8
C	pathfinding.py	10
D	clustering.py	12

List of Figures

1	Database representation of the relevant GTFS files	2
---	--	---

List of Tables

1	Execution time (ms) for the graph creation over several tries	3
2	Execution time (ms) for the pathfinding methods over several tries	5

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`.

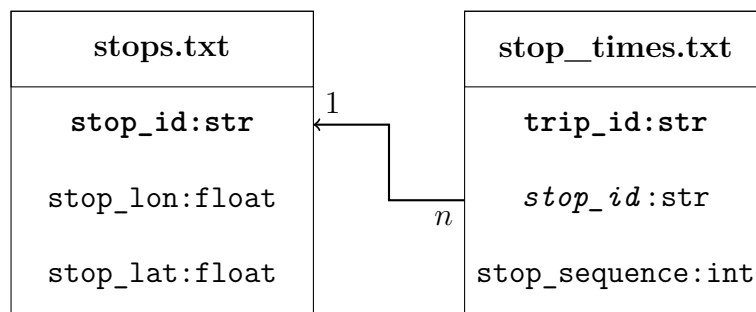


Figure 1 – Database representation of the relevant GTFS files

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` instantiation 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:

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

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

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

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`

```
1 from typing import Tuple
2 from sys import argv
3 from os import getcwd
4 from os.path import join
5 from time import perf_counter
6 from math import sqrt
7 from timing import timing
8 from pathfinding import *
9 from clustering import clustering
10
11 Position = Tuple[float, float]
12 class Stop:
13     """Stop representation
14
15     # Properties
16     - 'id' - Unique identifier
17     - 'position' - Position of the stop
18     """
19     def __init__(self, id: str, lat: float, lon: float):
20         self.__id: str = id
21         self.__position: Position = (lat, lon)
22     def __repr__(self) -> str:
23         return "{0} {1}".format(self.__id, self.__position)
24     def __lt__(self, other: 'Stop') -> bool:
25         return self.id < other.id
26     def __eq__(self, other: 'Stop') -> bool:
27         return self.id == other.id and self.position == other.position
28     def __hash__(self) -> int:
29         return hash((self.__id, self.__position))
30     @property
31     def id(self) -> str:
32         return self.__id
33     @property
34     def position(self) -> Position:
35         return self.__position
36
37 def import_stops(path: str) -> Tuple[List[Stop], Dict[str, int]]:
38     """Import stops from GTFS 'stops.txt'
39
40     # Arguments
41     - 'path' - Path to the file
42
43     # Return value
```

```

44 Tuple '(stops, id_map)' where 'stops' is a list of 'Stop' instances,
45 and 'id_map' is a dictionary 'stop.id => node_id' where 'node_id' is the ↵
    index of the node in 'stops'
46 """
47 stops: List[Stop] = []
48 id_map: Dict[str, int] = dict()
49 with open(path, "rt") as file:
50     for i, line in enumerate(file):
51         if i > 0:
52             data = line.strip().split(",")
53             if data[10] == "TE" and int(data[8]) < 2 and len(data[9]) == 0: # ↵
                Filter stops
54                 stop = Stop(data[0], float(data[4]), float(data[5]))
55                 stops.append(stop)
56                 id_map[stop.id] = len(stops) - 1
57 return stops, id_map
58 def import_edges(path: str, id_map: Dict[str, int]) -> Set[Tuple[int, int]]:
59     """Import edges from GTFS 'stop_times.txt'
60
61     # Arguments
62     - 'path' - Path to the file
63     - 'id_map' - 'id_map' returned from 'import_stops'
64
65     # Return value
66     Set of ordered tuples of stop IDs
67     """
68     trips: Dict[str, List[Tuple[int, int]]] = dict()
69     # Import raw data
70     with open(path, "rt") as file:
71         for i, line in enumerate(file):
72             if i > 0:
73                 data = line.strip().split(",")
74                 if data[3] in id_map:
75                     if data[0] not in trips:
76                         trips[data[0]] = []
77                     heappush(trips[data[0]], (int(data[4]), id_map[data[3]]))
78     # Transform data
79     edges: Set[Tuple[int, int]] = set()
80     for trip in trips.values():
81         while len(trip) > 1:
82             edges.add((heappop(trip)[1], trip[0][1]))
83     return edges
84
85 if __name__ == "__main__":
86     # Get data path
87     """The path to the data files can be set using a script argument.
88
89     For example, if you execute Python from the workspace root, you can enter: '↵
        python src/gtfs.py ./data/'.
90     Or, if you execute Python from the 'src/' directory: 'python gtfs.py ../data↵
        /'.
91     """
92     DATAPATH = argv[1] if len(argv) > 1 else getcwd()
93     # Import data
94     (stops, id_map), exetime = timing(import_stops)(join(DATAPATH, "stops.txt"))
95     print("Imported {0} stops in {1}ms".format(len(stops), exetime * 1e3))

