CS162 Final Project Report: Bus Network Analysis

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, May 3, 2024

Tasks by Week

Week 03

1. Appendix I.1, Create a *private* GitHub repository
2. Appendix I.2, Clone a repository from GitHub
3. Appendix I.3, Create new files and folders in GitHub
4. Appendix I.4, `.gitignore` 101
5. Appendix I.5, Basic Git commands `(add, commit, push, pull)`
6. Appendix II.1, Python project with multiple files
7. Appendix II.2, Importing external files
8. Appendix II.3, Python classes, properties and methods

Week 05

1. Section 5.1, Stop (the `stop` module); Subsection 10.2.1, `StopQuery`
2. Section 5.2, Variant (the `variant` module); Subsection 10.2.2, `VariantQuery`
3. Section 10, Querying List of Objects

Week 06

1. Chapter 1, PROJ: Coordinate Transformation
2. Chapter 2, GeoJSON: Geosptial Data Format
3. Section 5.3, Path (the `path` module); Subsection 10.2.3, `PathQuery`
4. Chapter 3, Shapely: Spatial Analysis
5. Chapter 4, R-tree: Data Structure for Spatial Data
6. Section 11.1, Querying a `pandas.DataFrame`

Week 07

1. Chapter 6, Bus Network Graph: Construction Stage

2. Chapter 7, Dijkstra: Shortest Paths and related Centrality Measures
3. Section 7.2, APSP: All Pairs Shortest Path problem
4. Section 7.3, Betweenness Centrality; Section 9.1.3, `NetworkAnalysisBetweenness` class

Week 10

1. Chapter 8, Space-efficient Shortest Path Querying: Bidirectional Dijkstra, Contraction Hierarchy
2. Section 9.3, Experiments on real data

Week 11

1. Chapter 11, LangChain: Apply Large Language Model to Querying Databases

Contents

List of Figures

List of Tables

List of Listings

Part A
Geospatial Analysis

Chapter 1

PROJ: Coordinate Transformation

PROJ[3] is a generic coordinate transformation software that transforms geospatial coordinates from one *coordinate reference system (CRS)* to another.

- **Cartographic projections**. Mathematical transformations that project the Earth, represented by a uniform sphere (or a spherical ellipsoid) into the 2D Cartesian coordinate system.
- **Geodetic transformations**. Mathematical transformations from one CRS to another.

`pyproj`[4] is the Python interface to PROJ. The package can be installed like any other Python libraries (via `pip`, `conda`, etc.)

```
pip install pyproj
```

Listing 1.1: PROJ: Installing `pyproj`

1.1 | Some common Coordinate Reference System (CRS)

1.1.1 | EPSG:4326 / WGS-84: The World Geodetic System 1984

WGS-84 is a reference coordinate system that consists of

- **A reference ellipsoid**. WGS-84 models the Earth as a **spherical ellipsoid** that approximates the Earth's **geoid**, with parameters:
  - *Semi-major axis* $a = 6\ 378\ 137.000$m
  - *Flattening* $f = 1 : 298\ 257\ 223\ 563$
- **A coordinate system**. WGS-84 uses **longtitudes** (λ) and **latitudes** (φ) to position a point on the globe. When the distance of a point to the Earth's surface is considered, **altitudes** (h) is also introduced.

WGS-84 is the best geodetic system up to date. The system has an absolute accuracy of $1 - 2$ metres. Hence, many geodetic applications rely on the WGS84.

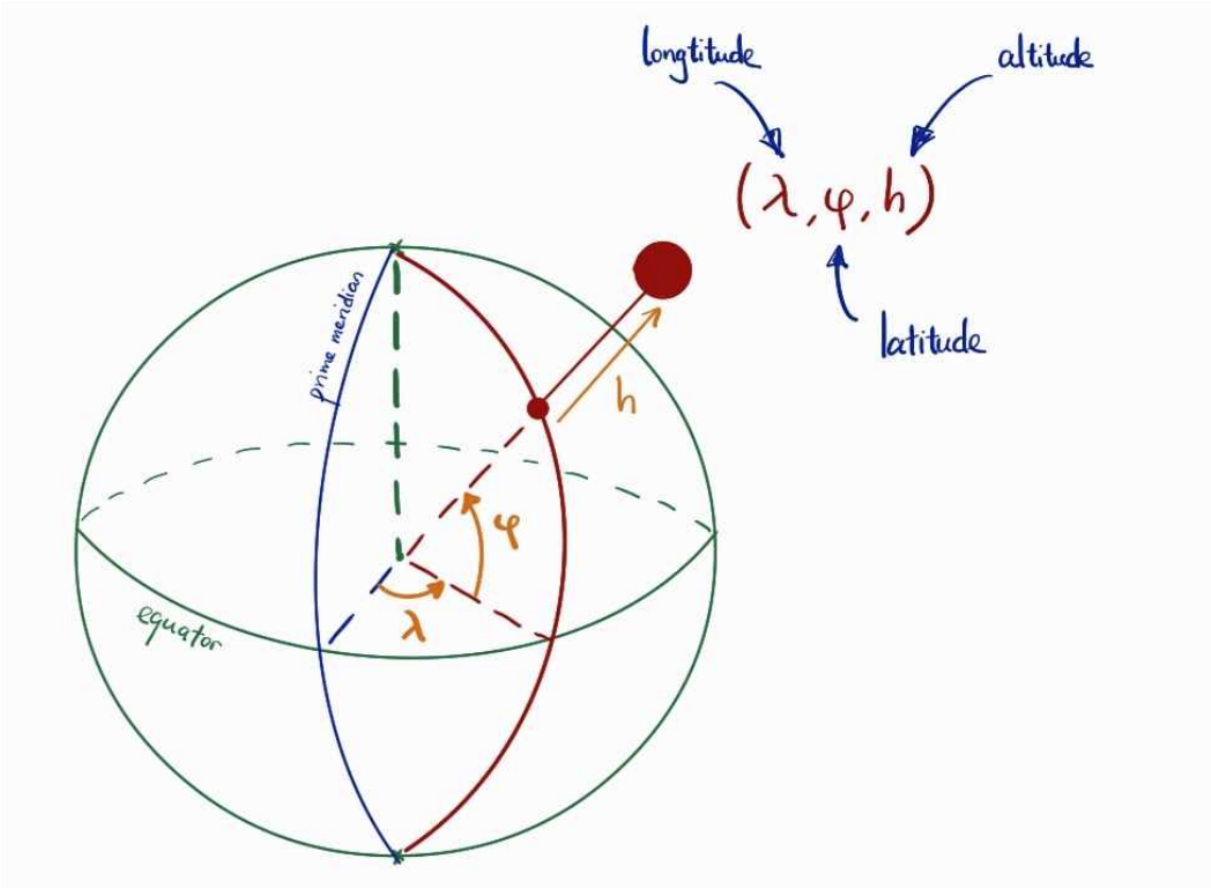


Figure 1.1: PROJ: The WGS-84 ellipsoid model and coordinate system

The WGS-84 standard is currently being used in the Global Positioning System (GPS), and its model of the Earth is adopted by many countries' CRS, including Vietnam's VN-2000.

1.1.2 | EPSG:3405 / VN-2000: Vietnam's National coordinate system 2000

Vietnam is currently using the VN2000 coordinates system, first established by *Decision No. 83/2000/QD-TTg on the use of Vietnam's National references and Coordinates Systems*[5]. Prior to VN2000, Vietnam had used the HN-72 system, a system that was commonly used in former socialist countries.

- **Reference ellipsoid**. VN-2000 shares its reference ellipsoid with WGS-84.
- **Coordinate system**. According to the international UTM Confirmed Cylindrical Transverse Mercator Projection. Vietnam uses the UTM projection on Zone 48N, which projects Vietnam onto a flat surface equipped with a Cartesian coordinate system, with minimum distortions.

Each CRS has their own advantages. Whilst WGS-84 excels in locating positions and is more common in GPS systems, tasks like calculating distances, finding shortest route, etc. require a flat coordinate system. Projected CRS such as VN-2000 is, therefore, a better-fit.

	WGS-84 (EPSG:4326)		**VN-2000** (EPSG:3405)	
	lat (λ)	**lng** (φ)	x	y
min	102.14	8.33	184767.75	999099.0
centre	-/-	-/-	589204.99	1754291.1
max	109.53	23.4	920895.91	2595190.43

Table 1.1: PROJ: Coordinate bounds in WGS-84 and VN-2000

1.2 | Converting from one CRS to another

1.2.1 | `Transformer.from_crs`

This is the `Transformer` class constructor. It takes two main arguments:

- `crs_from: pyproj.crs.CRS | str`: source CRS
- `crs_to: pyproj.crs.CRS | str`: desired CRS after transformation

1.2.2 | `Transformer.transform`

This function transform point(s) from the `crs_from` CRS to the `crs_to` CRS. The main parameters are

- `xx`: input x coordinate(s)
- `yy`: input y coordinate(s)
- `zz`: input z coordinate(s) (optional)
- `tt`: input t coordinate(s) (optional)
- `radians`: Input t coordinates (optional)
- `errcheck`: if `True`, raises errors; otherwise, returns `inf`
- `inplace`: if `True`, attempts to write the results onto the input array instead of returning a new array.

The function supports transformation in at least 2 dimensions and at most 4 dimensions.

```
class pyproj.transformer.Transformer:
    # ...
    @staticmethod
    def from_crs(crs_from: Any, crs_to: Any, **kwargs)
        -> Transformer:
        """
        Get all possible transformations from a :obj:`pyproj.crs.CRS`
        or input used to create one.
        ...
        """

    def transform(
        self,
        x: Any,
        y: Any,
        z: Any = None,
        t: Any = None,
        radians: bool = False,
        errcheck: bool = False,
        inplace: bool = False,
        **kwargs
    ) -> tuple[Any, Any] | tuple[Any, Any, Any] | tuple[Any, Any, Any, Any]:
        """
        Transform points between two coordinate systems.
        ...
        """

    # ...
```

Listing 1.2: PROJ: Implementation in `pyproj.Transformer` class. [2]

1.3 | Example: Converting from WGS-84 to VN-2000

```python
import pyproj

wgs84_vn2000_tf = pyproj.Transformer.from_crs('epsg:4326', 'epsg:3405')
vn2000_wgs84_tf = pyproj.Transformer.from_crs('epsg:3405', 'epsg:4326')

def wgs84_to_vn2000(lat: float, lng: float) -> tuple[float, float]:
    return wgs84_vn2000_tf.transform(lat, lng)

def vn2000_to_wgs84(x: float, y: float) -> tuple[float, float]:
    return vn2000_wgs84_tf.transform(x, y)
```

Listing 1.3: PROJ: Converting `(lat, lng)` in WGS-84 to `(x, y)` in VN-2000. Module defined in `helper/crs_convert.py`

Chapter 2

GeoJSON: Geosptial Data Format

GeoJSON is a format for encoding a variety of geographic data structures [6]. It is an extension of the JSON format, defined by the RFC7946 standard [7].

2.1 | GeoJSON Viewers

2.1.1 | geojson.io

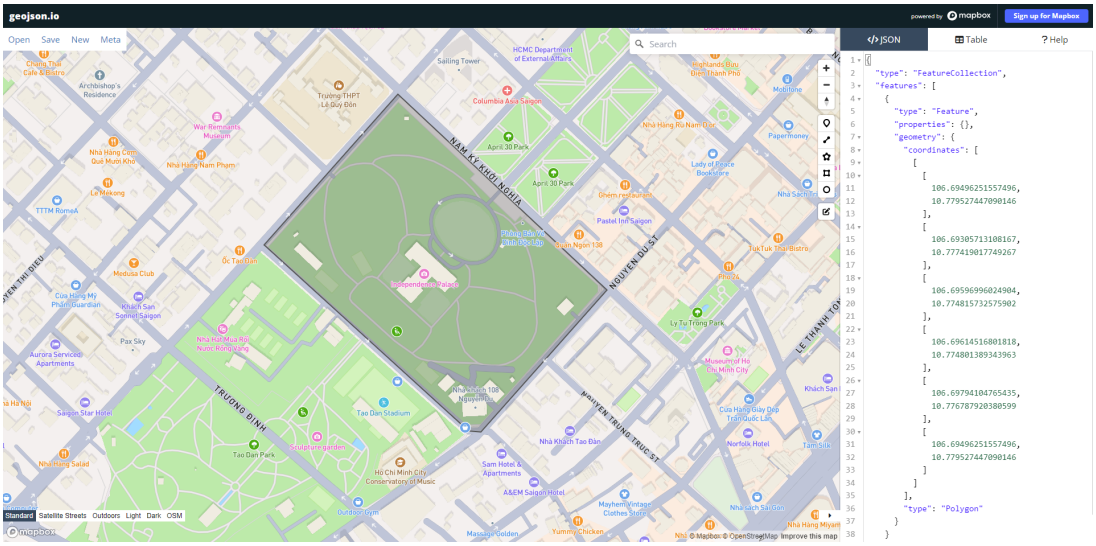


Figure 2.1: GeoJSON: *geojson.io*

geojson.io is a website designed for working with GeoJSON files. It supports:

- **Creating and Editing GeoJSON data**. GeoJSON objects can be editted manually or via graphical tools on the web.
- **Interchanging geodata between different formats**. geojson.io supports importing from and exporting to various formats, including *.geojson, *topojson, *.csv, *.kml, etc.
- **Visualising data on maps**. GeoJSON objects are overlayed on real maps such as satellite maps and street maps.

2.1.2 | Geo Data Viewer (a Visual Studio Code snippet)

This is a Visual Studio Code extension that helps visualing local GeoJSON files. To use the extension, simply install via VSCode Marketplace, then at the .geojson file, choose the *Geo: View Map* option from the *Command Pallete*.

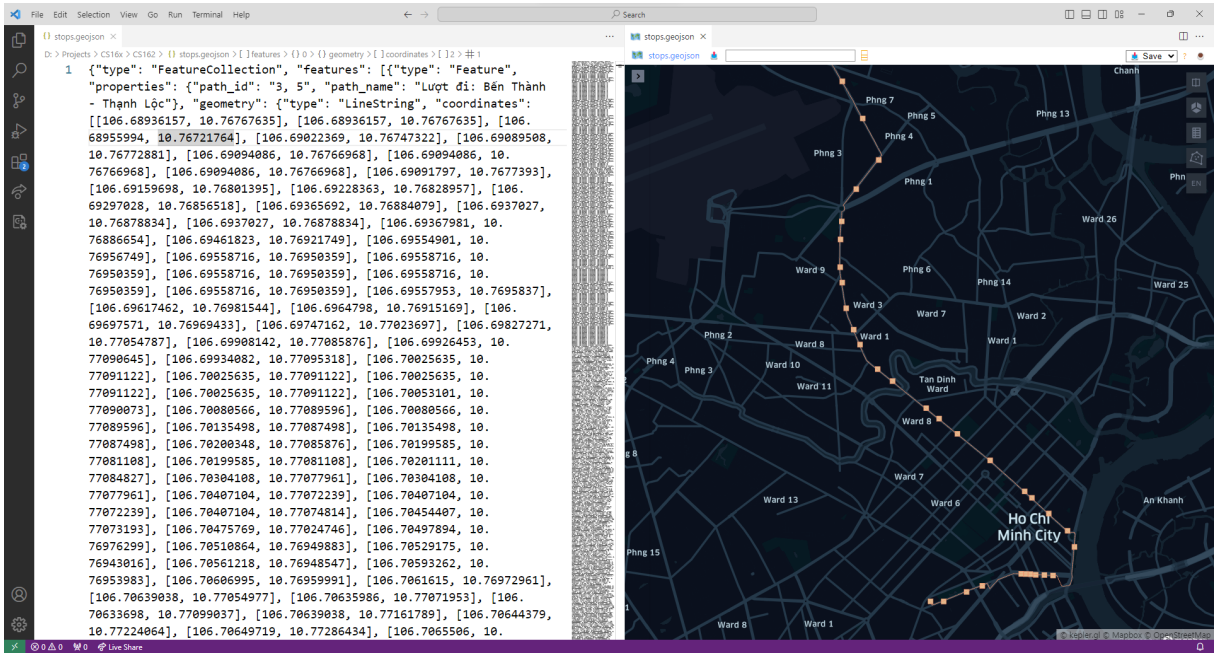


Figure 2.2: GeoJSON:
GeoJSON file in plain *(left)*, Visualisation in *Geo Data Viewer (right)*

2.2 | GeoJSON Features

2.2.1 | Feature

A Feature object represents a spatially bounded thing.

- **Type**. Every Feature object has a `"type"` attribute with value `"Feature"`.
- **Geometry**. Defines the shape of the object and its coordinates.
- **Properties**. Properties are optional and do not affect the shape of the object, but may come in handy when objects are visualised on the map.

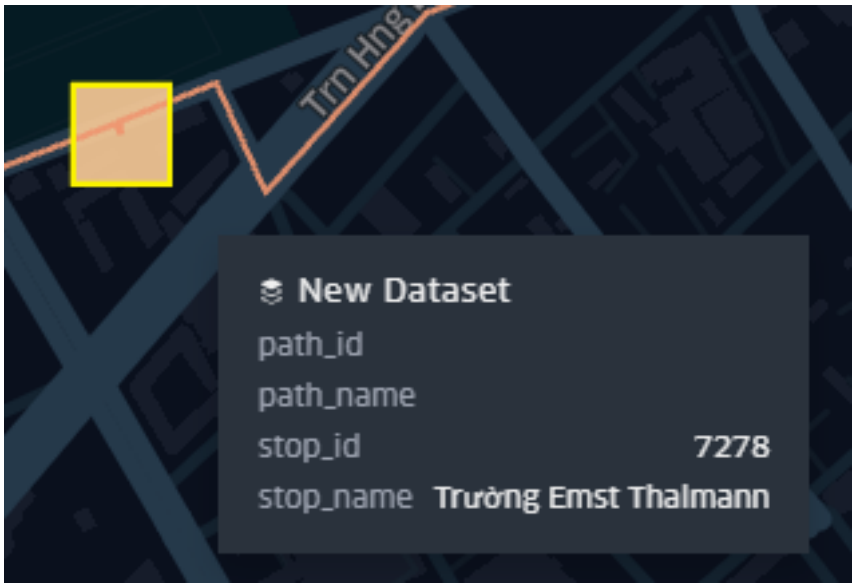


Figure 2.3: GeoJSON: Properties of a polygon show up when hovered

2.2.2 | GeometryCollection

A GeometryCollection object contains a list of `"geometries"`, each of which is a Geometry object (See GeoJSON Objects).

2.2.3 | FeatureCollection

A FeatureCollection object contains a list of `"features"`, each of which is a Feature object. This is the highest object in the hierarchy of the GeoJSON object tree.

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
          "flight": "SGN - SIN",
          "time": "2:05",
          "cost": 4959000
      },
      "geometry": {
        "coordinates": [
            [106.7179127, 10.8085596], [103.7174418, 1.4671041]
        ],
        "type": "LineString"
      }
    },
    {
      "type": "Feature",
      "properties": {
        "Airport": "Tan Son Nhat"
      },
      "geometry": {
        "coordinates": [106.7168838, 10.7699593],
        "type": "Point"
      }
    },
    {
      "type": "Feature",
      "properties": {
        "Airport": "Changi"
      },
      "geometry": {
        "coordinates": [103.6990521, 1.3358573],
        "type": "Point"
      }
    }
  ]
}
```

Listing 2.1: GeoJSON: Example of a complete .geojson file

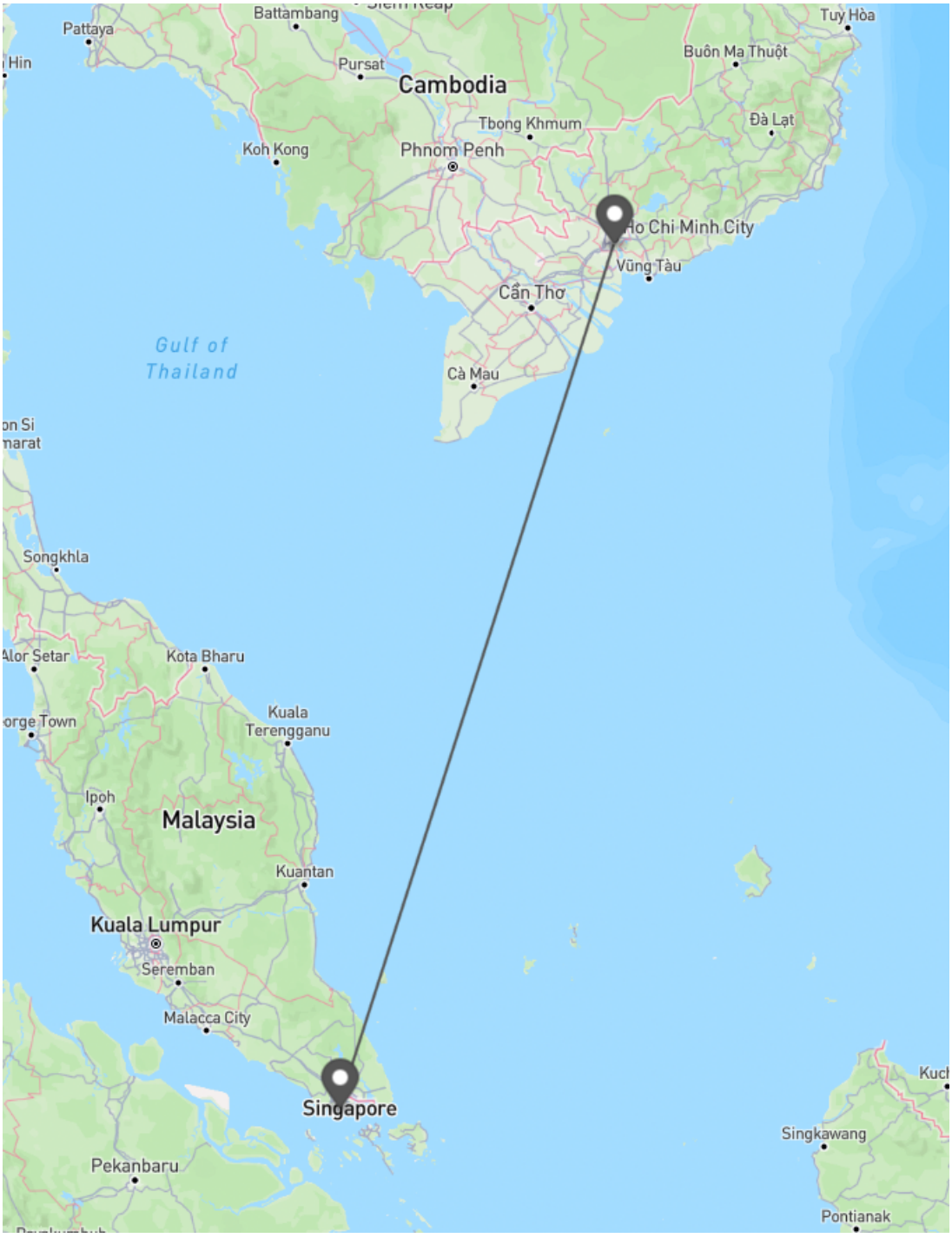


Figure 2.4: GeoJSON: Visualisation on satellite map

2.3 | GeoJSON Objects

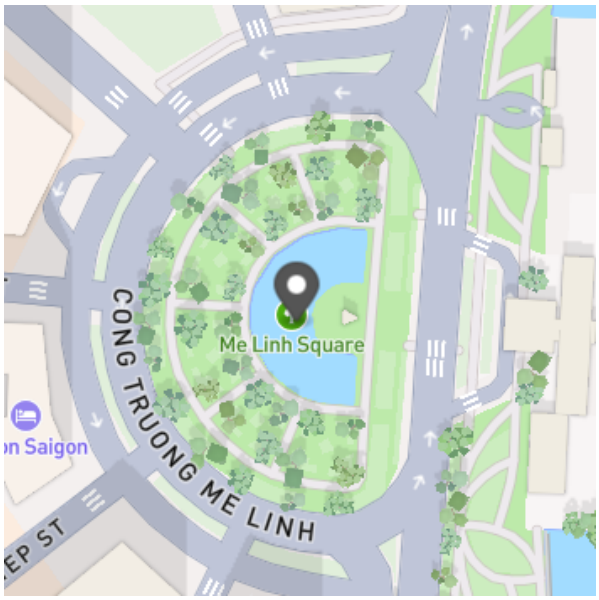
All GeoJSON gemetry objects have some common properties:

- **Type**. can be one of the geometry types: `Point`, `MultiPoint`, `LineString`, `MultiLineString`, and `Polygon`, `MultiPolygon`.
- **Coordinates**. A list of coordinates that defines the object. GeoJSON uses the WGS-84 coordinate system (latitude, longtitude, elevation), where elevation is optional.

  These basic geometry objects build up what is called a Spatial Data Model, which will be discussed in Section 3.1 when we dig into spatial data analysis and manipulation.

2.3.1 | Point

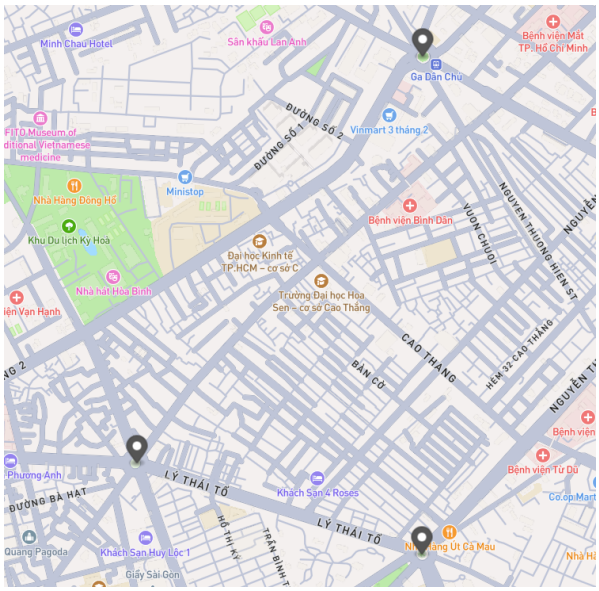
```
{
  "type": "Feature",
  "properties": {
    "Name": "Me Linh Square"
  },
  "geometry": {
    "coordinates": [
        106.7063463, 10.7752786
    ],
    "type": "Point"
  }
}
```



Listing 2.2: GeoJSON Point

2.3.2 | MultiPoint

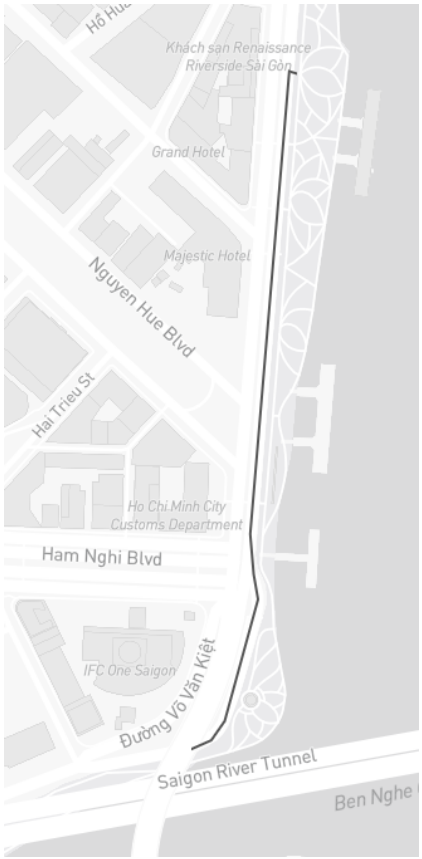
```
{
  "type": "Feature",
  "properties": {
    "Name": "Roundabouts"
  },
  "geometry": {
    "coordinates": [
      [106.6816905, 10.7778276],
      [106.6816538, 10.7654209],
      [106.6744136, 10.7677078]
    ],
    "type": "MultiPoint"
  }
}
```



Listing 2.3: GeoJSON MultiPoint

2.3.3 | LineString

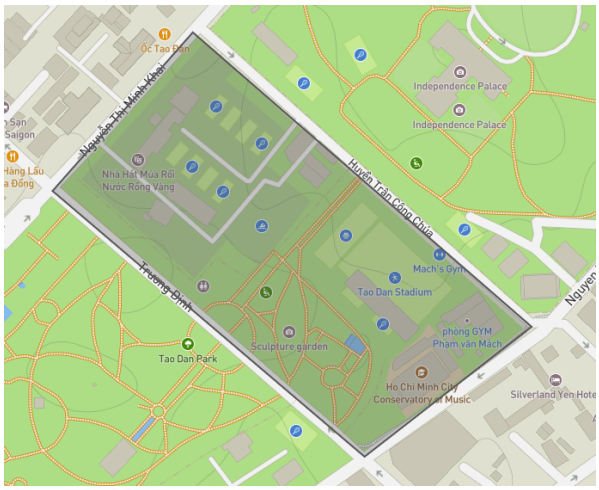
```
{
  "type": "Feature",
  "properties": {
    "PathName": "Ben Thanh-Thuan Loc"
  },
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [106.7059326, 10.7695398],
      [106.7060699, 10.7695999],
      [106.7061615, 10.7697296],
      [106.7063903, 10.7705497],
      [106.7063598, 10.7707195],
      [106.7063369, 10.7709903],
      [106.7063903, 10.7716178],
      [106.7064437, 10.7722406],
      [106.7064971, 10.7728643],
      [106.7065506, 10.7734870],
      [106.7066040, 10.7741098],
      [106.7066574, 10.7740936]
    ]
  }
}
```



Listing 2.4: GeoJSON LineString

2.3.4 | Polygon

