



Student Name: AFOLABI SAMAD	Matriculation Number: 1910611
Supervisor: KATE HAN	Second Marker:
Course: Information Technology with Cybersecurity	
Project Title: Data-Driven Modeling of Mobility in Scotland	
Project Investigation [CMM012] <input type="checkbox"/>	Start Date: 07/06/2021
MSc Project [CMM013] <input checked="" type="checkbox"/>	Submission Date: 20/12/2021
(Tick as appropriate)	

CONSENT

I agree ☒
I do not agree ☐

That the University shall be entitled to use any results, materials or other outcomes arising from my project work for the purposes of non-commercial teaching and research, including collaboration.

DECLARATION

I confirm:

- **That the work contained in this document has been composed solely by myself and that I have not made use of any unauthorised assistance.**
- **That the work has not been accepted in any previous application for a degree.**
- **All sources of information have been specifically acknowledged and all verbatim extracts are distinguished by quotation marks.**

Student Signature: AFOLABI SAMAD OLATOYE	Date Signed: 20/12/2021
---------------------------------------------	-------------------------



Abstract

The application of path finding algorithms to real world graph data, the use of modern standards to share information and the understanding of passenger transport networks. All these offer the opportunity to contribute to understanding and solving problems that occur in an cities infrastructure. HITRANS which is a regional Highlands and Islands transport partnership which gives grants and loans to implement regional transport strategy, participates in community planning to make transport in Scotland better. Partnered with the Robert Gordon University SDTC, they developed a software model to be able to perform analysis on transport in Scotland.

The report outlines a summary of the initial project investigation, a review of the software design, details of the implementation process along with the results of developing the programs.

This report provides a summary of a project to implement a path finding algorithm on a graph depicting Scotland's passenger transport network and a software program to convert files in CIF format to GTFS format.

Acknowledgement

I would like to show my gratitude.

To my parents for the constant support and for all the possibilities they provide for me.

To my supervisor for being patient with me and guiding through this project.

I would like to dedicate this work to everyone that helped me during my journey to this point.

Thank you.

Declaration

I confirm that the work contained in this MSc project report has been composed solely by myself and has not been accepted in any previous application for a degree. All sources of information have been acknowledged.

Signed:.....Afolabi Samad.....

Date: ..20 December 2021..

Contents

Abstract.....	2
Acknowledgement	3
Declaration.....	4
List of Tables	6
List of Figures.....	7
Listings.....	8
Introduction.....	9
Implementation	21
Testing and Evaluation.....	41
Conclusions.....	45
Bibliography	47
Project Management.....	49

List of Tables

Table 1: Terms and their definitions in the ATCO-CIF specification

Table 2: Terms and their definitions in the GTFS static reference (Google Developers, 2021)

Table 3: Text files and their descriptions in GTFS dataset (Google Developers, 2021)

Table 4: Showing a representation of the Node costs (Wotlmann S, 2021)

Table 5: Third-party libraries used and their description

Table 6: agency.txt fields, description and the CIF equivalent requirement

Table 7: stops.txt fields, description and the CIF equivalent requirement

Table 8: routes.txt fields, description and the CIF equivalent requirement

Table 9: trips.txt fields, description and the CIF equivalent requirement

Table 10: stop_times.txt fields, description and the CIF equivalent requirement

Table 11: calendar.txt fields, description and the CIF equivalent requirement

Table 12: calendar_dates.txt fields, description and the CIF equivalent requirement

List of Figures

Figure 1: Diagram showing how GTFS information interact (Opentransportdata.swiss 2021)

Figure 2: Diagram showing a weighted graph (Woltmann S, 2021)

Figure 3: Diagram showing a weighted graph with Coordinate System (Woltmann S, 2021)

Figure 4: A* Algorithm pseudocode

Figure 5: Flowchart diagram for A* algorithm (Zidane 2018)

Figure 6: Graphical User Interface (GUI) for executing the A* and Dijkstra Algorithm

Figure 7: GUI for executing the converter

Figure 8: GUI showing the generated result from the A* algorithm

Figure 9: GUI showing the cost and computation time of Dijkstra algorithm

Figure 10: GUI showing cost and computation time of A* algorithm

Figure 11: Column chart comparing the Dijkstra and A* algorithm Computation time

Figure 12: GTFS validation tool results

Listings

Listing 1: HashMap containing record identities

Listing 2: Functions executed depending on type

Listing 3: Code samples of functions to process records identified

Listing 4: Code sample to format latitude and longitude

Listing 5: Code sample for generating stops data for stops.txt

Chapter 1

Introduction

This chapter introduces the topics covered in this report and summarizes what has been discussed in the project investigation report. First, a short section will state the author, the context and the purpose of this body of work. Afterthat, an overview of the project will inform the reader about the objectives of this project and the content of this report. The third section will briefly discuss the theoretical background, summing up the relevant concepts.

After that, the outcomes of the investigation report will be presented as a starting point for the practical work at the center of this report. Lastly, the list of all the following chapters and their content will be illustrated.

1.1 Thesis Overview

This is the thesis of Afolabi Samad Olatoye, submitted as part of the requirements for the degree of Master of Science in Information Technology with Cybersecurity at the School of Computing, Robert Gordon University, Scotland. The project focuses on the implementation of the A* (A-star) algorithm for pathfinding on a graph representing Scotland's passenger transport network and the implementation of a software program to convert ATCO-CIF files which is considered a legacy format to the GTFS format using the Java programming language. This is the experimental report that follows what has been analyzed and discussed in the investigation report.

1.2 Project Overview

Passenger transport is at the heart of this project. It is the total movement of passengers from one location to another using inland transport such as passenger cars, buses, coaches, trams or coaches on a given network. Passenger transport systems operate mostly on intracity, suburban, intercity and international levels. The analysis and understanding of passenger transport data gives insight into human mobility which in turn aids in urban planning, predicting migratory flows, forecasting traffic and epidemic modelling. The authors interest in this project stems from a fascination with pathfinding algorithms and graphs as a whole. Pathfinding algorithms are a very important aspect of human mobility as this allows the best possible path to be determined to reach the desired destination. Graphs have become a means to represent complex relationships that occur in the real world. Passenger transport is no exception to this as massive road networks are represented as graphs to be consumed by algorithms to achieve a specific objective.

Robert Gordon University's Smart Data Technologies Centre (SDTC) research team and Highlands and Islands Transport Partnership (HITRANS) have a partnership which lead to the development of an interactive computerized model to visualize and analyze the Scottish passenger transport network. The result of this collaboration is a fully functioning software developed using the Java programming language for performing inbound and outbound reachability analysis using OpenStreetMap and GTFS data.

This project is intended to contribute to the improvement of this computerized model which in turn aids in performing larger analysis and creating an evolved version of the model of the Scottish passenger transport network.

1.3 Theoretical Background

This project aims to use the Java programming language to implement the A* algorithm and to develop a program to convert CIF file format to GTFS file format. Given that, the theoretical background mainly involves the understanding of the following features: ATCO-CIF specification, GTFS specification, Graph searching algorithms.

Some of these topics have been discussed in detail in the investigation report and the following subsections will summarize the content.

1.3.1 The ATCO-CIF Specification

This section focuses on introducing the most relevant aspects regarding the ATCO-CIF specification with respect to the development of this project, where ATCO-CIF files are converted to GTFS files.

First, the ATCO-CIF specification defines records on each row of the file for the interchange of timetable data. The records are distinguished by two letter identifiers in characters 1 and 2 of a total length of 120 characters. This transfer format doesn't define the quality of data being transferred or the coding schemes being used.

The specification provides us a guideline for recording timetable data. Some terms are defined in the specification in relation to this. Each term represents a different aspect of the specification.

Term	Definition
Journey	Movement of a vehicle such as a bus or a train described by a sequence of stop points and times from a starting point to an ending point
Service	A group of journeys with common stop points
Route	Comprises one or more services
Route number	Used to identify vehicle undertaking journey
Operator	In charge of services and identified by company trading name
Stops	Locations at which events happen during a journey

Events	Describes what happens at stops in a journey E.g., Arrives, Departs, doesn't stop, Stops
Valid days	Days and other special days of the week the journey operates
Valid dates	First and last date of operation of the journey
Clusters	Groupings of stops where journeys can be interchanged
Interchange time	Minimum time needed to change between journeys at a stop

Table 1: Terms and their definitions in the ATCO-CIF specification

Given these definitions, a simple way of converting this file format is to consider the attributes of the CIF file format that describe the characteristics of the transport system. The use of the CIF file format by transport agencies in the UK such as National Rail, Traveline is why the format is being considered for conversion. ATCO-CIF in its specification is a standard for representing and connecting timetable data among different public transport agencies. The standard is used as a general-purpose transfer means of the more common elements of timetable enquiry information between different databases of the proprietary type. It leaves some fields blank where the data may not be available in the exported database.

Having said that, the ATCO-CIF specification is a list of requirements that must be followed for the delivery of more solid and meaningful information. These requirements aid in the proper conversion and delivery of the timetable information to the GTFS (General Transit Feed Specification) requirements.

1.3.2 General Transit Feed Specification

General Transit Feed Specification (GTFS) is an exchange format developed by Google for public transport services timetables and other relevant geographic information such as stops or locations. The data used is provided by transport companies and published as accumulated database and used to develop applications related to public transport. There are different forms of GTFS data: GTFS real-time and GTFS static. GTFS real-time as indicated by the name provides real-time data in the GTFS format while GTFS static provides static data in GTFS format. The static data format is what is used in this project to present the transport

data. The GTFS document defines terms that are defined below and used to illustrate how ATCO-CIF requirements are matched to the GTFS requirements.

Terms	Definitions
Dataset	Complete set of files defined in this specification reference. Alteration of the dataset creates a new variation of the dataset
Record	Data structure containing several dissimilar field values describing a single entity represented in a table as a row.
Field	Property of an object or entity represented as a column
Field value	Individual entry in a field represented as a single cell
Required	This must be included in the dataset and a value provided in that field for each record. An empty string can be used as a value in some required fields
Optional	This can be omitted from the dataset.
Conditionally required	This field or file is required under certain conditions.
Service day	A time period used to indicate route scheduling. It varies from agency to agency, but they sometimes don't correspond with calendar days.