```



```

96 edges, exetime = timing(import_edges)(join(DATAPATH, "stop_times.txt"), id_map←
    )
97 print("Imported {0} edges in {1}ms".format(len(edges), exetime * 1e3))
98 # Construct graph
99 exetime = perf_counter()
100 GRAPH = Graph(stops, compute_weight=lambda u, v: sqrt(
101     (v.position[0] - u.position[0]) ** 2 + (v.position[1] - u.position[1]) ** ←
        2))
102 for start, end in edges:
103     GRAPH.add_edge(start, end)
104 print("Constructed graph in {0}ms".format((perf_counter() - exetime) * 1e3))
105 # Construct pathfinders
106 BFS = Pathfinder(GRAPH, bfs)
107 DIJKSTRA = Pathfinder(GRAPH, dijkstra)
108 # Create clustering
109 clustering(DIJKSTRA, set(id_map.values()), 11)

```

B graph.py

```

1 from typing import Callable, Dict, Generic, Iterable, Iterator, List, TypeVar
2
3 T = TypeVar("T")
4 class Node(Generic[T]):
5     """Graph node representation
6
7     # Generic
8     - 'T' - Type of the node value
9
10    # Properties
11    - 'value' - Value of the node
12    - 'neighbors_out' - Dictionnary 'node => weight' of outward neighbors
13    - 'neighbors_in' - Dictionnary 'node => weight' of inward neighbors
14    """
15    def __init__(self, value: T):
16        self.__value = value
17        self._neighbors_out: Dict[int, float] = dict()
18        self._neighbors_in: Dict[int, float] = dict()
19    def __repr__(self) -> str:
20        return repr(self.__value)
21    def __lt__(self, other: 'Node') -> bool:
22        return self.value < other.value
23    def __eq__(self, other: 'Node') -> bool:
24        return self.value == other.value
25    def __hash__(self) -> int:
26        return hash(self.__value)
27    @property
28    def value(self) -> T:
29        return self.__value
30    @property
31    def neighbors_out(self) -> Dict[int, float]:
32        return self._neighbors_out
33    @property
34    def neighbors_in(self) -> Dict[int, float]:
35        return self._neighbors_in

```

```

36 class Graph(Generic[T]):
37     """Graph (weighted directed) representation
38
39     # Generic
40     - 'T' - Type of the nodes
41
42     # Properties
43     - 'nodes' - List of nodes
44     - 'size' - Size of the graph
45     - 'compute_weight' - Function to compute edge weight from two nodes
46     """
47     def __init__(self, nodes: Iterable[T], compute_weight: Callable[[T, T], float] ←
         = None):
48         self.__nodes: List[Node[T]] = []
49         for node in nodes:
50             self.add_node(node)
51         self.__size: int = 0
52         self.__compute_weight = compute_weight
53     def __iter__(self) -> Iterator[Node[T]]:
54         return iter(self.__nodes)
55     def __getitem__(self, key: int) -> Node[T]:
56         return self.__nodes[key]
57     @property
58     def order(self) -> int:
59         return len(self.__nodes)
60     @property
61     def size(self) -> int:
62         return self.__size
63     def add_node(self, node: T):
64         """Add a node to the graph
65
66         # Arguments
67         - 'node' - Node value to add
68         """
69         self.__nodes.append(Node(node))
70     def add_edge(self, start: int, end: int):
71         """Add an edge 'u-(weight)->v' to the graph
72
73         Will compute the weight using the 'compute_weight' property.
74         If 'compute_weight' is 'None', the weight will be 0.
75
76         # Arguments
77         - 'start' - Key of the first node
78         - 'end' - Key of the second node
79
80         # Errors thrown
81         - 'ValueError' if both keys are equal
82         """
83         if start == end:
84             raise ValueError("start={0} and end={0} are equal".format(start, end))
85         else:
86             weight = 0 if self.__compute_weight is None else self.__compute_weight ←
                 (self.__nodes[start].value, self.__nodes[end].value)
87             self.__nodes[start].neighbors_out[end] = weight
88             self.__nodes[end].neighbors_in[start] = weight
89             self.__size += 1

