Appendix B

Maintenance Manual

This section provides details on the implementation of this project. The aim is to enable others to easily install, modify, and extend the project, as well as to trace and debug any issues that may arise during execution. A menu application was also developed for this project to use as a way to test the BiS4EV algorithm; it will be discussed here also. Overall, the manual will cover the following topics:

- How to install the system and system dependencies.
- Menu application.
- Source code files.
- Constants.
- Directions for future improvements.
- Bug Reports.
- Directory structure.
- List of files submitted.

B.1 How to install the system and system dependencies

To run the code for this project, the only part that is needed is installing the external dependencies used. The dependencies used for this project are:

- textwrap
- scipy.stats
- seaborn
- matplotlib
- time
- re
- os
- matplotlib.pyplot
- pyrosm
- typing
- math
- networkx
- pandas
- numpy
- heapq
- osmnx
- random
- pyrosm.data
- scipy

To manage all of these modules, it is recommended to use the Anaconda distribution of Python[1]. Once Anaconda is installed, installing Pyrosm is done using the following command:

```
conda install -c conda-forge pyrosm
```

Before installing OSMNx, you must first install GeoPandas. This can be done with Anaconda by using the following command:

```
conda install -c conda-forge geopandas
```

Once Geopandas is installed, OSMNx can then also be installed using the following two commands:

```
conda config --prepend channels conda-forge
conda create -n ox --strict-channel-priority osmnx
```

The remaining dependencies should come with installing Anaconda.

Some of the classes and source code files developed in-house for this project were also imported as modules and used across files. As such they are assumed to be accessible in the top directory, and are listed and their role is explained in Table B.1.

B.2 Menu Application

A menu application was developed for this project to offer an interface for the BiS4EV algorithm. The application is a command line application that offers the user a way to specify which region, or city they are looking to travel in, and then a way to specify origin and target location. The application then attempts to find a route between the origin and target locations specified by using the path-finding functionality in BiS4EV. The result is returned to the user as a graphic of the region or city with the route mapped out. More details on how to use the menu application can be found in the User Manual appendix of this report.

This menu application is inspired by the Google Coding challenge that was part of Bright Network Technology Internship 2021. The original source code used for that challenge can be found here: `https://github.com/internship-experience-uk/google-code-sample/tree/main/python`.

B.2.1 Menu Application: Implementation details

The menu application consists of three files:

- run.py file
- Command_parser_location.py
- route_planner.py

The run.py file contains the while loop which runs the application. This while loop takes input from the user and passes that input to the command_parser_location file. This file contains the logic to parse the command inputted by the user and activates the correct functionality. There are

[1] `https://www.anaconda.com/download`

six commands available to the user, with some commands offering additional sub-commands. The following three show all of the commands available to the user, including the sub-commands.

```
Main menu
├── region - Starts region selector
│   ├── back - Go back to main menu
│   ├── help - Displays this help menu
│   └── places - Displays available datasets
├── origin - Starts origin selector
│   ├── back - Go back to main menu
│   └── help - Displays this help menu
├── target - Starts target selector
│   ├── back - Go back to main menu
│   └── help - Displays this help menu
├── help - Displays a help menu
├── exit - Terminates the program execution
└── choices - Displays the current choices
```

Depending on the command inputted, command_parser_location.py will activate the corresponding functionality. If a command is inputted which cannot be mapped to one of these six commands, or a subcommand if a previous command has been executed, an error message is thrown. Once the user has filled out a region, an origin, and a target, the application automatically asks if the user wants to proceed to the mapping of a route. Running – and parsing – each of the subcommands is done in a similar way to how the main body of the application is run: a while loop that activates certain functionality if certain if-statements are true.

B.3 Source code files

The roles of the remaining source code files, which are not covered in Table B.1, are explained in Table B.2. These were either run as separately as Python scripts, or used as part of the menu application (i.e. run.py).

B.4 Constants

There were several constants used for this project; a subset are listed in Figure B.1 These constants specify details that relate to the routes used to evaluate the algorithms: the places_dict contains a Python dictionary of the location name and the corresponding node ID in the graph; the route lists contains pairings corresponding to the start-target locations of each route. These constants are found in the testing_suite.py file and get_project_statistics.py files.