Table 2: Terms and their definitions in the GTFS static reference (Google Developers, 2021)

Technical Description

The GTFS static is kept in zip format containing a series of text files. Each text file provides information about a certain aspect of the transit information, for example stops represented as stops.txt, routes represented as routes.txt transport companies represented as agency.txt and other related data. In this specification, some information is required while others are optional but still provide useful information such as changes to the timetable on specific public holidays found in calendar_dates.txt. The figure below illustrates how the different information types relate.

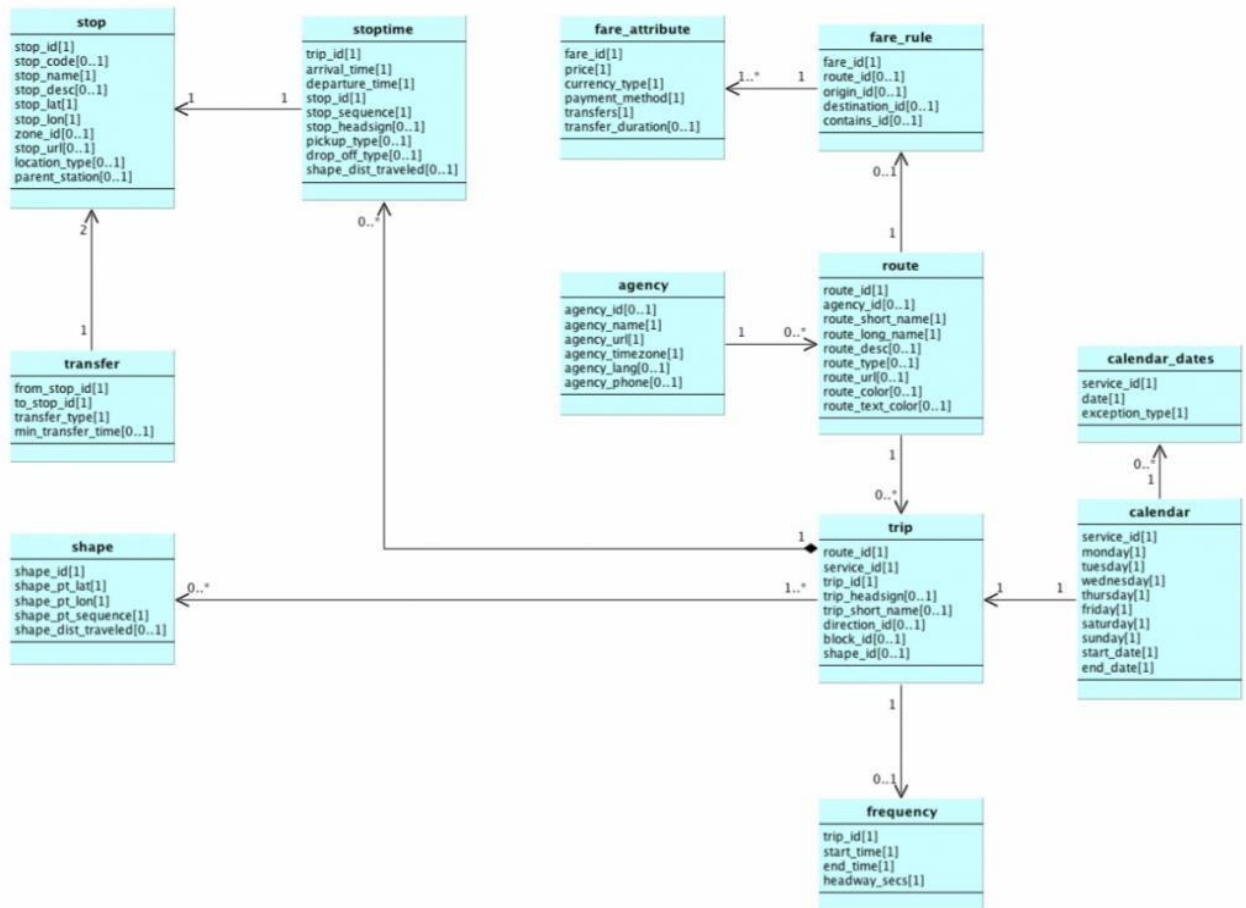


Figure 1: Diagram showing how GTFS information interact (Opentransportdata.swiss 2021)

Each text file in the zipped folder contains Comma-Separated values (CSV) of texts and numbers which combined, provide the information being searched for. The files contained in the zip folder are indicated in the table below.

File Name	Description
Agency.txt	Transport agencies with service involved in this dataset
Calendar.txt	Dates of service specified using a weekly schedule. Required unless dates of service are provided in calendar_dates.txt
Calendar_dates.txt	Contains exceptions define in calendar.txt but becomes required if calendar.txt is not provided
Feed_info.txt	Contains metadata such as publisher, version and expiration information
Routes.txt	A group of trips provided to users as a single service
Stops.txt	Where passengers are picked up or dropped off
Stop_times.txt	Times at which vehicles arrive and depart from stops

Trips.txt	A sequence of two or more stops during a particular time period
-----------	-----------------------------------------------------------------

Table 3: Text files and their descriptions in GTFS dataset (Google Developers, 2021)

There are other files that are contained in the zip folder such as fare_attributes.txt, fare_rules.txt, shapes.txt, frequencies.txt, pathways.txt, levels.txt, transfers.txt, translations.txt, but they are optional and not included in this project. A description of this files can be found in Appendix.

1.3.3 Graph Searching Algorithm

Graphs are an abstract representation that describe the organization of transportation systems, human interactions and telecommunication networks (Skiena 2020). A graph consists of a set of vertices V (sometimes referred to as nodes) together with a set of edges E or vertex pairs.

Graphs can essentially model any relationship. An example is a road network model with towns or cities being the vertices and the roads connecting them being the edges or an electric circuit model with the junctions as vertices and the components as edges. There are different flavors or types of graphs:

- **Undirected vs Directed:** The graph is undirected if it allows back and forth movement between two nodes or vertices. An example is road network in cities that have lanes that allow movement in both directions. One-way streets in cities can be considered directed since movement is from one point to the next i.e., it moves in one direction.
- **Weighted vs Unweighted:** The edges in a graph are weighted if there is a numerical value assigned to them. An example of this using road networks may be the drive time between vertices, speed limit or distance depending on the application. In unweighted graphs, the edges have no numerical value assigned to them.

Other types of graphs include:

- Simple vs non-simple
- Cyclic vs Acyclic
- Sparse vs Dense
- Implicit vs Explicit
- Labelled vs Unlabeled

- Embedded vs Topological

This project focuses on trying to find shortest path on road networks between two vertices. A path is a sequence of edges connecting two edges. In a road network, there are usually many probable paths connecting two vertices. The path that produces the lowest sum of edge weights is likely the fastest travel path between the two vertices. The algorithm must be able to efficiently traverse the graph in a systematic way. One key idea behind traversing a graph is to visit every node or vertex and edge and keep track of any node that has not yet been explored. An algorithm for finding shortest path in a weighted graph is described below:

- Dijkstra's Algorithm: This is one of the most popular choices for finding the shortest path between two nodes in an edge/vertex weighted graph. Taking a starting position at vertex s , it finds the shortest path from s to all other vertices in the graph including the destination node d . Suppose the shortest path from start node s to destination node d passes through an intermediate node y , the best path from s to d must contain a shortest path from s to y , if it doesn't, the path from s to d can be improved by finding the shortest path from s - y prefix to destination d .

The algorithm to be implemented in this project to find shortest paths in the road network graph for Scotland provided for this project is the A-Star algorithm (A* algorithm) which is a variation on the Dijkstra's algorithm described above. The Dijkstra algorithm has already been implemented in the software model developed by SDTC as mentioned in section 1.2.

The A* algorithm was discussed in the investigation report submitted for this project. A quick overview of this algorithm and more details on the heuristics to be used are looked at in the next section.

1.3.4 A* Algorithm

The Dijkstra algorithm finds shortest paths from a node s to a destination d by searching through the graph for all possible paths toward the destination d and returning the path with the smallest weight sum. The A* differs from Dijkstra by prematurely terminating paths leading in the wrong direction. To achieve this, it uses a heuristic that can calculate the shortest possible distance to the destination d from each node with small effort.

Using a sample weighted graph displayed in the figure below:

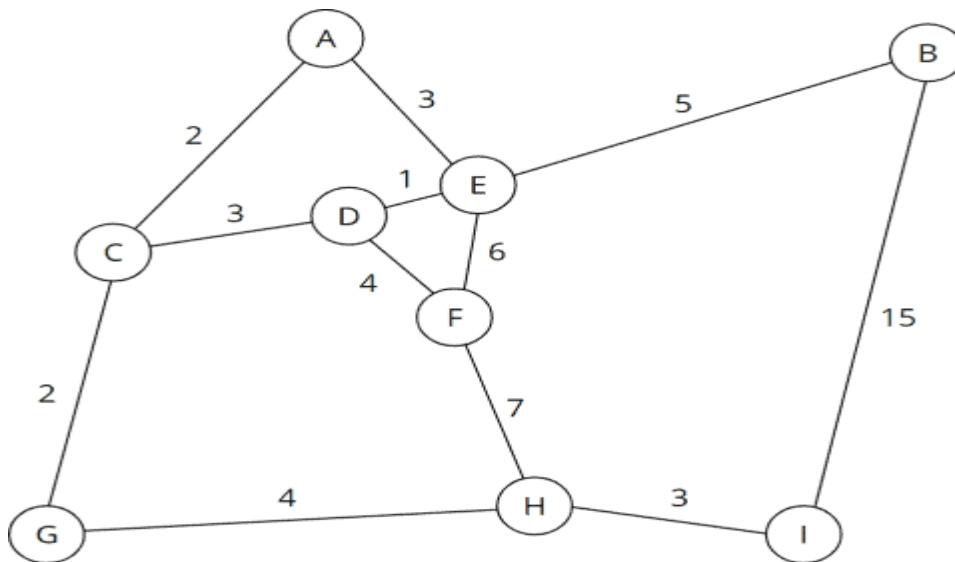


Figure 2: Diagram showing a weighted graph (Woltmann S, 2021)

A major difference I highlighted above between Dijkstra and A* algorithm is the use of a heuristic to calculate the fastest possible path from all nodes on the graph to the destination node *d*. The graph shown in the figure above represents a two-dimensional map, so a suitable heuristic is the Euclidean distance simply put a straight line to the destination. This is done so the algorithm prioritizes the nodes headed in the right direction later. The heuristic must never overestimate the actual costs that would be accumulated to reach the destination. Adding a coordinate system to the graph in the figure above would produce a means to determine an educated guess about the distance to the destination.

