CS162 Solo Project Weekly Report: Week 07

CS162, Intro. to Computer Science II
Semester 2, AY2023/24

Solo Project

Full Name	Student ID
NGUYEN Hoang The Kiet	23125023

Lecturer: DINH Ba Tien (PhD)
TAs: HO Tuan Thanh (MSc), NGUYEN Le Hoang Dung (MSc)

Ho Chi Minh City, April 7, 2024

Tasks by Week

Week 07

1. (Chapter 2) Bus Network Graph: Construction Stage
2. (Chapter 1) Dijkstra: Shortest Paths and related Centrality Measures
3. (Section 1.2) APSP: All Pairs Shortest Path problem
4. (Section 1.3) Betweenness Centrality, (Section 3.2) `NetworkAnalysisBetweenness` class

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Dijkstra: Shortest Paths and related Centrality Measures

1.1 | SSSP: Single Source Shortest Path problem

ⓐ | Formulation

Given a graph $G = (V, E, w)$, where

- V is the set of nodes
- $E \subseteq V \times V$ is the set of **directed** edges
- $w : E \mapsto \mathbb{R}$ is the weight function of an edge.[a]

A path from s to t is a sequence of nodes $P := s = u_0, u_1, \cdots, u_{k-1}, u_k = t$, where

- $u_i \in V, \forall 0 \leq i \leq k$
- Every pair of consecutive nodes identify an edge in E, that is, $e_i := (u_i, u_{i+1}) \in E, \forall 0 \leq i < k$

Find the shortest path in the graph G, that is, to find a valid path that minimises

$$\delta(u, v) = \sum_{i=0}^{k-1} w(e_i) = \sum_{i=0}^{k-1} w(u_i, u_{i+1})$$

[a] Let $e = (u, v)$. Sometimes we also write $w(e) = w(u, v)$, considering w as a function of two nodes. If the edge (u, v) does not exists, then $w(u, v) := \infty$

In this problem, our interest is on non-negative weighted graphs, that is, $w(e_i) \geq 0, \forall e_i \in E$

ⓑ | Dijkstra's Algorithm

In his original paper *"A Note on Two Problems in Connexion with Graphs" (1959)*[1], Dijkstra explains the tuition of his algorithm as follows,

> ***Problem 2***[1]***. Find the path of minimum total length between two given nodes P and Q.***

[1] The first problem was to find the minimum weight spanning tree, for which Dijkstra, in his paper, had rediscovered Prim's algorithm

> *We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to other nodes are constructed in order of increasing length until Q is reached.*

The original $O(|V|^2)$ algorithm

We can easily reach an $O(|V|^2)$ algorithm based only on the original intuition of Dijkstra. Let S be a set of nodes to be considered. We excessively extract the node with minimum distance from the source, then use that source to update the currently-found minimal path for other adjacent nodes.

In each iteration, we need $O(|V|)$ time to find the minimum distant node in the set[2]. With V iterations, the complexity of the algorithm is $O(|V|^2)$

```python
def dijkstra(V, E, w, src):
    """
    Run Dijkstra's Algorithm on graph G = (V, E, w) from src
    """

    S = set()
    for u in V:
        dist[u] = INFINITY
        prev[u] = None
        S.add(u)

    dist[src] = 0
    while len(S) > 0:
        _, u = min((dist[u], u) for u in S)
        S.remove(u)

        for v in neighbours(u):
            if v not in S:
                continue

            if dist[v] > dist[u] + w(u, v)
                dist[v] = dist[u] + w(u, v)
                prev[v] = u

    return dist, prev
```

Listing 1.1: Dijkstra: The originally proposed algorithm

[2] In Python, `set` is a hash-table, hence `find-min` is $O(n)$. In contrast, C++'s `std::set` is a red-black BST which supports `find-min` in $O(\log n)$

```python
from priority_queue import PriorityQueue

def dijkstra(V, E, w, src):
    """
    Run Dijkstra's Algorithm on graph G = (V, E, w) from src
    """

    for u in V:
        dist[u] = INFINITY
        prev[u] = None
        S.add(u)

    dist[src] = 0
    pq = PriorityQueue()
    pq.put((0, src))

    while not pq.empty():
        _, u = pq.get()

        for v in neighbours(u):
            if v not in S:
                continue

            if dist[v] > dist[u] + w(u, v)
                dist[v] = dist[u] + w(u, v)
                prev[v] = u
                pq.put((dist[v], v))

    return dist, prev
```

Listing 1.2: Dijkstra: Improvements with `PriorityQueue`

The improved $O(|E| \log |V|)$ algorithm

We can improve the algorithm by changing the data structure that stores the set S. Using a priority queue implemented by a binary heap leads to a complexity of $O((|E|+|V|) \log |V|)$.

Better data structures like Fibonacci heap can improve the complexity to $O(|E| + |V| \log |V|)$. However, due to the complex implementation of Fibonacci heaps and its high constant factors, binary heaps are usually perferred.

In this report we assume that the $O(|E| \log |V|)$ version of Dijkstra's Algorithm is used.

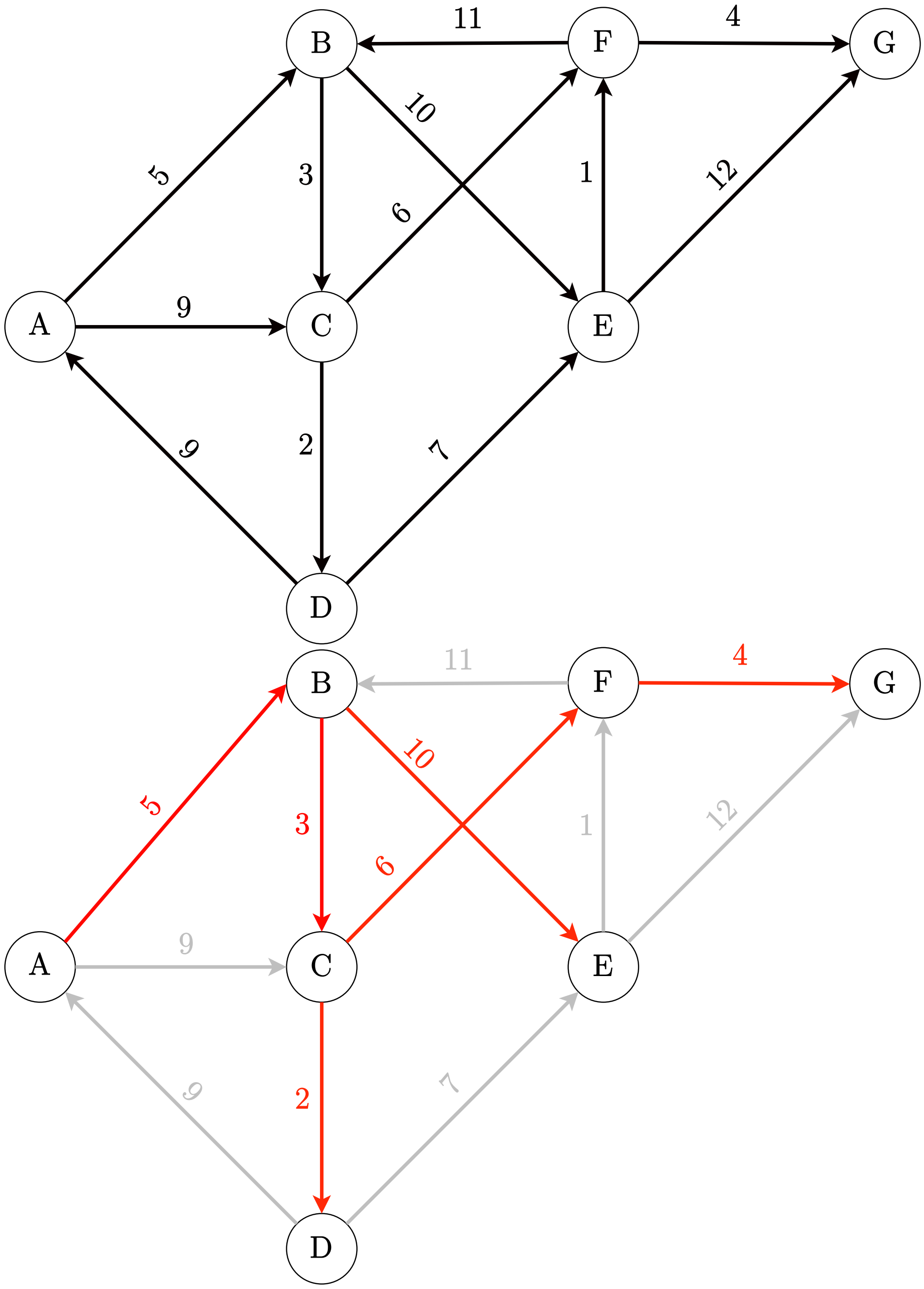


Figure 1.1: Dijkstra: Shortest paths from source node A (red paths) generate the **shortest-path tree**.