```
{
  "type": "Feature",
    "properties": {
        "Name": "Tao Dan Park"
    },
    "geometry": {
      "coordinates": [
        [
          [106.6929551, 10.7773571],
          [106.6917032, 10.7759683],
          [106.6944880, 10.7736426],
          [106.6959784, 10.7747803],
          [106.6929551, 10.7773571]
        ]
      ],
      "type": "Polygon"
    }
  }
}
```



Listing 2.5: GeoJSON Polygon

```
{
  "type": "Feature",
  "geometry": {
    "coordinates": [
      [
        [106.6974673, 10.7440676],
        [106.6964724, 10.7441327],
        [106.6964787, 10.7436847],
        [106.6973864, 10.7435340],
        [106.6974673, 10.7440676]
      ], [
        [106.6966164, 10.7439634],
        [106.6966384, 10.7437807],
        [106.6969000, 10.7438044],
        [106.6970271, 10.7437115],
        [106.6972078, 10.7436630],
        [106.6972887, 10.7437105],
        [106.6973139, 10.7438798],
        [106.6972246, 10.7439510],
        [106.6966164, 10.7439634]
      ]
    ],
    "type": "Polygon"
  }
}
```

Listing 2.6: GeoJSON Polygon with holes. The resulting shape is obtained by uniting polygons with positive signed area ($A > 0$), then subtracting polygons with negative signed area ($A < 0$)

Chapter 3

Shapely: Spatial Analysis

This chapter is based mostly on the official Shapely documentation, Release 2.0.3[8]

Shapely is a Python package for **set-theoretic analysis** and **manipulation of planar features**, wrapping the widely deployed GEOS library. As usual, installation can be done via `pip` or `conda`:

```
pip install shapely
```

Listing 3.1: Shapely: Installing Python `shapely` package

Shapely, by default, only uses the Cartesian coordinate system. For spatial analysis, it is practical to convert spherical coordinates to Cartesian coordinates, apply the calculations/transformations, then revert back to geographic coordinates. See Example: Converting from WGS-84 to VN-2000.

3.1 | Spatial Data Model

Fundamental geometric objects are characterised by three set of points, *interior*, *exterior* and *boundary* the union of which coincides with the plane.

3.1.1 | Points: `Point` class

Definition of a Point:

- Interior: One point
- Boundary: $\emptyset$
- Exterior: All other points
- Dimension: 0

A Point is implemented in the `Point` class. Its 'collectional' version is the `MultiPoint` class.

```
class Point(*args):
    # A geometry type that represents a single coordinate
    # with x,y and possibly z values.
```

Listing 3.2: Shapely: `Point` class

```
>>> from shapely.geometry import *
>>> p = Point(3, 5)
>>> p
<POINT (3 5)>
>>> p.x, p.y
(3.0, 5.0)
>>> p.area, p.bounds
(0.0, (3.0, 5.0, 3.0, 5.0))
```

Listing 3.3: Shapely: Point

3.1.2 | Curves: `LineString` class, `LinearRing` class

Definition of a Curve:

- Interior: All points along the curve's length
- Boundary: Two endpoints
- Exterior: All other points
- Dimension: 1

A Curve is implemented in the `LineString` class and the `LinearRing` class. Note that a smooth curve can only be approximated in this model.
Their collectional versions are `MultiLineString` and `MultiLinearRing`.

```
class LineString(coordinates=None):
    # A geometry type composed of one or more line segments.
```

Listing 3.4: Shapely: `LineString` class

```
>>> from shapely.geometry import *
>>> line = LineString([
...     [0, 0],
...     [0, 1],
...     [1, 1],
...     [1, 2],
...     [2, 2],
... ])
>>> line
<LINESTRING (0 0, 0 1, 1 1, 1 2, 2 2)>
>>> line.length
4.0
>>> line.area, line.bounds
(0.0, (0.0, 0.0, 2.0, 2.0))
```

Listing 3.5: Shapely: LineString

3.1.3 | Surface: `Polygon` class

Definition of a Surface:

- Interior: All points within the curves
- Boundary: One or more curves
- Exterior: All other points, including those within **holes**
- Dimension: 2

A Surface is implemented in the `Polygon` class. Note that a smooth surface can only be approximated in this model.

```
class Polygon(shell=None, holes=None):
    # A geometry type representing an area that is enclosed by a
    # linear ring. It may have one or more negative-space holes
    # which are also bounded by linear rings
```

Listing 3.6: Shapely: `Polygon` class

```
>>> from shapely.geometry import *
>>> shell = [[0, 0], [0, 4], [4, 4], [4, 0]]
>>>  hole = [[1, 1], [1, 3], [3, 3], [3, 1]]
>>> poly = Polygon(shell=shell, hole=hole)
>>> poly
<POLYGON ((0 0, 0 4, 4 4, 4 0, 0 0), (1 1, 1 3, 3 3, 3 1, 1 1))>
>>> poly.area, poly.bounds
4.0
>>> poly.area, poly.bounds
(12.0, (0.0, 0.0, 4.0, 4.0))
```

Listing 3.7: Shapely: Polygon

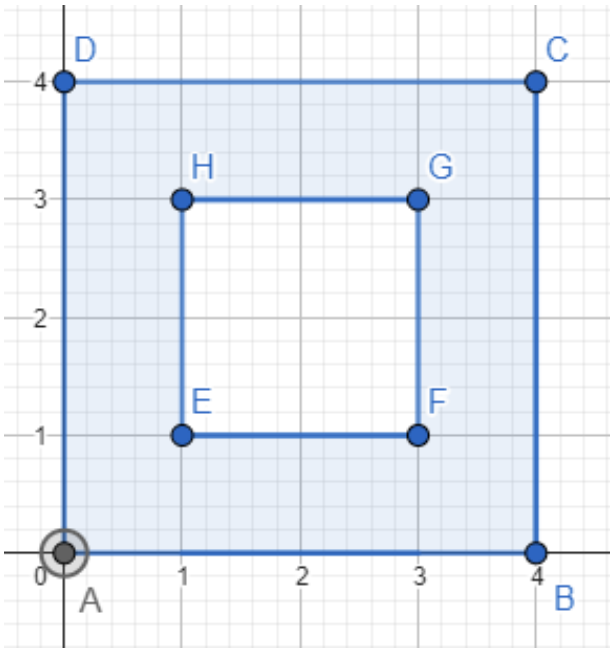


Figure 3.1: Shapely: The `Polygon` in the sample code is obtained by subtracting the polygon $ABCD$ by the polygon $EFGH$.

3.2 | Geometric Relationships

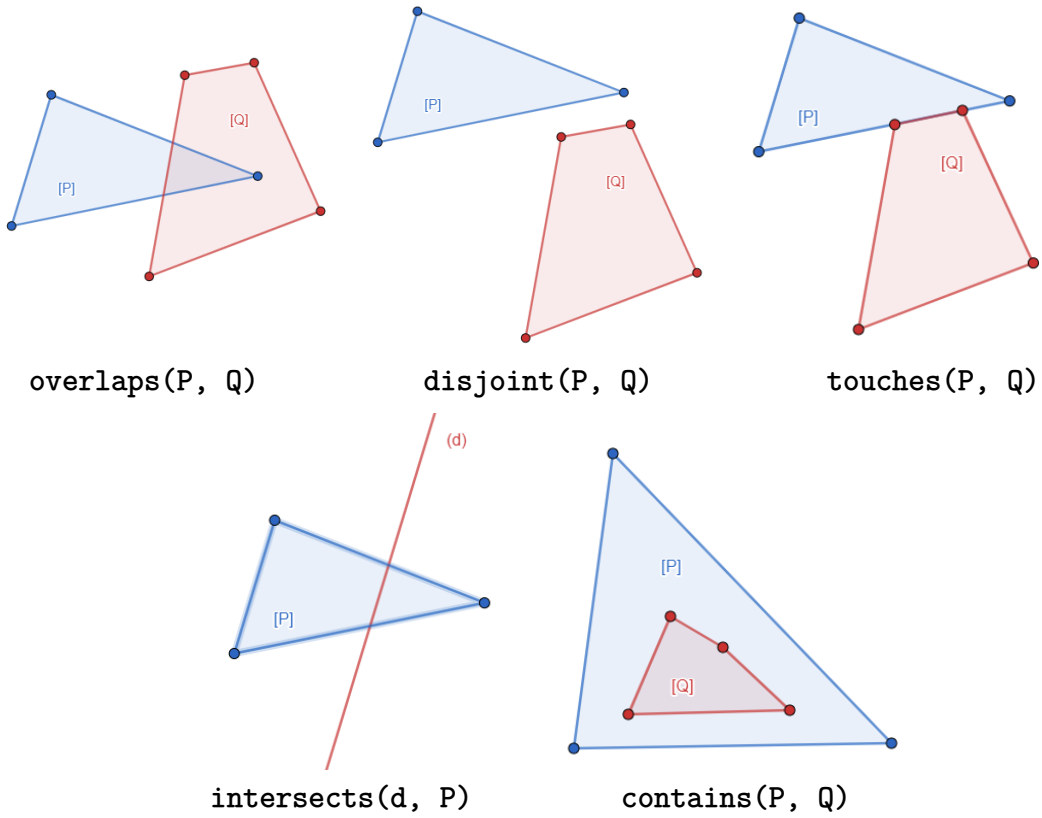
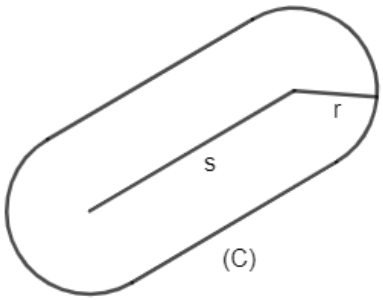
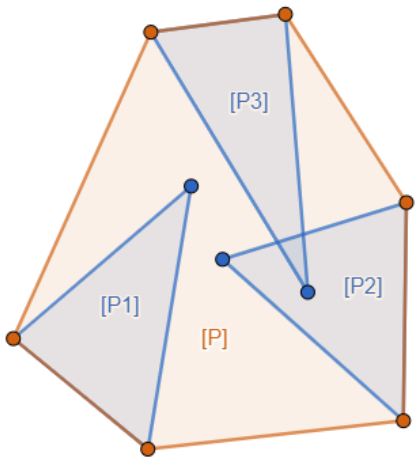


Figure 3.2: Shapely: Geometric relationships and their Shapely equivalent code snippet

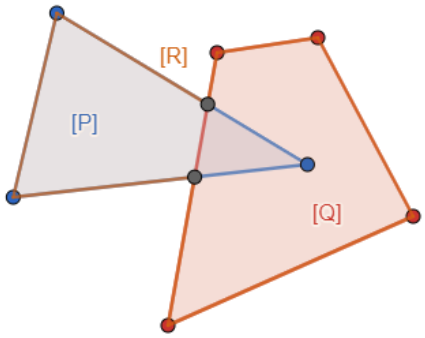
3.3 | Geometric Operations

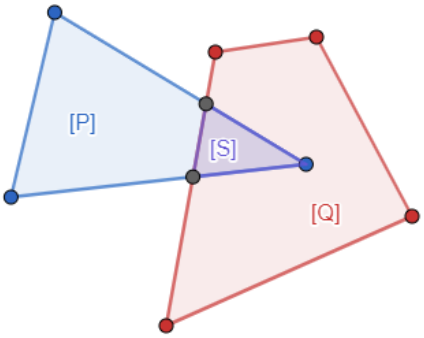





```
C = buffer(s, r)
```

```
P = convex_hull(MultiPolygon([P1, P2, P3]))
```






```
R = union(P, Q)
```

```
R = intersection(P, Q)
```

Figure 3.3: Shapely: Geometry operations and their Shapely equivalent code snippet

Chapter 4

R-tree: Data Structure for Spatial Data

This chapter is based mostly on the official R-tree Documentation, Release 1.2.0[9]

R-trees are tree data structures used in spatial analysis. Its most notable applications are finding the nearest point(s) and retrieving points within a given distance.

The R-tree data structure is based on the B-tree: Each leaf node is an object, nearby nodes are grouped and represented by their minimum bounding rectangle at a higher level node. Although having linear worst-case complexity for most operations, R-trees have been shown to perform well on real world data.

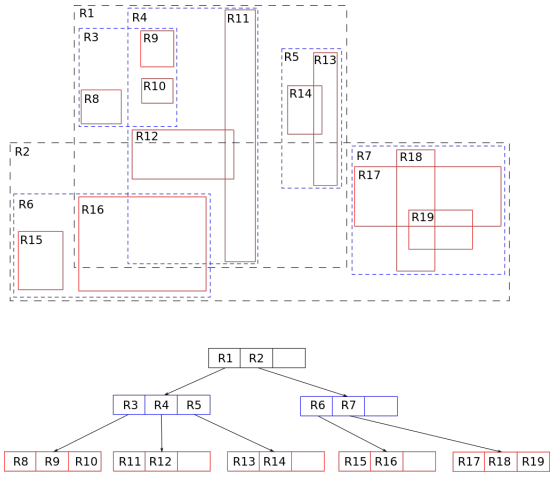


Figure 4.1: R-tree: Image of an R-tree. *(Source: Wikipedia)*

In this chapter we shall not dive deep into the theorical aspect of R-trees, rather we will introduce their applications via the `Rtree` Python package, a Python wrapper for libspatialindex, providing a number of advanced spatial indexing features, such as *intersection search* and *nearest neighbour search*. Installation can be done via `pip` or `conda`:

```
pip install rtree
```

Listing 4.1: R-tree: Installing Python `rtree` package

```
class rtree.index.Index:
    def __init__(self, *arg, **kwargs):
        """
        Creates a new R-tree Index object.
        """

    @property
    def bounds(coordinate_interleaved: bool) -> Iterable[int]:
        """
        Returns the bounds of the R-tree.
        """

    def insert(id: int, coodinates: Any, obj: object = None) -> None:
        """
        Insert a new object with id and coordinates.
        """

    def delete(id: int, coordinates: Any) -> None:
        """
        Delete an object with id and coordinates.
        """

    def count(coordinates: Any) -> int:
        """
        Returns the count of objects intersecting the given coordinates.
        """

    def intersection(coordinates: Any, object = Literal[X]) -> Iterator[Y]:
        """
        Returns ids/objects intersecting the given coordinates.
        """

    def nearest(coordinates: Any, num_results: int, object = Literal[X])
        -> Iterator[Y]:
        """
        Returns ids/objects of the k-nearest objects (k = num_results)
        """

    def close() -> None:
        """
        Frees R-tree from memory.
        """
```

Listing 4.2: R-tree: Implementation of `rTree.index.Index` class

4.1 | Creation

4.1.1 | `rtree.index.Index` constructor

- `filename` / `stream`: Defines the file-based storage for the RTree. If left blank, RTree will be stored on default memory.
- `interleaved=True`
  Returns in the form `[Xmin, Ymin, ..., Xmax, Ymax, ...]` if `interleaved=True`, otherwise returns in the form `[Xmin, Xmax, Ymin, Ymax,...]`.
- `properties`: An `index.Property` object, containing creation and instantiation properties for the object.

For now, we will be using the blank constructor.

```
>>> from rtree import index
>>> index.Index()
rtree.index.Index(
    bounds=[
         1.7976931348623157e+308,  1.7976931348623157e+308,
        -1.7976931348623157e+308, -1.7976931348623157e+308
    ],
    size=0
)

>>> index.Index(interleaved=False)
rtree.index.Index(
    bounds=[
         1.7976931348623157e+308, -1.7976931348623157e+308,
         1.7976931348623157e+308, -1.7976931348623157e+308
    ],
    size=0
)
```

Listing 4.3: R-tree: Sample Code on Constructor

Note that in the above example, range_x = range_y = $(+\infty, -\infty) = \emptyset$.

4.2 | Insertion and Deletion

4.2.1 | Insertion: `Index.insert(id, coordinates, object)`

Insert one item into the R-tree with parameters:

- `id`: The id of the object, and need not be unique.
- `coordinates`: A tuple of numbers defining the bounding rectangle.
- `obj`: An object associated with the coordinates.

The method does not guarantee uniqueness, therefore two objects with identical `id` and `coordinates` can co-exist. Uniqueness must be enforced by the user.

4.2.2 | Deletion: `Index.delete(id, coordinates)`

Deletes ONE item from the R-tree with parameters:

- `id`: The id of the object, and need not be unique.
- `coordinates`: A tuple of numbers defining the bounding rectangle.

If multiple objects satisfy the criteria, only one item will be deleted.

```
>>> idx = index.Index()
>>> idx.insert(id=1, coordinates=(0, 0, 2, 2), obj="A")
>>> idx.insert(id=2, coordinates=(1, 1, 3, 3), obj="B")
>>> idx.insert(id=3, coordinates=(2, 2, 4, 4), obj="C")
>>> idx
rtree.index.Index(bounds=[0.0, 0.0, 4.0, 4.0], size=3)
>>> idx.delete(id=1, coordinates=(0, 0, 2, 2))
>>> idx          # Deletion succeeds, size decreases to 2
rtree.index.Index(bounds=[1.0, 1.0, 4.0, 4.0], size=2)
>>> idx.delete(id=2, coordinates=(0, 0, 2, 2))
>>> idx          # Deletion fails, size remains at 2
rtree.index.Index(bounds=[1.0, 1.0, 4.0, 4.0], size=2)
```

Listing 4.4: R-tree: Sample Code on Insertion and Deletion

4.3 | Querying

4.3.1 | *property* `Index.bounds(interleaved=True)`

Returns the bounds of the RTree.

- **`coordinate_interleaved=True`**.

```
>>> idx
rtree.index.Index(bounds=[1.0, 1.0, 4.0, 4.0], size=2)
>>> idx.bounds
[1.0, 1.0, 4.0, 4.0]
```

Listing 4.5: R-tree: Sample Code on property `bounds`

4.3.2 | `Index.count(coordinates)` `Index.intersection(coordinates, objects)`

Counts/Returns objects that intersects the given coordinates.

- **`coordinates`**. Coordinates in query
- **`objects = True | False | 'raw'`**. Describe the return type for the satisfied objects.

- If `True`, returns object wrapped in `rtree.index.Item` class
- If `False`, returns objects' `id`.
- If `'raw'`, returns objects in raw form (i.e., without the `Item` wrapper).

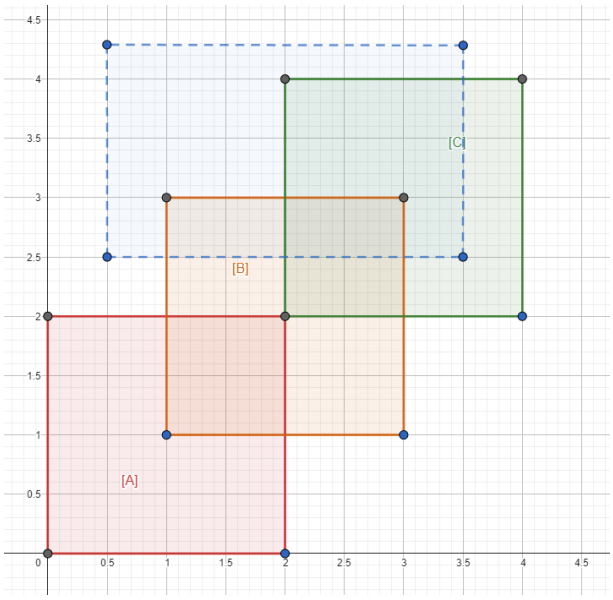


Figure 4.2: R-tree: Sample Code on Intersection operations *(Illustration)*

```
>>> idx = index.Index()
>>> idx.insert(id=1, coordinates=(0, 0, 2, 2), obj="A")
>>> idx.insert(id=2, coordinates=(1, 1, 3, 3), obj="B")
>>> idx.insert(id=3, coordinates=(2, 2, 4, 4), obj="C")
>>> query = (0.5, 2.5, 3.5, 4.5)
>>> idx.count(query)
2
>>> idx.intersection(query)
<generator object Index._get_ids at 0x0000026337738590>
>>> list(idx.intersection(query))
[2, 3]
>>> list(idx.intersection(query, objects='raw'))
['B', 'C']
>>> list(idx.intersection(query, objects=True))
[
    <rtree.index.Item object at @address1>,
    <rtree.index.Item object at @address2>
]
>>> [item.object for item in idx.intersection(query, objects=True)]
['B', 'C']
```

Listing 4.6: R-tree: Sample Code on Intersection operations

4.3.3 | Index.nearest(coordinates, num_objects, objects)

Runs the k-nearest neighbour algorithm.

- **coordinates**. Coordinates in query
- **num_results=1**. The number of objects to return nearest to the coordinates. It is the parameter k in the $k-$nearest neighbour algorithm.
- **objects = True | False | 'raw'**. Describe the return type for the satisfied objects, similar to what in subsection 4.3.2.

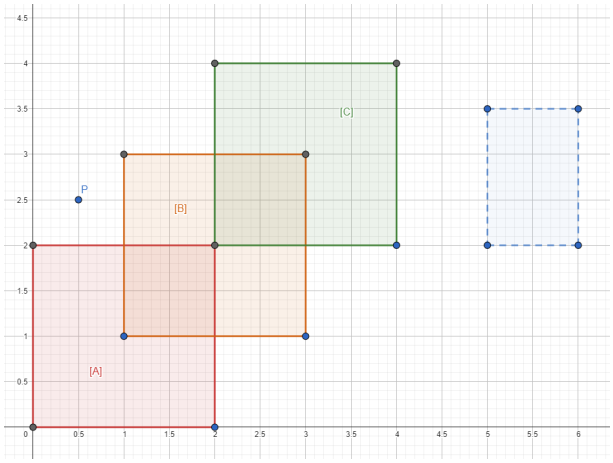


Figure 4.3: R-tree: Sample Code on Nearest Neighbour *(Illustration)*

```
>>> idx = index.Index()
>>> idx.insert(id=1, coordinates=(0, 0, 2, 2), obj="A")
>>> idx.insert(id=2, coordinates=(1, 1, 3, 3), obj="B")
>>> idx.insert(id=3, coordinates=(2, 2, 4, 4), obj="C")
>>> list(idx.nearest((5, 1)))
[3]
>>> list(idx.nearest((0.5, 2.5), objects='raw'))
['B', 'A']
>>> list(idx.nearest((5, 2, 6, 3.5), num_results=2, objects='raw'))
['C', 'B']
```

Listing 4.7: R-tree: Sample Code on Nearest Neighbour

4.3.4 | Index.close()

Disposes the RTree, frees up memory.

Chapter 5

Elements of a Bus Network

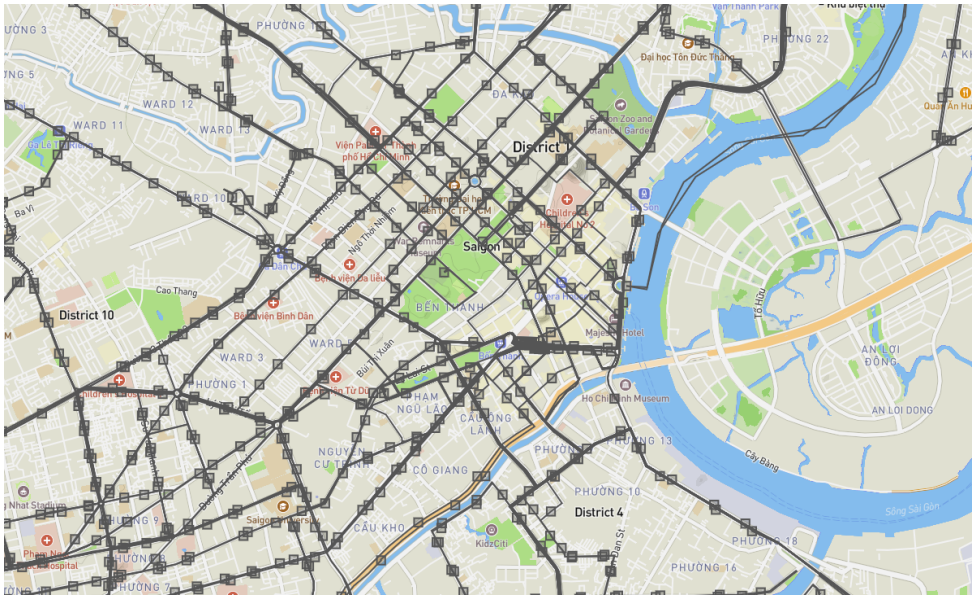


Figure 5.1: Elements of a Bus Network: The public bus network in Ho Chi Minh City. Squares represent bus stops, paths represent bus routes

5.1 | Stop (the stop module)

*Stop*s are the basic elements of a bus network. It defines places where buses stop temporarily to let passengers get on or off the bus.

5.1.1 | Properties

- ***property*** `stop_id: int`
- ***property*** `code: str`
- ***property*** `name: str`
- ***property*** `stop_type: str`
- ***property*** `zone: str`
- ***property*** `ward: str`
- ***property*** `address_no: str`
- ***property*** `street: str`
- ***property*** `support_disability: bool`
- ***property*** `status: str`
- ***property*** `latitude: float`
  property `longtitude: float`
  Coordinates of the bus stop in WGS-84 coordinate system.
- ***property*** `coord`
  Coordinates of the bus stop in VN-2000 coordinate system.
- ***property*** `search: list[str]`
  List of tokens that can be used to search for the bus stop
- ***property*** `routes: list[str]`
  List of route names crossing the bus stop

5.1.2 | Methods

- `to_string() -> str`
  Display information of a `Stop` in a string format.
  This function is called by `__repr__()`.
- ***static*** `from_dict(obj: dict) -> Stop`
  Create `Stop` from a Python dictionary.
- `to_dict() -> dict`
  Convert `Stop` to a Python dictionary.
- ***static*** `from_json(file: str) -> Stop`
  Create `Stop` from JSON file. The function loads a JSON file into a Python dictionary, then call `from_dict()`.
- `to_json(file: str) -> None`
  Export `Stop` information to JSON file. The function calls `to_dict()` to convert `Stop` to a dictionary, then dumps it into a JSON file.