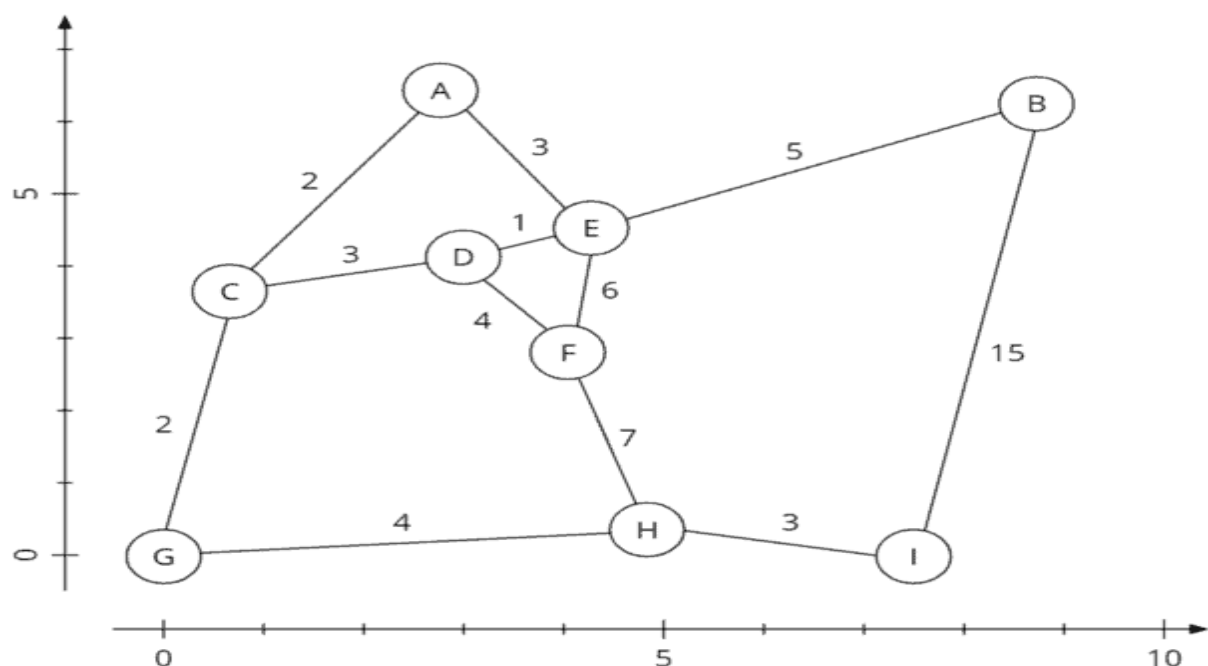


Figure 3: Diagram showing a weighted graph with Coordinate System (Woltmann S, 2021)

Determining a suitable value for the heuristic can be done using the coordinate system. An example would be trying to determine a suitable distance value from node G to node B which in this case can be estimated to be in the range of 9 to 10. When the algorithm gets to the next node, the heuristic is checked again, and if its higher than the previous node, this indicates that the current node is not in the right direction. This process is repeated until the shortest path with the least weight sum is reached. A known formula for determining which node is next to be visited is the $f(n) = g(n) + h(n)$ where f represents the sum of all costs, g represents the cost of the current node and h represents the heuristic value.

Node	Parent Node	Total Cost from Start $g(n)$	Minimum Remaining Costs to Target $h(n)$	Sum of Costs $f(n)$
D	–	0.0	8.5	8.5
A	–	∞	8.9	∞
B	–	∞	0.0	∞
C	–	∞	9.2	∞
E	–	∞	7.5	∞
F	–	∞	5.5	∞
G	–	∞	9.6	∞
H	–	∞	4.0	∞
I	–	∞	1.6	∞

Table 4: Showing a representation of the Node costs (Wotlmann S, 2021)

The table describes how the cost is represented with the addition of the heuristic. The node with the minimum $f(n)$ value is picked as the next node, with this repeating process the algorithm moves in the right direction to the destination reducing the number of nodes that need to be traversed.

The example used to describe how A* worked made use of a heuristic method called Euclidean Distance. There are different heuristics which are outlined below along with details on when they should be used:

- **Euclidean Distance:** The straight-line distance between two points (Euclidean distance, 2021). It is often used when movement is allowed in all directions. This is used in our project because of no limitation on the directions.
- **Manhattan Distance:** Distance between two points measured along axis at right angles. It is often used in integrated circuits where the wires run parallel to the X or Y axis. It is often used when movement is only allowed in four directions.
- **Diagonal Distance:** Mostly used when movement is limited to 8 directions like how the king moves in a chess piece (GeeksforGeeks, 2021).

1.4 Investigation Report Summary

The investigation phase of this project is summarized in Sections 1.3 and 1.4, the final outcomes of this investigation report are outlined below before moving on to the actual design and implementation of the project:

- The ATCO-CIF format is a format used by public transportation agencies in the UK. The requirements outlined in the ATCO-CIF specification act as a standard for communicating timetable information data. The specification outlines a lot of records detailing different stages of transporting passengers using public transport system.
- The General Transit Feed Specification GTFS is a specification developed by Google to communicate timetable information data from public transport agencies. It achieves the same purpose as ATCO-CIF for storing transport timetable data but presents in a different format. It makes use of text files containing CSV (Comma-Separated Values).
- Graph searching algorithms possess a variety of applications such as delivering data across the internet, traveling between locations. These project makes use of graph searching algorithms to find shortest paths between locations for public transport systems.

1.5 Chapter List

The introductory aspect of this report is concluded by presenting a brief description of the following chapters as provided

Chapter 2 Implementation.

Chapter 3 Evaluation.

Chapter 4 Conclusion.

Chapter 2

Implementation

This chapter makes use of the research that has been discussed in the project investigation report and summarized in chapter 1 as the beginning of describing the design and practical implementation of a software program to convert ATCO-CIF files to GTFS and to implement a graph searching algorithm known as A-Star (A*) on a graph modeling Scotland's passenger transport network. An outline of what this section covers is listed:

- Section 2.1 – Implementation plan and design for the software programs.
- Section 2.2 – Software description which looks at the pseudocode of the A* algorithm and breaks down how each section is implemented.
- Section 2.3 – Data Processing which looks at sections of the file formats to be converted and what parts of each document requirement matches to the other document requirement.
- Section 2.4 – Data Converter which looks at the software program created to convert the files in CIF format to the files in GTFS format.
- Section 2.5 – Conclusion for the chapter.

2.1 Implementation

The project leverages the use of some aspects of the waterfall methodology to design and implement this project. The project is entirely coded in the Java programming language using the NetBeans Code Editor. The Java programming language was selected because the API's used to interact with the graph use Java and it would allow better integration with the software developed by RGU and HITRANS.

A list of the third-party libraries used in the software are outlined in the table below:

Third-Party Library	Description
rgu-algorithms-2.0.0-beta	Contains implementations for BinaryMinHeap data structure, Dijkstra's algorithm, an interface for exploring the graphs
rgu-transport-2.0.0-beta	Contains implementations to for holding latitude, longitude, parsing GTFS files
onebusaway-uk-atco-cif-parser	Contains functions for reading ATCO-CIF files

Table 5: Third-party libraries used and their description

2.2 A* Algorithm

This section looks at the pseudocode for the A* algorithm, with explanations on the algorithm and the choices made in the implementation of the algorithm.

2.2.1 Pseudocode of A* Algorithm

The A* algorithm builds on the principles of Dijkstra's shortest path algorithm to provide a faster result when trying to find the shortest path between 2 points on a graph. (Zobenika 2021)

- One of the first requirements is a graph which has noted in section 2.1 is available along with third party libraries that provide application programmable interfaces (API) for interacting with the graph.
- Another requirement is a data structure to hold the list of visited points on the graph and another data structure to hold the list of unvisited points on the graph.

```

function A*(start,goal)
  closedset := the empty set    // The set of nodes already evaluated.
  openset := {start}           // The set of tentative nodes to be evaluated, initially containing the start node
  came_from := the empty map    // The map of navigated nodes.
  g_score[start] := 0           // Cost from start along best known path.
  f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

  while openset is not empty
    current := the node in openset having the lowest f_score[] value
    if current = goal
      return reconstruct_path(came_from, goal)

    remove current from openset
    add current to closedset
    for each neighbor in neighbor_nodes(current)
      tentative_g_score := g_score[current] + dist_between(current,neighbor)
      if neighbor in closedset
        if tentative_g_score >= g_score[neighbor]
          continue

      if neighbor not in openset or tentative_g_score < g_score[neighbor]
        came_from[neighbor] := current
        g_score[neighbor] := tentative_g_score
        f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
        if neighbor not in openset
          add neighbor to openset

  return failure

function reconstruct_path(came_from, current_node)
  if came_from[current_node] in set
    p := reconstruct_path(came_from, came_from[current_node])
    return (p + current_node)
  else
    return current_node