(c) | Shortest-path Tree

Assume a graph with real, positive weights where for every two nodes s, t, there exists a single shortest path. Then all shortest paths starting from a source s forms a tree T. As in Dijkstra: Improvements with `PriorityQueue`, we can recover the tree using the `prev` labels:

$$(u, v) \in T \iff \text{prev}(v) = u \iff \delta(s, u) + w(u, v) = \delta(s, v)$$

The shortest-path tree has many applications in centrality measures, a characterisation of the "importance" of a node. One of which that makes it into this report is the Betweenness Centrality.

Note: Generally, the graph created by removing "redundant" edges (edges that when removed do not change any shortest path) is a Directed Acyclic Graph (DAG). We are assuming there is a single shortest path from any given pair of nodes (s, t), hence the resulting graph is a tree, a special version of a DAG.

1.2 | APSP: All Pairs Shortest Path problem

(a) | Formulation

Given a graph $G = (V, E, w)$. Find the shortest path between all pairs of nodes, that is, to find $\delta(u, v)$ for all $(u, v) \in V^2$.

(b) | Dijkstra vs. alternatives

Dijkstra, $O(|V||E| \log |V|)$

Dijkstra is advantegous for sparse graph, where $|E| \ll |V|^2$.

Floyd-Warshall, $O(|V|^3)$

When the graph is dense, i.e., $|E| = O(|V|^2)$, then Floyd-Warshall's algorithm in $O(|V|^3)$ outperforms Dijkstra's in $O(|V|^3 \log |V|)$.

```python
import itertools as iters

def floyd_warshall(V, E, w):
    dist = {(i, j): INFINITY for i, j in iters.product(V) }
    for k in V:
        for i, j in iters.product(V):
            if dist[(i, j)] > dist[(i, k)] + dist[(k, j)]:
                dist[(i, j)] = dist[(i, k)] + dist[(k, j)]
    return dist
```

Listing 1.3: Dijkstra: Floyd-Warshall Algorithm for APSP

1.3 | Betweenness Centrality

Betweenness centrality is a measure of centrality in a graph based on shortest paths, formally defined first in 1977 by Freeman L. C. in his article in the journal *Sociometry*, *A Set of Measures of Centrality Based on Betweenness*[2]. We will be using a simpler variant of this metric: By assuming that the graph is real and stochastically valued, there is only a single shortest path between every pair of nodes.

ⓐ | Formulation of the Betweenness centrality

Given a graph $G = (V, E, w)$. The Betweenness centrality of a node is defined as the number of shortest paths passing that node. Formally, let $\delta(s, t)$ be the shortest path from s to t, then

$$g(u) = \sum_{(s,t)\in V^2} \frac{\text{\# of shortest paths } s \to u \to t}{\text{\# of shortest paths } s \to t}$$

Assuming all weights are real numbers with high precision which implies that there is a single shortest path between any two nodes, we can redefined the Betweenness Centrality as such:

$$g(u) = \sum_{(s,t)\in V^2} \text{\# of shortest paths } s \to u \to t$$

ⓑ | The naive $O(|V|^2|E|\log|V|)$ algorithm

This version assumes a $O(|E|\log|V|)$ implementation of Dijkstra algorithm to find the shortest path between any two nodes s and t.

With N^2 pairs of nodes, the algorithm runs in total time of $O(|V|^2 \cdot |E|\log|V|)$.

```python
import itertools as iters

def shortest_path(s, t) -> list[int]:
    # ...

def betweenness_centralities(V, E, w):
    g = {u: 0 for u in V}
    for s, t in iters.product(V, V):
        for x in shortest_path(s, t):
            g[x] += 1
    return g
```

Listing 1.4: Dijkstra: Computing $g(u)$ by iterating across all shortest paths

ⓒ | An improvement to $O(|V|^2 + |V||E|\log|V|)$ using Shortest-path tree

Consider a shortest-path tree obtained by applying Dijkstra's Algorithm on source s. The betweenness centrality of a node u with respect to source s is defined as

$$g_s(u) = \sum_{t \in V} \#\text{ of shortest paths } s \to u \to t$$

Hence, on the shortest-path tree, s is the root node, and t must be a descendant of u. The number of nodes t satisfing the above condition is equal to the size of the subtree of u.

Finding the size of the subtree of every node is a simple Dynamic Programming problem: Let $g_s(u)$ = size of subtree of u, then we have the well known recursive formula:

$$\boxed{g_s(u) = 1 + \sum_{(u,v) \in E_T} g_s(v)}$$

where E_T is the list of **directed** edge on the shortest-path tree, i.e., edges (u, v) that satisfies $\delta(s, v) = \delta(s, u) + w(u, v)$.

The complexity is $O(|V|^2 + |V||E|\log|V|)$. Had we used the theoretical fastest version of Dijkstra, the complexity would only be $O(|V||E| + |V|^2 \log|V|)$, which is what proposed in the Brandes' algorithm[3].

ⓓ | Other Centrality Measures based on Shortest Paths

Closeness Centrality (Alex Bavelas, 1950)[4]

$$g(u) = \frac{|V| - 1}{\sum_{v \in V} \delta(u, v)}$$

Harmonic Centrality (Marchiori and Latora, 2000)[5]

$$g(u) = \sum_{v \in V - \{u\}} \frac{1}{\delta(u, v)}$$

(assuming $1/\infty = 0$)

This measure is later independently discovered by Dekker (2005)[6] under the name "valued centrality", and by Rochat (2009).