```

C pathfinding.py

```
1 from typing import Callable, Dict, Generic, List, Set, Tuple, TypeVar
2 from math import inf
3 from heapq import heapify, heappop, heappush
4 from graph import Graph
5
6 T = TypeVar("T")
7 class Pathfinder(Generic[T]):
8     """Wrapping class to allow pathfinding in graphs
9
10    # Generic
11    - 'T' - Type of the graph nodes
12
13    # Properties
14    - 'graph' - Graph used to compute pathfinding
15    - 'previous' - Dictionary 'from => to => previous' where 'previous' is a list↵
16      of the previous nodes of 'to' when searching from 'from'
17    - 'distance' - Dictionary 'from => to => distance'
18    - 'method' - Function to compute shortest paths from a node
19    """
20    def __init__(self, graph: Graph[T], method: Callable[['Pathfinder[T]', int], ↵
21      None]):
22        self.graph = graph
23        self._previous: Dict[int, Dict[int, Set[int]]] = dict()
24        self._distance: Dict[int, Dict[int, float]] = dict()
25        self._method = method
26    def reset(self):
27        """Reset the pathfinding results"""
28        Pathfinder.__init__(self, self.graph, self._method)
29    def compute(self, start: int):
30        """Execute the pathfinding method from a certain node
31
32        Save the newly computed data to the save file
33
34        # Arguments
35        - 'start' - Key of the starting node
36        """
37        if start not in self._previous:
38            self._method(self, start)
39    def has_path(self, start: int, end: int) -> bool:
40        """Check if a path exist between two nodes
41
42        # Arguments
43        - 'start' - Key of the starting node
44        - 'end' - Key of the ending node
45
46        # Errors thrown
47        - 'ValueError' if both keys are equal
48
49        # Return value
50        'True' if a path exists; 'False' otherwise
51        """
52        if start == end:
53            raise ValueError("start={0} and end={0} are equal".format(start, end))
54        else:
```

```

53         if start not in self._previous:
54             self.compute(start)
55         return end in self._previous[start]
56 def get_paths(self, start: int, end: int) -> List[List[int]]:
57     """Get the shortest path between two nodes
58
59     # Arguments
60     - 'start' - Key of the starting node
61     - 'end' - Key of the ending node
62
63     # Errors thrown
64     - 'ValueError' if both keys are equal
65
66     # Return value
67     List of paths, with a path being a list of node indexes
68     """
69     if start == end:
70         raise ValueError("start={0} and end={0} are equal".format(start, end))
71     else:
72         if start not in self._previous:
73             self.compute(start)
74         paths: List[List[int]] = []
75         def __recurse(path: List[int], pos: int):
76             if path[pos] == start:
77                 paths.append(path[:pos + 1][::-1])
78             else:
79                 for previous in self._previous[start][path[pos]]:
80                     if len(path) < pos + 2:
81                         path.append(previous)
82                     else:
83                         path[pos + 1] = previous
84                         __recurse(path, pos + 1)
85         if self.has_path(start, end):
86             __recurse([end], 0)
87         return paths
88 def get_distance(self, start: int, end: int) -> float:
89     """Get the distance between two nodes
90
91     # Arguments
92     - 'start' - Key of the starting node
93     - 'end' - Key of the ending node
94
95     # Return value
96     Distance from 'start' to 'end', in edges
97     """
98     if start not in self._previous:
99         self.compute(start)
100     return self._distance[start][end] if end in self._distance[start] else inf
101
102 def bfs(self: Pathfinder[T], start: int):
103     """Breadth-First Search method for 'Pathfinder'"""
104     if start not in self._previous:
105         self._previous[start] = dict()
106         self._distance[start] = dict()
107         self._distance[start][start] = 0
108         queue = [start]

```

```

109         heapify(queue)
110         while len(queue) > 0:
111             current = heappop(queue)
112             for u in self.graph[current].neighbors_out:
113                 if u not in self._distance[start]:
114                     self._previous[start][u] = set()
115                     self._distance[start][u] = self._distance[start][current] + 1
116                     heappush(queue, u)
117                     if self._distance[start][u] == self._distance[start][current] + 1:
118                         self._previous[start][u].add(current)
119 def dijkstra(self: Pathfinder[T], start: int):
120     """Dijkstra method for 'Pathfinder'"""
121     if start not in self._previous:
122         self._previous[start] = dict()
123         self._distance[start] = dict()
124         self._distance[start][start] = 0
125         marked: Set[int] = set()
126         queue: List[Tuple[float, int]] = [(0, start)]
127         heapify(queue)
128         while len(queue) > 0:
129             _, current = heappop(queue)
130             marked.add(current)
131             for (u, weight) in self.graph[current].neighbors_out.items():
132                 tentative_distance = self._distance[start][current] + weight
133                 if u not in self._distance[start] or tentative_distance < self._distance[start][u]:
134                     self._previous[start][u] = set()
135                     self._distance[start][u] = tentative_distance
136                     if self._distance[start][u] == tentative_distance:
137                         self._previous[start][u].add(current)
138                     if u not in marked:
139                         heappush(queue, (self._distance[start][u], u))