```

Figure 4: A* Algorithm pseudocode

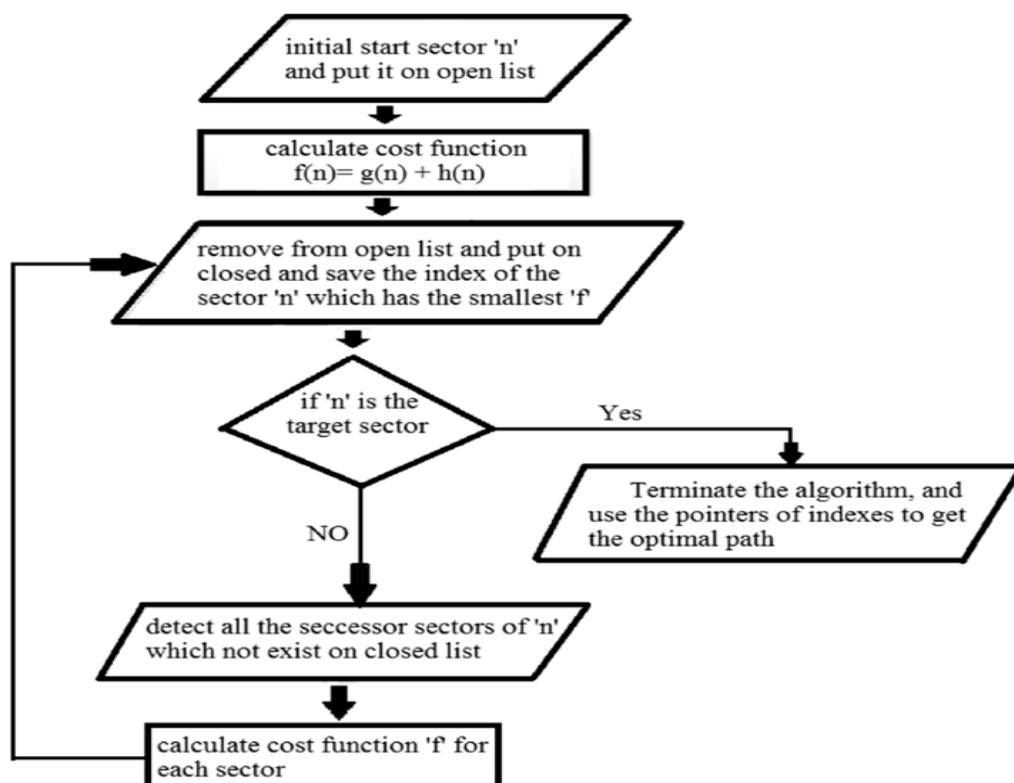


Figure 5: Flowchart diagram for A* algorithm (Zidane 2018)

2.2.2 A* Code Implementation

This section looks at the Java code implementation of the A* algorithm. The pseudocode shown above is used to implement the A* algorithm on the graph used for this project. The nodes for the graph are represented using an object called “GeoLocation” as will be seen throughout this section. The “GeoLocation” object represents the latitude and longitude of each point on the graph network. The graph provide for this project can be loaded to output weight between nodes in duration (seconds, minutes or hours) or distance (metres, kilometres). This can be seen below where we make use of duration as the unit of weight of edges.

```
Graph<GeoLocation, Duration> graph = DurationAdapter.adapt(roadGraph);
```

The example below shows the graph being loaded to allow for distance as the unit of the edges.

```
Graph<GeoLocation, Distance> graph1 = getDistanceGraph(roadGraph);
```

The graph used for this project represents nodes using longitude and latitude which indicate real world locations in Scotland. The figure below shows an example of the longitude and latitude inputs representing nodes. The latitude and longitude of the start point indicate some random location near inverness but the start point is not selected as that point because it is not represented on the graph network.

The screenshot displays the 'RGU SCHOOL PROJECT' web application. It features two main sections for algorithm input: 'DIJKSTRA ALGORITHM' and 'ASTAR ALGORITHM'. Each section has a 'START POINT' and an 'END POINT' with input fields for 'LATITUDE' and 'LONGITUDE'. The 'DIJKSTRA ALGORITHM' section has a 'RUN' button. The 'ASTAR ALGORITHM' section also has a 'RUN' button. At the bottom, there are two large empty boxes labeled 'DIJKSTRA COST' and 'COMPUTATION TIME' on the left, and 'ASTAR COST' and 'COMPUTATION TIME' on the right.

RGU SCHOOL PROJECT			
DIJKSTRA ALGORITHM		END POINT	
LATITUDE	<input type="text" value="57.43"/>	LATITUDE	<input type="text" value="57.5953"/>
LONGITUDE	<input type="text" value="-4.36"/>	LONGITUDE	<input type="text" value="-4.4284"/>
<input type="button" value="RUN"/>			
ASTAR ALGORITHM		END POINT	
LATITUDE	<input type="text" value="57.43"/>	LATITUDE	<input type="text" value="57.5953"/>
LONGITUDE	<input type="text" value="-4.36"/>	LONGITUDE	<input type="text" value="-4.4284"/>
<input type="button" value="RUN"/>			
DIJKSTRA COST		COMPUTATION TIME	
<div></div>		<div></div>	
ASTAR COST		COMPUTATION TIME	
<div></div>		<div></div>	

Figure 6: Graphical User Interface (GUI) for executing the A and Dijkstra Algorithm.*

The NearestFinder API is called to find the nearest point to the latitude and longitude that are provided as inputs. This API finds the nearest point to the input which exists on the network. This is shown below.

```
NearestFinder<GeoLocation> finder = GeoLocation.nearestHaversine(graph.vertices());  
  
    // find a point near to our example point, but actually on the network  
    // we should find a point quite close to our example point  
    GeoLocation startPoint = finder.nearest(examplePoint);
```

The A* algorithm involves the use of data structures to hold the nodes on the graph that have been explored, the nodes that will be explored next and the parents of nodes that have been explored. The data structure which holds the set of the nodes to be explored is called the Open list and is implemented using the BinaryMinHeap provided as an API in the RGU algorithms third party library.

```
MinHeap<GeoLocation, Long> open = new BinaryMinHeap<>((a, b) -> { return a.compareTo(b) < 0;  
});
```

The BinaryMinHeap is not the only data structure that could be used to accomplish the purpose of holding nodes to be explored. Other data structures that could be used include the PriorityQueue included in the Java library and the TreeSet data structure. These data structures are used for their additional functionalities of being able to return the minimum element according to conditions defined. The reason the BinaryMinHeap was selected was because it did not require the creation of new classes to wrap the nodes which made it easier to fill it with nodes that required processing or being processed.

The term data structure is used to refer to the underlying data structures that the above implement. BinaryMinHeap implements the Heap data structure, the PriorityQueue implements the Queue data structure and the TreeSet implements the Set data structure.

The other list required for the A* algorithm is the closed list which contains a list of nodes to hold the nodes that have been explored. As indicated in Chapter 1, an explored node means a node for which its cost to a previous node and its neighbours have been determined. The data structure used to hold this set of nodes is the HashSet which implements the Set data structure. This data structure holds just one set of objects which are nodes that have been explored as shown below.

```
Set<GeoLocation> closed = new HashSet<>();
```

The other list is the parent list which holds the previous nodes of each node explored. This list is used to trace back the path used by the algorithm when the destination has been

found. The data structure holds two objects, the parent node and the child node. The list can be seen initialised below.

```
Map<GeoLocation, GeoLocation> previous = new HashMap<>();
```

As noted in the explanation of the A* algorithm in Section 1.3.4, the A* algorithm determines its next point on the graph using the formula $G(n) = F(n) + H(n)$. This is initialised in the code using the HashMap which implements the Map data structure. The Map data structure works like a dictionary, there is a key and a value. In this context, the node is the key and its duration is its value.

```
Map<GeoLocation, Long> gScore = new HashMap<>(); ----- G(n)
Map<GeoLocation, Long> fScore = new HashMap<>(); -----F(n)
```

The data structures required have been initialised. The empty data structures are filled with initial values. The parent list is filled with the starting node and a null value because the starting node has no parent node. The node in this case is the point on the graph network closest to the location input that is provided. The G(n) is filled with the starting node and a score of 0 since it cost nothing to get to the start. The F(n) is filled with the starting node and the heuristic value to get to the destination. The open list is also filled with the starting node and a cost of 0.

```
previous.put(source, null);
gScore.put(source, (long) 0.0);
fScore.put(source, Speed.ofKilometersPerHour(100).seconds(heuristic(source, target)).toSeconds());
open.addElement(source, (long) 0.0);
```

The heuristic is determined using the HAVERSINE api provided in the third-party library. The Haversine library implements Haversine distance heuristic which is widely used. As noted in Section 1.3.4, there are different heuristics which can be implemented such as Euclidean distance and Manhattan distance. This heuristic function is shown below.

```
private static Distance heuristic(GeoLocation start, GeoLocation end) {
    return GeoLocation.HAVERSINE.distance(start, end);
}
```

The addition of the heuristic occurs again in the loop to go through the graph. The line below shows the addition of G(n) to a new heuristic value generated using the current node being evaluated and the target location to create the new F(n) value to be used to determine the next node to be evaluated.

```
Long newFScore = newGScore + Speed.ofKilometersPerHour(100).seconds(heuristic(child,
target)).toSeconds();
```

This process is repeated till the final destination has been reached. A function then takes the parent list and the current node to generate a list of the paths taken to reach the destination. This function is shown below. A java method in the Collections library is used to generate the path taken.

```
private static List<GeoLocation> reConstruct(Map<GeoLocation, GeoLocation> previous, GeoLocation
current) {
    List<GeoLocation> path = new ArrayList<>();
    GeoLocation currentNode = current;

    while (currentNode != null) {
        path.add(currentNode);

        currentNode = previous.get(currentNode);
    }

    Collections.reverse(path);
    return path;
}
```

2.3 Data Processing

This section focuses on the phase of the implementation which consists of all the operations that go into making the CIF file format from how its originally provided to input data that can be converted into GTFS format. The two document format requirements are analyzed, and the equivalent requirements are determined to make conversion an easier process. The approach that will be used is by considering GTFS file to be populated and then the sections in the CIF files are identified to match the records.

2.3.1 agency.txt

The agency.txt file is a required file in the GTFS requirements. It contains 8 fields of which 3 are required, 1 is conditionally required and the rest are optional. Indicated in the table below are the different field names, description and the equivalent value in a CIF file. The blank sections that do not show a CIF equivalent are hardcoded or provided.