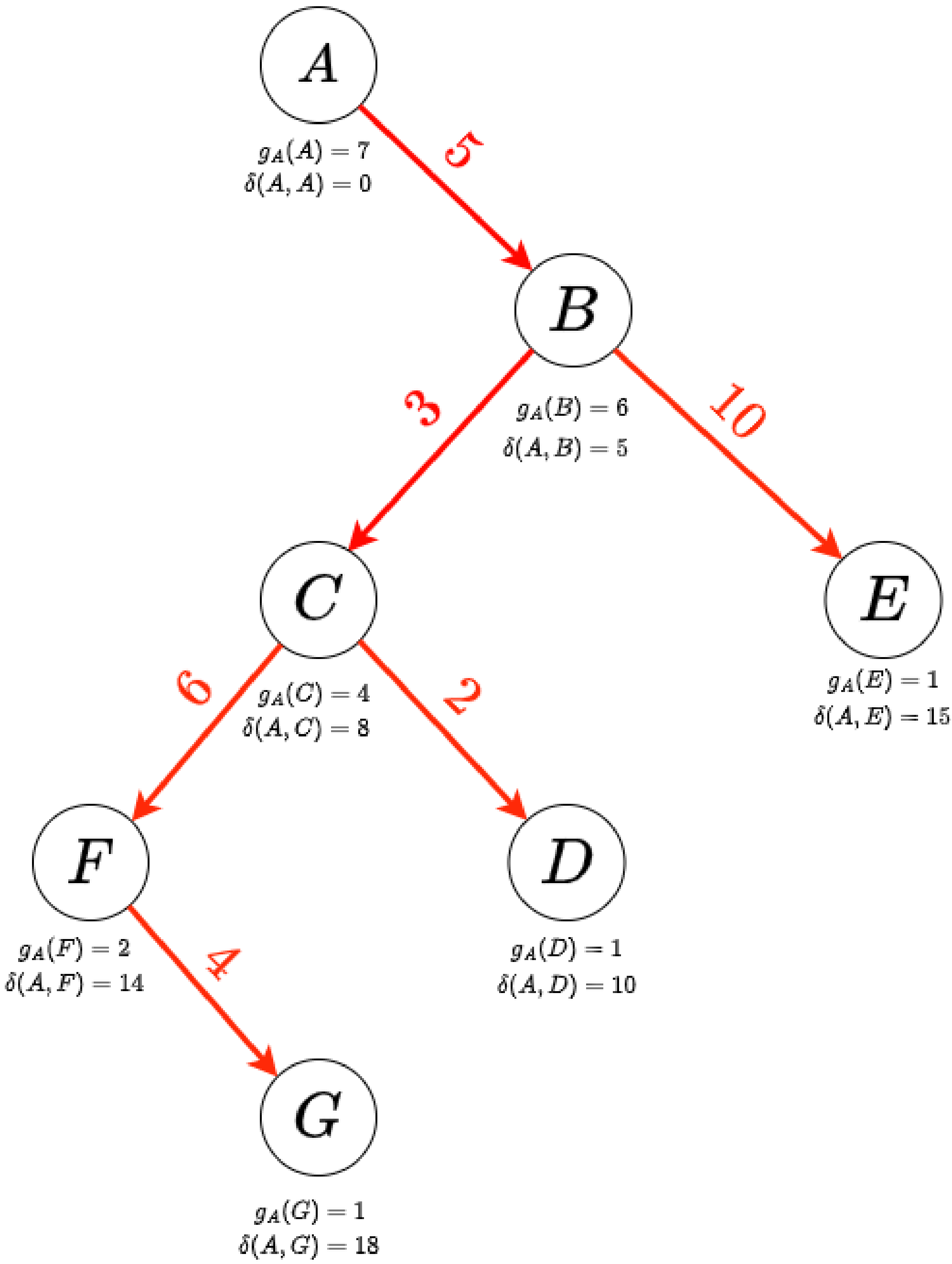


Figure 1.2: Dijkstra: Usage of the **shortest-path tree** to compute $g_s(u)$.

```python
from priority_queue import PriorityQueue

def dijkstra(V, E, w, src):
    """
    Run Dijkstra's Algorithm on graph G = (V, E, w) from src
    """

    for u in V:
        dist[u] = INFINITY
        prev[u] = None
        g[u] = 1
        S.add(u)

    dist[src] = 0
    pq = PriorityQueue()
    pq.put((0, src))

    while not pq.empty():
        _, u = pq.get()

        for v in neighbours(u):
            if dist[v] > dist[u] + w(u, v)
                dist[v] = dist[u] + w(u, v)
                prev[v] = u
                pq.put((dist[v], v))

    order = sorted((dist[u], u) for u in V, reverse=True)
    # Traverse tree in reverse topological order
    for _, u in order:
        g[prev[u]] += g[u]

    return dist, prev, g
```

Listing 1.5: Dijkstra: Computing $g_s(u)$

Chapter 2

Bus Network Graph: Construction Stage

We present key results in the construction and analysis of the bus graph. A deep analysis with code is presented in the ***`main.ipynb`*** *Jupyter notebook in this repository.*

2.1 | Definition

ⓐ | Bus Network

A bus network is a tuple of three sets:

- A set of *Stops*;
- A set of *(Route) Variants* containing *Stop*s in chronological order;
- A set of *Paths*, each of which describes the topological shape of a corresponding *(Route) Variant*.

ⓑ | Bus (Network) Graph

A bus network graph (or bus graph) is a graph, where

- Each node represents a bus *Stop*.
- Each **directed** edge represents a connection between two bus *Stops* along a bus *Variant*.

2.2 | Exploratory Data Analysis on real bus network

The bus network is read from three JSON files:

- `'stops.json'`, containing a set of *Stop*s, each stop has their own properties and belongs to multiple bus routes / variants.
- `'vars.json'`, containing a set of bus routes and variants, each record consists of a `RouteId`, `RouteVarId` and their perspective parameters (distance of travel, name, time of travel, route number, etc.)

- 'paths.json', describing the geometry of each route/variant in the form of a LineString in WGS-84 coordinate system.

(a) | *Stops*

StopType

StopType (in Vietnamese)	StopType (in English)	Count
Bến xe	*Station*	143
Nhà chờ	*Shelter*	628
Trụ dừng	*Stop*	2951
Ô sơn	*Marking sign*	603
Biển treo	*Overhead sign*	61
Trạm tạm	*Temporary*	7
-/-	Others	4
	Total	4397

Table 2.1: Bus Network Graph: Types of bus *Stops*

Degree distribution

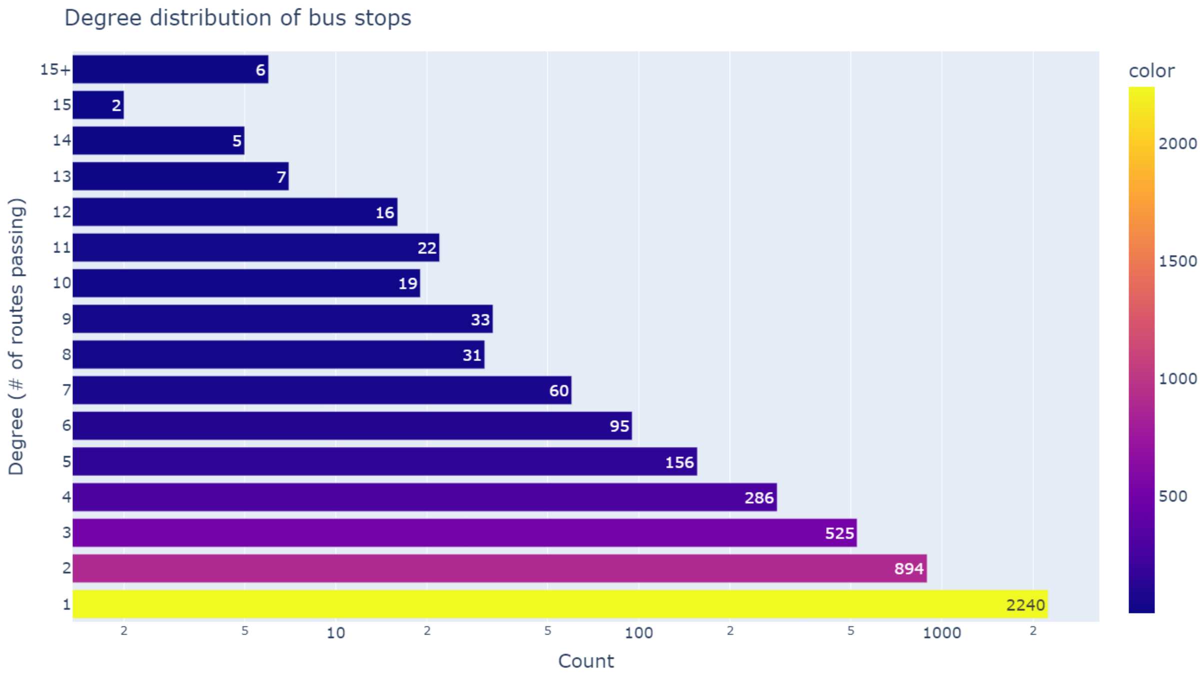


Figure 2.1: EDA: Degree distribution of bus *Stops* (total).