The other constants used in this project relate to the specific electric vehicle model that was used. They were:

- Efficiency: $\approx$ 1.87654 kWh/100km
- Total battery capacity: 77.0 kWh

B.5 Directions for future improvements

Three main directions are identified for future improvements as far as the source code is concerned:

- Finish (and improve) the implementation of BiS4EV;

Source code file	Explanation
auxilary	This file contains auxiliary functions used across the other files. This includes a range of functions from how to construct a contraction graph to reading paths to calculating charging time and more.
ListDict	ListDict is an externally found class used for managing a collection for efficient adding, removing, and pseudo-random choosing
BiS4EV	BiS4EV contains the implementation of BiS4EV for this project.
Dijkstra4EV	Dijkstra4EV contains the implementation of Dijkstra4EV for this project
graph	Contains the graph class which manages the pre-processing of the NetworkX graph representing the road network of the region or city used.
path	path is a class that represents a path alongside information on the start node, target node, and the battery consumption required to traverse the path.
Command_parser_location	This class is part of the menu application. It parses the commands passed into the command line and activates the appropriate functionality.
route_planner	This file is part of the menu application. It activates the path finding of BiS4EV based on the information given to the command line previously.

Table B.1: Table explaining the source code files that were imported as modules and used across files in this project.

- Extend the implementation of Dijkstra4EV;
- Extend the menu application.

This project was unsuccessful in implementing case C of BiS4EV. The first direction for future improvement is therefore to finish the implementation of BiS4EV. The implementation of BiS4EV could also be improved by using a priority queue to monitor which node should be considered next in each search direction.

Another future improvement could be to extend Dijkstra4EV to enable re-routing the path to a node via the nearest charging station. This would significantly improve the capabilities of the algorithm.

The menu application is very simple as of now. Adding further functionality to the application could be a future improvement. Additionally, as of now any charging stations the vehicle should stop at are not shown on the returned graphics. Including these in the graphics would help demonstrate the use of charging stations along the produced rote.

In addition to the future improvements listed above, the following directions are also suggested to improve the project as a whole:

- Enable dynamic changes in the availability of charging stations.

Source code file	Explanation
get_project_statistics	This source code file contains all of the code necessary to run and extract results from the statistical testing.
testing_suite	This file contains the testing suite developed for this project. All the code necessary to run the different tests evaluating the algorithms is contained here.
print_andorra	This file contains all the code used to produce the graphics of Andorra that were included in the project.
print_paths	This file contains all the code used to produce the graphics of the routes that were included in the project.
run.py	This is part of the menu application. Together with the parser, described in Table B.1, this file constitutes the main logic behind the menu application and interactions done with it.

Table B.2: Table explaining the roles of the remaining source files.

- Enable more electric vehicle models to be modelled as vehicles traversing the graph.

B.6 Bug Reports

There is a ShapelyDeprecationWarning that arises due to the version of Shapely used; this needs fixing.

The menu application assumes that the location inputted by the user actually exists in the specified region or city: no efforts are made to fact-check this. The application instead tries to map the inputted location to the nearest node given the coordinates returned by the OSMNx package and its geocoder module. This nearest node is found by using the nodes of the graph representing the road network of the region or city specified using the spatial package and its spatial module and the implementation of KD tree. As these coordinates returned by the geocoding can be incorrect, the wrong node could be found to be the nearest location, and as such the location inputted by the user can be incorrectly mapped to a location in the graph. If the user input cannot be geocoded at all, a Nominatim could not geocode query will be thrown by the geocoder module, and caught by the menu application. Adding functionality to see how close the geocoded coordinates are to the coordinates of the nearest approximated node would ensure that locations are mapped more correctly. Figure B.2 shows a suggested start to this bug fix. Additionally, when running the path finding of BiS4EV based on the user-inputted data, there is no error catching. There could be errors thrown by the algorithm and these should be caught appropriately.

Lastly, upon calculating the averages of the metrics for iterations that were capped, an average was calculated by summing and dividing the values by the number of values, despite there only being one value. This should not be done since the value that should be used is the value collected during that one iteration.

```
places_dict = {
    "Andorra La Vella": 2758504422,
    "Escaldes-Engordany": 51974050,
    "Sant Julia de Loria": 3610820021,
    "Encamp": 1934179573,
    "La Massana": 2975949958,
    "Ordino": 9721876525,
    "Canillo": 53275523,
    "El Pas de la Casa": 7296679289,
    "Arinsal": 52204280,
    "La Margineda": 277697480,
    "Sispony": 3268159387,
    "Llorts": 268615705,
    "Erts": 51554703,
    "Bixessarri": 52677211,
    "Aubinya": 52262417,
    "El Serrat": 51582448,
}
hard_routes = [("El Pas de la Casa", "Arinsal"), ("El Serrat", "Aubinya")]
easy_routes = [
    ("Encamp", "Canillo"),
    ("Ordino", "La Massana"),
    ("Bixessarri", "Sant Julia de Loria"),
    ("Arinsal", "Ordino"),
    ("Escaldes-Engordany", "Encamp"),
    ("Andorra La Vella", "Sispony"),
    ("La Margineda", "Aubinya"),
    ("Llorts", "Ordino"),
    ("Canillo", "Ordino"),
    ("Andorra La Vella", "Erts"),
]
```