```

D clustering.py

```

1 from pathfinding import Pathfinder
2
3 def clustering(DIJKSTRA: Pathfinder, nodes, n):
4     """
5     Clustering method (first try)
6     """
7     print("\nCreating", n, "clusters...")
8     clusters = [] # Content of the clusters
9     n_nodes = len(nodes) # Number of nodes
10    iteration = 0
11    while len(clusters) < n: # As long as the right number of clusters has not ←
12        been constituted
13        iteration += 1
14        print("\nIteration", iteration)
15        edge_betweenness = {} # Betweenness of each edge
16        DIJKSTRA.reset()
17        nodes_found = set() # List of discovered nodes
18        nodes_to_explore = nodes.copy() # List of nodes to explore
19        progress = 0

```

```

19 # clusters = []
20 current_cluster = -1
21 while nodes_to_explore: # As long as there is still unexplored nodes
22     # Displaying the progress
23     new_progress = int(round(100 * (1 - len(nodes_to_explore) / n_nodes), ←
24                             -1))
25     if new_progress > progress:
26         progress = new_progress
27         print(progress, "%", sep='')
28     starting_node = nodes_to_explore.pop() # Take a node to explore
29     # Update the list of clusters
30     if starting_node in nodes_found: # If the node has already been ←
31         discovered...
32         # Search the node in every cluster list
33         for i in range(len(clusters)):
34             if starting_node in clusters[i]:
35                 clusters[i].add(starting_node)
36                 current_cluster = i
37     else: # If the node hasn't been discovered report the discovery of a ←
38         new cluster
39         clusters.append({starting_node}) # Create a new entry for the new ←
40         cluster containing its first node
41         current_cluster = len(clusters) - 1
42     nodes_found.add(starting_node) # Add the node to the discovered nodes ←
43     list
44     for target_node in nodes_to_explore: # Search every other node
45         # (paths), exetime = timing(DIJKSTRA.get_paths)(starting_node, ←
46         target_node)
47         # print("Dijkstra of {0} in {1}ms".format(starting_node, exetime * ←
48         1e3))
49         paths = DIJKSTRA.get_paths(starting_node, target_node) # Search a ←
50         path between the two nodes
51         if len(paths) > 0: # If there is a path...
52             # Generating list of contents of each cluster
53             if target_node in nodes_found: # If the node has already been ←
54                 found
55                 # Search the node in every cluster list
56                 for other_cluster in range(len(clusters)):
57                     if target_node in clusters[other_cluster]:
58                         if current_cluster == other_cluster:
59                             clusters[current_cluster].add(target_node)
60                         else:
61                             clusters[current_cluster].update(clusters[←
62                             other_cluster])
63                             del clusters[other_cluster]
64                             # If the deleted cluster was older take into ←
65                             account the shift of values in the list
66                             if other_cluster < current_cluster:
67                                 current_cluster -= 1
68                             break
69             else:
70                 # Set the second node as discovered at add it to the same ←
71                 cluster as the first one
72                 nodes_found.add(target_node)
73                 clusters[current_cluster].add(target_node)
74     n_paths = len(paths) # Number of paths of equal length found

```

```

63         for path in paths: # For each path
64             last_node = path.pop() # Get the last node of the path ←
                                     between the two nodes
65         while path: # For each edge of the path...
66             previous_node = path.pop() # Get the previous node
67             edge_name = (previous_node, last_node) # Compute the ←
                                     name of the edge between them
68             # Compute the new edge betweenness
69             if edge_name in edge_betweenness:
70                 edge_betweenness[edge_name] += 1 / n_paths
71             else:
72                 edge_betweenness[edge_name] = 1 / n_paths
73             last_node = previous_node
74     n_clusters = len(clusters)
75     print(n_clusters, "clusters found")
76     # Delete as many edges as there are clusters to create
77     for i in range(n - n_clusters):
78         highest_betweenness = max(edge_betweenness, key=edge_betweenness.get) ←
                                     # Get the edge with the highest betweenness
79         print("Deleting the edge between ", highest_betweenness[0], " and ", ←
               highest_betweenness[1], " (Betweenness: ", edge_betweenness[←
               highest_betweenness], ") ", sep='')
80         del edge_betweenness[highest_betweenness] # Delete it from the list ←
               of edges betweenness
81         # Deleting the edge by deleting its name from the neighbors_out list ←
               of the start node
82         del DIJKSTRA.graph[highest_betweenness[0]].neighbors_out[←
               highest_betweenness[1]]
83         """
84         Doing so should delete de facto the edge from the graph.
85         A ghost edge will still be listed in neighbors_in but it shouldn't be ←
               used by the program
86         """
87     print("\n", n, "clusters obtained:")
88     for i in range(len(clusters)):
89         print("Cluster #", i + 1, " found, size: ", len(clusters[i]), " (", round←
               (100 * len(clusters[i]) / n_nodes), "%) : ", sorted(clusters[i]), sep='←
               ')

```