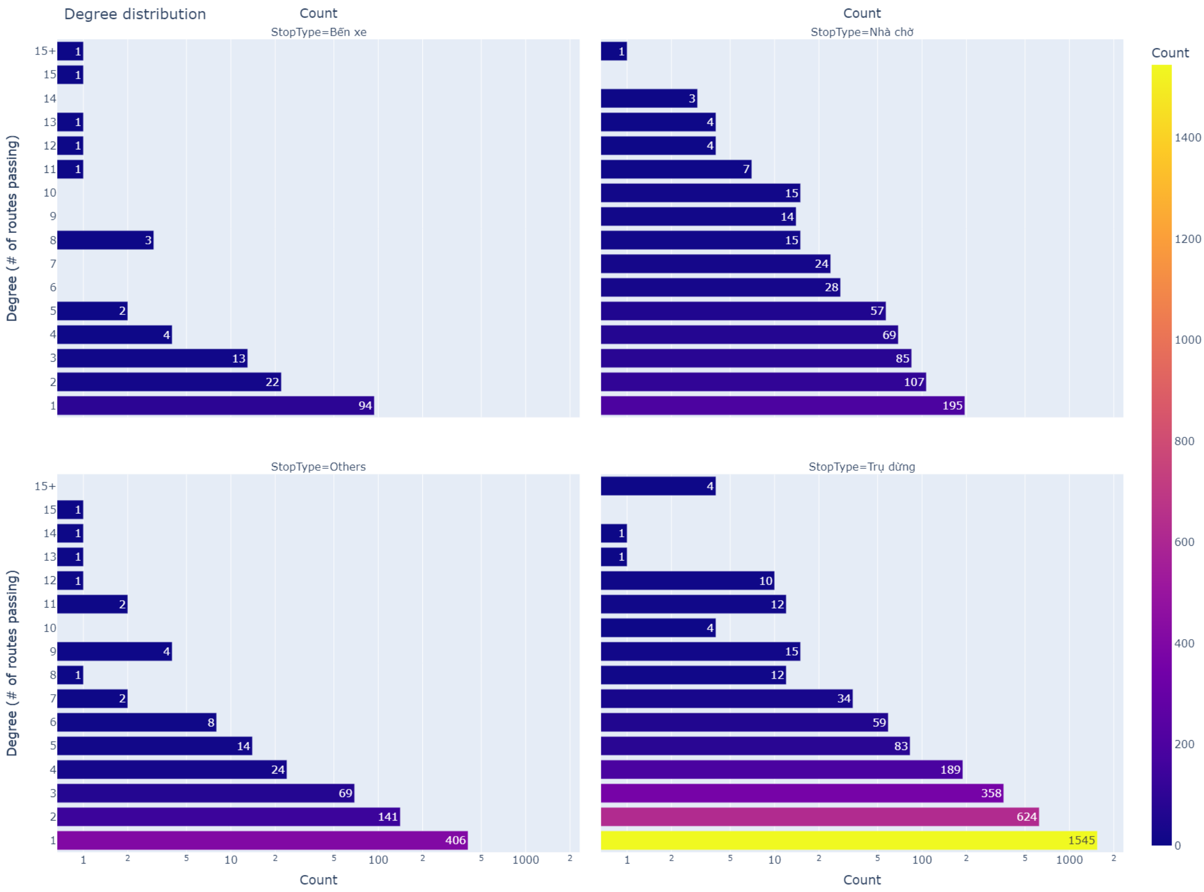


Figure 2.2: EDA: Degree distribution of bus *Stops* (by `StopType`)

ⓑ | *Routes*

Relationship between *Route* distance and running time

Figure 2.3: EDA: Relationship between *Route* distance and running time

- Trendline: $\boxed{\text{Distance} = 396.199 \cdot \text{RunningTime}}$
- Average speed: 23.68 km/hr (396.199 m/min.)
- R^2: 0.915583

ⓒ | Relationship between *Stops* and *Paths*

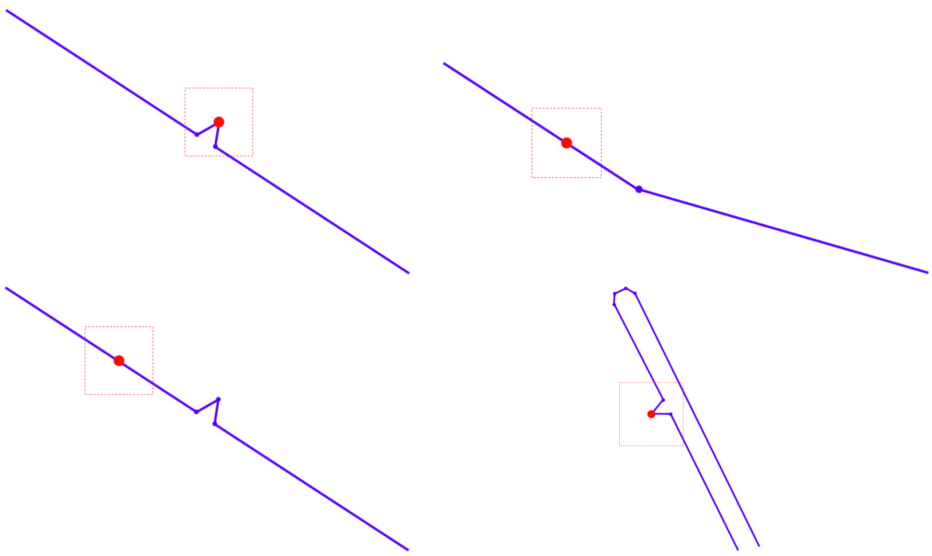


Figure 2.4: EDA: Case 1 - a *Stop* lies on a *Path*

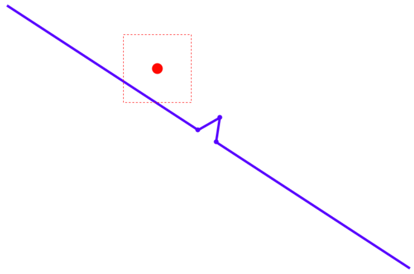


Figure 2.5: EDA: Case 2 - a *Stop* lies at a distant from the *Path*

2.3 | Graph construction algorithm

ⓐ | Fixing data errors

From EDA, we know that the data given to us suffer from measurement errors, specifically when some *Stops* do not lie on the path of the corresponding *Route*. This report will fix the errors based on two principles:

- The coordinates of any *Stop* are precise and **must be respected**.
- The shape of any *Path* are just an approximation and **can be modified** in a way that insignificantly changes the basic shape of the path and the order of the *Stop*s that it goes through.

With that, we propose two methods:

Method 1

Insert every stop into the path in the **"least disruptive way"**.

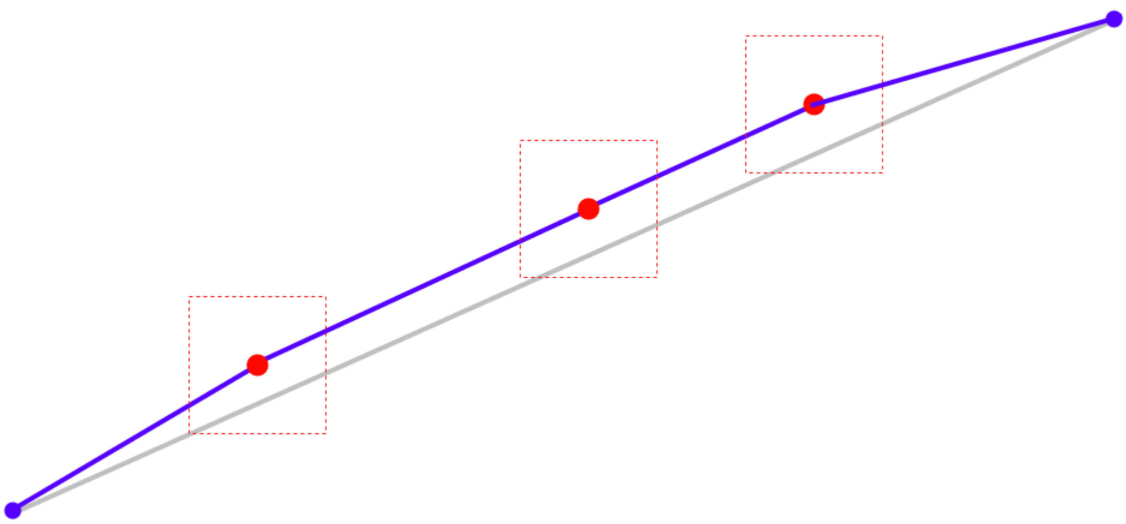


Figure 2.6: Graph Construction: Error fixing, Method 1

Method 2

Modify the path so that at every stop there is a **"ramp"** connecting every *Stop* with its nearest edge.