```
>>> from stop import Stop
>>> stop = Stop.from_json('stop.json')
>>> stop.to_string()
================[Nguyen Van Linh]=================
| StopID:              7182                       |
| Code:                Q7 BD2                     |
| Name:                Nguyen Van Linh            |
| Type:                O son                      |
| Zone:                Quan 7                     |
| Ward:                Phuong Tan Phong           |
| Address No.:         R1-49                      |
| Street:              Bui Bang Doan              |
| Support Disability:  True                       |
| Status:              Dang khai thac             |
| Lng:                 106.708499                 |
| Lat:                 10.729471                  |
| Search tokens:       NVL, R, BBD                |
| Routes:              D2                         |
==================================================
>>> stop.routes.append('D3')
>>> stop.to_json('out.json')
>>> stop2 = Stop.from_json('out.json')
>>> stop2
================[Nguyen Van Linh]=================
| StopID:              7182                       |
| Code:                Q7 BD2                     |
| Name:                Nguyen Van Linh            |
| Type:                O son                      |
| Zone:                Quan 7                     |
| Ward:                Phuong Tan Phong           |
| Address No.:         R1-49                      |
| Street:              Bui Bang Doan              |
| Support Disability:  True                       |
| Status:              Dang khai thac             |
| Lng:                 106.708499                 |
| Lat:                 10.729471                  |
| Search tokens:       NVL, R, BBD                |
| Routes:              D2 -> D3                   |
==================================================
```

Listing 5.1: Elements of a Bus Network: Example of a `Stop` object

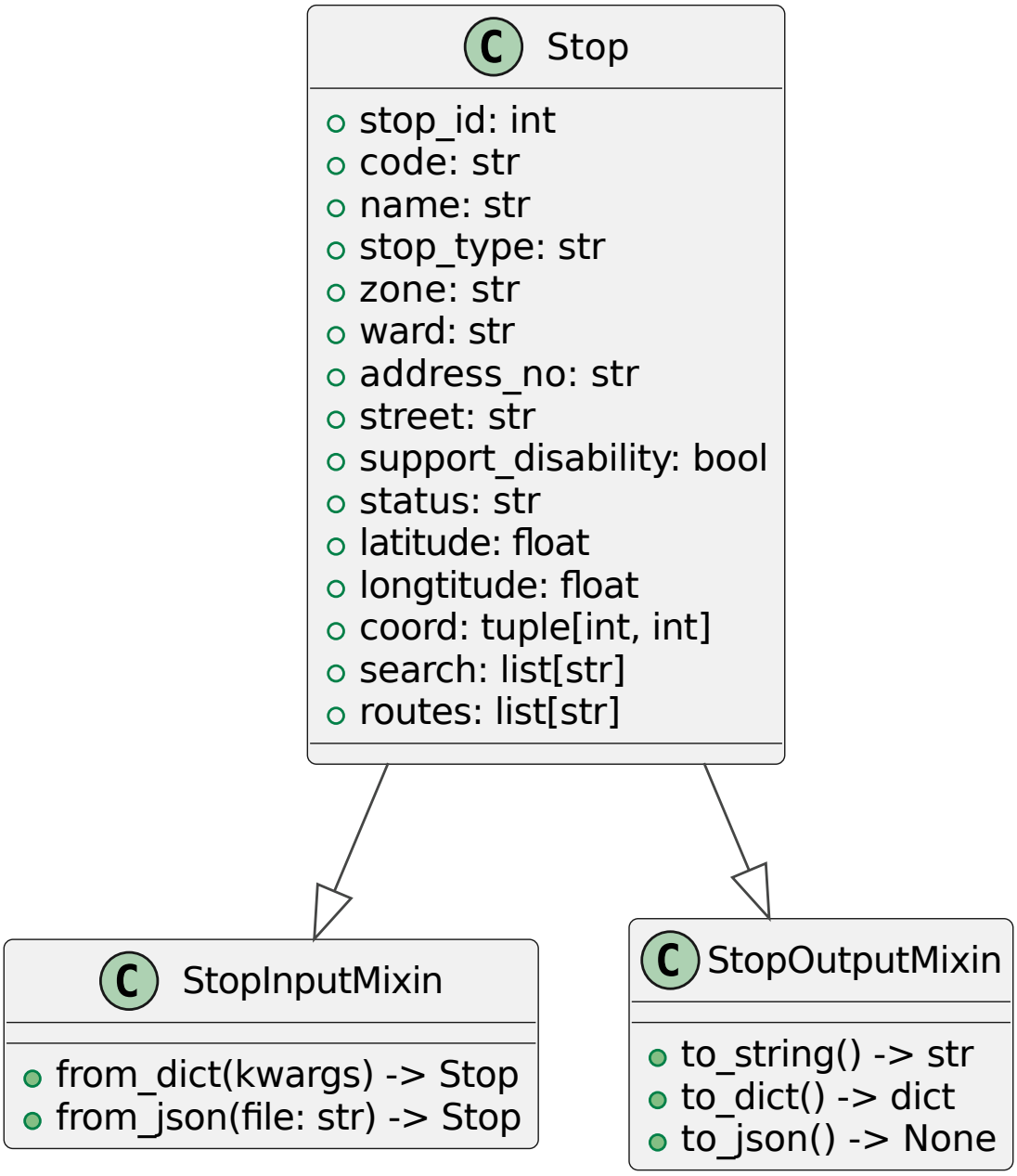


Figure 5.2: Elements of a Bus Network: Stop class diagram

5.2 | Variant (the variant module)

*Variant*s represent bus routes in a specific direction. A bus route has two variants, outbound and inbound. Some bus routes may only have one variant.

5.2.1 | Properties

- ***property*** `route_id: int`
  property `route_var_id: int`
  property `route_ids: tuple[int, int]`
  IDs of a `Variant`.
  A route has two variants, each shares the same `route_id` but different `route_var_id`. `route_ids` is a tuple of `route_id` and `route_var_id`.
- ***property*** `number: str`
  Number of the bus variant. Must be of `str` type to store values such as `61-1, D2`.
- ***property*** `name: str`
  property `short_name: str`
  Name/Short name of the bus variant.
- ***property*** `start_stop: str`
  property `end_stop: str`
  Starting and ending stops of a variant
- ***property*** `distance: float`
  property `running_time: float`
  Distance and running time of the bus variant.

5.2.2 | Methods

- `to_string() -> str`
  Display information of a `Variant` in a string format.
  This function is called by `__repr__()`.
- ***static*** `from_dict(obj: dict) -> Variant`
  Create `Variant` from a Python dictionary.
- `to_dict() -> dict`
  Convert `Variant` to a Python dictionary.
- ***static*** `from_json(file: str) -> Variant`
  Create `Variant` from JSON file. The function loads a JSON file into a Python dictionary, then call `from_dict()`.
- `to_json(file: str) -> None`
  Export `Variant` information to JSON file. The function calls `to_dict()` to convert `Variant` to a dictionary, then dumps it into a JSON file.

```
>>> from variant import Variant
>>> var = Variant.from_json('var.json')
>>> var
========[Route D2, Paris Baguette -> Cressent mall]========
| RouteNo:        D2                                       |
| StartStop:      Paris Baguette                           |
| EndStop:        Cressent mall                            |
| Name:           Luot di                                  |
| ShortName:      Luot di                                  |
| Distance:       3677.0000000000005                       |
| RunningTime:    14                                       |
| RouteIds:       (212, 1)                                 |
============================================================
>>> var.route_id
212
>>> var.route_var_id
1
>>> var.route_ids
(212, 1)
>>> var.to_dict()
{
    'RouteNo': 'D2',
    'StartStop': 'Paris Baguette',
    'EndStop': 'Cressent mall',
    'Name': 'Luot di'
    'ShortName': 'Luot di'
    'Distance': 3677.0000000000005,
    'RunningTime': 14,
    'RouteIds': (212, 1)
}
```

Listing 5.2: Elements of a Bus Network: Example of a `Variant` object

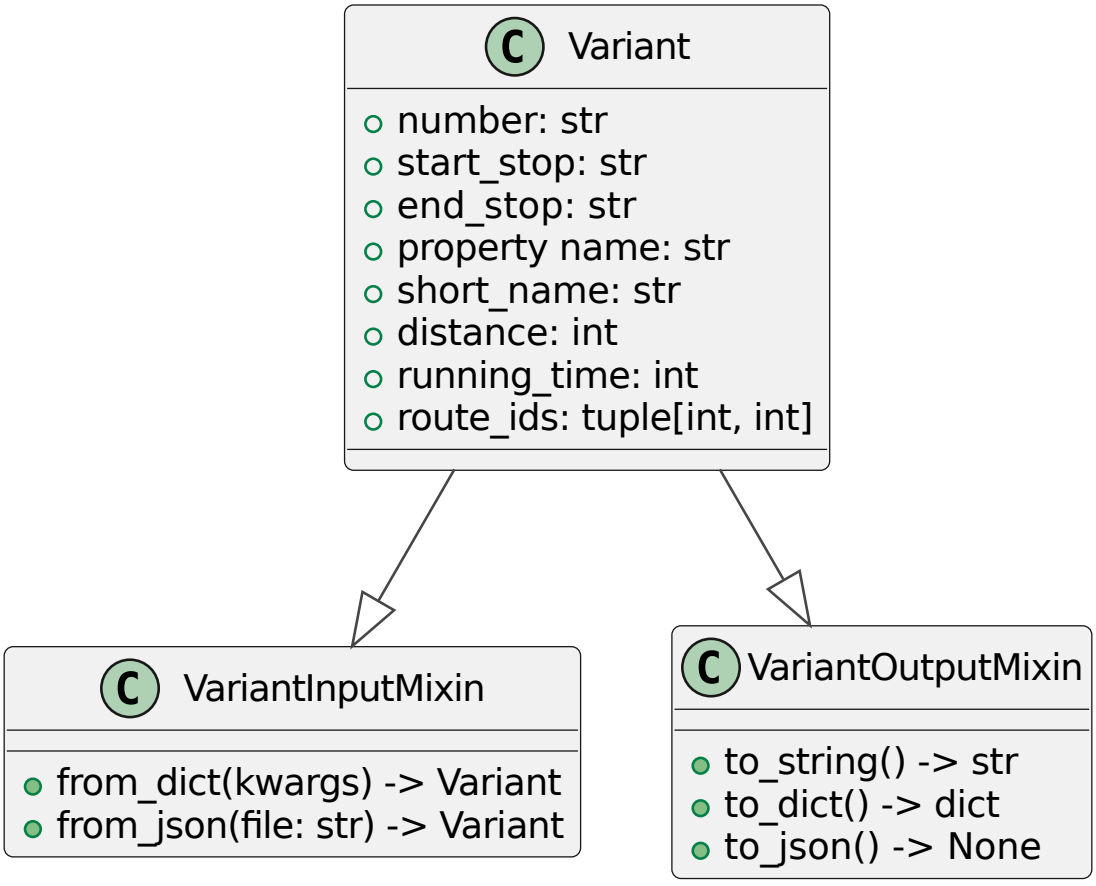


Figure 5.3: Elements of a Bus Network: Variant class diagram

5.3 | Path (the path module)

*Path*s represent the specific path of a bus variant in the form of a LineString.

5.3.1 | Properties

- ***property*** `route_id: int`
  property `route_var_id: int`
  property `route_ids: tuple[int, int]`
  IDs of the route that the `Path` represents.
  A route has two variants, each shares the same `route_id` but different `route_var_id`. `route_ids` is a tuple of `route_id` and `route_var_id`.
- ***property*** `coords: list[tuple[float, float]]` Coordinates of a `LineString` representing the path of the bus variant.

5.3.2 | Methods

- `polysides() -> Iterable[tuple[tuple[float, float], tuple[float, float]]]`
  Return an `Iterable` of tuples of coordinates, representing each segments of the `LineString`.
- `to_string() -> str`
  Display information of a `Path` in a string format.
  This function is called by `__repr__()`.
- ***static*** `from_dict(obj: dict) -> Path`
  Create `Path` from a Python dictionary.
- `to_dict() -> dict`
  Convert `Path` to a Python dictionary.
- ***static*** `from_json(file: str) -> Path`
  Create `Path` from JSON file. The function loads a JSON file into a Python dictionary, then call `from_dict()`.
- `to_json(file: str) -> None`
  Export `Path` information to JSON file. The function calls `to_dict()` to convert `Path` to a dictionary, then dumps it into a JSON file.