Figure B.1: Constants used in this project

```
def in_region(point, point_approx, degree = 0.2):
print(point)
print(point_approx)
for x in range(len(point)):
    procent_difference = abs(point[x] - point_approx[x]) / (abs(point[x]
        + point_approx[x]) / 2)
    print(procent_difference)
    if procent_difference > degree:
        return False
return True
```

Figure B.2: Suggested code to validate of the approximated node based on the geocoded coordinates.

B.7 Directory structure

The code assumes the following directory structure:

```
Root directory
├── auxilary.py
├── BiS4EV.py
├── Dijkstra4EV.py
├── get_project_statistics.py
├── graph.py
├── ListDict.py
├── path.py
├── print_andorra.py
├── print_paths.py
├── README.md
├── testing_suite.py
├── data
│   ├── bis4ev
│   └── dijkstra
├── paths
│   ├── bis4ev
│   └── dijkstra
├── pics
├── report_pics
└── src
    ├── Command_parser_location.py
    ├── route_planner.py
    ├── run.py
    └── __init__.py
```

The data subdirectory contains a subdirectory for each algorithm containing the collected data for that algorithm. In this project, two different types of routes were used – easy and hard routes – and as Section B.8 will show, each type has its xlsx file. The paths subdirectory also contains a subdirectory for each algorithm. In these, the unique paths found were saved to .txt files. These can be read using the read_paths function in auxilary. They contain each node used in the path, each node separated by a new line. The pics subdirectory is where path .png files are saved throughout testing; the report_pics subdirectory is where photos used for the report are found.

B.8 List of files submitted

The source code submission, including data files, contains the following files:

```
Root directory
├── auxilary.py
├── BiS4EV.py
├── Dijkstra4EV.py
├── get_project_statistics.py
├── graph.py
├── ListDict.py
├── path.py
├── print_andorra.py
├── print_paths.py
├── README.md
├── testing_suite.py
├── pics
├── report_pics
├── data
│   ├── bis4ev
│   │   ├── easy.xlsx
│   │   └── hard.xlsx
│   └── dijkstra
│       ├── easy.xlsx
│       └── hard.xlsx
├── src
│   ├── Command_parser_location.py
│   ├── route_planner.py
│   ├── run.py
│   └── __init__.py
└── paths
    ├── bis4ev
    │   ├── bis4ev_Andorra La Vella_Erts_16_0_easy.txt
    │   ├── bis4ev_Andorra La Vella_Sispony_16_0_easy.txt
    │   ├── bis4ev_Arinsal_Ordino_16_0_easy.txt
    │   ├── bis4ev_Arinsal_Ordino_29_3_easy.txt
    │   ├── bis4ev_Bixessarri_Sant Julia de Loria_16_0_easy.txt
    │   ├── bis4ev_Canillo_Ordino_16_0_easy.txt
    │   ├── bis4ev_El Pas de la Casa_Arinsal_16_0_hard.txt
    │   ├── bis4ev_El Pas de la Casa_Arinsal_29_3_hard.txt
    │   ├── bis4ev_El Serrat_Aubinya_16_0_hard.txt
    │   ├── bis4ev_Encamp_Canillo_16_0_easy.txt
    │   ├── bis4ev_Encamp_Canillo_4_3_easy.txt
    │   ├── bis4ev_Escaldes-Engordany_Encamp_16_0_easy.txt
    │   ├── bis4ev_La Margineda_Aubinya_16_0_easy.txt
    │   ├── bis4ev_Llorts_Ordino_16_0_easy.txt
    │   └── bis4ev_Ordino_La Massana_16_0_easy.txt
    └── dijkstra
        └── dijkstra_Andorra La Vella_Sispony_16_0_easy.txt
```