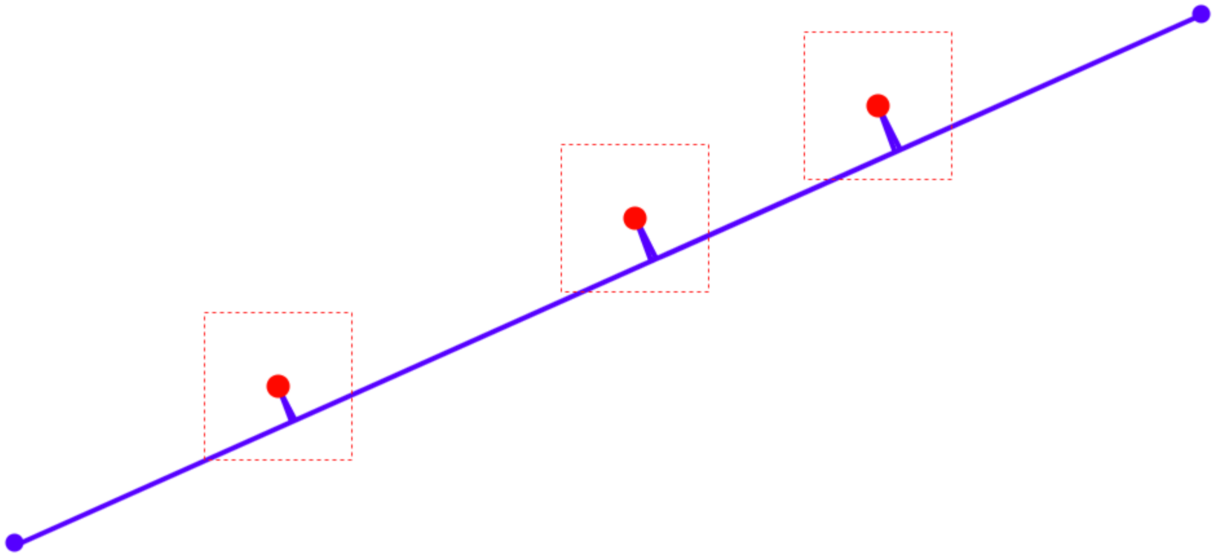


Figure 2.7: Graph Construction: Error fixing, Method 2

We will be using Method 2 in this report.

ⓑ | A simple algorithm

Graph construction algorithm I

For each *Variant* `(RouteId, RouteVarId)`

- Let $V = \mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n$ be the sequence of *Stops* in order from start to finish.
- Let $P = \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_m$ be the sequence of junctions that defines the shape of the *Path*. The i-th edge of the path is $e_i = (\mathbf{p}_{i-1}, \mathbf{p}_i)$ $(1 \le i \le m)$.
- For each *Stop* $\mathbf{v}_i$, find the **closest edge** to the *Stop*; in other words, find an index s_i such that $\text{dist}(\mathbf{v}_i, e_{s_i})$ is minimised. Then determine the point $\mathbf{r}_i$ on edge e_{s_i} that minimises $\text{dist}(\mathbf{v}_i, e_{s_i})$.
- For each pair of consecutive *Stops* $\mathbf{v}_i, \mathbf{v}_{i+1}$, define the edge $(\mathbf{v}_i, \mathbf{v}_{i+1})$ with the corresponding path $p(\mathbf{v}_i, \mathbf{v}_{i+1}) = \mathbf{v}_i, \mathbf{r}_i, \mathbf{p}_{s_i}, \mathbf{p}_{s_i+1}, \cdots, \mathbf{p}_{s_{i+1}-1}, \mathbf{r}_{i+1}, \mathbf{v}_{i+1}$. From that we can then find the corresponding length and time of the path.

The complexity of the above algorithm is $O((n+m) \cdot m)$ per variant. The extra m factor comes from the **closest edge finding** step which can be further optimised.

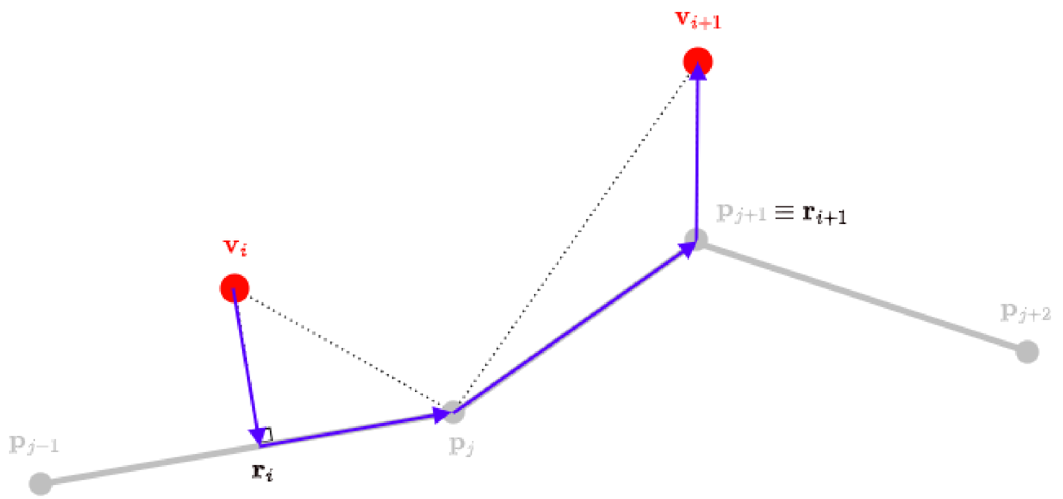
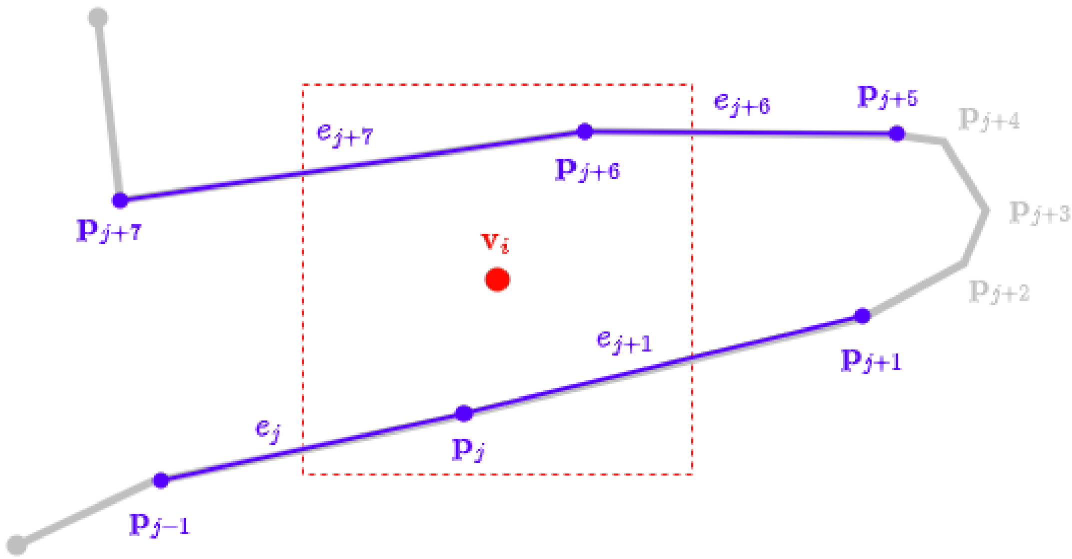


Figure 2.8: Graph Construction: Algorithm I

ⓒ | An optimisation using R-tree

Real life data shows that a bus variant should not be "near" a bus stop too many times. Hence, we can eliminate edges that are obviously irrelavant beforehand, and only check amongst edges that are "near" enough.