```
>>> from path import Path
>>> import math
>>> path = Path.from_json('path.json')
>>> path.route_ids
(212, 1)
>>> len(path.coords)
30
>>> sum(math.dist(p1, p2) for (p1, p2) in path.polysides())
3664.914478222333
```

Listing 5.3: Elements of a Bus Network: Example of a `Path` object

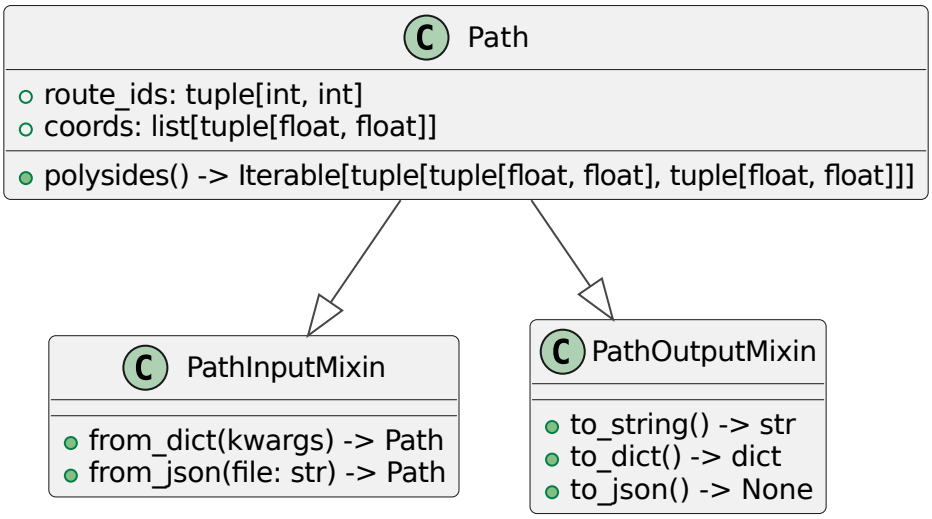


Figure 5.4: Elements of a Bus Network: Path class diagram

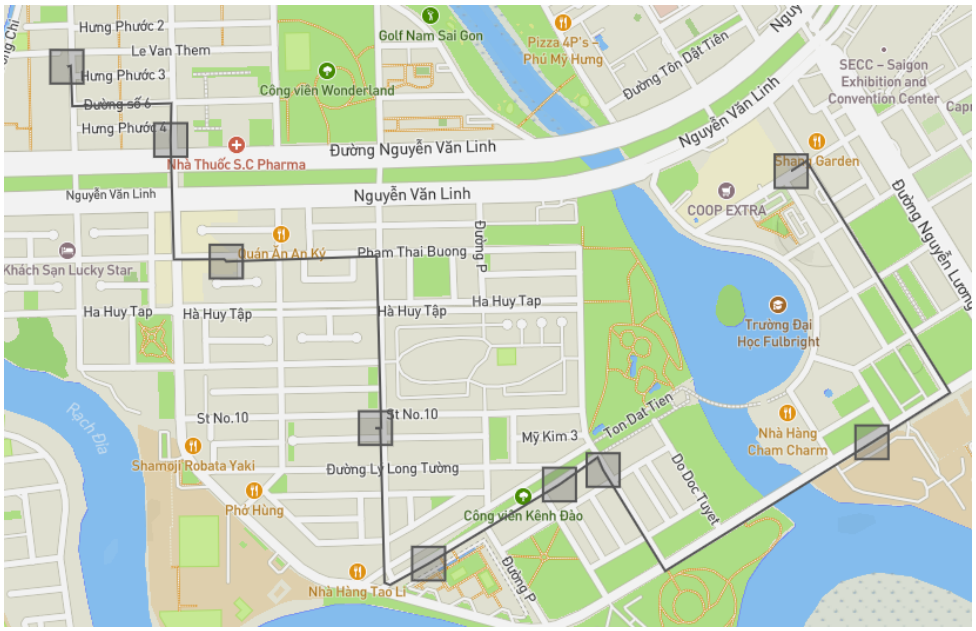


Figure 5.5: Elements of a Bus Network: The variant from *Paris Baguette* to *Cressent mall* has a total distance of approx. 3665km, whilst the distance in database is 3677km. Their relative difference is 0.33%.

Chapter 6

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.*

6.1 | Definition

6.1.1 | 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*.

6.1.2 | 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*.

6.2 | 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.

6.2.1 | *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 6.1: Bus Network Graph: Types of bus *Stops*

Degree distribution

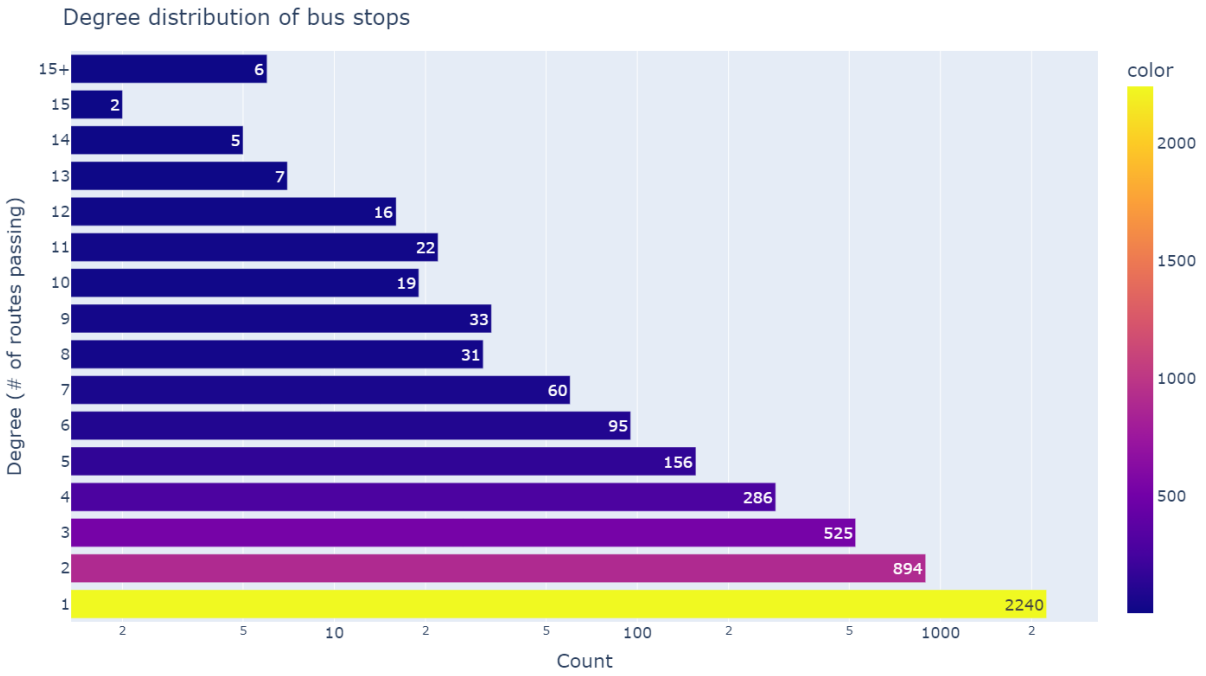


Figure 6.1: EDA: Degree distribution of bus *Stops* (total).

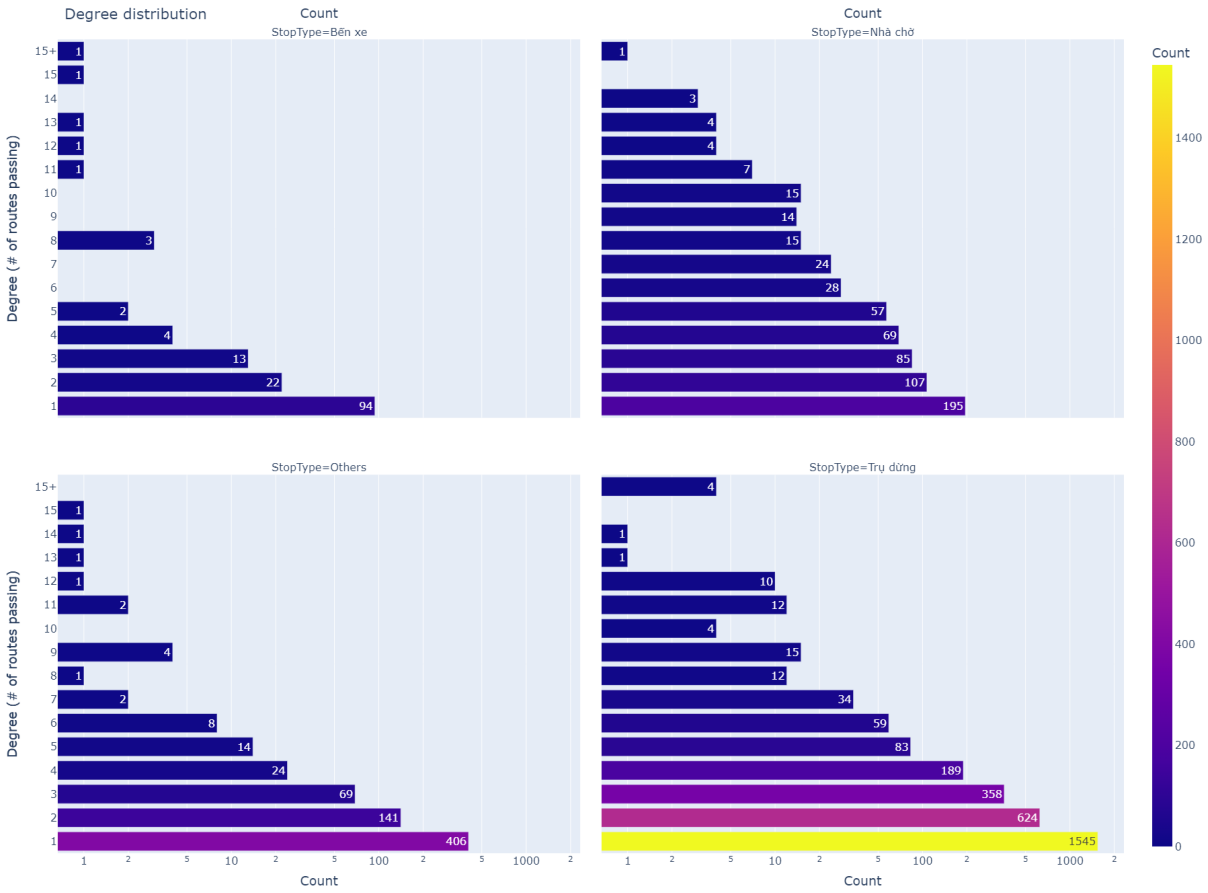


Figure 6.2: EDA: Degree distribution of bus *Stop*s (by `StopType`)

6.2.2 | *Routes*

Relationship between *Route* distance and running time

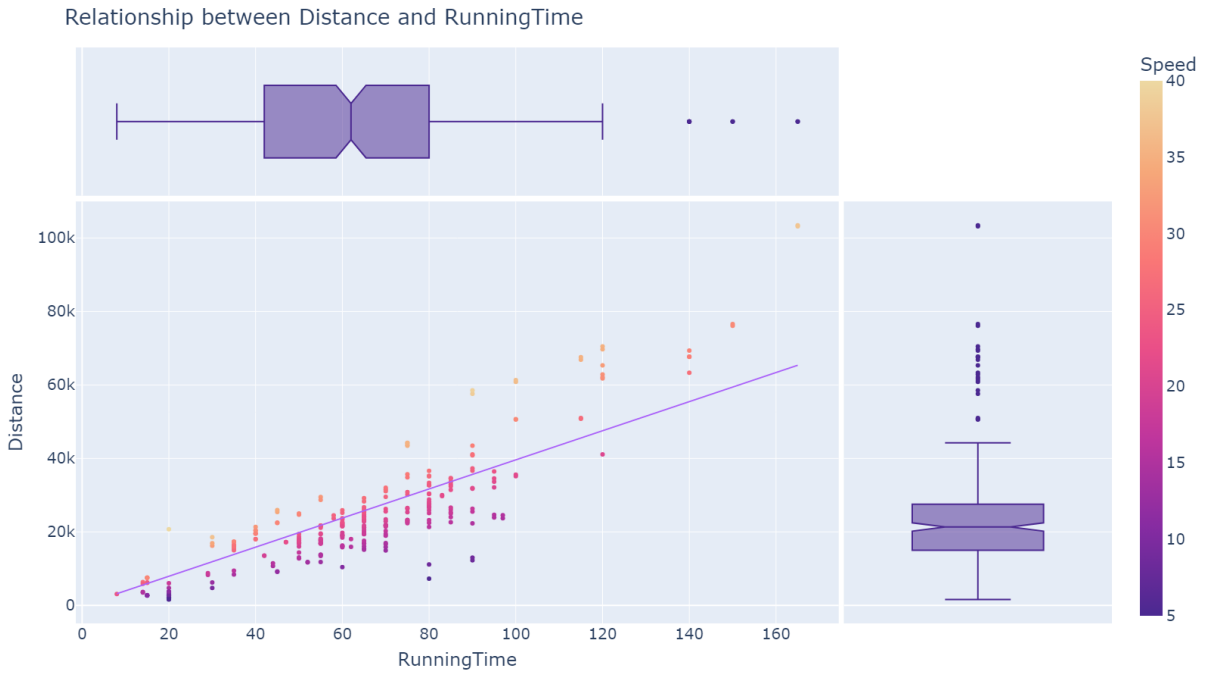


Figure 6.3: EDA: Relationship between *Route* distance and running time

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

6.2.3 | Relationship between *Stops* and *Paths*

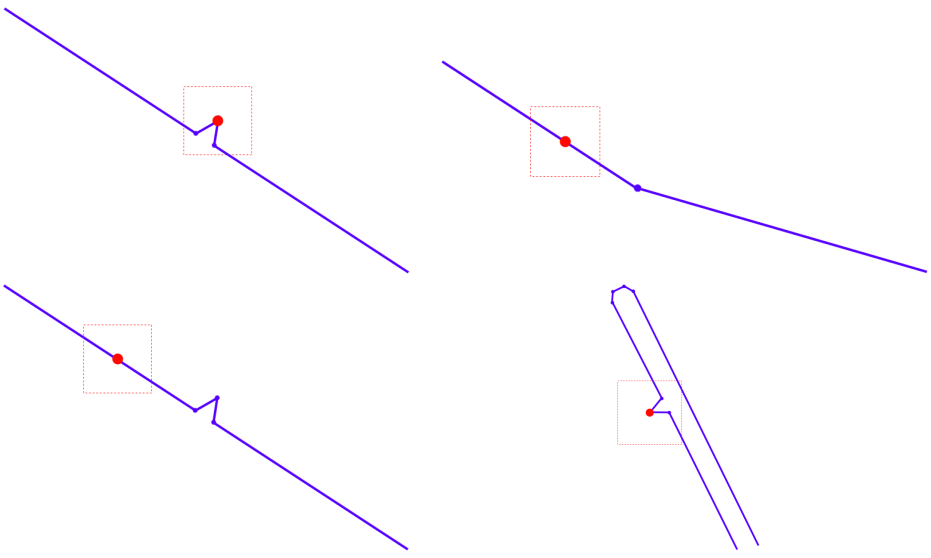


Figure 6.4: EDA: Case 1 - a *Stop* lies on a *Path*

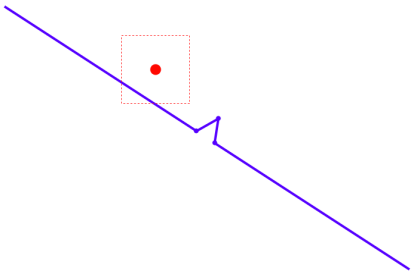


Figure 6.5: EDA: Case 2 - a *Stop* lies at a distant from the *Path*

6.3 | Graph construction algorithm

6.3.1 | 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"**.

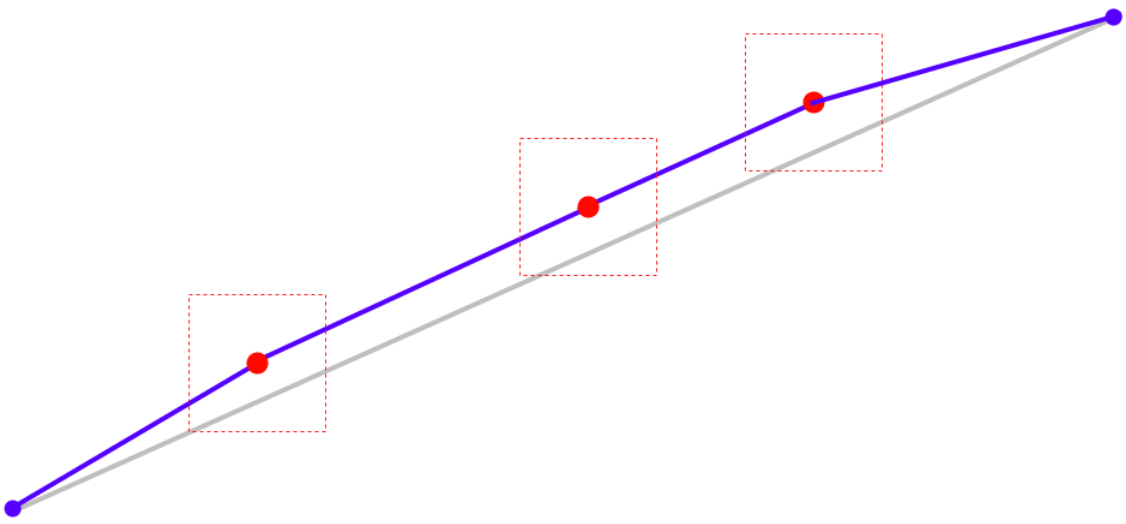


Figure 6.6: Graph Construction: Error fixing, Method 1

Method 2

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

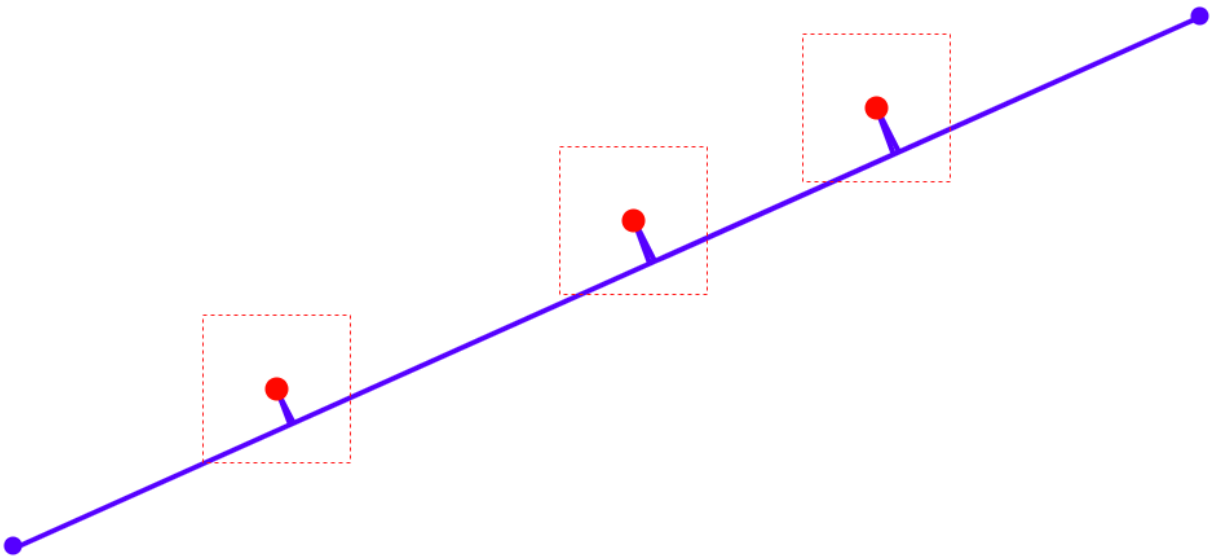


Figure 6.7: Graph Construction: Error fixing, Method 2

We will be using Method 2 in this report.

6.3.2 | 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 *Stop*s 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 $e_i = (\mathbf{p}_{i-1}, \mathbf{p}_i)$ $(1 \leq i \leq m)$.
- 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 *Stop*s $\mathbf{v}_i, \mathbf{v}_{i+1}$, 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.

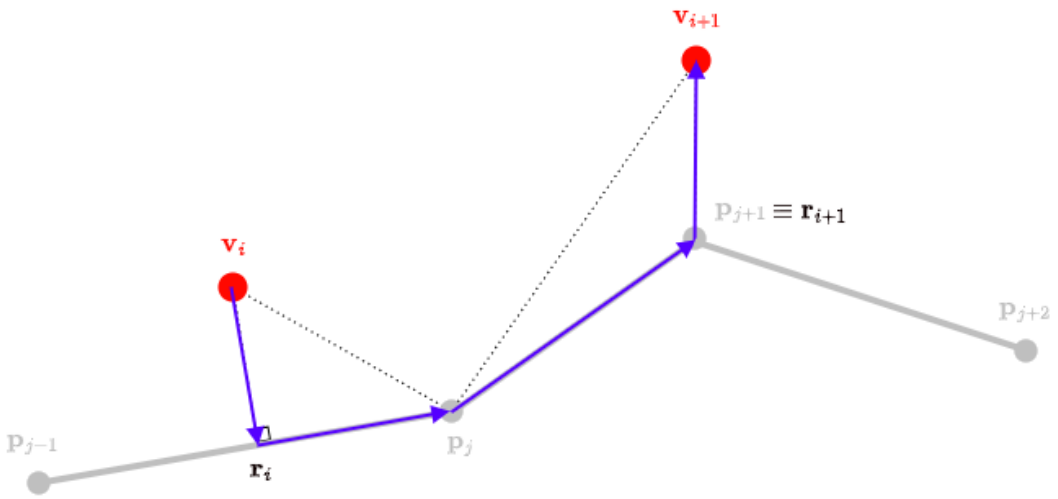
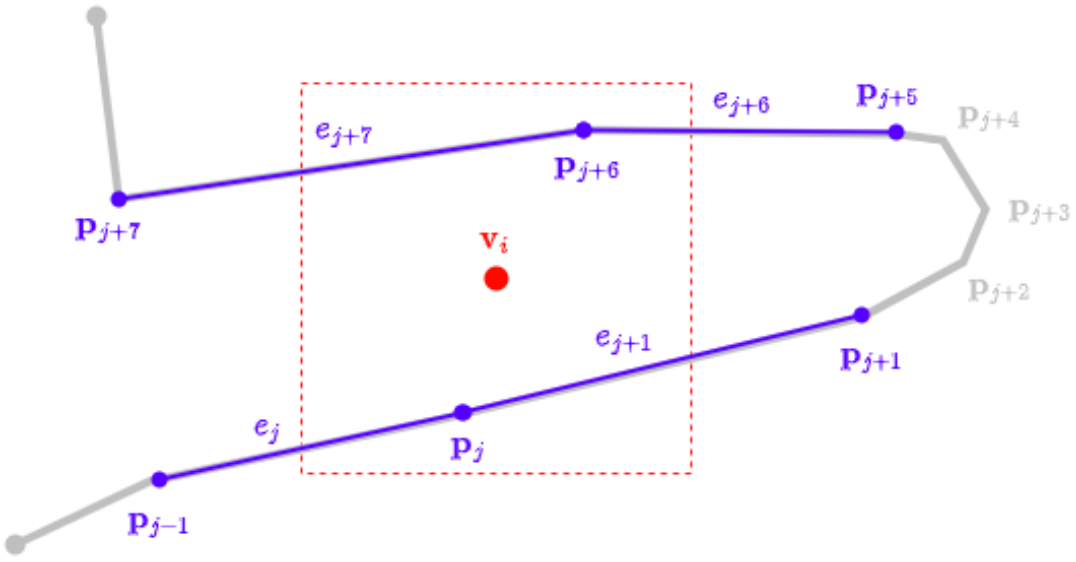


Figure 6.8: Graph Construction: Algorithm I

6.3.3 | 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 6.9: 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 *Stop*s 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 $e_i = (\mathbf{p}_{i-1}, \mathbf{p}_i)$ $(1 \leq i \leq m)$.
- 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.

6.4 | Implementation

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

6.4.2 | `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 6.1: 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 6.2: Graph Construction: `BusNetwork` class inherited from generic `Network`

6.5 | Experiments

6.5.1 | 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 6.1: Bus Graph Experiments: Running time of construction algorithms

6.5.2 | # 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 6.2: Bus Graph Experiments: Stats on # of edge candidates $|E|$ per *Stop* on each *Route*

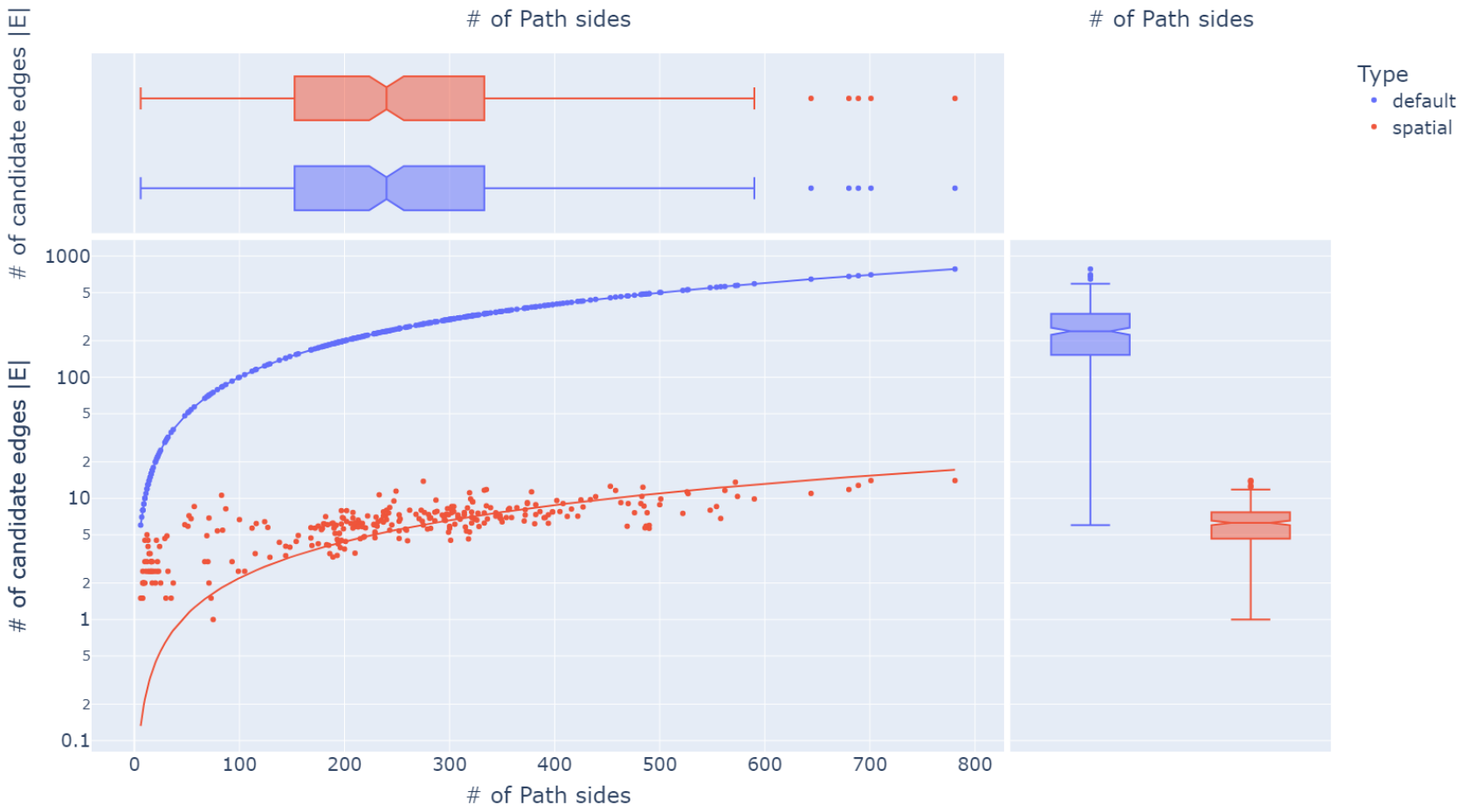


Figure 6.10: Bus Graph Experiments: # of edge candidates $|E|$ per *Stop* on each *Route*

Part B
Shortest Path Problem on Real life Network

Chapter 7

Dijkstra: Shortest Paths and related Centrality Measures

7.1 | SSSP: Single Source Shortest Path problem

7.1.1 | 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, $w(e_i) \geq 0, \forall e_i \in E$

7.1.2 | Dijkstra's Algorithm

In his original paper *"A Note on Two Problems in Connexion with Graphs" (1959)*[10], 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***.***
> *We use the fact that, if* R *is a node on the minimal path from* P *to* Q*, knowledge*

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

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 $O(|V|^2)$

```
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 7.1: Dijkstra: The originally proposed algorithm

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 $O((|E|+|V|)\log|V|)$.

[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

    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))

    return dist, prev
```

Listing 7.2: Dijkstra: Improvements with `PriorityQueue`

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.

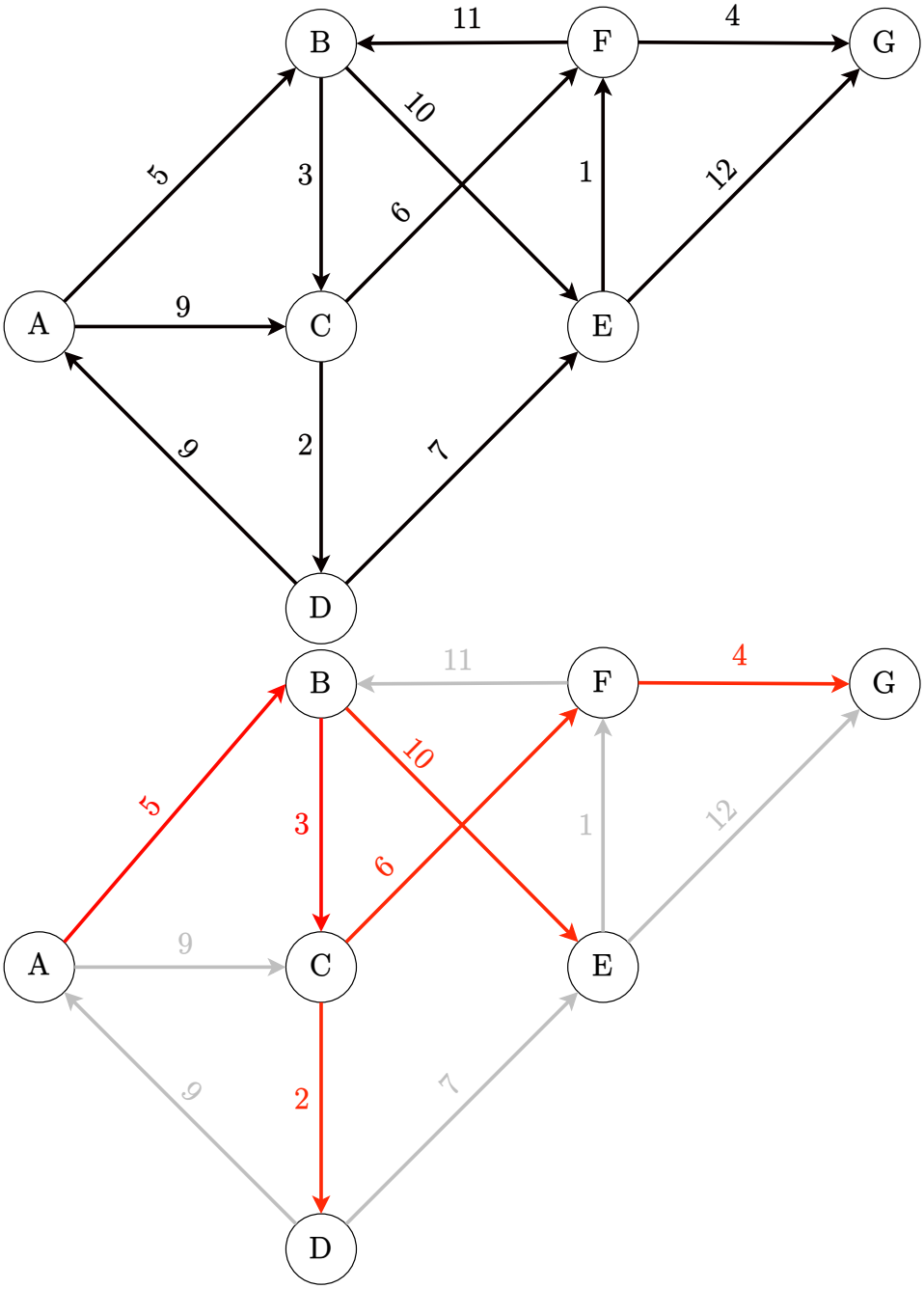


Figure 7.1: Dijkstra: Shortest paths from source node A (red paths) generate the **shortest-path tree**.

7.1.3 | 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 Bidirectional Dijkstra: Implementation, 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.

7.2 | APSP: All Pairs Shortest Path problem

7.2.1 | 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 $(u, v) \in V^2$.

7.2.2 | 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 7.3: Dijkstra: Floyd-Warshall Algorithm for APSP

7.3 | 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*[11]. 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.

7.3.1 | 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:

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

7.3.2 | 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 7.4: Dijkstra: Computing $g(u)$ by iterating across all shortest paths

7.3.3 | 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[12].

7.3.4 | Other Centrality Measures based on Shortest Paths

Closeness Centrality (Alex Bavelas, 1950)[13]

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

Harmonic Centrality (Marchiori and Latora, 2000)[14]

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

(assuming $1/\infty = 0$)
This measure is later independently discovered by Dekker (2005)[15] under the name "valued centrality", and by Rochat (2009).

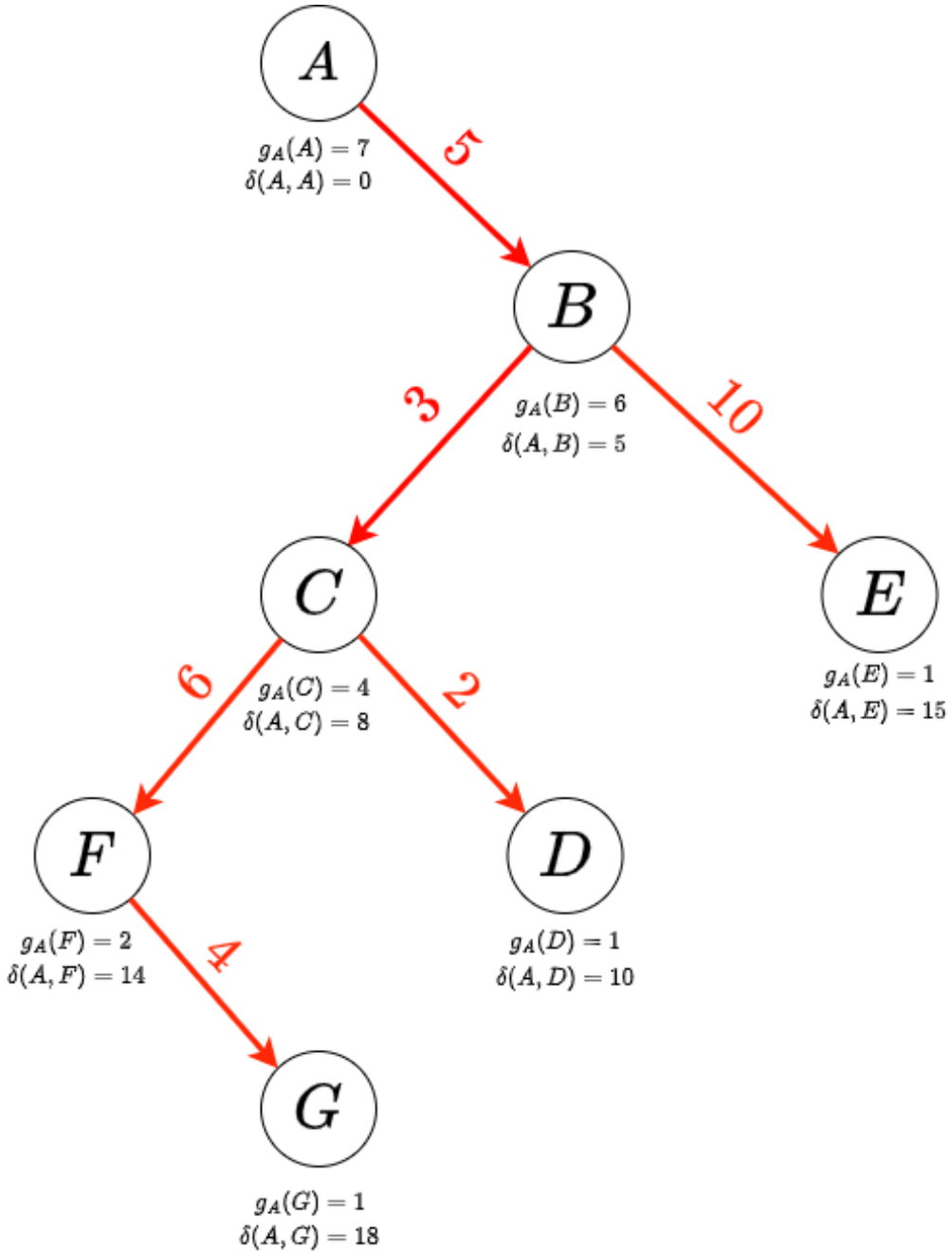


Figure 7.2: Dijkstra: Usage of the **shortest-path tree** to compute $g_s(u)$.

```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 7.5: Dijkstra: Computing $g_s(u)$

Chapter 8

Space-efficient Shortest Path Querying: Bidirectional Dijkstra, Contraction Hierarchy

Dijkstra's Algorithm solves the single-source-all-destination problem by extracting subsequent nodes in order of distance from the source. Thus, every node is guaranteed to be searched, from nearest to furthest. In other words, the search space of Dijkstra's algorithm is strictly $|V|$.
In reality, navigation apps have to process thousands, if not millions of nodes and edges; they also have to perform a huge number of single-source-single-destination queries simultaneously. Hence, it is insufficient to let a query search through all of the possible nodes in the search space.
In this chapter, we will present modifications on the original Dijkstra's algorithm that have shown to perform well on road networks, by taking advantage of the special network structure:

- **Single-destination Dijkstra**. Dijkstra is stopped whenever the destination is reached.
- **Bidirectional Dijkstra**. Perform two concurrent instances of Dijkstra from the source and destination.
- **Contraction Hierarchies**. Assign to each node a level of importance. With sufficient 'shortcuts', any shortest path can be represented by a 'up-path' (nodes of increasing importance) followed by an 'down-path' (nodes of decreasing importance).