Field Name	Description	CIF Equivalent
------------	-------------	----------------

Agency_id (Conditionally required)	Required when the dataset contains multiple transport agencies. It is optional when it contains only one transport agency.	Record identity: QS (Journey header) Position: starts at 4 th position Max characters: 4
Agency_name (required)	Full name of the transport agency	Record identity: QP (Operator) Position: starts at 32 nd position Max characters: 48
Agency_url (required)	URL of the transport agency	Provided
Agency_timezone (required)	Time zone where transport agencies are located. Only one time zone allowed per dataset even if multiple transport agencies are indicated.	Provided

Table 6: agency.txt fields, description and the CIF equivalent requirement

2.3.2 stops.txt

The stops.txt file is a required file in the GTFS requirements. It contains 14 fields of which 1 is required, 5 are conditionally required and the rest are optional. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
Stop_id (required)	Used to identify a stop, station or station entrance. Stops, stations or station entrances are referred to as locations	Record identity: QL (Bus Location) Position: Starts at 4 th position Max characters: 12
Stop_name (conditionally required)	Name of the location	Record identity: QL Position: Starts at 16 th position Max characters: 48
Stop_lat (conditionally required)	Latitude of the location	Record identity: QB Position: Starts at 16 th position Max characters: 8

Stop_lon (conditionally required)	Longitude of the location	Record identity: QB Position: Starts at 24 th position Max characters: 8
--------------------------------------	---------------------------	-------------------------------------------------------------------------------------------

Table 7: stops.txt fields, description and the CIF equivalent requirement

2.3.3 routes.txt

The routes.txt file is a required file in the GTFS requirements. It contains 12 fields of which 2 are required, 3 are conditionally required and the rest are optional. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
Route_id	Used to identify a route	Record identity: QS Position: 39 th position Max characters: 4
Agency_id	References agency id in agency.txt, required if multiple agency routes are present in the dataset	Same requirement as agency_id in agency.txt
Route_short_name	Short name of a route	
Route_long_name	Full name of a route	
Route_type	Indicates type of transportation operating on the route	Record identity: QS Position: 49 th position Max characters: 8

Table 8: routes.txt fields, description and the CIF equivalent requirement

2.3.4 trips.txt

The trips.txt file is a required file in the GTFS requirements. It contains 10 fields of which 3 are required, 1 is conditionally required and the rest are optional. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
Route_id (required)	Route identifier	Same requirement as route_id in routes.txt

Service_id (required)	References service.id in calendar or calendar_dates	Same requirement as service_id in calendar.txt
Trip_id (required)	Trip identifier	

Table 9: trips.txt fields, description and the CIF equivalent requirement

2.3.5 stop_times.txt

The stop_times.txt file is a required file in the GTFS requirements. It contains 12 fields of which 3 are required, 2 are conditionally required and the rest are optional. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
Trip_id (required)	References trip_id in trips.txt	Same requirement as trips_id in trips.txt
Arrival_time (conditionally required)	This indicates arrival time at a specific stop for a specific trip on a route	Record identity: QI Position: 15 th position Max characters: 4
Departure_time (conditionally required)	This indicates departure time from a specific stop for a specific trip on a route	Record identity: QI Position: 19 th position Max characters: 4
Stop_id (required)	References stop_id in stops.txt	Same requirement as stop_id in stops.txt
Stop_sequence (required)	This is the order of stops for a particular trip	

Table 10: stop_times.txt fields, description and the CIF equivalent requirement

2.3.6 calendar.txt

The routes.txt file is a required file in the GTFS requirements. It contains 10 fields of which all are required. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
Service_id	Identifies set of dates when service is available	

Monday (required)	Shows whether service operates on all Mondays between the start_date and end_date values	Record identity: QS Position: 30 th position Max characters: 1
Tuesday (required)	Functions like Monday but on Tuesdays	Record identity: QS Position: 31 st position Max characters: 1
Wednesday (required)	Functions like Monday but on Wednesdays	Record identity: QS Position: 32 nd position Max characters: 1
Thursday (required)	Functions like Monday but on Thursdays	Record identity: QS Position: 33 rd position Max characters: 1
Friday (required)	Functions like Monday but on Fridays	Record identity: QS Position: 34 th position Max characters: 1
Saturday (required)	Functions like Monday but on Saturdays	Record identity: QS Position: 35 th position Max characters: 1
Sunday (required)	Functions like Monday but on Sundays	Record identity: QS Position: 36 th position Max characters: 1
Start_date (required)	Date service interval starts	Record identity: QS Position: 14 th position Max characters: 8
End_date (required)	Date service interval starts	Record identity: QS Position: 22 nd position Max characters: 8

Table 11: calendar.txt fields, description and the CIF equivalent requirement

2.3.7 calendar_dates.txt

The routes.txt file is a required file in the GTFS requirements. It contains 3 fields of which all are required. Indicated in the table below are the different field names, description and the equivalent requirement in a CIF file.

Field Name	Description	CIF Equivalent
------------	-------------	----------------

Service_id (required)	References service_id in calendar.txt or creates its own ID if calendar.txt not available	Same requirement as service_id in calendar.txt
Date (required)	Date when the service exception occurred	Record identity: QE Position: 3 rd position Max characters: 8
Exception_type (required)	shows whether service is available on date indicated in the date field. Can be 1 (service added) or 2 (service removed)	Record identity: QE Position: 19 th position Max characters: 1

Table 12: calendar_dates.txt fields, description and the CIF equivalent requirement

2.4 Data Converter

The first relevant conversion in the software program consists of building the representation for each record in an HashMap. In the process of trying to build the HashMap, the ATCO-CIF files could not be read directly. Fortunately, the one-bus-away repository on GitHub provides a library for obtaining the information in the ATCO-CIF file.

2.4.1 Reading ATCO-CIF files

When parsing ATCO-CIF files, a couple of class files must be created to hold varying elements. These files include an element file to hold different records, a location file to hold location data, an operator file to hold operator data, an origin file to hold origins of each journey and other files are indicated in my [GitHub](#) repository. In this project, not all records in the ATCO-CIF files are processed. The records that have a class file associated with them are assigned a type in the HashMap. The records that do not have class files to hold the elements are assigned an UNKNOWN type.

```
//Inputting the ATCO-CIF record identity into the hashmap
static {
    _typesByKey.put("QS", roadElement.Type.JOURNEY_HEADER);
    _typesByKey.put("QE", roadElement.Type.JOURNEY_DATE_RUNNING);
    _typesByKey.put("QO", roadElement.Type.JOURNEY_ORIGIN);
    _typesByKey.put("QI", roadElement.Type.JOURNEY_INTERMEDIATE);
    _typesByKey.put("QT", roadElement.Type.JOURNEY_DESTINATION);
}
```



```

        _typesByKey.put("QL", roadElement.Type.LOCATION);
        _typesByKey.put("QB", roadElement.Type.ADDITIONAL_LOCATION);
        _typesByKey.put("QV", roadElement.Type.VEHICLE_TYPE);
        _typesByKey.put("QC", roadElement.Type.UNKNOWN);
        _typesByKey.put("QP", roadElement.Type.OPERATOR);
        _typesByKey.put("QQ", roadElement.Type.UNKNOWN);
        _typesByKey.put("QJ", roadElement.Type.UNKNOWN);
        _typesByKey.put("QD", roadElement.Type.ROUTE_DESCRIPTION);
        _typesByKey.put("QY", roadElement.Type.UNKNOWN);
        _typesByKey.put("ZM", roadElement.Type.UNKNOWN);
        _typesByKey.put("ZS", roadElement.Type.UNKNOWN);
    }

```

Listing 1: HashMap containing record identities

Indeed, listing 1 above shows the varying records and their type being put in the HashMap.

Now that the record identity has been constructed and added to the HashMap, the next step is to perform a second conversion that uses this HashMap to determine how each record should be processed. The records are processed as strings and each element of the record is removed according to the position specified by the ATCO-CIF specification. The listing below illustrates an example of this conversion.

```

private void parseLine(roadContentHandler handler) {String typeValue
    = pop(2);
    roadElement.Type type = _typesByKey.get(typeValue); if (type ==
    null) {
        throw new roadException("unknown record type: " + typeValue
            + " at line " + _currentLineNumber);
    }
    switch (type) {
        case JOURNEY_HEADER -> parseJourneyHeader(handler);
        case JOURNEY_DATE_RUNNING -> parseJourneyDateRunning(handler); case
        JOURNEY_ORIGIN -> parseJourneyOrigin(handler);
        case JOURNEY_INTERMEDIATE -> parseJourneyIntermediate(handler); case
        JOURNEY_DESTINATION -> parseJourneyDestination(handler); case LOCATION
        -> parseLocation(handler);
        case ADDITIONAL_LOCATION -> parseAdditionalLocation(handler); case
        VEHICLE_TYPE -> parseVehicleType(handler);
        case ROUTE_DESCRIPTION -> parseRouteDescription(handler); case
        OPERATOR -> parseOperator(handler);
        case UNKNOWN -> {
            }
        default -
    }
    > throw new roadException("unhandled record type: " + type);
    }
}