The prechecking part can be done using an R-tree: We insert every segment of the path into an R-tree, then query for segments that are "near" enough by performing an intersection check; the closest edge is guaranteed to be amongst the queried segments.



$$E = \{e_j, e_{j+1}, e_{j+6}, e_{j+7}\}$$

Figure 2.9: Graph Construction: Algorithm II (optimisation with R-tree)

Graph construction algorithm II

For each *Variant* `(RouteId, RouteVarId)`

- Let $V = \mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n$ be the sequence of *Stops* in order from start to finish.
- Let $P = \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_m$ be the sequence of junctions that defines the shape of the *Path*. The i-th edge of the path is $e_i = (\mathbf{p}_{i-1}, \mathbf{p}_i)$ $(1 \le i \le m)$.
- Let T be an `RTree.Index` tree.
- For each edge e_i, insert the minimum bounding rectangle (MBR) of e_i to T.
- Set $\varepsilon = 150$ (m).
- For each stop $\mathbf{v}_i$,
    - Let E be the set of edges whose MBR intersects the rectangle $\mathbf{v}_i \pm \mathbf{1}\varepsilon$
    - Find the **closest edge** to the *Stop* $\mathbf{v}_i$. Now we only have to perform checks in the list of candidates E.
    - With projection points $\mathbf{r}_i$, build the graph as the original algorithm follows.

The complexity is $O((n + m) \cdot (\log m + |E|))$ per variant in average. As stated, $|E|$ is usually very small in real data.

2.4 | Implementation

(a) | A generic `Network` class

`@dataclass NetworkConnector`

- ***property*** `src: int, dest: int`
  Source and destination of the edge.

- ***property*** `weight: int`
  Weight function of the edge.

`class Network[TNode, TConnector]`

- ***property*** `nodes: Dict[int, TNode]`
  A mapping from node IDs to set of `Nodes`.

- ***property*** `adjs: Dict[int, TConnector]`
  A mapping from node IDs to set of adjacent `NetworkConnectors` (adjacency list).

ⓑ | `BusNetwork` as an inheritance of generic `Network` class

`@dataclass BusNetworkConnector(NetworkConnector)`

- ***property*** `src: int, dest: int`
  Source and destination of the edge.
- ***property*** `time: float, length: float`
  Running time and length of the edge.
- ***property*** `weight: int`
  Weight function of the edge, defined as the running time of the edge.

`class BusNetwork`

- ```
  from_ndjsons(
      stops_json_file: str = 'stops.json',
      vars_json_file:  str = 'vars.json',
      paths_json_file: str = 'paths.json',
      sides_set_type:  str = 'spatial'
  ) -> BusNetwork
  ```

  Input the network from 3 JSON files describing a list of *Stop*s, *Variant*s and *Path*s.

  Parameters:

  - `stops_json_file`: JSON file describing a list of *Stop*s.
  - `vars_json_file`: JSON file describing a list of *Variant*s.
  - `paths_json_file`: JSON file describing a list of *Path*s.
  - `sides_set_type = 'default' | 'spatial'`:
    - If `sides_set_type = 'default'`: Construct the graph using the Graph construction algorithm I algorithm.
    - If `sides_set_type = 'spatial'`: Construct the graph using the Graph construction algorithm II algorithm (more optimised).
  ```

```python
@dataclass
class NetworkConnector():
    src:   int
    dest:  int

    @property
    def weight(self):
        raise NotImplementedError

@dataclass
class BusNetworkConnector(NetworkConnector):
    route_ids:  tuple[int, int]
    time:       float
    length:     float
    real_path:  list[tuple[float, float]]

    @property
    def weight(self) -> float:
        return self.time

    @classmethod
    def from_dict(cls, obj: dict) -> 'BusNetworkConnector':
        # ...

    def to_dict(self) -> dict:
        # ...
```

Listing 2.1: Graph Construction: `BusNetworkConnector` class inherited from generic `NetworkConnector`

```python
class Network(Generic[TNode, TNetworkConnector]):
    @property
    def nodes(self) -> Dict[int, TNode]:
        # A mapping from node IDs to set of Nodes.

    @property
    def adjs(self) -> Dict[int, TNetworkConnector]:
        # A mapping from node IDs to list of adjacent NetworkConnectors
        # (adjacency list).

class BusNetwork(Network[Stop, BusNetworkConnector]):
    @classmethod
    def from_ndjsons(self, sides_set_type: str = 'spatial', **kwargs)
        -> 'NetworkConnector':
        # Read the bus network from three json files

    @classmethod
    def from_dict(cls, obj: dict) -> 'NetworkConnector':
        # ...

    @classmethod
    def from_json(cls, file: str = 'net.json') -> 'NetworkConnector':
        # Read the bus network from a 'net.json' file

    def to_dict(self) -> dict:
        # ...

    def to_json(self, file: str):
        # ...
```

Listing 2.2: Graph Construction: `BusNetwork` class inherited from generic `Network`

2.5 | Experiments

ⓐ | Running time of construction algorithms

	Graph construction I		Graph construction II		Improvement
	sec	iterations/sec	sec	iterations/sec	%
Run #1	43.384115	6.845824	8.083512	36.741458	
Run #2	42.890710	6.924576	8.161453	36.390578	
Run #3	39.855744	7.451874	8.059147	36.852537	
Run #4	39.514839	7.516164	7.915563	37.521020	
Run #5	38.971134	7.621025	7.955797	37.331271	
Average	40.923309	7.271893	8.035094	36.967373	509.31%

Table 2.1: Bus Graph Experiments: Running time of construction algorithms

ⓑ | # of edge candidates $|E|$ per *Stop*

`Default`	Trendline	cand. $= 1.000 \cdot$ (#of sides)
	R^2	1.000
	Min. cand.	6.00
	Mean cand.	247.27
	Max. cand.	14.04
	std.	155.00
`Spatial`	Trendline	cand. $= 0.012 \cdot$ (#of sides) $+ 3.125$
	R^2	0.608
	Min. cand.	1.00
	Mean cand.	6.32
	Max. cand.	14.04
	std.	2.56

Table 2.2: Bus Graph Experiments: Stats on # of edge candidates $|E|$ per *Stop* on each *Route*

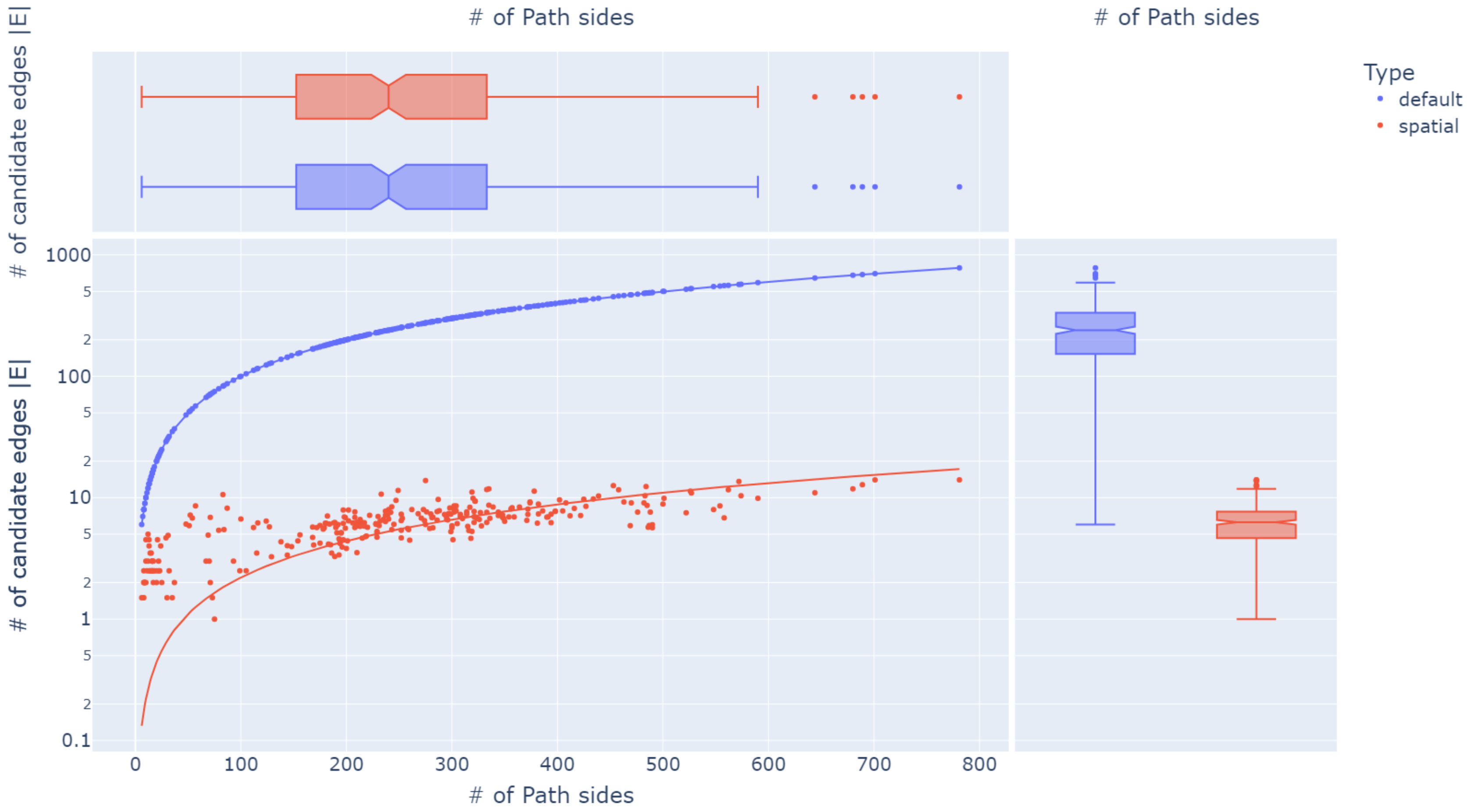


Figure 2.10: Bus Graph Experiments: # of edge candidates $|E|$ per *Stop* on each *Route*

Chapter 3

Bus Network Graph: Analysis Stage

*We present key results in the construction and analysis of the bus graph. A deep analysis with code is presented in the **main.ipynb** Jupyter notebook in this repository.*

3.1 | NetworkDijkstra, BusNetworkDijkstra

ⓐ | Generic NetworkDijkstra class