8.1 | Bidirectional Dijkstra

8.1.1 | Intuition

Bidirectional Dijkstra performs two instances of Dijkstra algorithm, one from the source node, and one from the destination node (on the transposed graph). Each instance stops after reaching half of the optimal distance, therefore reducing the search space from $O(b^d)$ to $O(b^{d/2})$, where b is the branching factor, and d is the optimal distance.[16]

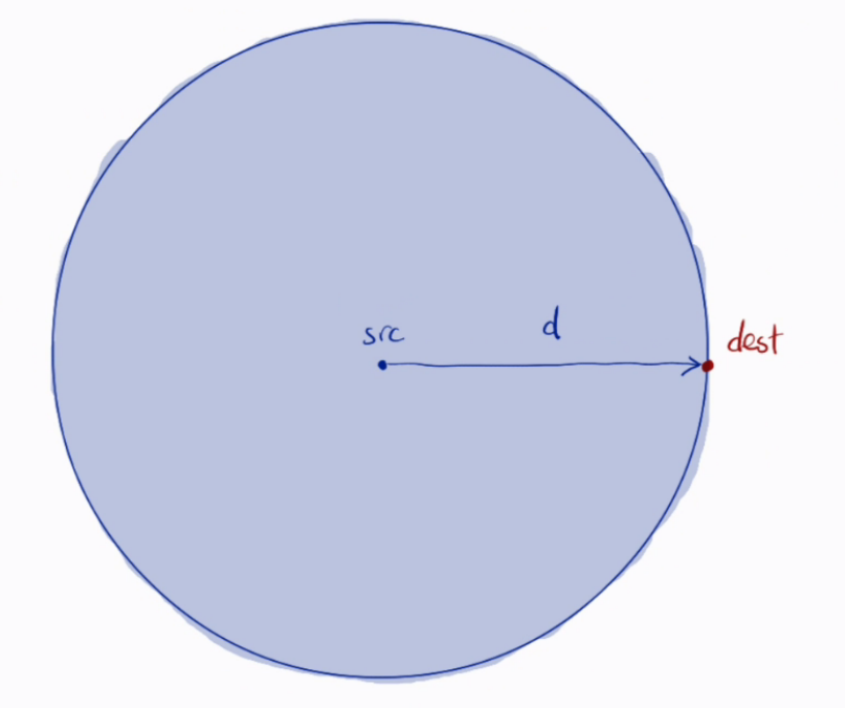


Figure 8.1: Search-space efficiency: Dijkstra expands nodes from source in the form of a sphere, the area of which is πd^2.

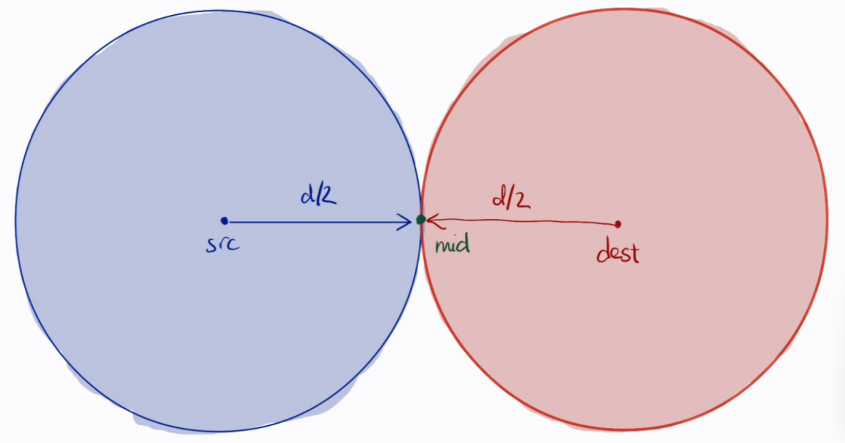


Figure 8.2: Search-space efficiency: Bidirectional Dijkstra expands nodes from both the source and the destination in the form of two circles, the area of each of which is $\pi(d/2)^2 = \frac{1}{4}\pi d^2$. Hence the total search area is $\frac{1}{2}\pi d^2$, which is half of Dijkstra's.

8.1.2 | Implementation

```
from priority_queue import PriorityQueue

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

    prev_s, prev_t = {}, {}
    dist, mid = INFINITY, None
    for u in V:
        dist_s[u] = dist_t[u] = INFINITY

    dist_s[src], dist_t[dest] = 0, 0
    pq_s, pq_t = PriorityQueue(), PriorityQueue()
    pq_s.put((0, src))
    pq_t.put((0, dest))

    while True:
        is_running = False

        if not pq_s.empty():
            is_running = True
            _, u = pq_s.get()
            for v in neighbours(u):
                if dist_s[v] > dist_s[u] + w(u, v)
                    dist_s[v] = dist_s[u] + w(u, v)
                    prev_s[v] = u
                    pq_s.put((dist_s[v], v))

            dist, mid = min((dist, mid), (dist_s[u] + dist_t[u], u))

        if not pq_t.empty():
            is_running = True
            _, u = pq_t.get()
            for v in neighbours_rev(u):
                if dist_t[v] > dist_t[u] + w(v, u)
                    dist_t[v] = dist_t[u] + w(v, u)
                    prev_t[v] = u
                    pq_t.put((dist_t[v], v))

            dist, mid = min((dist, mid), (dist_s[u] + dist_t[u], u))

        if not is_running:
            break

    return dist, prev_s, prev_t, mid
```

Listing 8.1: Bidirectional Dijkstra: Implementation

8.2 | Contraction Hierarchies

8.2.1 | Intuition

Traffic roads are usually divided according to their significance in the network: motorways (red), trunk (dark orange), primary (orange), secondary (yellow), tertiary (white), etc. When travelling from one point to another, we may expect to travel from less "busier" roads (usually tertiary or secondary roads) to more important ones, (usually a trunk highway or a primary), then back to smaller roads to reach the destination.
Although the above intuition is not always strictly true, it gives a heuristic approach to finding end-to-end shortest path:

- Each node u (instead of edges or roads) is given an arbitary score of importance $h(u)$, hence the name *"hierarchies"*.
- At any time, the chosen node is given the lowest score (or the lowest hierarchy), then it is removed from the graph. Shortcuts are added to preserve the shortest paths between each pairs of nodes in the remaining graph, hence the name *"contraction"*.
- From the original graph G, create two subgraphs: a graph consists of only **upward** arcs G^+, and a graph consists of only **downward** arcs G^-. An upward arc is an arc from a lower node to a higher node in the hierarchy.
- Now to query a shortest path from s to t, run two concurrent instances of Dijkstra: one from node s on graph G^+, and one from node t on graph G^-.

Unlike A* or Bidirectional Dijkstra, Contraction Hierarchies is not a well-defined "algorithm", but rather a speed-up technique, that consists of two phases: *preprocessing* and *querying*. It can be thought of as a **data structure** that organises nodes in hierarchical order and supports faster querying.

Figure 8.3: Hierarchical Nature of Road Networks: The optimal path starts from the road with least importance (white), subsequently to roads with higher importance (yellow - orange - red), then down to a road with less importance (white).

8.2.2 | The high-level algorithm and implementation decisions

Contraction Hierarchies algorithm - Preprocessing

- Let $G = (V, E)$ be the original graph
- Let $E_{\text{shortcuts}}$ be the set of shortcuts to be added. Initially, $E_{\text{shortcuts}} = \emptyset$
- Let $\ell = 0$
- While $|V| > 0$:
  - $\ell := \ell + 1$
  - Select a node $x \in V$ to be contracted. (2)
  - Let $Y :=$ the set of incoming nodes to u, $Z :=$ the set of outcoming nodes from u.
  - For every pair $(y_i, z_j) \in Y \times Z$:
    - If the removal of u increases the shortest path from y_i to z_j: (1)
      - $E := E \cup (y_i, z_j)$
      - $w(y_i, z_j) := w(y_i, u) + w(u, z_j)$
  - Set $h(u) := \ell$
  - Remove u from V
- The **upward** graph is $G^+ = (V^+, E^+)$, where

$$E^+ := \{(u, v) \in E \cup E_{\text{shortcuts}} \mid h(u) < h(v)\}$$

- The **downward** graph is $G^- = (V^-, E^-)$, where

$$E^- := \{(u, v) \in E \cup E_{\text{shortcuts}} \mid h(u) > h(v)\}$$

Contraction Hierarchies algorithm - Querying from s to t

- Run Bidirectional Dijkstra on source s and destination t to receive path $\Pi = [e_1, e_2, \cdots, e_k]$.
- Let π be the real path from s to t. Initially $\pi = [\,]$.
- For each edge e_i in path Π:
  - If $e_i \in E_{\text{shortcuts}}$:
    - Let π_0 be the real path represented by the shortcut e_i.
    - Append π_0 to π
  - Else
    - Append e_i to π

The algorithm description leaves us with decisions to be made:

1. How can we determine if the removal of a node causes changes in any shortest paths?
2. Which nodes should we contract at each iteration?

8.2.3 | Decision 1: Tracking shortest paths changes after node removal

As long as we preserve the shortest path of every pair of nodes in G, the query phase is guaranteed to be accurate. Thus, it does not hurt to add **unnecessary shortcuts** into the graph, hence our removal test can accept **false positives** as a tradeoff for efficiency.

So our check goes as follows: For every incoming node y_i, run Dijkstra from y_i without node x. Now because we accept **false positives**, we only need to run Dijkstra for a specific (small) number of iterations. For each outgoing node z_j, if $\delta'(y_i, z_j) > \delta(y_i, z_j)$, then a shortcut $(y_i, z_j, \delta(y_i, z_j))$ is added. Here δ represents the original shortest path, δ' represents the shortest path after removing node x.

```
def shortcuts_added_at(node: int, local_steps: int = 50):
    """
    Return the list of shortcuts with the Dijkstra engine
    running within a limit of (default = 50) steps.
    """
    lefts = net.adjs_rev[node]
    rights = net.adjs[node]

    net.hide_node(node)
    for left, left_connector in lefts.items():
        engine = NetworkDijkstraLocalSteps(limit=local_steps) \
            .from_src(net=net, src=left)

        for right, right_connector in rights.items():
            dist = engine.dists.get(right, INFINITY)
            if dist > left_connector.weight + right_connector.weight:
                shortcuts.append((left_connector, right_connector))

    net.unhide_node(node)

    return shortcuts
```

Listing 8.2: Contraction Hierarchies: Shortcuts added given the removal of a node

8.2.4 | Decision 2: Node contraction order

Strategy 2.1: Random contraction order

The strategy is as simple as selecting a random node to be contracted. Nonetheless, it has been proven theoretically that the number of shortcuts added and the expected search space is better than a genuine Bidirectional Dijkstra[17].

```
def build_contraction_net_random():
    level = {}
    for node in random.sample(nodes, n):
        level[node] = len(level)
        contract(node, shortcuts_added_at(node))
```

Listing 8.3: Contraction Hierarchies: Implementation of the random node order heuristic

	Preprocessing	Querying
Bidirectional Dijkstra	0	$0.25n^2$
C.H. (random, $p = 0.5$)	$0.25(\sqrt{n}\log_2\sqrt{n})^2 + 0.67n$	$4.16\sqrt{n}\log\sqrt{n} + 2.00n$

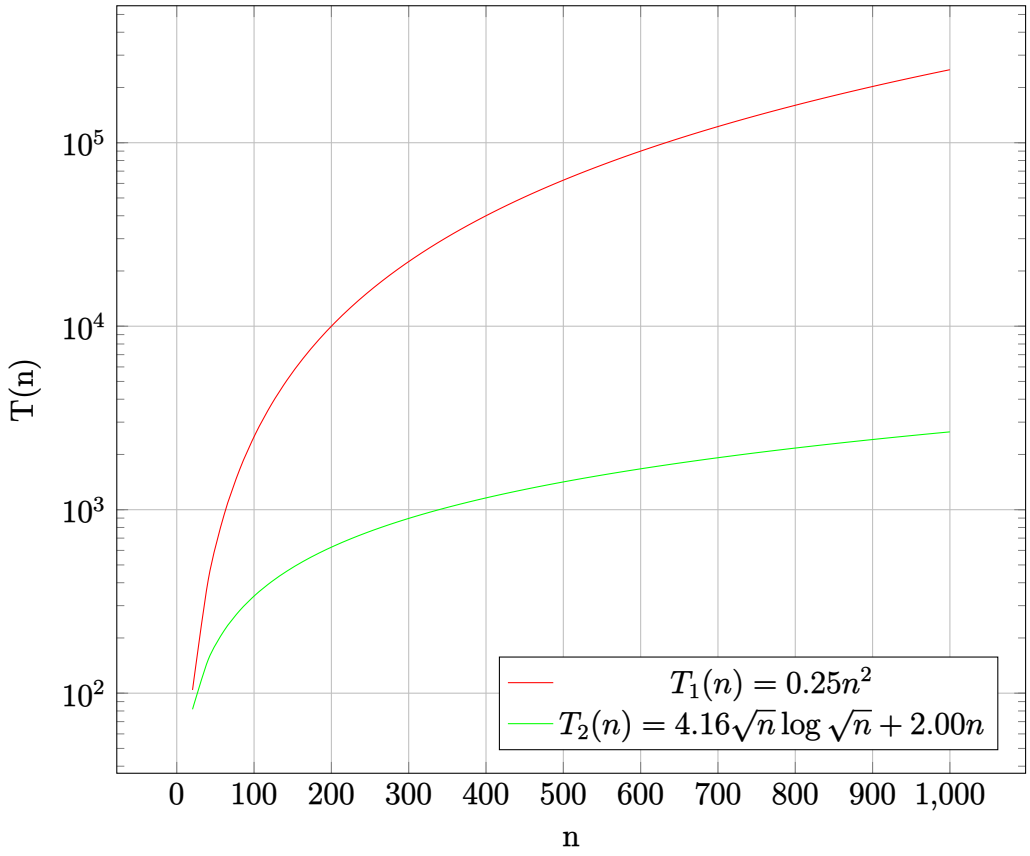


Figure 8.4: Search-space Comparison between Bidirectional Dijkstra and Contraction Hierarchies with Random node contraction.

the expected search space for this strategy is $O(\sqrt{n}\log\sqrt{n})$, which is asymptotically better than that of a Bidirectional Dijkstra search of $O(n^2)$.

Strategy 2.2: Least edge difference (ED)

Edge difference (ED)

The edge difference of node x, $\text{ED}(x)$, is defined as $S(x) - E(x)$, where
- $S(x)$ = the number of shortcuts to be added after the contraction of node x.
- $E(x)$ = the number of arcs on the original graph incident to node x.

The intuition here is we want to minimise the number of shortcuts added to the graph, therefore making the querying phase more efficient. A heuristic is then choosing the node with the **least computed edge difference** at every iteration.
Note that in Decision 1: Tracking shortest paths changes after node removal, we may accept unnecessary shortcuts, thus the computation of edge difference may result in a value exceeding the actual ED.

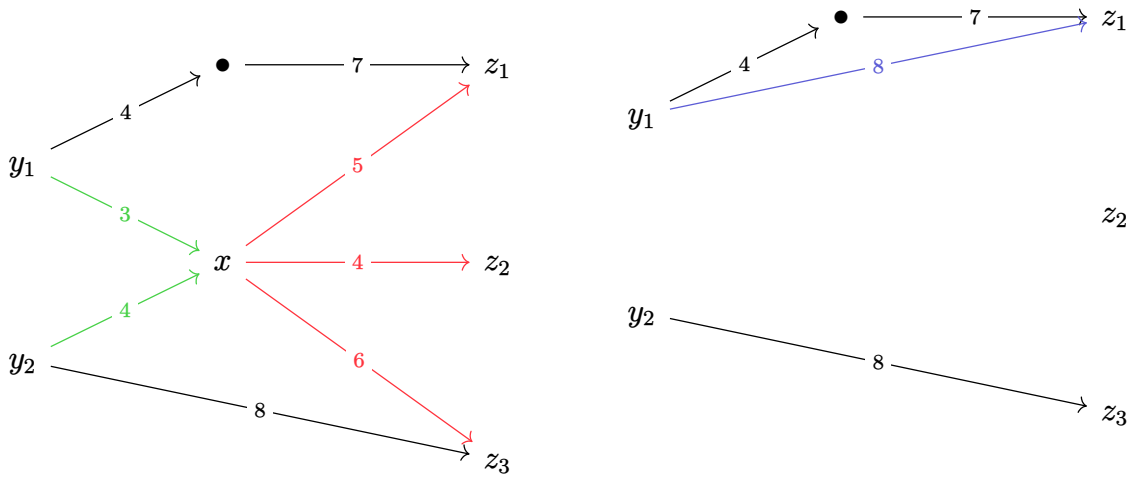


Figure 8.5: Edge Difference: $\mathrm{ED}(x) = 1 - 5 = -4$

All EDs might change when a node is contracted; however, it is infeasible to keep an up-to-date ED of every node at every iteration. We provide in this report two sub-strategies[18] using the least edge difference heuristic[1].

1. **Periodic ED update**. Recompute ED for all nodes in V every k iterations.
2. **Lazy ED update**. Recompute ED for the currently extracted node.

We will present the performance comparisons between the above algorithms in successive chapters.

[1]Referenced from the Course 'Efficient Route Planning' given by the University of Freiburg.

```
def build_contraction_net_periodic_ED(k: int):
    costs_of = {node: ED(node) for node in nodes}
    level = {}

    while costs_of:
        ranked_nodes = sorted(
            list(costs_of.keys()),
            key=lambda x: costs_of[x]
        )

        for node in ranked_nodes[:k]:
            level[node] = len(level)
            contract(node, shortcuts_added_at(node))
            costs_of.pop(node, 0)

        costs_of = {node: ED(node) for node in costs_of.keys()}
```

Listing 8.4: Contraction Hierarchies: Implementation of **Periodic ED update** heuristic

```
def build_contraction_net_lazy_ED():
    pq = PriorityQueue()
    level = {}

    for node in net.nodes:
        pq.put((ED(node), node))

    while not pq.empty():
        _, node = pq.get()
        shortcuts = shortcuts_added_at(node)
        cost = ED(node)

        if (not pq.empty()) and cost > pq.queue[0][0]:
            pq.put((cost, node))
            continue

        level[node] = len(level)
        contract(node, shortcuts)
        lvl += 1
```

Listing 8.5: Contraction Hierarchies: Implementation of **Lazy ED update** heuristic

Chapter 9

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.*

9.1 | Implementation of Dijkstra algorithm and its uses

9.1.1 | Generic `NetworkDijkstra` class

```python
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 9.1: 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.

9.1.2 | 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 9.2: Graph Analysis: `BusNetworkDijkstra` class inherited from generic `NetworkDijkstra`

9.1.3 | NetworkAnalysisBetweenness class

```
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 9.3: 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 9.4: 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.
- `top_scores(k: int) -> list[tuple[int, int]]`
  Return k-most influential *Stop*s with their corresponding scores.

9.2 | Implementation of search-space efficient Shortest path algorithms

See the `network.shortest_paths` module.

9.3 | Experiments on real data

9.3.1 | 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 9.1: Bus Graph Experiments: Running time of analysis algorithms

9.3.2 | 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 9.2: Bus Graph Experiments: Top $k = 10$ most influential *Stops*

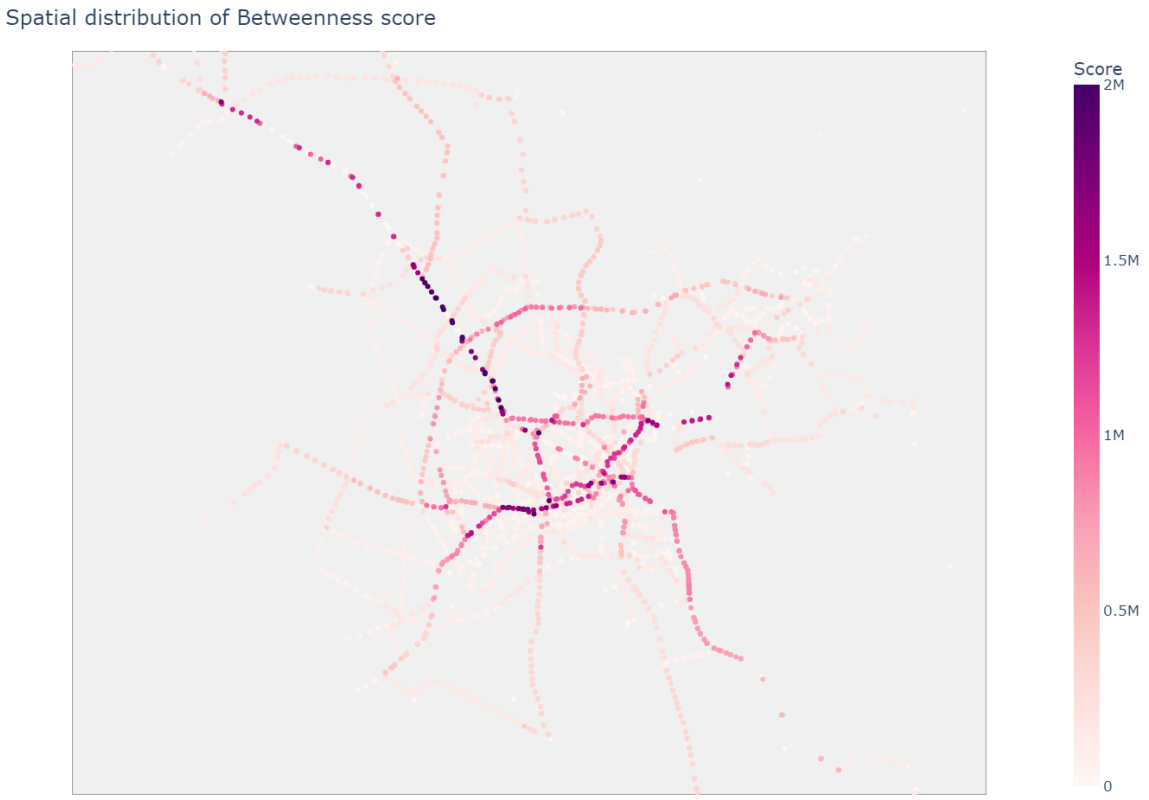


Figure 9.1: Bus Graph Experiments: Spatial distribution of Betweenness score

9.3.3 | Shortest Path algorithms

The time to run a single shortest path query is negligible, hence we will use search-space size to compare the performance between algorithms.
We generated 3000 different pairs of source and destination nodes, then feed it into 5 different shortest path algorithms:

- Dijkstra (single-destination)
- Bidirectional Dijkstra
- Contraction Hierarchies, random node order
- Contraction Hierarchies, node order by least ED, periodically updated after every 100 nodes
- Contraction Hierarchies, node order by least ED, lazily updated

After excluding invalid pairs (shortest path $= \infty$), we retrieve the search space of each algorithm, and then plot the search space size with the actual distance.

	Distance	Search space size				
		Dijkstra	*Bidir. Dijkstra*	*CH (random)*	*CH (periodic)*	*CH (lazy)*
count	2946.00	2946.00	2946.00	2946.00	2946.00	2946.00
mean	72.26	2481.48	611.56	278.29	125.60	68.76
std	43.65	1233.94	380.82	93.87	55.22	20.24
min	1.03	5.00	5.00	9.00	5.00	6.00
25%	42.08	1470.50	290.25	214.00	86.00	56.00
50%	61.21	2642.50	570.50	296.00	118.00	71.00
75%	92.35	3546.00	887.00	349.00	155.00	84.00
max	310.85	4361.00	1669.00	490.00	346.00	126.00

Table 9.3: Bus Graph Experiments: Statistics of search space size of different algorithms

Dijkstra vs. Bidirectional Dijkstra

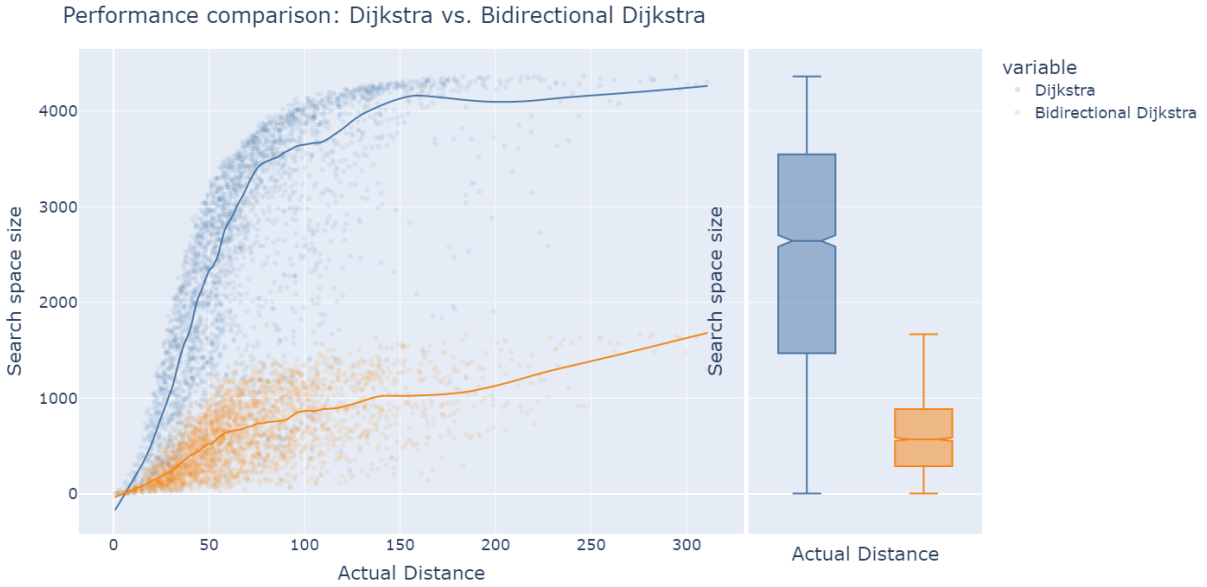


Figure 9.2: Bus Graph Experiments: Performance comparison between Dijkstra and Bidirectional Dijkstra

- The performance of Dijkstra is proportional to the distance.
- The performance of Bidirectional Dijkstra is also proportional to the distance, but with a smaller factor of around 1/4 of that of Dijkstra.
- When the actual distance gets larger, the growth performance slows down, probably due to the exhaustion of nodes (i.e. most of the nodes are visited already).