```

Listing 2: Functions executed depending on type

The code displayed above selects the first 2 characters of each line to determine their type. This type is matched to their respective values in the HashMap. Using a switch, each type executes a specific function to process the record. Some of the functions that are executed are illustrated in Listing 3 below.

```
private void parseJourneyHeader(roadContentHandler handler) {
    roadHeader element = element(new roadHeader());

    String transactionType = pop(1);
    element.setOperatorId(pop(4));
    element.setJourneyIdentifier(pop(6));
    element.setStartDate(serviceDate(pop(8)));
    element.setEndDate(serviceDate(pop(8)));
    element.setMonday(integer(pop(1)));
    element.setTuesday(integer(pop(1)));
    element.setWednesday(integer(pop(1)));
    element.setThursday(integer(pop(1)));
    element.setFriday(integer(pop(1)));
    element.setSaturday(integer(pop(1)));
    element.setSunday(integer(pop(1)));

    String schoolTermTime = pop(1);
    String bankHolidays = pop(1);
    element.setRouteIdentifier(pop(4));
    String runningBoard = pop(6);

    element.setVehicleType(pop(8));

    String registrationNumber = pop(8);
    element.setRouteDirection(pop(1));

    closeCurrentJourneyIfNeeded(element, handler);
    _currentJourney = element;
    handler.startElement(element);
}

//
private void parseJourneyDateRunning(roadContentHandler handler) {
    roadDateRun element = element(new roadDateRun());
    element.setStartDate(serviceDate(pop(8)));
    element.setEndDate(serviceDate(pop(8)));
    element.setOperationCode(integer(pop(1)));
    if (_currentJourney == null) {
        throw new roadException("journey timepoint without header at line
        + _currentLineNumber);
    }
}
```

```

    }
    _currentJourney.getCalendarModifications().add(element);
    fireElement(element, handler);

}

//
private void parseJourneyOrigin(roadContentHandler handler) { roadOrigin element
    = element(new roadOrigin()); element.setLocationId(pop(12));
    String departString = pop(4); element.setDepartureString(departString);
    elements.add(element);
//    element.setDepartureTime(time(departTime));
//    pushTimepointElement(element, handler);
}

//
private void parseJourneyIntermediate(roadContentHandler handler) { roadIntermediate
    element = element(new roadIntermediate()); element.setLocationId(pop(12));
    String arriveString = pop(4); String
    departString = pop(4);
    element.setDepartureString(departString);
    element.setArrivalString(arriveString); elements.add(element);
}

//
private void parseJourneyDestination(roadContentHandler handler) { roadDestination
    element = element(new roadDestination()); element.setLocationId(pop(12));
    String arriveString = pop(4);
    element.setArrivalString(arriveString); elements.add(element);
    validateJourney(handler);
}

//
private void pushTimepointElement(roadTimePoint element, roadContentHandler handler) {
    if (_currentJourney == null) {

```

```

        throw new roadException("journey timepoint without header at line
"
        + _currentLineNumber);
    }
    element.setHeader(_currentJourney);
    _currentJourney.getTimePoints().add(element);
    fireElement(element, handler);
}

//
private void parseLocation(roadContentHandler handler) {
    roadLocation element = element(new roadLocation());
    String transactionType = pop(1);
    element.setLocationId(pop(12));
    element.setName(pop(48));
    fireElement(element, handler);
}

//
private void parseAdditionalLocation(roadContentHandler handler) {
    roadAdditional element = element(new roadAdditional());
    String transactionType = pop(1);
    element.setLocationId(pop(12));

    String xValue = pop(8);
    String yValue = pop(8);
    Point2D.Double location = getLocation(xValue, yValue, true);
    if (location != null) {
        element.setLat(location.y);
        element.setLon(location.x);
    }
    fireElement(element, handler);
}

```

Listing 3: Code samples of functions to process records identified

While Listing 3 above shows different functions for processing separate records, the function for processing location and additional location is not complete as the value for latitude and longitude are presented in a different format to their GTFS equivalent. The Java programming language possesses a built-in class in its library to rectify this called `Point2D`. This class defines methods for working with 2-D coordinate space. The class is adapted to convert the latitude and longitude value in the ATCO-CIF file.

```

private Point2D.Double getLocation(String xValue, String yValue,
    boolean canStripSuffix) {

```

```

        if (xValue.isEmpty() && yValue.isEmpty()) {
            return null;
        }

        Point2D.Double from = null;

        try {
            double x = Long.parseLong(xValue);
            double y = Long.parseLong(yValue);
            from = new Point2D.Double(x, y);
        } catch (NumberFormatException ex) {
            throw new roadException("error parsing additional location: x="
                + xValue + " y=" + yValue + " line=" + _currentLineNumber)
;
        }

        try {
            Point2D.Double result = new Point2D.Double();
            CoordinateSystemToCoordinateSystem.transform(_fromProjection,
                _toProjection, from, result);
            return result;
        } catch (ProjectionException ex) {
            _log.warn("error projecting additional location: x=" + xValue + "
y="
                + yValue + " line=" + _currentLineNumber);
            return null;
        }
    }
}

```

Listing 4: Code sample to format latitude and longitude

Thanks to the application of the Point2D class, the longitude and latitude of the dataset is appropriately formatted to meet GTFS requirements.

2.4.2 Converting ATCO-CIF files

At this point in the conversion pipeline, the goal is to take the data representation of ATCO-CIF files and present them in the GTFS format. The functions presented in the Listings above present the process of reading each record from the ATCO-CIF files. While engineering and testing the code for this, some of the sections in the ATCO-CIF documents do not have values.

Due to these missing values, exceptions were added. An instance of this is in the Listing above where the conversion of longitude and latitude occur, an exception is thrown if the

code cannot process the values. Exceptions such as these are included in aspects of the code to avoid getting the software terminated prematurely. The means to handle these exceptions help to keep the software running even if errors occur while executing the program.

Once the final record has been parsed, it is possible to proceed to the conversion to GTFS which combines separate values from different records to produce a unique dataset.

The challenge is to develop code with the purpose of:

- Taking the converted data as input.
- Split the values in each record into substrings that are adjoined using the hyphen (-) to form new unique datasets.
- Identify the next character for each substring

An example of a function which achieves this purpose is shown in Listing 5 below on how the stops text file is generated in the dataset:

```
private Stop findStop(String stopId) {  
  
    for (String prefix : _pruneStopsWithPrefixes) {  
        if (stopId.startsWith(prefix)) {  
            return null;  
        }  
    }  
    roadLocation location = getLocationForId(stopId);  
    if (location == null) {  
        throw new roadException("no stop found with id " + stopId);  
    }  
  
    String locationId = location.getLocationId();  
    AgencyAndId id = id(locationId);  
    Stop stop = _dao.getStopForId(id);  
    if (stop == null) {  
        roadAdditional additionalLocation = _additionalLocationById.get(location  
Id);  
        if (additionalLocation == null) {  
            throw new roadException("found location with id=" + locationId  
                + " but no additional location information found");  
        }  
  
        if (additionalLocation.getLat() == 0.0  
            || additionalLocation.getLon() == 0.0) {  
            if (_stopsWithNoLocationInfo.add(stopId)) {  
                _log.info("stop with no location: " + locationId);  
                _prunedStopsWithNoLocationInfoCount++;  
            }  
        }  
    }  
}
```

```

    }
    if (_pruneStopsWithNoLocationInfo) {
        return null;
    }
}

stop = new Stop();
stop.setId(id(locationId));
stop.setName(location.getName());
stop.setLat(additionalLocation.getLat());
stop.setLon(additionalLocation.getLon());

_dao.saveEntity(stop);
}
return stop;
}

```

Listing 5: Code sample for generating stops data for stops.txt

To sum this up, at this point the functions for performing the conversion of the ATCO-CIF files to GTFS files has been coded and is ready to be used in a main method to run them.

2.4.3 Executing the program

Instead of first applying the conversion functions to the whole ATCO-CIF files and then separating the obtained records into their separate data, the strategy being followed consists of splitting first and then converting after. This can be seen section 2.3.1 and section 2.3.2. Additionally, the conversion functions can be leveraged using the split dataset to adjust different properties of the data. This could come in handy when dealing with a different format of CIF file.

Finally, the figure below shows the graphical user interface (GUI) used to run the program. Two values must be provided in the text box, the input path which points to the location of the ATCO-CIF files and the output path which points to the location where the GTFS files are to be stored.

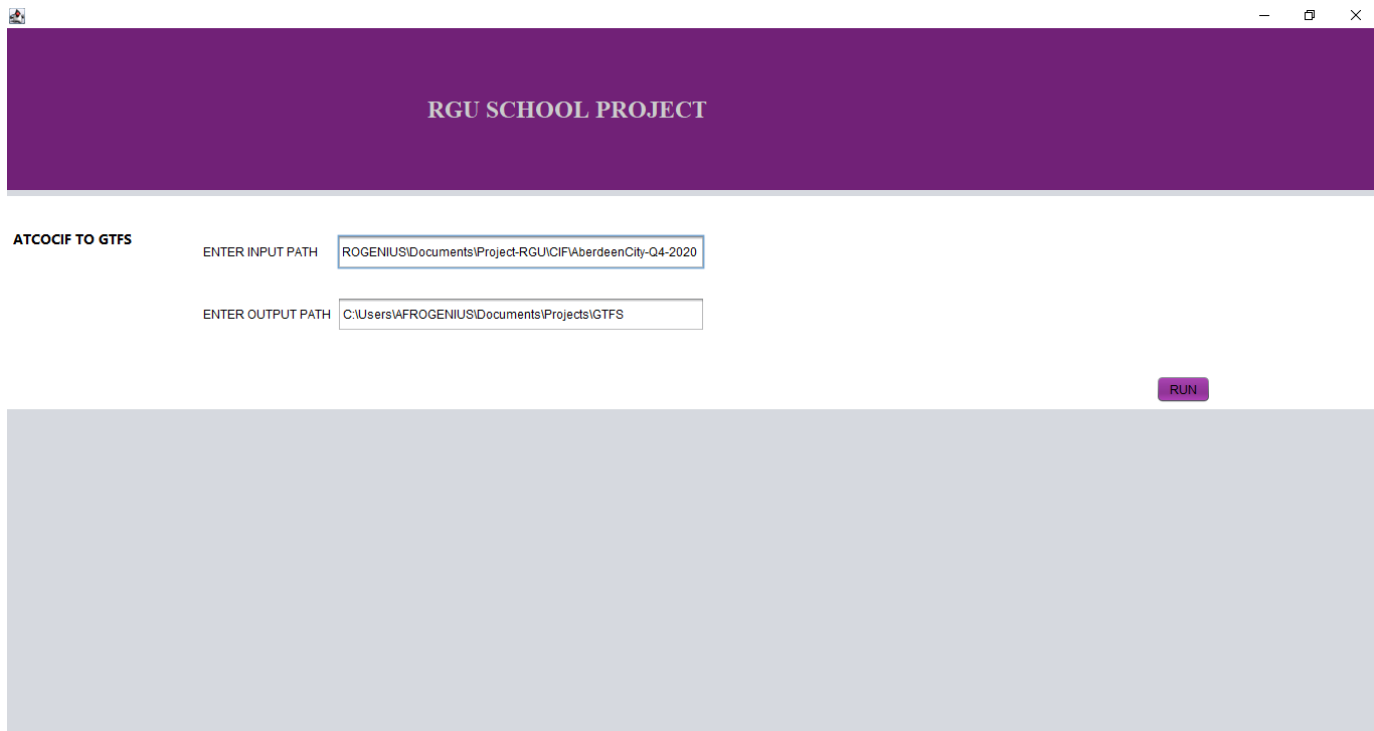


Figure 7: GUI for executing the converter

2.5 Conclusions

This chapter focused on the practical phase of the project.