```
class NetworkDijkstra:
    def from_net(self, net: Network, src: int):
        # Run Dijkstra on network net and from source src

     def reverse_path_from(self, dest: int):
        while dest != self._src:
            connector = self.pars[dest]
            yield connector
            dest = connector.src

    def path_to(self, dest: int):
        return reversed(list(self.reverse_path_from(dest=dest)))
```

Listing 3.1: Graph Analysis: Generic `NetworkDijkstra` class

- `from_net(self, net: Network, src: int)`
  Run Dijkstra (using the implementation in Dijkstra: Computing $g_s(u)$) on network `net` and from source `src`.
- ***property*** `dists: list[float]`
  List of all shortest paths length from `src` to every node.
- ***property*** `pars: list[int]`
  Mapping from a node to its parent in the shortest-path tree.

- ***property*** `count_paths: list[int]`
  Mapping from a node to the number of descendants of that node in the shortest-path tree. Analogous to the $g_s(u)$ function.
- `reverse_path_from(self, dest: int)`
  Trace the shortest path from source to destination `dest` using the shortest-path tree stored in `pars` dictionary. Shortest path generated from the last edge to the first.
- `path_to(self, dest: int)`
  Trace the shortest path from source to destination `dest` using the shortest-path tree stored in `pars` dictionary. Shortest path generated from the first edge to the last.

ⓑ | BusNetworkDijkstra class

```python
class BusNetworkDijkstra(NetworkDijkstra):
    def path_to_json(self, dest: int, file: str):
        # Export the shortest path from in-state source to file

    def shortest_path_to_json(
        self,
        net: Network,
        src: int,
        dest: int,
        file: str
    ):
        # Run Dijkstra from given source,
        # then export the shortest path to file

        self.from_src(net=net, src=src)
        return self.path_to_json(dest=dest, file=file)
```

Listing 3.2: Graph Analysis: `BusNetworkDijkstra` class inherited from generic `NetworkDijkstra`

3.2 | NetworkAnalysisBetweenness class

```python
class NetworkAnalysisBetweenness:
    def _from_net_brute_force(
        self, net: Network,
        engine: NetworkDijkstra = None
    ):
        if engine is None:
            engine = NetworkDijkstra()

        raw_scores = { node_id: 0 for node_id in net.nodes }

        for src in tqdm(net.nodes):
            engine.from_src(src=src, net=net)
            for dest in net.nodes:
                for connector in engine.reverse_path_from(dest=dest):
                    raw_scores[connector.dest] += 1
                    if connector.src == src:
                        raw_scores[src] += 1

        self.scores = dict(
            sorted(raw_scores.items(), key=lambda x: x[1], reverse=True)
        )

    def _from_net(
        self, net: Network,
        engine: NetworkDijkstra = None
    ):
        if engine is None:
            engine = NetworkDijkstra()

        raw_scores = { node_id: 0 for node_id in net.nodes }

        for src in tqdm(net.nodes):
            engine.from_src(src=src, net=net)
            for (node_id, added_score) in engine.count_paths.items():
                raw_scores[node_id] += added_score

        self.scores = dict(
            sorted(raw_scores.items(), key=lambda x: x[1], reverse=True)
        )
```

Listing 3.3: Graph Analysis: NetworkAnalysisBetweenness class inner function implementation

```python
class NetworkAnalysisBetweenness:
    def from_net(
        self, net: Network,
        engine: NetworkDijkstra = None,
        alg: str = 'tree'
    ):
        if alg == 'tree':
            self._from_net_shortest_tree(net, engine)
        else:
            self._from_net_brute_force(net, engine)

    def top_scores(self, k: int = 10):
        it = iter(self._scores)
        return [next(it) for _ in range(k)]