Contraction Hierarchies variants

	Random node order	Periodic ED update	Lazy ED update
# of nodes	4397.00	4397.00	4397.00
# of shortcuts	54210.00	13567.00	7447.00
Density	12.33	3.09	1.69

Table 9.4: Bus Graph Experiments: Added shortcuts by Contraction Hierarchies variants

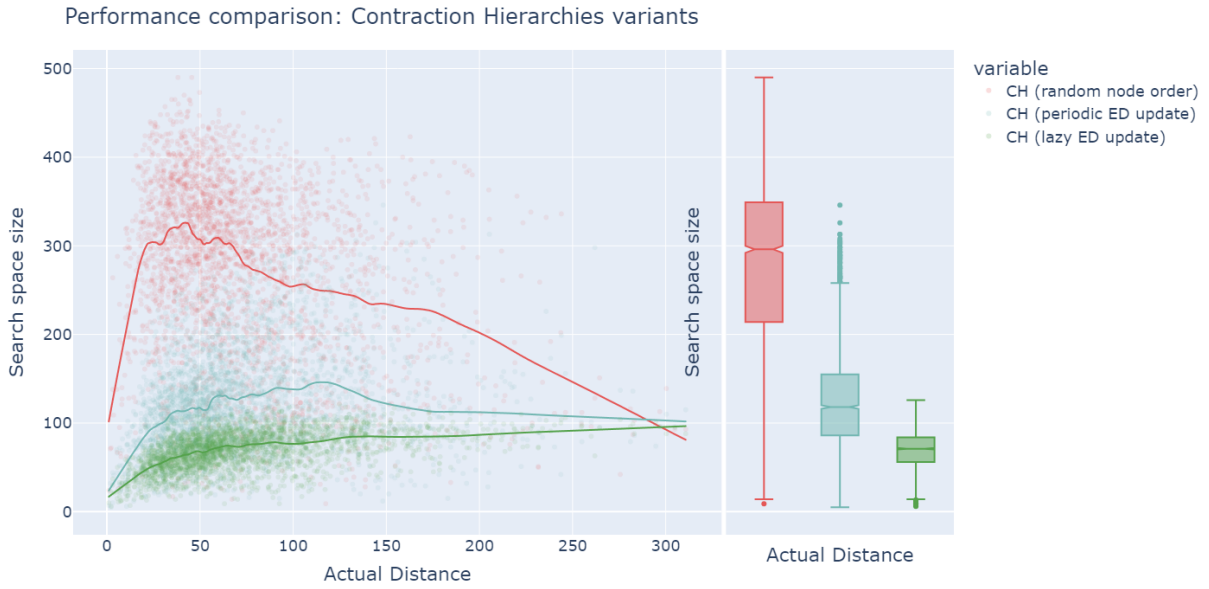


Figure 9.3: Bus Graph Experiments: Performance comparison between Contraction Hierarchies variants

- All contraction hierarchies variants have their performance peaked when the distance is medium-length, whilst, at larger distances, the three variants' performance converges. This can be explained by added shortcuts, providing a faster search for medium-length paths.
- The 'lazy ED update' variant is the most efficient of the three, probably since this variant added the least shortcuts out of the three (The density of the 'lazy ED update' variant is only 1/7 of that of the 'random node order' variant).

Bidirectional Dijkstra vs. Contraction Hierarchies

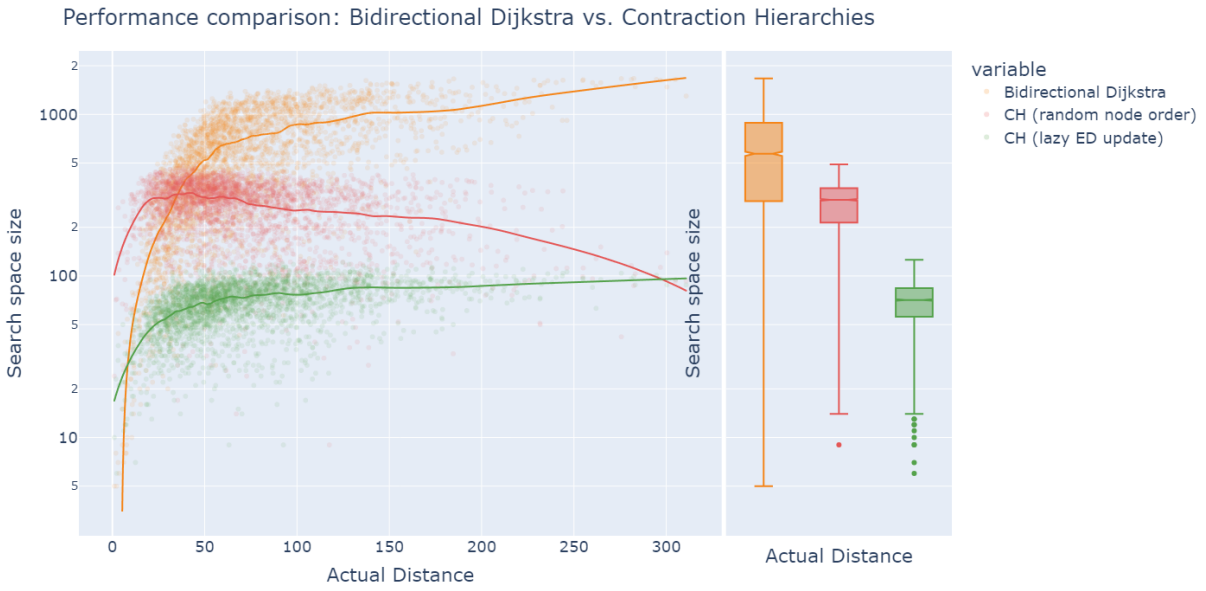


Figure 9.4: Bus Graph Experiments: Performance comparison between Bidirectional Dijkstra and Contraction Hierarchies

- Recall that the search space size of Bidirectional Dijkstra is $O(n^2)$, whilst that of Contraction Hierarchies is $O(\sqrt{n}\log\sqrt{n})$
- Contraction Hierarchies generally have large overhead, whilst Bidirectional Dijkstra outperforms at smaller distances.

9.4 | Module diagrams

9.4.1 | `network` module diagram

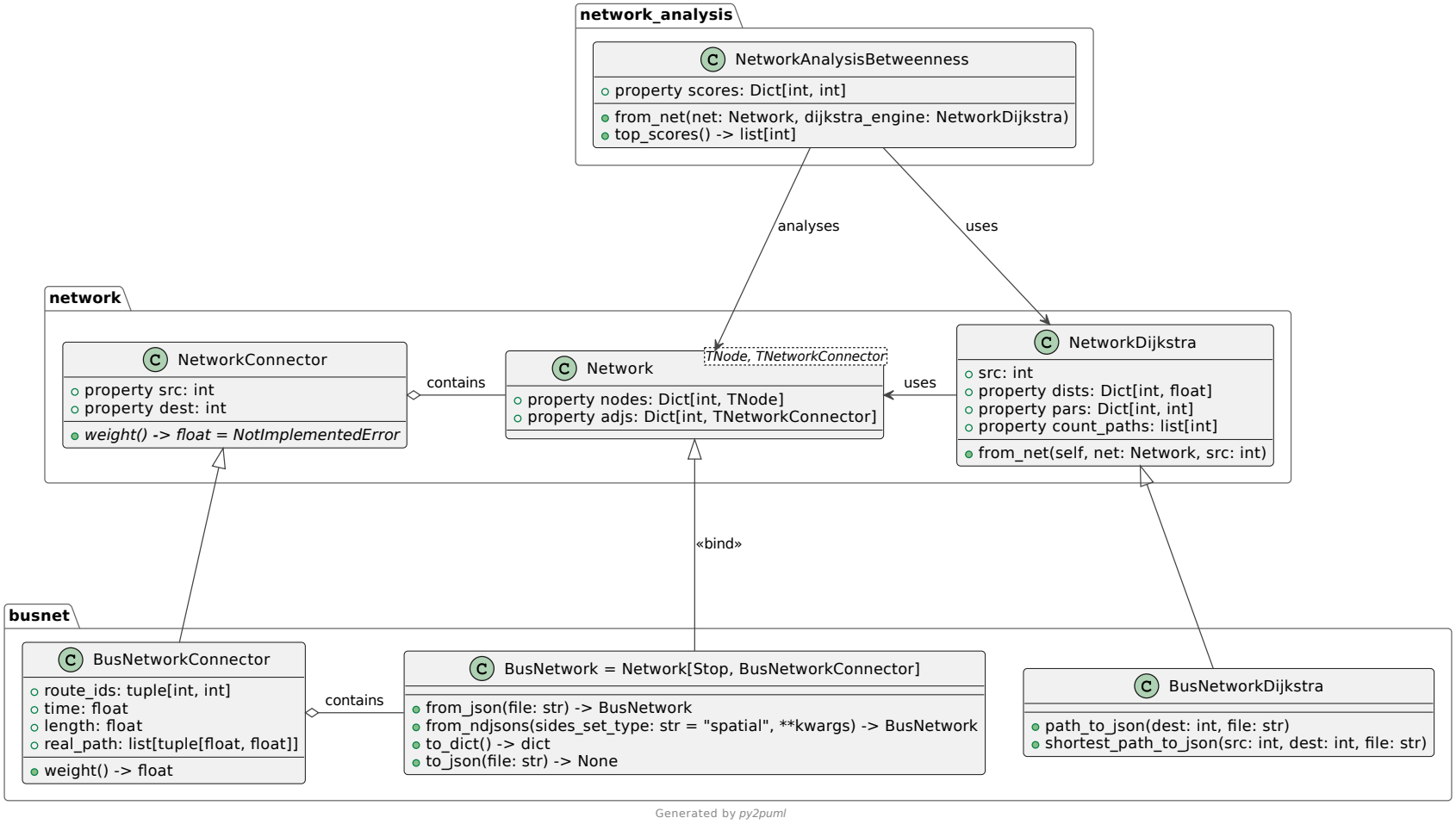


Figure 9.5: Bus Network Analysis: `network` module diagram

9.4.2 | `network.shortest_paths` module diagram

Figure 9.6: Bus Network Analysis: `network.shortest_paths` module diagram

Part C
Querying by Natural Language

Chapter 10

Querying List of Objects

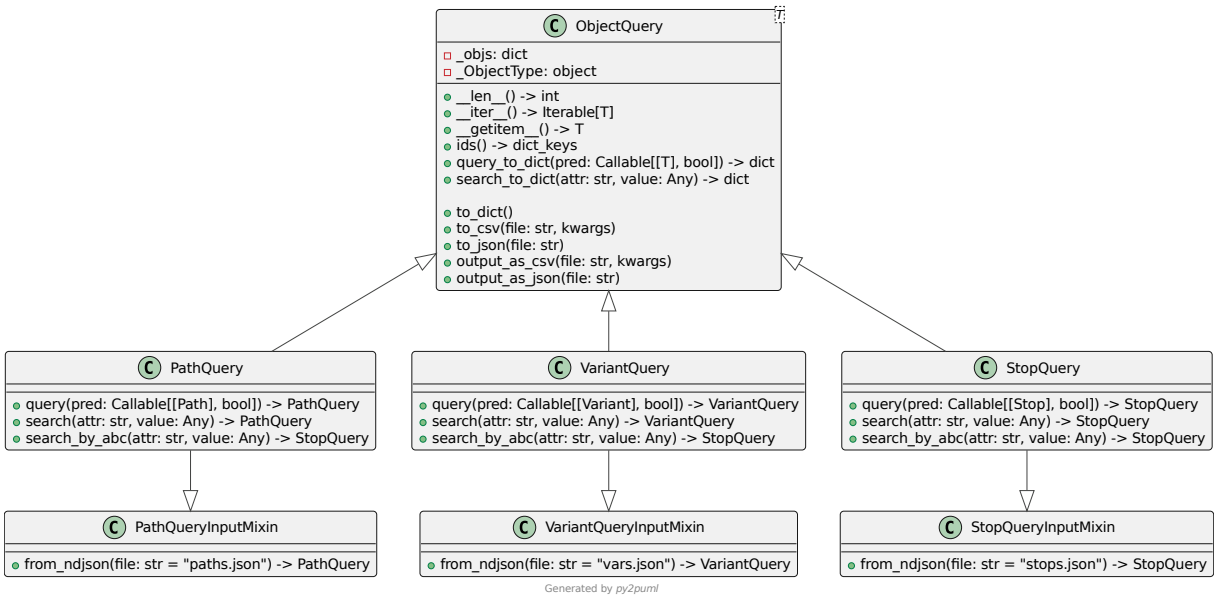


Figure 10.1: Querying List of Objects: ObjectQuery base class and derived classes for specific querying data types

10.1 | A generic object querying type: `ObjectQuery`

10.1.1 | Properties

- `_objs: dict`
  Items in query are stored in a dictionary with their associated IDs.
- `_ObjectType: object`
  Type of items.
- ***property*** `ids: list[int]`
  Return ids of items.
- ***property*** `values: list[Any]`
  Return values of items.

10.1.2 | Methods

- `query_to_dict(pred: Callable) -> dict`
  Query all items satisfying the predicate `pred`, return as a dictionary.
- `search_to_dict(attr: str, value: Any) -> dict`
  Search for all items with attributes matching `value`, return as a dictionary.

```
def search_to_dict(self, attr: str, value: Any) -> dict:
    if not hasattr(self._ObjectType, attr):
        raise AttributeError

    return self.query_to_dict(lambda obj: getattr(obj, attr) == value)
```

- `to_dict() -> dict`
  Returns internal state `_objs`.
- `to_json(file: str) -> None`
  Export `ObjectQuery` information to JSON file. The function dumps the dictionary `_objs` into a JSON file.
- `to_csv(file: str) -> None`
  Export `ObjectQuery` information to CSV file. The function converts the dictionary `_objs` into a `pandas` table before exporting to CSV.
- `output_as_json = to_json`
  `output_as_csv = to_csv`
  Aliases for `to_json` and `to_csv` functions.

10.2 | Example use of ObjectQuery

10.2.1 | StopQuery

```
class StopQuery(ObjectQuery, StopQueryInputMixin):
    def __init__(self, objs):
        super.__init__(objs=objs, ObjectType=Stop)

    def query(self, pred: Callable[[Stop], bool]):
        return StopQuery(self.query_to_dict(pred))

    def search(self, attr: str, value: Any):
        return StopQuery(self.search_to_dict(attr, value))

    # aliases
    search_by_abc = search
```

Listing 10.1: Querying List of Objects: `StopQuery` class

10.2.2 | VariantQuery

```python
class VariantQuery(ObjectQuery, VariantQueryInputMixin):
    def __init__(self, objs):
        super.__init__(objs=objs, ObjectType=Variant)

    def query(self, pred: Callable[[Variant], bool]):
        return VariantQuery(self.query_to_dict(pred))

    def search(self, attr: str, value: Any):
        return VariantQuery(self.search_to_dict(attr, value))

    # aliases
    search_by_abc = search
```

Listing 10.2: Querying List of Objects: `VariantQuery` class

10.2.3 | PathQuery

```python
class PathQuery(ObjectQuery, PathQueryInputMixin):
    def __init__(self, objs):
        super.__init__(objs=objs, ObjectType=Path)

    def query(self, pred: Callable[[Path], bool]):
        return PathQuery(self.query_to_dict(pred))

    def search(self, attr: str, value: Any):
        return PathQuery(self.search_to_dict(attr, value))

    # aliases
    search_by_abc = search
```

Listing 10.3: Querying List of Objects: `PathQuery` class

Chapter 11

LangChain: Apply Large Language Model to Querying Databases

This chapter is based mostly on the official Python Langchain Documentation[19]

LangChain is a framework for developing applications powered by language models. LangChain provides the ability for the program to be:

- **Context-aware**: connect a language model to sources of context
- **Reasonable**: rely on a language model to reason (answering queries, deciding actions, etc.)

This report will briefly discuss the application of LangChain only to the task of querying databases. *It is to be noticed that Large Language Models (LLMs) are still at their early stages, thus most LLM libraries expect users to use with special care and revision. That being said, LLMs do bring a lot of potential to empower existing programs and enable them to do tasks that have never been thought of before.*

To install LangChain, install the following dependencies:

```
pip install langchain
pip install langchain_experimental
pip install langchain_openai
```

11.1 | Querying a `pandas.DataFrame`

LangChain supports querying a `pandas.DataFrame` directly through the `langchain_experimental.agents.agent_toolkits` module.

```
>>> import pandas as pd

>>> from langchain.agents.agent_types import AgentType
>>> from langchain_experimental.agents.agent_toolkits \
...     import create_pandas_dataframe_agent
>>> from langchain_openai import OpenAI
```

Listing 11.1: LangChain: Importing libraries

Let say we have a `DataFrame` of all *Bus stops* in Ho Chi Minh City called `stops_df`.

```
>>> stops_df = pd.read_csv('stops.csv')
>>> stops_df.sample(5)
      StopId     Code          Zone
388      483   Q9 213        Quan 9
5852     478   Q2 072        Quan 2
8575    1128  HNB 048  Huyen Nha Be
8225    3559  QCCT128  Huyen Cu Chi
4254     841   Q7 102        Quan 7
```

Listing 11.2: LangChain: Importing libraries

Now we create an "agent" that will link our dataframe with a LLM.

```
>>> agent = create_pandas_dataframe_agent(
...     OpenAI(temperature=0, openai_api_key=OPENAI_API_KEY),
...     stops_df,
...     verbose=True
... )
```

Listing 11.3: LangChain: Create an "agent"

Note that we can replace the `OpenAI` model by any large language models, most of which are included in the `langchain.llms` module.

Asking a question is as simple as "invoking" the agent.

```
>>> agent.invoke("List 5 stops in District 5")
```

Listing 11.4: LangChain: "Invoke" a query to get the answer

```
> Entering new AgentExecutor chain...
Thought: I need to filter the dataframe for stops in District 5
Action: python_repl_ast
Action Input: df[df['Zone'] == 'Quan 5'].head()    StopId    Code    Zone
93      569  Q5 036  Quan 5
94      573  Q5 035  Quan 5
95      433  Q5 017  Quan 5
96      434  Q5 018  Quan 5
97      436  Q5 019  Quan 5I now know the final answer
Final Answer: The 5 stops in District 5 are:
1. StopId: 569, Code: Q5 036
2. StopId: 573, Code: Q5 035
3. StopId: 433, Code: Q5 017
4. StopId: 434, Code: Q5 018
5. StopId: 436, Code: Q5 019
```

Listing 11.5: LangChain: Response from the agent

And the method will return a JSON object of the user input and the model final output of the query.

```
{'input': 'List 5 stops in District 5',
'output': 'The 5 stops in District 5 are:\n1. StopId: 569, \
Code: Q5 036\n2. StopId: 573, Code: Q5 035\n3. StopId: \
433, Code: Q5 017\n4. StopId: 434, Code: Q5 018\n5. \
StopId: 436, Code: Q5 019'}
```

Listing 11.6: LangChain: Final input and output

11.2 | Large Language Model

Large Language Model (LLM) is a class of models that deals with tasks that involve language processing and generation.

A standard task for a LLM is to predict the subsequent token[1] of a sentence or a paragraph. It does so by choosing the token that minimises the perplexity, i.e. the uncertainty of the model to predict the next token.

$$\log(\text{perplexity}) = -\frac{1}{n}\sum_{i=1}^{n}\log \mathbb{P}[\text{token}_i|\text{token}_1, \text{token}_2, \cdots, \text{token}_{n-1}]$$

He is a student of the University of _____. (Science / Social Science / Technology / ...)

Figure 11.1: LLM: The basic task of LLM

[1]Token is the basic unit of language that conveys meaning. A token is smaller than a word.

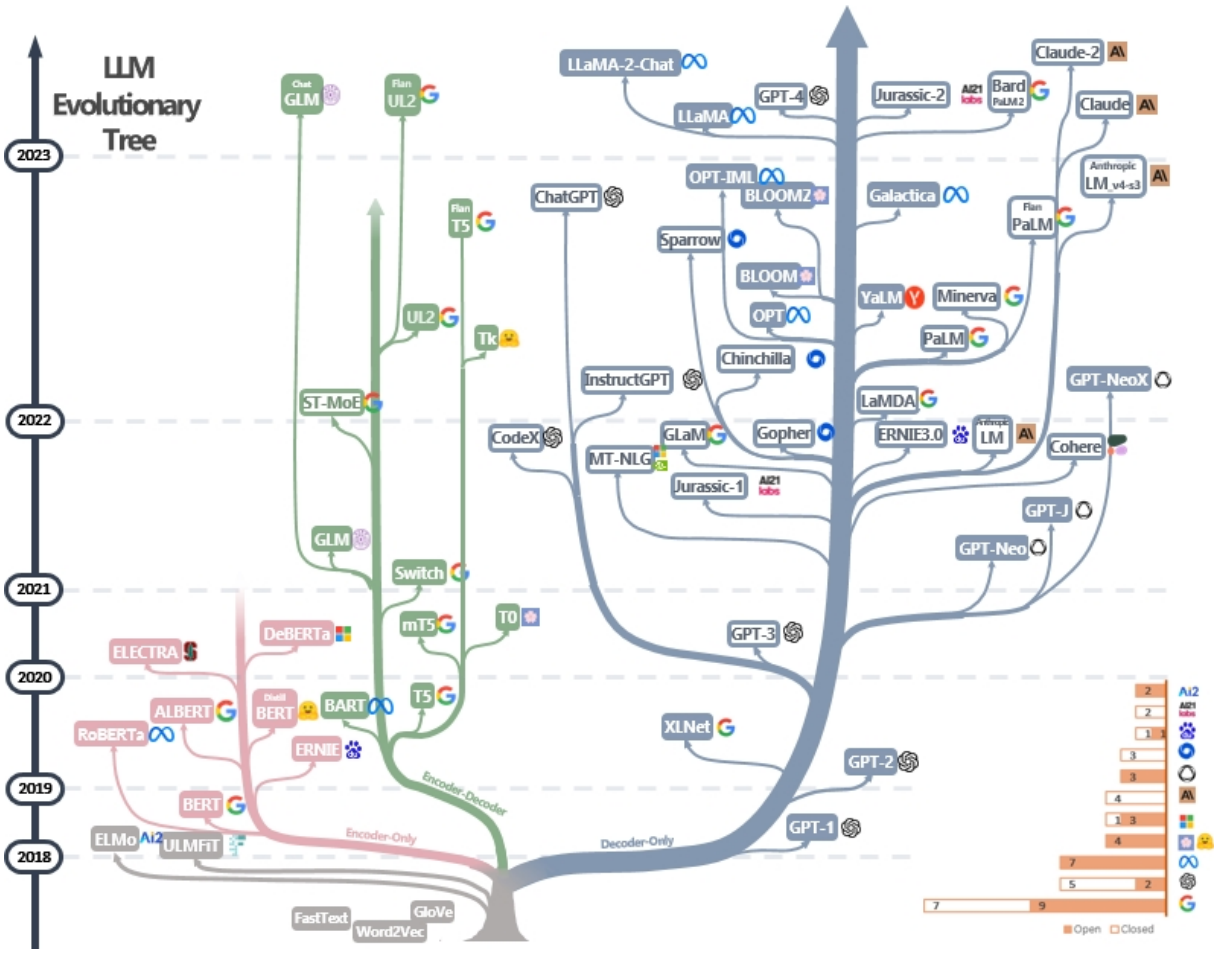


Figure 11.2: LLM: The evolutionary tree of LLM[1]

Through experiments, LLMs are shown to be capable of doing various language-heavy tasks, including

- **Text generation**. Given a context, the model predicts the next possible token, then the token will be added into the context for the next prediction.
- **Text masking**. The context is the tokens before and after the masked position.
- **Data retrieval**. The model compares each data point with a target using similarity scores (e.g. cosine similarity). The most similar data are then selected.
- **Reasoning agent**. The model receives the context, then produces the best possible action.

To illustrate meaningful results in this report, we will be using the GPT-4 model, provided by OpenAI.

11.3 | Agents

11.3.1 | Agents

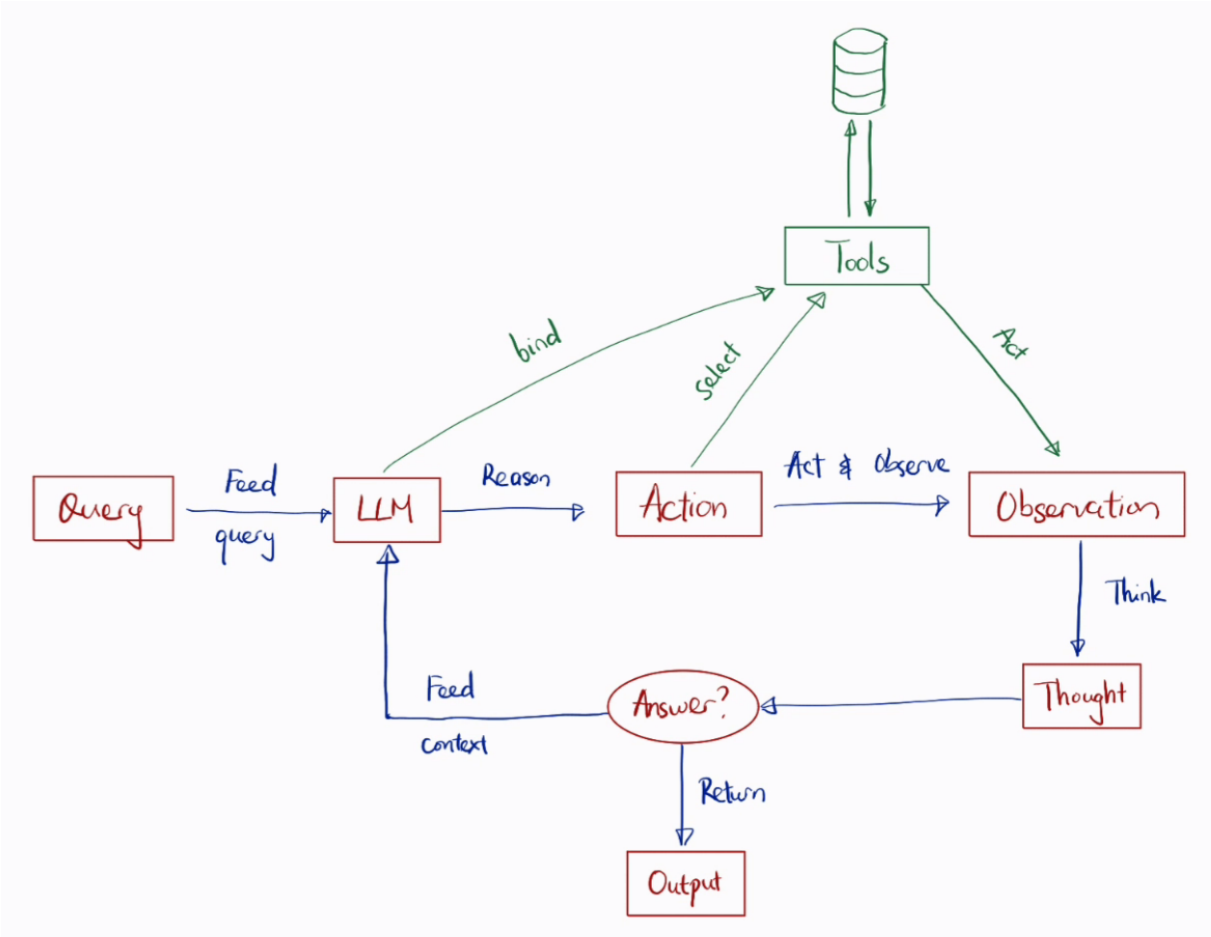


Figure 11.3: Langchain Agents: Workflow

An Agent is a model that uses LLM to decide which 'tools' to use for a certain task.