First, the breakdown of each aspect into sections was illustrated in order to provide a general view of the projects design.

Then the process for each section has been described with details for each phase:

- A* Algorithm. Here, the pseudocode for the A* algorithm was outlined, the implementation was listed, and the execution discussed to show how the algorithm works.
- Data Processing. Here, the focus was on the identification of the required GTFS files and their required fields, done to link them to their ATCO-CIF file equivalent records.
- Data Converter. Here, the main tasks involved: reading of records in the CIF files into the HashMap data structure, creation of Java class files to hold elements from CIF records, linking of records in HashMap to their respective class files, passing records to functions for conversion.

Chapter 3

Testing and Evaluation

This chapter introduces and discusses the results of the project implementation and analyses the work done by comparing the achievements with the aim and objectives of the work as declared in the investigation report. Moreover, it contains a view into the legal, ethical, safety, social and professional issues.

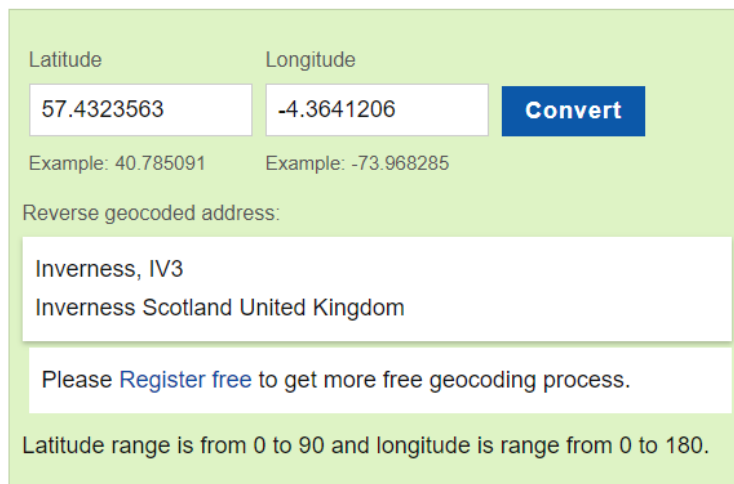
3.1 Test Data Sources

The graph network was tested by running the Dijkstra algorithm implemented in the third-party library on it. The algorithm ran properly and the generated latitude and longitude location on the network was compared to real life location data as shown below.

example point: (57.43, -4.36) ---- Location inputted
start point: (57.4323563, -4.3641206) ----- Nearest location on the graph network

The nearest location on the graph network was inputted into a tool that converts latitude and longitude to addresses (called reverse geocoding). The resulting address is indicated as Inverness which is correct. This result is shown below.

Reverse geocoding is the process to convert the latitude and longitude coordinates to a readable address. Type the lat long coordinates and press Convert button. Reverse geocoded address will shown below. Also the municipality, subdivision and country name can be found.



The screenshot shows a web-based reverse geocoding tool. It has two input fields for 'Latitude' and 'Longitude'. The 'Latitude' field contains '57.4323563' and the 'Longitude' field contains '-4.3641206'. Below these fields are example coordinates: 'Example: 40.785091' for latitude and 'Example: -73.968285' for longitude. A blue 'Convert' button is positioned to the right of the input fields. Below the button, the text 'Reverse geocoded address:' is followed by a text box containing 'Inverness, IV3' and 'Inverness Scotland United Kingdom'. Below this text box is a message: 'Please Register free to get more free geocoding process.' At the bottom, a note states: 'Latitude range is from 0 to 90 and longitude is range from 0 to 180.'

The ATCO-CIF files available were of public transport timetables for Scotland. The data was validated by comparing the records and confirming that they conformed to the specifications laid out in the ATCO-CIF specification document. The records in the CIF files were confirmed not to contain any values that does not correspond to those indicated in the specification document.

During this process of validating the files provided, it also became understood that these files may not contain all the record types as defined in the specification, this does not hinder the program in accomplishing its objective because fields that were required in the GTFS dataset were present in the CIF files.

3.2 Validation and checks

The GTFS files generated by the converter was validated using the FeedValidator tool developed by Google (the authors of the GTFS file format). The generated output from the Feed Validator tool is shown in the results section.

For each implemented code with a specific function, the outputs from these functions were checked to ensure the right outputs were produced. This was done by putting print statements at different points in the code to confirm their outputs. This helped to make sure that the different functions did not return wrong values to other functions dependent on their outputs.

Testing user inputs by placing print statements ran during the development of this software programs and removed after development to prevent excess outputs to the terminal which slows the program considerably. This print statements are a simple means to check outputs from varying functions.

3.3 Results

The A* algorithm ran successfully without any reported errors. The figure below shows the output from the program comparing the computation time of the Dijkstra algorithm and the A* algorithm on the graph network.

The screenshot shows a web application titled "RGU SCHOOL PROJECT". It has two main sections for algorithm input: "DIJKSTRA ALGORITHM" and "ASTAR ALGORITHM". Each section has input fields for "START POINT" (Latitude: 57.43, Longitude: -4.36) and "END POINT" (Latitude: 57.5953, Longitude: -4.4284). A "RUN" button is present next to each set of inputs. Below the input fields is a table comparing the results of both algorithms.

DIJKSTRA COST	COMPUTATION TIME	ASTAR COST	COMPUTATION TIME
PT18M25S	0.4497693	PT18M25S	0.2400862
PT18M25S	0.2739561	PT18M25S	0.1215416
PT18M25S	0.2956669	PT18M25S	0.069600401
PT18M25S	0.221751701	PT18M25S	0.1091379
PT18M25S	0.1750095	PT18M25S	0.0767337
PT18M25S	0.2088193	PT18M25S	0.060269399
PT18M25S	0.211844401	PT18M25S	0.0661099
PT18M25S	0.1838651	PT18M25S	0.0720686
PT18M25S	0.172006999	PT18M25S	0.0603152
PT18M25S	0.213158301	PT18M25S	0.059998099

Figure 8: GUI showing the generated result from the A* algorithm.

The computation time for the A* algorithm and the Dijkstra algorithm can be seen in the GUI table below.

DIJKSTRA COST	COMPUTATION TIME
PT18M25S	0.4497693
PT18M25S	0.2739561
PT18M25S	0.2956669
PT18M25S	0.221751701
PT18M25S	0.1750095
PT18M25S	0.2088193
PT18M25S	0.211844401
PT18M25S	0.1838651
PT18M25S	0.172006999
PT18M25S	0.213158301

Figure 9: GUI showing the cost and computation time of Dijkstra algorithm.

ASTAR COST	COMPUTATION TIME
PT18M25S	0.2400862
PT18M25S	0.1215416
PT18M25S	0.069600401
PT18M25S	0.1091379
PT18M25S	0.0767337
PT18M25S	0.060269399
PT18M25S	0.0661099
PT18M25S	0.0720686
PT18M25S	0.0603152
PT18M25S	0.059998099

Figure 10: GUI showing cost and computation time of A* algorithm.

As can be seen from the figures above, the average runtime of the A* algorithm was 0.0931 compared to the Dijkstra algorithm runtime of 0.240.

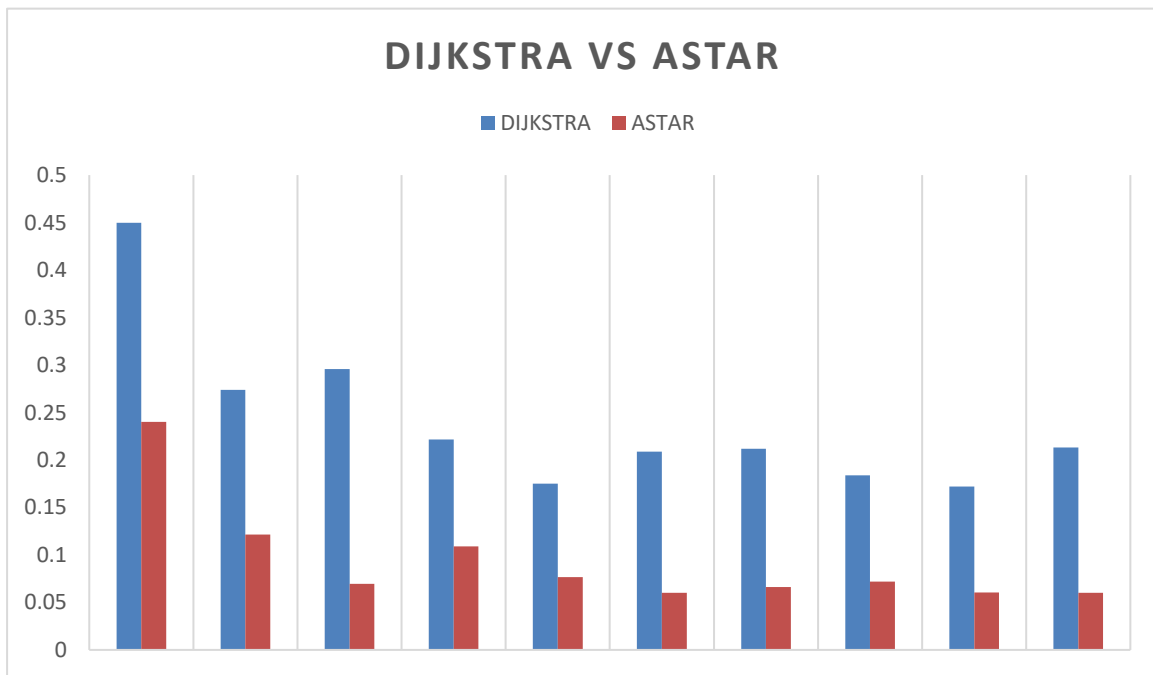


Figure 11: Column chart comparing the Dijkstra and A* algorithm Computation time.