    @property
    def scores(self):
        # Returns the mapping from a node to its score in order of
        # decreasing score
```

Listing 3.4: Graph Analysis: `NetworkAnalysisBetweenness` class

- `from_net(self, net: Network, src: int)`
  Compute the betweenness scores of every node.
  - If `alg == 'tree'`, run the An improvement to $O(|V|^2 + |V||E|\log|V|)$ using Shortest-path tree algorithm.
  - Otherwise, run the The naive $O(|V|^2|E|\log|V|)$ algorithm algorithm.
- ***property*** `top_scores(k: int) -> list[tuple[int, int]]`
  Return k-most influential *Stop*s with their corresponding scores.

3.3 | Experiments on real data

ⓐ | Shortest Path between arbitary nodes

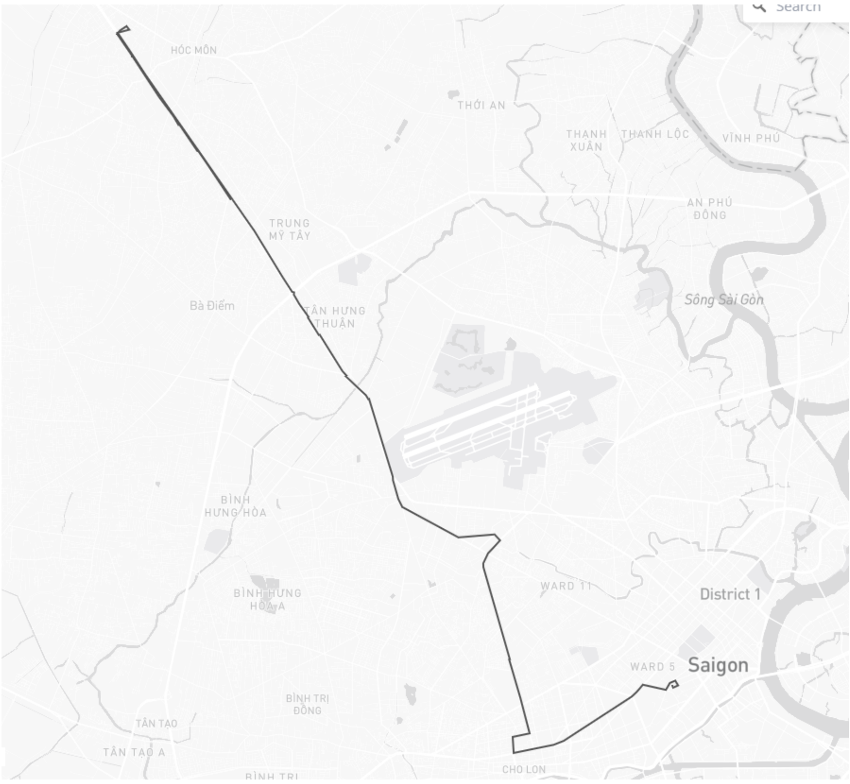


Figure 3.1: Shortest path between nodes `35` and `1234`

ⓑ | Running time between Betweenness Analysis algorithms

	`alg = 'default'`		`alg = 'tree'`		**Improvement**
	sec	iterations/sec	sec	iterations/sec	%
Run #1	568.207484	7.738370	148.743473	29.560961	
Run #2	534.781242	8.222054	141.632670	31.045097	
Run #3	460.380685	9.550792	143.763187	30.585020	
Run #4	450.744785	9.754966	137.384343	32.005103	
Run #5	436.979286	10.062262	137.740463	31.922355	
Average	490.218697	9.065689	141.852827	31.023707	345.58%

Table 3.1: Bus Graph Experiments: Running time of analysis algorithms

ⓒ | Top $k = 10$ most influential *Stops*

Rank	StopId	Name	Street	District	StopType	Score
1	268	Mũi tàu Cộng Hòa	Trường Chinh	Tân Bình	Trụ dừng	2625614
2	267	Ngã ba Chế Lan Viên	Trường Chinh	Tân Bình	Nhà chờ	2625614
3	1239	Bến xe An Sương	Quốc lộ 22	Hóc Môn	Trụ dừng	2614406
4	1115	Bến xe An Sương	Quốc lộ 22	Quận 12	Trụ dừng	2613916
5	1393	Ngã tư Trung Chánh	Quốc lộ 22	Hóc Môn	Nhà chờ	2607219
6	1152	Trung tâm Văn hóa Quận 12	Quốc lộ 22	Quận 12	Nhà chờ	2604321
7	270	UBND Phường 15	Trường Chinh	Tân Bình	Nhà chờ	2311589
8	271	Khu Công Nghiệp Tân Bình	Trường Chinh	Tân Bình	Nhà chờ	2310622
9	174	Trạm Dệt Thành Công	Trường Chinh	Tân Phú	Nhà chờ	2294866
10	510	Bệnh viện Quận Tân Bình	Hoàng Văn Thụ	Tân Bình	Nhà chờ	2261665

Table 3.2: Bus Graph Experiments: Top $k = 10$ most influential *Stop*s

Figure 3.2: Bus Graph Experiments: Spatial distribution of Betweenness score

3.4 | Network class diagram

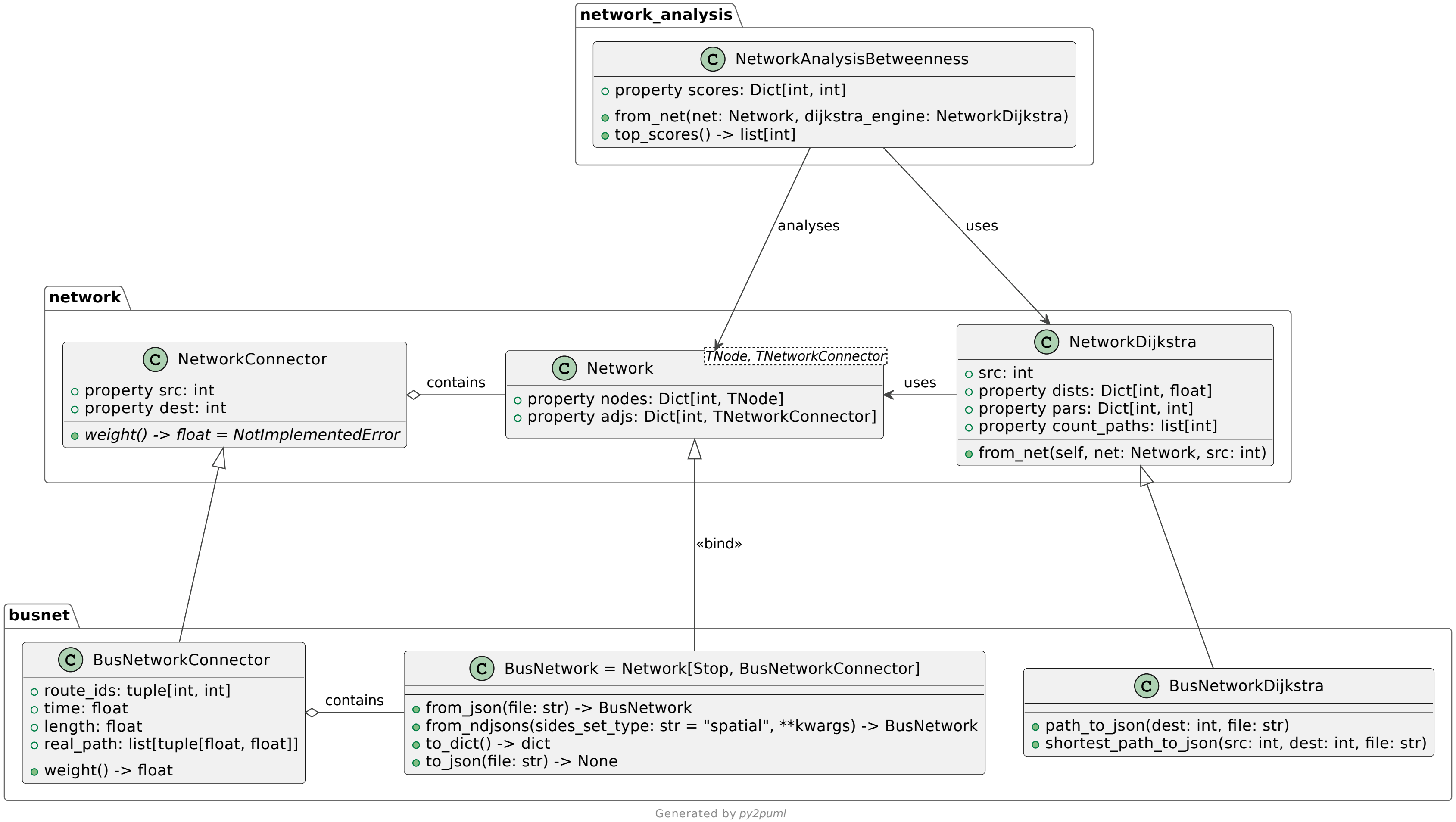


Figure 3.3: Bus Network Analysis: `Network` class diagram

Bibliography

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[2] F. L. Clarke, "A set of measures of centrality based on betweenness," *Sociometry*, 1977. [Online]. Available: https://doi.org/10.2307/3033543

[3] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, 03 2004.

[4] A. Bavelas, "Communication Patterns in Task-Oriented Groups," *The Journal of the Acoustical Society of America*, vol. 22, no. 6, pp. 725–730, 11 1950. [Online]. Available: https://doi.org/10.1121/1.1906679

[5] M. Marchiori and V. Latora, "Harmony in the small-world," *Physica A: Statistical Mechanics and its Applications*, vol. 285, no. 3–4, p. 539–546, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0378-4371(00)00311-3

[6] "Conceptual distance in social network analysis," *Journal of Social Structure*, 2005.