1. The query is fed into a Large Language Model (LLM), which has been 'bounded' or 'aware of' the available tools.
2. The LLM try to reason to come up with the next possible action. An action is built upon the tools in its awareness.
3. The action is then taken, the result of which is observed by the agent.
4. The agent tries to 'think' by giving a structured analysis of the observations.
5. From the generated thoughts, the agent must decide if any sensible output can be returned. If not, and if allowed, the agent will rerun the whole process by feeding the new context into the LLM.

To define a functional LangChain agent with tool-calling capability, use the `create_tool_calling_agent` function and the `AgentExecutor` class.

```
from langchain.agents import
    (create_tool_calling_agent, AgentExecutor)
from langchain.tools import BaseTool

llm: LanguageModelLike
tools: Sequence[BaseTool]
prompt: str

agent = create_tool_calling_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

agent_executor.invoke({"input": "Enter query here..."})
```

Listing 11.7: Langchain Agents: Basic use

11.3.2 | Tools

A simple tool is built upon Python functions with the decorator `@tool`. The function behaves like a normal Python function, with additional properties:

- `name: str`. The name of the function, must be unique in a set of tools.
- `description: str`. The function docstring, used by the agent to determine the appropriate tool(s).
- `args_schema: pydantic.BaseModel`. Parameters validation, few-shot examples, etc.

```python
"""
module chatter.tools.shortest_path
"""

import pandas as pd
from langchain.tools import tool

from network.bus import BusNetwork
from network.shortest_paths.contraction_hierarchies import
    NetworkContractionHierarchiesLazyED

ch_model: NetworkContractionHierarchiesLazyED = None

@tool
def distance_to_one(src: Any, dest: Any) -> float:
    """
    Find the shortest distance between node src and node dest.
    Arguments:
    * src:      ID of the source node
    * dest:     ID of the destination node
    """
    src = int(src)
    dest = int(dest)
    return ch_model.dist(src, dest)
```

Listing 11.8: Langchain Tools: Create a custom tool

11.4 | Prompt Engineering

Sometimes there are complications that we want the LLM to know beforehand, for example:

- The default coordinates in the database are geographical, not Cartesian, hence Euclidean distances do not apply.
- A user may mistype names, so an LLM should be accountable for that.

We can teach the LLM those ad-hoc rules using *Prompt engineering*, that is, to pass in a prompt that provides the wanted tasks and any details to watch out for.

11.5 | Querying a `pandas.DataFrame`, advanced

We implemented the `chatter` module, containing the database of bus stops (`stops_df`) and the corresponding `agent`.

The `agent` takes in an OpenAI GPT-4 model as the Large Language Model, a prefix prompt and a set of extra tools (beside tools from the `pandas_agent`).

```
PROMPT = """
[Notes for searching Stops]:
Terminologies:
* Phường/Xã = Ward
* Quận/Huyện = Zone

Searching Process:
* In most cases, it is best to find the appropriate ID of a Stop before performing any further searching.
* A user may mistype a word, especially names. If there is any typo, find the most reasonable alternatives before applying the search.
* Sometimes the user may type a Vietnamese name without tones. If so, you may want to ignore the tones while searching.
* A query may result in an error due to a lack of options, in which case, output all the possible values found so far.
* If only one value is needed, select one randomly.

Locations and Distances:
* To calculate the distance between two Stops:
    1. Find the IDs of the source (src) node and the destination (dest) node.
    2. Use the given distance function with the node IDs.

* To calculate the distance from one Stop (source) to all other Stops, use the following expression:
    df['distance'] = df['StopId'].apply(lambda node_id: distance_to_one(src, node_id))
"""

stops_agent = create_pandas_dataframe_agent(
    ChatOpenAI(temperature = 0, model = 'gpt-4-turbo-preview'),
    self.stops_df,
    verbose = True,
    handle_parsing_errors=True,
    max_iterations=10,
    prefix: PROMPT,
    extra_tools: [
        chatter.tools.distance_to_one
    ]
)
```

Listing 11.9: Langchain: Create an "agent" with additional prompting and extra tools

11.6 | Experiments

The full experiment is in the llm_test.ipynb Jupyter notebook in this repository.

11.6.1 | Simple Task 1: Asking the ID of a Stop

- **Query**. *What is the ID of the Stop named "Cầu vượt An Sương"*
- **Answer**.

```
> Entering new AgentExecutor chain...
Thought: I need to filter the dataframe to find the row where the Name
↪  column is "Cầu vượt An Sương" and then retrieve the StopId.
Action: python_repl_ast
Action Input: df[df['Name'] == 'Cầu vượt An
↪  Sương']['StopId'].values[0]2252I now know the final answer
Final Answer: The ID of the Stop named "Cầu vượt An Sương" is 2252

> Finished chain.
The ID of the Stop named "Cầu vượt An Sương" is 2252
```

- **Comment**. The agent got the correct answer!.

```
===============[Cầu vượt An Sương]================
| StopID:              2252                       |
| Code:                Q12 088                    |
| Name:                Cầu vượt An Sương          |
| Type:                Nhà chờ                    |
| Zone:                Quận 12                    |
| Ward:                                           |
| Address No.:         Xương Z735                 |
| Street:              Quốc lộ 1A                 |
| Support Disability?: False                      |
| Status:              Đang khai thác             |
| Lng:                 106.616081                 |
| Lat:                 10.843623                  |
| Search tokens:       CvAS, XZ, Ql1A             |
| Routes:              24 -> 33 -> 60-1 -> 61-3   |
=================================================
```

11.6.2 | Simple Task 2: Querying the information of routes

- **Query**. *'How many routes pass the Stop KTX Tran Hung Dao?'*
- **Answer**.

```
> Entering new AgentExecutor chain...
Thought: I need to find the row corresponding to the Stop KTX Tran Hung
↪  Dao and then count the number of routes passing through it.
```

```
Action: Use pandas to filter the dataframe for the row corresponding to
↪  the Stop KTX Tran Hung Dao.
Action Input: df[df['Name'] == 'KTX Trần Hưng Đạo']Use pandas to filter
↪  the dataframe for the row corresponding to the Stop KTX Tran Hung
↪  Dao. is not a valid tool, try one of [python_repl_ast].I should use
↪  the python_repl_ast tool to filter the dataframe for the row
↪  corresponding to the Stop KTX Tran Hung Dao.
Action: python_repl_ast
Action Input: df[df['Name'] == 'KTX Trần Hưng Đạo']     StopId    Code
↪  Name StopType    Zone  Ward     AddressNo  \
110     121  Q1 125  KTX Trần Hưng Đạo  Nhà chờ  Quận 1  None
↪  10
261      26  Q1 137  KTX Trần Hưng Đạo  Nhà chờ  Quận 1  None  135/40
↪  (135)

            Street SupportDisability          Status         Lng
             ↪  Lat  \
110  Trần Hưng Đạo             Không  Đang khai thác  106.694695
↪  10.767363
261  Trần Hưng Đạo             Không  Đang khai thác  106.694824
↪  10.767064

                 Search                         Routes        coord_x  \
110       KTHD, 10, THD          01, 139, 152, 45, 56  685107.170760
261  KTHD, 135/40(, THD  01, 139, 152, 45, 53, 56, 88  685121.462751

          coord_y
110  1.190880e+06
261  1.190847e+06  I have found the row corresponding to the Stop KTX
↪  Tran Hung Dao. Now, I need to count the number of routes passing
↪  through it.
Action: python_repl_ast
Action Input: len(df[df['Name'] == 'KTX Trần Hưng
↪  Đạo']['Routes'].iloc[0].split(','))5I have found that there are 5
↪  routes passing through the Stop KTX Tran Hung Dao.
Final Answer: 5

> Finished chain.
```

- **Comment.**
  1. The agent knew to look for 'almost equal' values, thanks to our pretraining prompt. In this case, it had derived from the name 'KTX Tran Hung Dao' without accents to 'KTX Trần Hưng Đạo' with accents.
  2. There are two possible correct answers: one with code `Q1 125` and one with code `Q1 137`. The agent took one at random (instruction from pretraining prompt). In the full ipynb notebook, there is case when another agent returned the other bus stop.

11.6.3 | Advanced Task: Finding the geodesic distance between two Stops

- **Query**. *Distance between "Bến xe Miền Tây" and "Bến xe Miền Đông"*
- **Answer**.

```
> Entering new AgentExecutor chain...
Thought: I need to find the IDs of "Bến xe Miền Tây" and "Bến xe Miền
↪ Đông" before I can calculate the Euclidean distance between them.
↪ Since the names are not directly visible in the provided data
↪ snippet, I will search for these names ignoring tones as suggested in
↪ the notes for searching Stops.
Action: python_repl_ast
Action Input: print(df[df['Name'].str.contains('Miền Tây', regex=False,
↪ na=False)])     StopId     Code             Name  StopType
↪ Zone           Ward  \
592     116  QBT 012  Bến xe Miền Tây  Trụ dừng  Quận Bình Tân  Phường An
↪ Lạc
644     725    BX 46  Bến xe Miền Tây    Bến xe  Quận Bình Tân
↪ None
694     119  QBT 008  Bến xe Miền Tây   Nhà chờ  Quận Bình Tân  Phường An
↪ Lạc
          AddressNo            Street SupportDisability          Status
          ↪ \
592          358-360  Kinh Dương Vương             Không  Đang khai thác
644  Bến xe Miền Tây  Kinh Dương Vương             Không  Đang khai thác
694  Bến xe Miền Tây  Kinh Dương Vương             Không  Đang khai thác
           Lng        Lat              Search  \
592  106.619096  10.741973  BxMT, 358-360, KDV
644  106.618317  10.740705     BxMT, BxMT, KDV
694  106.618468  10.741035     BxMT, BxMT, KDV
                                                 Routes        coord_x  \
592  09, 139, 14, 148, 25, 32, 61, 62-10, 62-2, 62-...  676853.625586
644  10, 102, 139, 14, 148, 151, 32, 39, 41, 60-1, ...  676769.158331
694           09, 25, 61, 62-1, 62-10, 62-2, 62-8, 84  676785.482321
...
3602  1.196252e+06
3621  1.196238e+06
I found multiple entries for "Bến xe Miền Đông" as well, but the entry
↪ with Code "BX44" and "Bến xe Miền Đông mới" with Code "BX 33" are the
↪ actual bus stations (Bến xe) based on their StopType. Since the
↪ question does not specify which "Bến xe Miền Đông" to use, I will
↪ calculate the distance to both and provide the distances separately.
↪ Now, I will calculate the distance between "Bến xe Miền Tây" (ID 725)
↪ and both "Bến xe Miền Đông" (ID 1403) and "Bến xe Miền Đông mới" (ID
↪ 7695).
Action: distance_to_one
Action Input: 725, 1403
```

- **Comment**. The agent knew to call the `distance_to_one` function with the ap-

propriate input 725 and 1403, but had errors when parsing the input into the function.

11.7 | Conclusion

LLMs have shown potential in assisting humans to explore using no more than a grasp of language. However, many limitations exist that make this process not so straightforward. It is best to combine this approach with other traditional methods with human involvement to produce the best results.

Appendices

Developing tools

Appendix I

GitHub 101

1.0.1 | The necessity of a Version Control System (VCS)

When a project enlarges and/or is contributed by a large team of developers, many problems arise that differ from a personal, small-scale project:

- **Non-linear development**. Developers may start from different versions of the code, from which they would grow the project, resulting in a Directed Acyclic Graph (DAG) based development.
- **Distribution**. Developers may work on different aspects of the project, thus there is a need for collaboration platform.
- **Failure management**. When the current version fails, the project may need to roll back to previous versions.

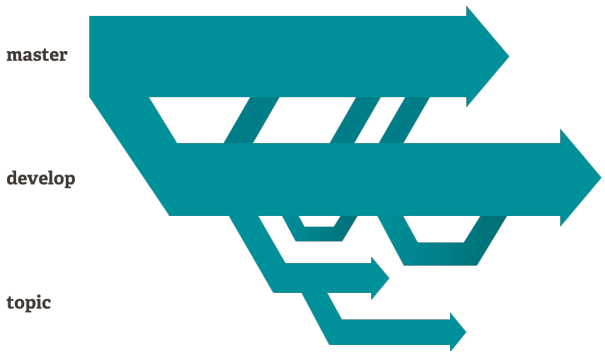


Figure 1.1: Github: Large-scale project version control.
Source: https://git-scm.com/

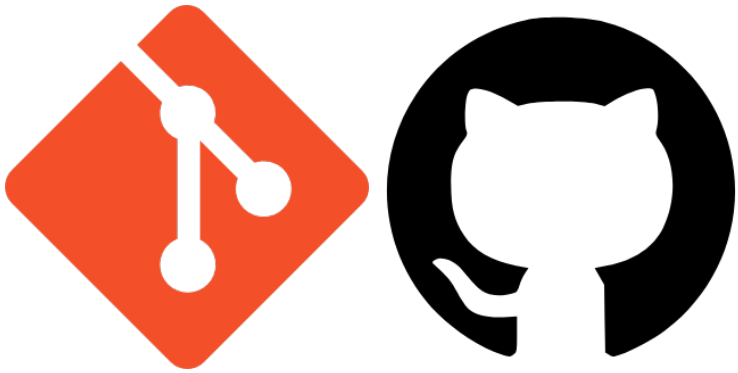


Figure 1.2: Github: The Git logo resembles a version control diagram of a project. The GitHub logo resembles... a cat. Turns out, that cat is no normal cat, it is the Octocat - GitHub's mascot.

1.0.2 | Git vs. GitHub

Git

Git is a Distributed Version Control System (DVCS), a local tool used for tracking code changes in a project. Git supports:

- **Snapshotting**. Creating snapshots of the project at different timestamps.
- **Monitoring**. Tracking changes made by different developers.
- **Revertability**. Rolling back to previous versions.

Git exists in both the form of Graphical User Interface (GUI) or Command Line Interface (CLI). *This report will mostly use the command-line version of Git.*

To install Git, download an installer at https://git-scm.com/, or install via packages like `apt`, `yarn`, `chocolatey`, etc.

To confirm that Git is installed, enter `git version` in the Terminal.

```
$ git version
git version 2.33.0.windows.2
```

Listing 1.1: Github: Check if `git` is installed. Display the version of Git.

GitHub

GitHub is a cloud-based service that provides functionalities built on top of Git.

In short, GitHub acts like a cloud storage medium which also supports Git-style version control. To make full use of GitHub, it is recommended to have Git installed.

1.0.3 | Terminologies

This subsection is based mostly on the official Github's Repositories documentation[20].

repository (Repo)

A repository is the most basic element of GitHub. It's a place where you can store your code, your files, and each file's revision history. Repositories can have multiple collaborators and can be either public or private. A repo can be either *public* or *private*.

clone

To clone is to download a full copy of a repository's data from GitHub.com, including all versions of every file and folder.

commit

To commit is to create a *snapshot* of the code.

branch

A branch is a parallel version of the code that is contained within the repository, but does not affect the primary or `main` branch.

fork

A fork is a new repository that shares code and visibility settings with the original "upstream" repository.

pull

A pull is a request to merge changes from one branch into another.

I.1 | Create a *private* GitHub repository

- **Step 1.** On the top-right corner, click on the + *Create new...* button to reveal the menu. Then click on *New repository*.

  Alternatively, go to https://github.com/new.

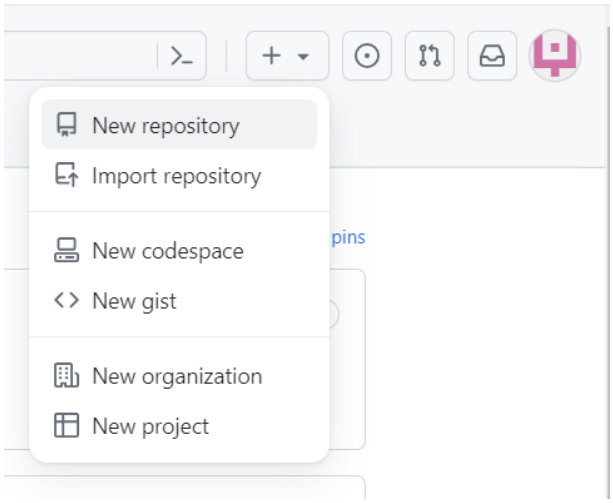


Figure 1.3: Github: Creating a new repository

- **Step 2.** Fill the information into the fields. An example is given in Figure 1.4 Some commonly used fields are
  - **Owner**. By default it is set to the creator.
  - **Repository name**. Must be at most 100 code points (a code point is either an alphanumeric character, a hyphen `'-'`, an underscore `'_'` or a period `'.'`)
  - **Description** *(Optional)*. A brief description of the repository.
  - **Public/Private**. Choose the visibility of the repo. In this specific example, choose *Private*.
  - **Initialise this repository with**. Create default files (`README.md` and `.gitignore`) and select licensing options. For simplicity, it can be left blank until later.

The result should look like in Figure 1.5.

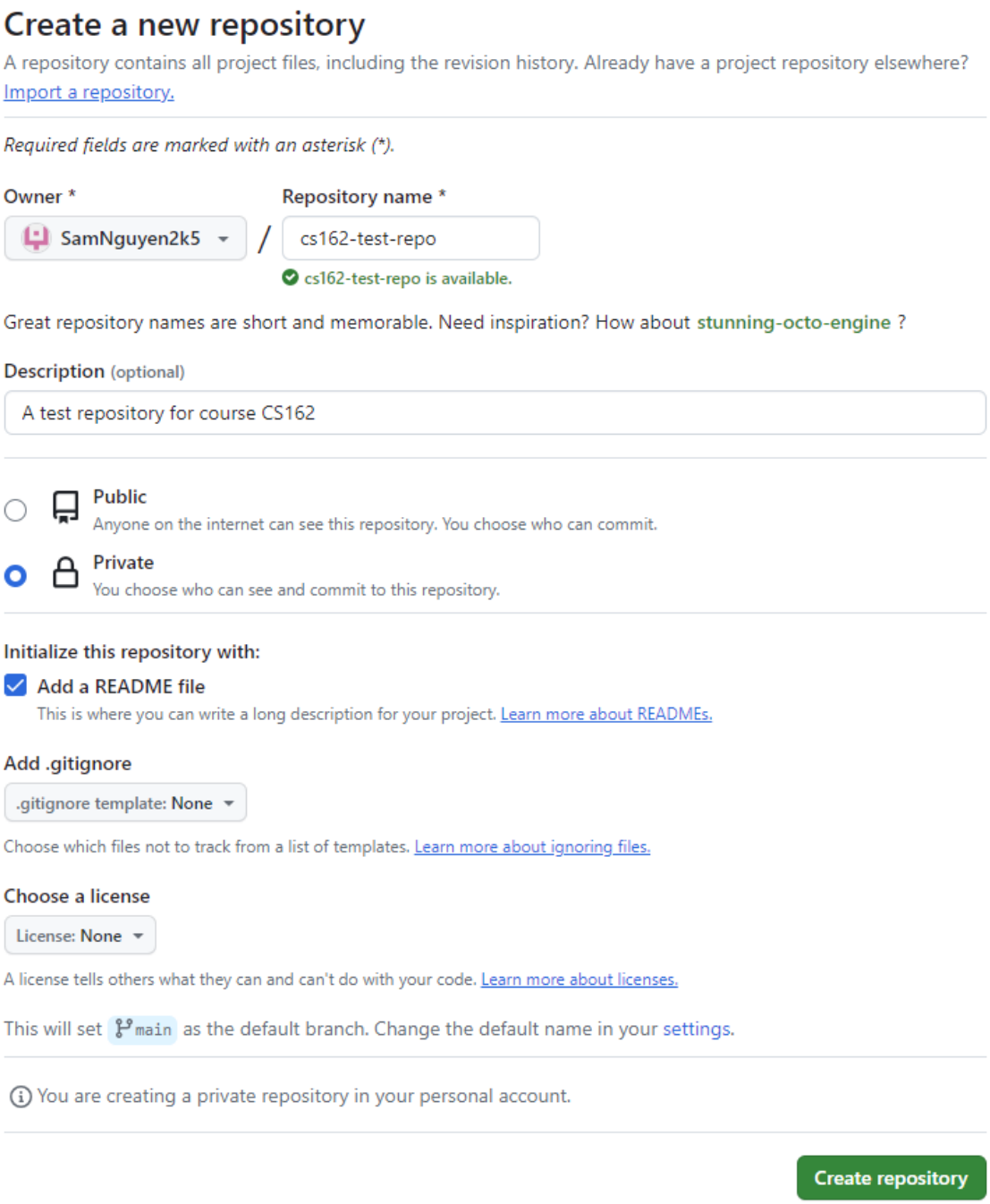


Figure 1.4: Github: Repository information field.

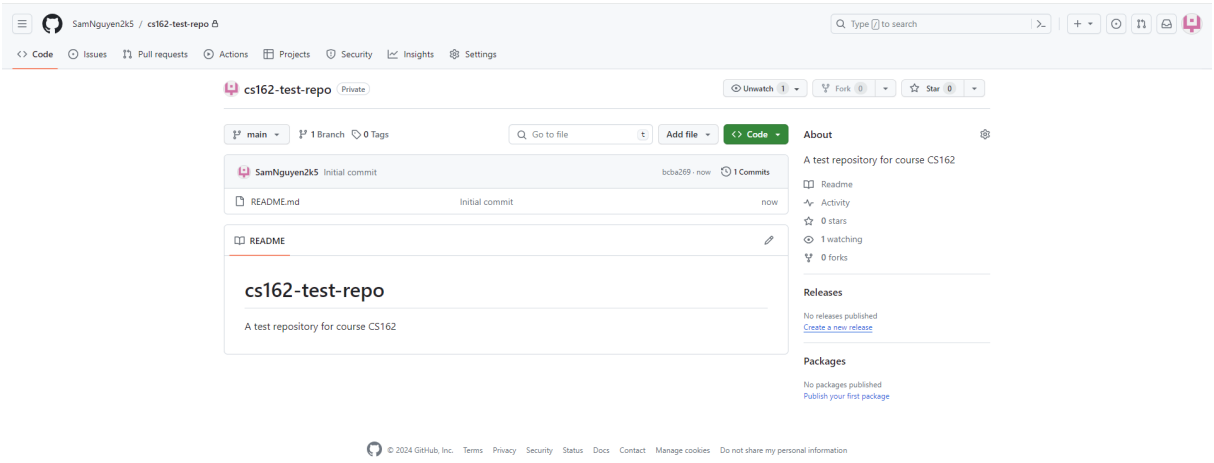
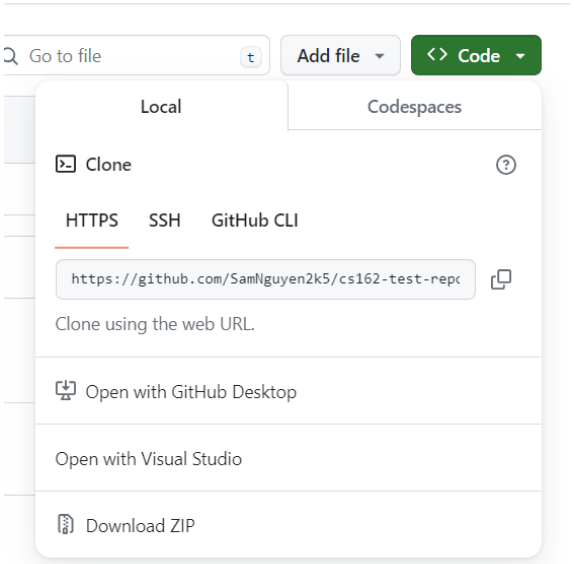


Figure 1.5: Github: Repository successfully created.

I.2 | Clone a repository from GitHub

- **Step 1**. At the main page of the repository, click on 



- **Step 2**. Copy the `.git` URL. *(This URL is for cloning using HTTPS).*
- **Step 3**. In the Terminal, open the folder into which the repository is to be cloned.
- **Step 4**. Use the `git clone <link-to-repository>` command.