As we can see in the chart above, the A* algorithm runs faster on every iteration in finding the best path to reach the destination.

The ATCO-CIF to GTFS converter was tested with the validated CIF files. The text files were generated containing csv format data with the records converted from the CIF files. The two outputs shown below are from the route.txt file and the trips.txt file.

```
agency_id,route_id,route_short_name,route_long_name,route_type
BAIN,BAIN-305,BAIN-305,,1
BAIN,BAIN-52,BAIN-52,,1
BAIN,BAIN-777,BAIN-777,,1
CS,CS-00,CS-00,,1
CS,CS-10,CS-10,,1
CS,CS-20,CS-20,,1
CS,CS-30,CS-30,,1
```

```
route_id,trip_id,service_id
BAIN-52,BAIN-1-0,20201005-20201011-_MTWHF_-00
BAIN-777,BAIN-1-1,20201005-20201011-_MTWHF_-00
BAIN-305,BAIN-1-2,20201005-20201011-_MTWHF_-00
BAIN-52,BAIN-2-0,20201005-20201011-_MTWHF_-00
BAIN-305,BAIN-2-1,20201005-20201011-_MTWHF_-00
BAIN-52,BAIN-3-0,20201005-20201011-_MT____-00
BAIN-305,BAIN-3-1,20201005-20201011-_MTWHF_-00
BAIN-52,BAIN-4-0,20201005-20201011-_MT____-00
```

The validation of the GTFS files using a python tool developed by Google called FeedValidator generated an HTML detailing what errors were found with the generated GTFS files.

GTFS validation results for feed:

..\Projects\GTFS\

FeedValidator extension used: None

Agencies: [Demo Transmit Authority](#), [Demo Transmit Authority](#), [Demo Transmit Authority](#), [Demo Transmit Authority](#)
[Transmit Authority](#), [Demo Transmit Authority](#), [Demo Transmit Authority](#)

Routes: 92

Stops: 3120

Trips: 21754

Shapes: 0

Effective: October 05, 2020 to October 11, 2020

Found these problems:

92 errors

4354 warnings

- 92 [Invalid Values](#)
- 1 [Expiration Date](#)
- 64 [Invalid Values](#)
- 23 [Stops Too Closes](#)
- 4238 [Too Fast Travels](#)
- 28 [Too Many Consecutive Stop Times With Same Times](#)

Errors:

Invalid Value

- Invalid value -1 in field route_type
in line 2 of routes.txt

agency_id	route_id	route_short_name	route_long_name	route_type
BAIN	BAIN-305	BAIN-305		-1

- Invalid value -1 in field route_type
in line 3 of routes.txt

agency_id	route_id	route_short_name	route_long_name	route_type
BAIN	BAIN-52	BAIN-52		-1

- Invalid value -1 in field route_type
in line 4 of routes.txt

agency_id	route_id	route_short_name	route_long_name	route_type
BAIN	BAIN-777	BAIN-777		-1

- Invalid value -1 in field route_type
in line 5 of routes.txt

agency_id	route_id	route_short_name	route_long_name	route_type
CS	CS-00	CS-00		-1

Figure 12: GTFS validation tool results.

3.4 Objectives, Achievement and Issues

This section focuses on determining whether the objectives, functional requirements and non-functional requirements for the project have been accomplished by comparing them with the achievements of the implementation, testing and results section of the project.

1. The objective of building and implementing a software program to transform ATCO-CIF files to GTFS was achieved.
2. The objective of implementing an A* algorithm for use with the graph data representing Scotland's passenger transport network was achieved.
3. The objective of comparing the A* pathfinding algorithm to the Dijkstra algorithm was achieved.
4. The objective of testing and validating the GTFS files was not achieved. The GTFS files did not pass the validations and had a lot of errors discovered.
5. The functional requirement of allowing a user to select which CIF file to be processed was achieved since the GUI can accept file locations as inputs.
6. The functional requirement of allowing a user to press a button to initiate the process of converting CIF files to GTFS was achieved.
7. The functional requirement to allow for a means to exit the window was achieved.
8. The functional requirement to allow the user to change the starting location on the graph was achieved.
9. The functional requirement to provide a visual of transport networks was not achieved.
10. The non-functional requirement to properly generate dialog windows to display errors and a menu section to allow for easy navigation was not achieved.
11. The non-functional requirement for the software to be user friendly, self-explanatory and small in size was achieved.

3.5 Ethical and Legal Issues

This section discusses the potential issues and risks related to the development of this project. From a legal point of view, all materials such as the third-party libraries, the CIF files, the graph network have been clearly referenced and compliant with intellectual property laws. Proper handling of publicly available material has also been followed. Developing a software program that uses data provided from a national organization, it is important that the CIF files provided were not disclosed to outside users as the files have not been provided openly to the public.

The third-party libraries provided to be used in the codebase were kept secure and the repositories to which they are uploaded is private to prevent unauthorized access. From an ethical perspective, this project report that no living entity (human or animal) was involved in the development of this project report. Also, the interaction with humans on this project has been to seek knowledge on the graph network and its API's. This project has been undertaken out of personal interest in pathfinding algorithms and networks and also to further an existing project being developed from a professional point of view. I do not gain any commercial value in completing this project.

Chapter 4

Conclusions

This chapter provides the conclusions for this project by first summarizing the purpose and the results of this project and then discusses the possibilities of extending the work done in it by outlining the limitations in order to provide alternative paths in this field.

4.1 Summary

This project does not conform to the regular projects which require an analysis of the problem domain, defining requirements with stakeholders, creating a suitable design and the implementation and testing but possesses its own challenges in taking an existing codebase and trying to extend the functionalities as well as add new functionalities.

The project objectives were simple and well understood. Given that data sources and third-party libraries were readily available that gave a sense that this would be a straightforward project. However, this proved not to be the case as indicated by some of the unmet objectives, functional and non-functional requirements

The solution to be implemented was a known factor and the methodology selected due to this understanding was the waterfall methodology. The software was completed during which the testing phase showed that the GTFS files generated had errors.

Nevertheless, the knowledge gained from this project would serve me well in future projects and this project showed me certain inadequacies with the waterfall methodology in practice such as lack of feedback during phases of a project can lead to discovery of major errors later in a project.

4.2 Limitations and Future Work

Implementing an algorithm is a part of the modern-day interview process which makes it a popular process but the application of this algorithm to real world problems or datasets can constitute a major hurdle to solving problems. With respect to this project, this was the case as resources were not lacking in how to implement the A* graph searching algorithm but an application of this algorithm to a real-world graph data proved to be the limitation of this project.

Likewise, information on the varying types of CIF files limited the overall ability of the converter implemented for this project. Future work for this project is to reduce the errors generated by the validation tool.

Bibliography

Zidane, Issa & Ibrahim, Khalil. (2018). Wavefront and A-Star Algorithms for Mobile Robot Path Planning. 69-80. 10.1007/978-3-319-64861-3_7.

Zeng, W., & Church, R. L. (2009). Finding shortest paths on real road networks: the case for A. *International journal of geographical information science*, 23(4), 531-543.

Delling, D., Sanders, P., Schultes, D., & Wagner, D. (2009). Engineering route planning algorithms. In *Algorithmics of large and complex networks* (pp. 117-139). Springer, Berlin, Heidelberg.

Skiena, S. S. (2020). Graph Traversal. In *The Algorithm Design Manual* (pp. 197-242). Springer, Cham.

Skiena, S. S. (2020). Data Structures. In *The Algorithm Design Manual* (pp. 69-108). Springer, Cham.

Opentransportdata.swiss. 2021. GTFS | Open data platform mobility Switzerland. [online] Available at: <<https://opentransportdata.swiss/en/cookbook/gtfs/>> [Accessed 19 August 2021].

Google Developers. 2021. Reference | Static Transit | Google Developers. [online] Available at: <<https://developers.google.com/transit/gtfs/reference>> [Accessed 20 August 2021].

Zobenika, D., 2021. Graphs in Java - A* Algorithm. [online] Stack Abuse. Available at: <<https://stackabuse.com/graphs-in-java-a-star-algorithm/>> [Accessed 19 August 2021].

Woltmann, S., 2021. A* Algorithm (With Java Example) | HappyCoders.eu. [online] HappyCoders.eu. Available at: <<https://www.happycoders.eu/algorithms/a-star-algorithm-java/>> [Accessed 19 August 2021].

Isaac Computer Science. 2021. Isaac Computer Science. [online] Available at: <https://isaaccomputerscience.org/concepts/dsa_search_a_star> [Accessed 19 August 2021].

GeeksforGeeks. 2021. A* Search Algorithm - GeeksforGeeks. [online] Available at: <<https://www.geeksforgeeks.org/a-search-algorithm/>> [Accessed 19 August 2021].

Edureka. 2021. Exception Handling in Java | A Beginners Guide to Java Exceptions | Edureka. [online] Available at: <<https://www.edureka.co/blog/java-exception-handling>> [Accessed 19 August 2021].

Liu, Ruifang & Feng, Shan & Shi, Ruisheng & Guo, Wenbin. (2014). Weighted Graph Clustering for Community Detection of Large Social Networks. *Procedia Computer Science*. 31. 85–94. 10.1016/j.procs.2014.05.248.

Xlinux.nist.gov. 2021. Euclidean distance. [online] Available at: <<https://xlinux.nist.gov/dads/HTML/euclidndstnc.html>> [Accessed 20 August 2021].

Appendix A

Project Management

This project work was done remotely due to the safety measures implemented in response to the COVID pandemic gripping the world.

The meetings with my supervisor took place every Monday on Microsoft teams, this was essential in pushing me in the right direction and not wasting time exploring solutions that would not contribute to achieving the project objectives.

I managed to attend the meetings with my supervisor and asking questions in regards to the project.