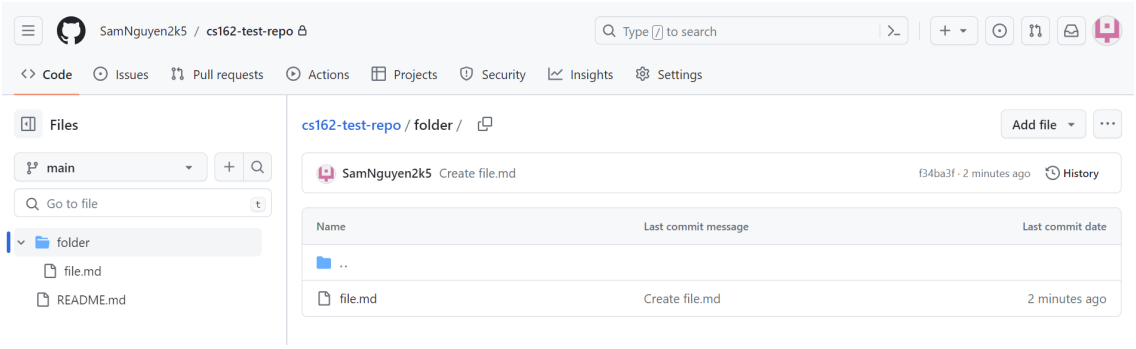
I.3 | Create new files and folders in GitHub

```
$ git clone https://github.com/SamNguyen2k5/cs162-test-repo.git
Cloning into 'cs162-test-repo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

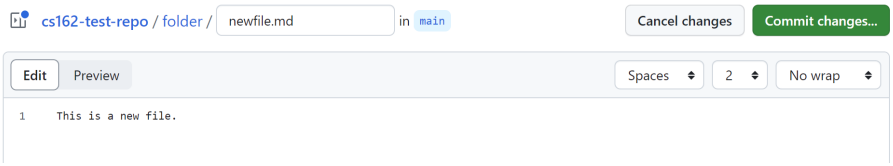
Listing 1.2: Github: Link to `.git` for repository cloning.

1.3.1 | Create files from scratch

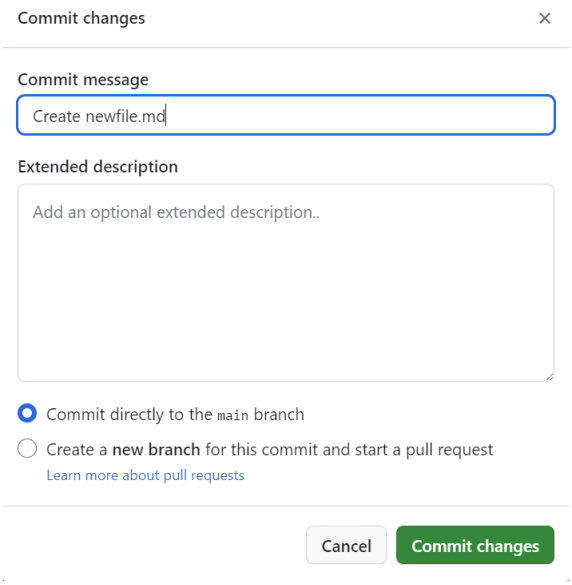
- **Step 1**. Navigate to the folder to which files/folders are added.
- **Step 2**. Select the Add file menu, then click *Create new file*, or select + in the tree view on the left.



- **Step 3**. Set the filename. Subfolders can be created by adding a slash (`'/'`).
- **Step 4**. Enter the content of the file.

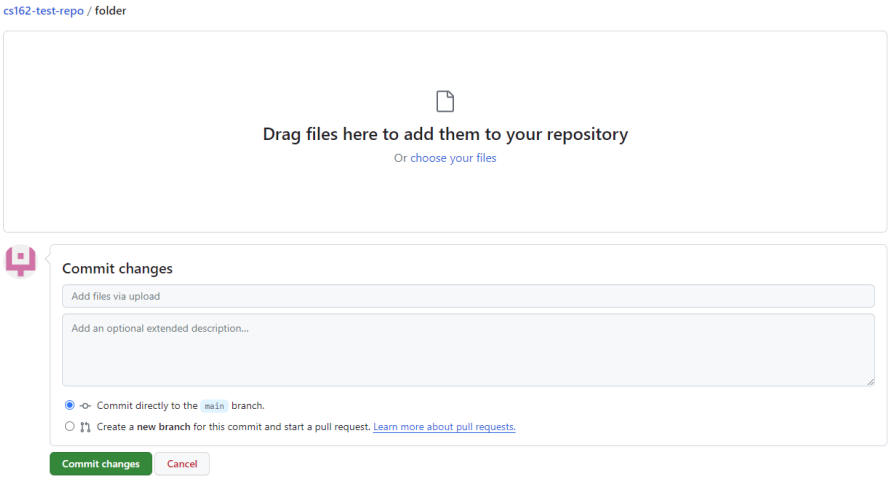


- **Step 5**. Select Commit changes. Write a brief *Commit message*, then choose which branch to commit. The new file can either be
  - Commited directly to the `main` branch.
  - Commited to a new branch, and a pull request is made.



1.3.2 | Upload local files

- **Step 1**. Navigate to the folder to which files/folders are added.
- **Step 2**. Select the Add file menu, then click *Upload files*.
- **Step 3**. Drag files or folders to upload. Then write *commit message* and choose a commit branch, similar to above.



I.4 | .gitignore 101

`.gitignore` is a Git file that lists out all files **to be passed** by Git when commited and pushed to the remote repository. These files usually:

- are unimportant for development, but require substantial amount of repository storage (`.vscode` folder, compiled executable files, etc.)
- contain sensitive when publicised (password database, etc.)

`.gitignore` is a text file in which each line is a folder path or a file to be ignored by Git.

```
## This is a comment
#### A comment starts with a hash character ('#')

# Ignore a folder
.vscode

# Ignore a file
main.aux

# Ignore a file with wildcard
*.log
debug.log*
```

Listing 1.3: Github: Example of `.gitignore`

I.5 | Basic Git commands (add, commit, push, pull)

1.5.1 | Syntax

- **`git add`**. Add files/folders to the staging environment (start tracking files/folders).

```
git add <file>
```

- **`git commit`**. Snapshot a change to the **local repository**.

```
git commit -m "Commit message"
```

- **`git push`**. Transfer local commits to the remote repository.

```
git push <remote_name> [<local-branch_name>:]<remote-branch_name>
```

- **`git pull`**. Update local repository from the remote repository.

```
git pull <remote> <branch>
```

1.5.2 | Example

git add

```
$ tree /f
Folder PATH listing for volume Windows-SSD
Volume serial number is 62AA-618C
C:.
|   README.md
|
+---folder
        file.md

$ vim added_file.md

$ type added_file.md
This file is added by
```
    git add added_file.md
```

$ git add added_file.md

$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   added_file.md
```

Listing 1.4: Github: Example of `git add`

`git commit` and `git push` to the current working branch

```
$ git commit -m "Commit added_file.md"
[main 444661f] Commit added_file.md
 1 file changed, 4 insertions(+)
 create mode 100644 added_file.md

$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 365 bytes | 365.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SamNguyen2k5/cs162-test-repo.git
   f34ba3f..444661f  main -> main

nothing to commit, working tree clean
```

Listing 1.5: Github: Example of `git commit` and `git push` on current `main` branch

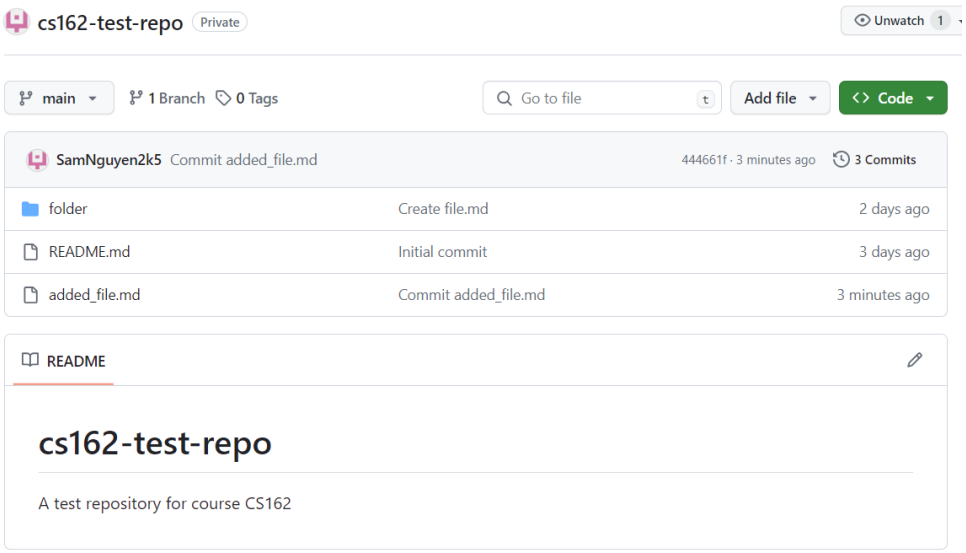


Figure 1.6: Github: After `git push`, `added_file.md` is added into the `main` branch.

`git commit` and `git push` to another working branch

```
$ git commit -a -m "Commit to new branch"
$ git checkout -b other
Switched to a new branch 'other'

$ git push origin other
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 403 bytes | 403.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'other' on GitHub by visiting:
remote:    https://github.com/SamNguyen2k5/cs162-test-repo/pull/new/other
remote:
To https://github.com/SamNguyen2k5/cs162-test-repo.git
 * [new branch]      other -> other
```

Listing 1.6: Github: Example of `git commit` and `git push` on another branch `other`

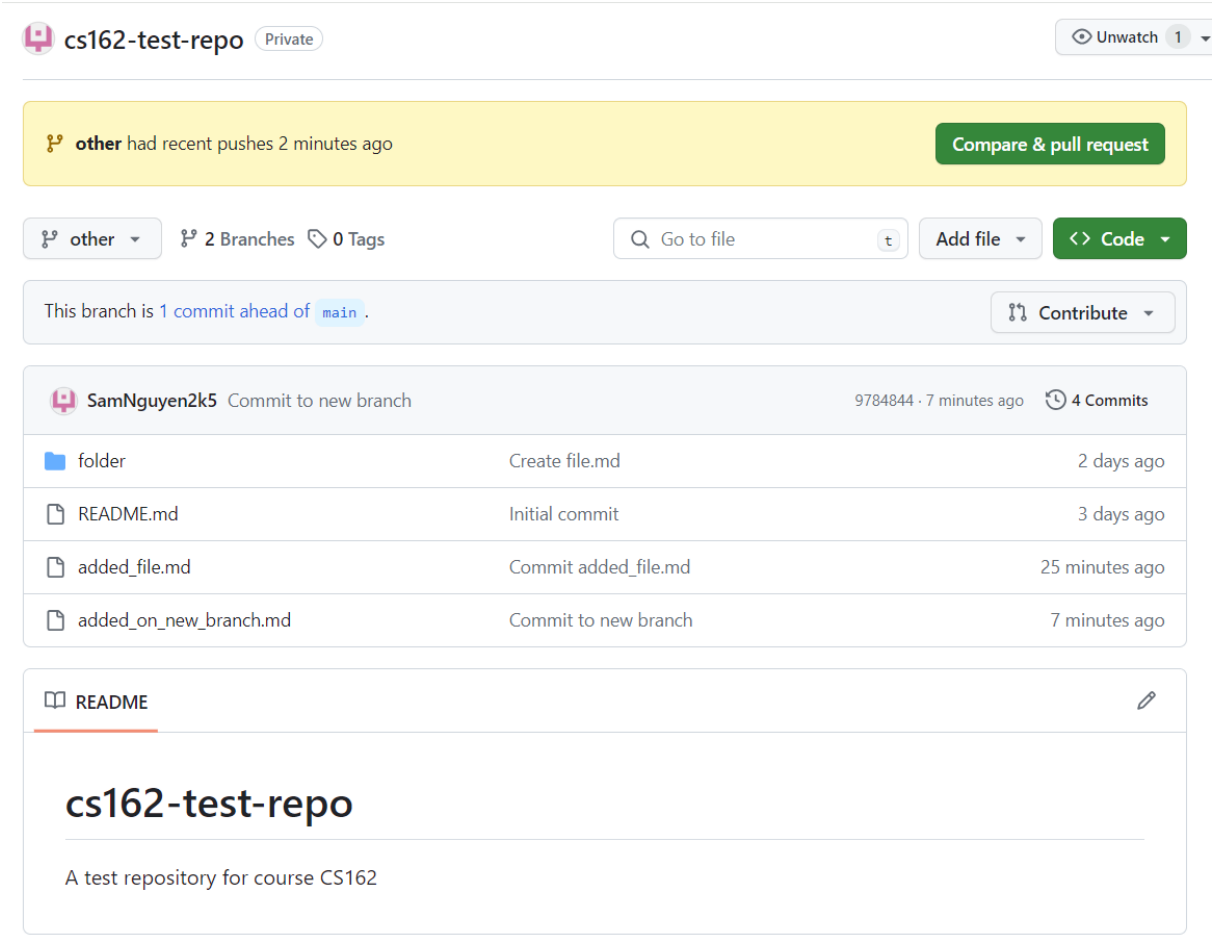


Figure 1.7: Github: Branch (`other`) created with `added_on_new_branch.md` added.

```
git pull
```

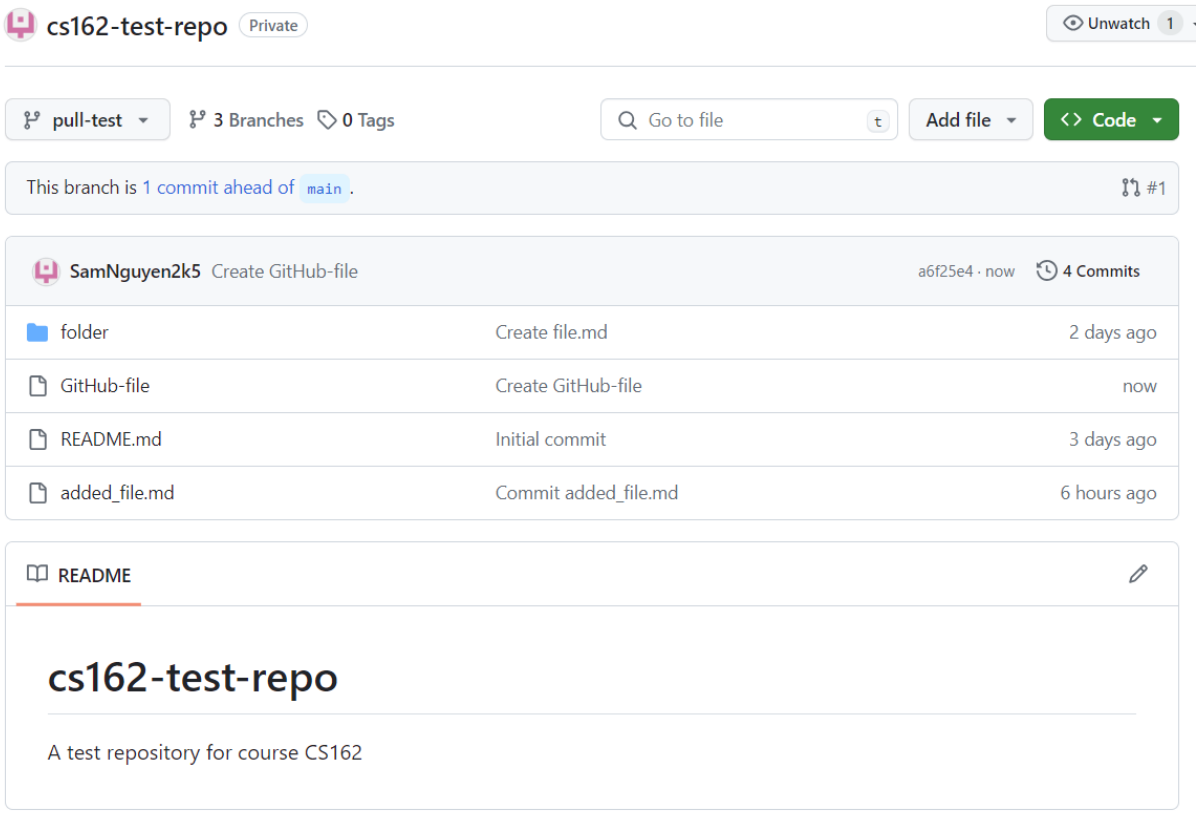


Figure 1.8: Github: A file is created on the GitHub website. The current local repository has no knowledge of this newly created file.

```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 1.03 KiB | 176.00 KiB/s, done.
From https://github.com/SamNguyen2k5/cs162-test-repo
 * [new branch]      pull-test  -> origin/pull-test
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so
with:

    git branch --set-upstream-to=origin/<branch> other

$ tree /f
Folder PATH listing for volume Windows-SSD
Volume serial number is 62AA-618C
C:.
   added_file.md
   added_on_new_branch.md
   GitHub-file
   README.md

folder
        file.md
```

Listing 1.7: Github: After `git pull`-ing, `GitHub-file` has been recognised.

Appendix II

Python 101

With regards to Artificial Intellgence and Machine Learning, Python is a great choice to start, even for researchers with little coding background. Python offers:

- **Easy syntax**. Pythonic syntax is close to human readable text
- **Extensive AI/ML library support**. `numpy`, `pandas`, `scikit-learn`, etc. are examples of beginner-friendly frameworks for Data Analysis and the Basics of Machine Learning. For professionals, many libraries are backed by large tech companies, such as Google (`Tensorflow`, `Keras`), Facebook (`Pytorch`), etc.
- **Platform flexibility**. Python performs uniformly on various operating systems, unlike C/C++ which is platform dependent, i.e. may run differently in different machines.
- **Scalability**. Easy for prototypes, easy to scale.

Figure 2.1: Python: Extensive AI/ML libraries. *Source: Internet*

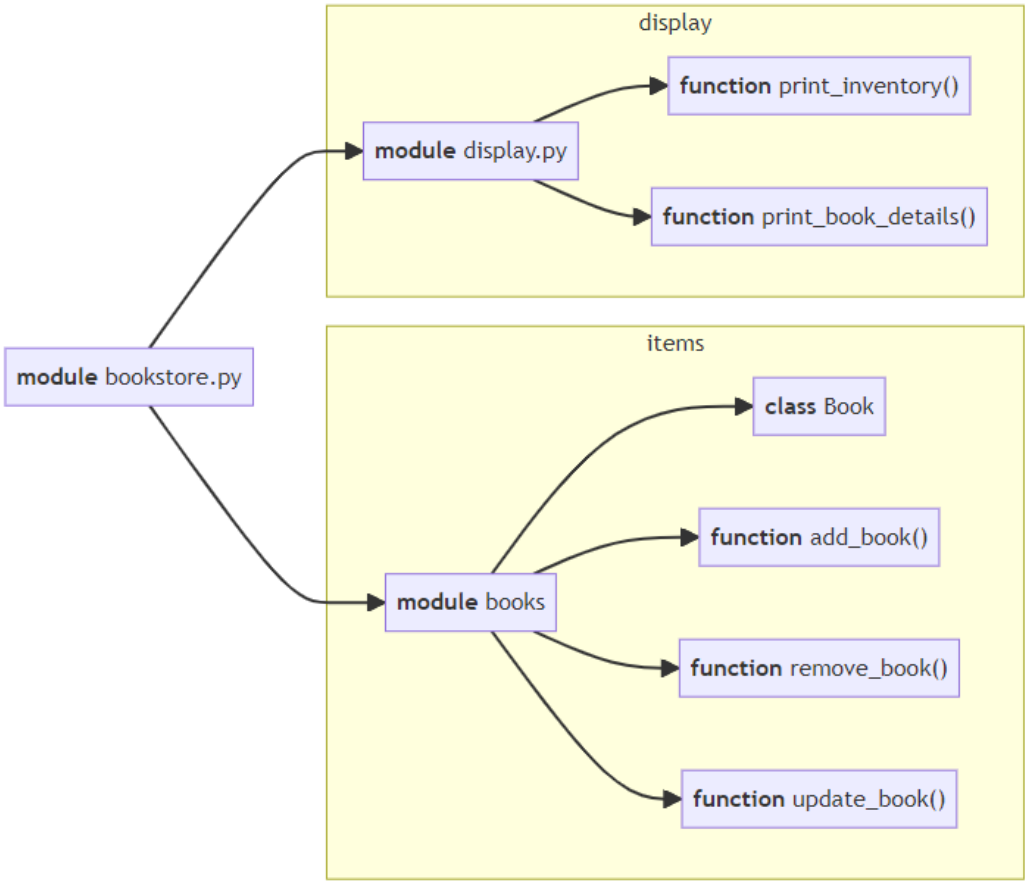


Figure 2.2: Python: Module structure

II.1 | Python project with multiple files

Python code structure is **module-based**, that is, each project contains many `*.py` files, called **modules**, with one module **imports** another.

The project is run from a main script. For example, a user might run the project by entering into the command line:

```
$ python main.py <arguments>
```

and `main.py` would import other modules if needed without any further action. Note that the executed file need not be `main.py`, it can be named arbitrarily.

II.2 | Importing external files

2.2.1 | The `import` keyword

To import a module, simply include the line

```
import <module-name>
```

at the beginning of the file. The `<module-name>` is simply the filename, without the `.py` extension.

To use a function in a module, access via the syntax

`<module-name>.<function-name>(<arguments>)`

```
import math                 # math.py imported. This module is located
                            # at the Python/Lib folder that is
                            # accessible by default
g = math.gcd(35, 14)        # function 'gcd' of module 'math' is called
print(g)          # prints out 7
```

Listing 2.1: Python: Using the `math` module

2.2.2 | Other ways to import files

`from` keyword

The `from` keyword allows importing a selection (or all) of the `classes` and `functions` from a module.

To include specific `classes` and `functions` from a module, include

```
from <module-name> import <class-names>, <fn-names>
```

at the beginning of the script.

To access all, replace the `classes` and `functions` by the asterisk `(*)`.

```
from <module-name> import *
```

`as` keyword

The `as` keywords allows specifying aliases for `functions` when imported from another module.

```
import math as maths
g = maths.gcd(35, 14)
print(g)          # prints out 7
```

Listing 2.2: Python: Using the `math` module, with module alias

It is common practice to use frequently used modules by their short-hand aliases for better readability, for example, `numpy as np`, `pandas as pd`, etc.

```python
from math import gcd as greatest_common_divisor
g = greatest_common_divisor(35, 14)
print(g)        # prints out 7
```

Listing 2.3: Python: Using the `math` module, with function alias

2.2.3 | Script mode vs. module mode

Due to the flexible role of a `.py` file, it is important to distinguish if a file is run directly from the `python` command **(script mode)** or via calls from other files **(module mode)**.

Usually the code runs when launched from command, but are not meant to be executed when imported by other files. To deal with the issue, every module has a variable `__name__` that stores either

- `__main__`, if the module is run from the command line, or
- **the name of the importer**, if the module is called from another module.

Therefore, it is common practice to check `if __name__ == __main__` at the beginning of a script, to avoid code run when imported by other modules.

```python
# -- sell.py --
def sell_to(customer):
    print('This book is sold to', customer, '!')

if __name__ == '__main__':
    sell_to('The Owner')
```

```python
# -- main.py --
from sell import sell_to

if __name__ == '__main__':
    sell_to('Everyone')
```

Listing 2.4: Python: Script mode (`main.py`) vs. Module mode (`sell.py`)

```
$ python sell.py
This book is sold to The Owner!

$ python main.py
This book is sold to Everyone!
```

Listing 2.5: Python: Different output at different direct call points

II.3 | Python classes, properties and methods

2.3.1 | Declaration

```
class ClassName:
    # class description
    class_var_1 = class_value_1
    class_var_2 = class_value_2

    def __init__(self, value_1, value_2):
        self.var_1 = value_1
        self.var_2 = value_2

obj = ClassName(args...)            # create an instance
```

Listing 2.6: Python: `class` template

```
class Book:
    count = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def print_book_details(self):
        print('Book Title:', self.name, ', Price: $', self.price)
```

Listing 2.7: Python: A simple Python `class` example

2.3.2 | Propeties

Propreties (or attributes) are variables associated with an object of the class. They are initialised in the `__init__()` method.

Class attributes are **public** by default, unlike most programming languages with strict OOP paradigm like C++/Java, where variable visibility must be explicitly declared. Instead, by convention:

- variables with a single underscore `(_)` prefix are considered **`protected`**;
- variables with a double underscore `(__)` prefix are considered **`private`**.

Note that these are conventions only, and is not strictly managed by the intepreter.

2.3.3 | Methods

Methods are functions defined within the class that operate on objects of the class.

- **Instance method**. declared inside a class, must take `self` as an argument. This method acts on an **instance**.
- **Class method**. declared inside a class with the `@classmethod` decorator, must take `cls` as an argument. This method returns a new instance of the class, without having to call the default constructor.
- **Static method**. declared inside a class with the `@staticmethod` decorator, take neither `self` or `cls` argument. This method is independent of the class, the class name sometimes acts more like a method namespace.

```python
class Book:
    count = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def print_book_details(self):
        print('Book Title:', self.name, ', Price: $', self.price)

    @classmethod
    def priceless_book(cls, name):
        return cls(name, 0)

    @staticmethod
    def minimum_price():
        return 2000

    @staticmethod
    def minimum_price_dollars():
        return '$2000'

a_priceless_book = Book.priceless_book("Priceless")
a_priceless_book.print()
print('Minimum price of a valuable book: ', Book.minimum_price_dollars())

# Output:
# Book Title: Priceless, Price: $0
# Minimum price of a valuable book: $2000
```

Listing 2.8: Python: An example of `@classmethod` and `@staticmethod`.

2.3.4 | @dataclass

Introduced in Python 3.9, `@dataclass` is a special `class` that automatically implements the `__init__()` and `__repr__()` methods. These are the most common methods for classes that focuses on storing data of a larger object, therefore the `@dataclass` syntax offers a syntactic sugar to the developer, removing the excess boiler-plate code.

```
class Book:
    def __init__(self, name: str, price: float, count: int):
        self.name = name
        self.price = price
        self.count = count

    def __repr__(self):
        print('Book Title:', self.name, ', Price: $', self.price,
              ', Quantity: ', self.count)

book = Book('No Title', 19.99, 4)
print(book)
# Sample Output:
# Book Title: No Title, Price: $19.99, Quantity: 4
```

Listing 2.9: Python: A Python regular `class`

```
from dataclasses import dataclass

@dataclass
class Book:
    name: str
    price: float
    count: int

book = Book('No Title', 19.99, 4)
print(book)
# Sample Output:
# Book(name='No Title', price=19.99, count=4)
```

Listing 2.10: Python: A Python `@dataclass`

Bibliography

[1] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, B. Yin, and X. Hu, "Harnessing the power of llms in practice: A survey on chatgpt and beyond," 2023.

[2] Source code for `pyproj.transformer`. [Online]. Available: https://pyproj4.github.io/pyproj/stable/api/transformer.html#pyproj-transformer

[3] PROJ contributors, "PROJ coordinate transformation software library," Open Source Geospatial Foundation, 2024. [Online]. Available: https://proj.org/

[4] `pyproj`. [Online]. Available: https://pypi.org/project/pyproj/

[5] "Decision No. 83/2000/QD-TTg on the use of Vietnam's National references and Coordinates Systems," July 2000. [Online]. Available: https://chinhphu.vn/default.aspx?pageid=27160&docid=7812

[6] [Online]. Available: https://geojson.org/

[7] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and S. Hagen, "The GeoJSON Format," RFC 7946, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7946

[8] [Online]. Available: https://shapely.readthedocs.io/_/downloads/en/stable/pdf/

[9] [Online]. Available: https://readthedocs.org/projects/rtree/downloads/pdf/latest/

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

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

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

[13] 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

[14] 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

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

[16] S. Sawlani, "Explaining the performance of bidirectional dijkstra and a* on road networks," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:195970067

[17] S. Funke and S. Storandt, "Provable efficiency of contraction hierarchies with randomized preprocessing," 12 2015, pp. 479–490.

[18] P. D. H. Bast, "Lecture 7: Contraction hierarchies (part 2)," University of Freiburg, 2012. [Online]. Available: https://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2012

[19] "Langchain." [Online]. Available: https://python.langchain.com

[20] Github, "Repositories documentation." [Online]. Available: https://docs.github.com/en/repositories