COMPUTER SCIENCE
ASSESSMENT DESCRIPTION 2018/19

UNIVERSITY OF Hull

MODULE DETAILS:

Module Number:	800053	Trimester:	2
Module Title:	Advanced Programming		
Lecturer:	Darren McKie / John Rayner / Warren Viant		

COURSEWORK DETAILS:

Assessment Number:	2	of	3		
Title of Assessment:	Software Portfolio				
Format:	Program	Report	Online test		
Method of Working:	Individual				
Workload Guidance:	Typically, you should expect to spend between	55	and	65	hours on this assessment
Length of Submission:	This assessment should be **no** more than: *(over-length submissions **will be** penalised as per University policy)*	2500 **words** *(excluding diagrams, appendices, references, source code)*			

PUBLICATION:

Date of issue:	w/c 25th Feb 2019

SUBMISSION:

ONE copy of this assessment should be handed in via:	Canvas		If Other (state method)	
Time and date for submission:	**Time**	2pm	**Date**	See Below
If **multiple hand–ins** please provide details:	Program, 30 April Report, 30 April Online test – trimester 2 exam period			
Will submission be scanned via TurnitinUK?	No	If submission is via TurnitinUK students MUST only submit Word, RTF or PDF files. Students MUST NOT submit ZIP or other archive formats. Students are reminded they can **ONLY** submit **ONE** file and must ensure they upload the correct file.		

The assessment must be submitted **no later** than the time and date shown above, unless an extension has been authorised on a *Request for an Extension for an Assessment* form which is available from: http://www2.hull.ac.uk/student/registryservices/currentstudents/usefulforms.aspx

If submission is via TurnitinUK within Canvas staff must set resubmission as standard, allowing students to resubmit their work, though only the last assessment submitted will be marked and if submitted after the coursework deadline late penalties will be applied.

MARKING:

Marking will be by:	Student Name

ASSESSMENT:

The assessment is marked out of:	100	and is worth	50	% of the module marks
N.B If multiple hand-ins please indicate the marks and percentage apportioned to each stage above, i.e. Stage 1–50, Stage 2–50. It is these marks that will be presented to the exam board.				

ASSESSMENT STRATEGY AND LEARNING OUTCOMES:

The overall assessment strategy is designed to evaluate the student's achievement of the module learning outcomes, and is subdivided as follows:

08025

LO	Learning Outcome	Method of Assessment
1	Identify and apply advanced programming concepts.	Program, Report
2	Identify and demonstrate an understanding of how to develop a robust, efficient and real-time application.	Program, Report, Online Test
3	Analyse and demonstrate an understanding of how tools allow a developer to create, debug and optimise an application.	Program, Report, Online Test

Assessment Criteria	Contributes to Learning Outcome	Mark
Data Structures Implementation	1, 2, 3	80
Report	1, 2, 3	20

FEEDBACK

Feedback will be given via:	Verbal (via demonstration)	Feedback will be given via:	Mark Sheet
Exemption			
Feedback will be provided no later than 4 'teaching weeks' after the submission date.			

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

You are advised to read the **NOTES** regarding late penalties, over-length assignments, unfair means and quality assurance in your student handbook, which is available on Canvas - https://canvas.hull.ac.uk/courses/17835/files/folder/Student-Handbooks-and-Guides.

In particular, please be aware that:

- Your work has a 10% penalty applied if submitted up to 24 hours late
- Your work has a 10% penalty applied and is capped to 40 (50 for level 7 modules) if submitted more than 24 hours late and up to and including 7 days after the deadline
- Your work will be awarded zero if submitted more than 7 days after the published deadline.
- The over-length penalty applies to your written report (which includes bullet points, and lists of text you have disguised as a table. It does not include contents page, graphs, data tables and appendices). Your mark will be awarded zero if you exceed the word count by more than 10%.

Please be reminded that you are responsible for reading the University Code of Practice on the use of Unfair means (http://www2.hull.ac.uk/student/studenthandbook/academic/unfairmeans.aspx) and must understand that unfair means is defined as any conduct by a candidate which may gain an illegitimate advantage or benefit for him/herself or another which may create a disadvantage or loss for another. You must therefore be certain that the work you are submitting contains no section copied in whole or in part from any other source unless where explicitly acknowledged by means of proper citation. In addition, **please note** that if one student gives their solution to another student who submits it as their own work, **BOTH** students are breaking the unfair means regulations, and will be investigated.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility, not the Department's, to produce the assignment in question.

Transport Network Data Structure and Process

Overview

The objective of this ACW is to implement, exercise and assess the performance of data structures that represent a transport network and support route-finding and evaluation. A network is a collection of nodes and arcs so, in the context of a transport network the nodes will correspond to road/rail junctions; towns/cities/villages; bus/rail stations; air/sea ports, etc., while the network arcs will be road/rail route segments or air corridors/sea lanes as appropriate.

Data will be supplied (in formats as described below) for the nodes and arcs representing a transport network, together with a 'command' file specifying operations to be performed on the network data. These operations may include, for example, reporting the distance between two nodes (places) on the network; finding neighbour nodes to a starting point; finding or validating a node sequence to form a journey between two places (pairs of nodes for origin and destination). For each command type, a specific output format will be defined. To streamline the assessment process, your software *must* implement all input and output formats precisely.

You will construct software in C++ with suitable class definitions for the necessary data structures, to input the supplied data and work through the commands. You will also be expected to present diagnostic data on process efficiency, for example the number of node/arc records visited in the data structure.

Provided Code

Initial code has been provided and this **MUST** be used as a starting point for your implementation.

Main.cpp

This file must **NOT** be edited. The main.cpp file will be replaced by a different main.cpp file during the marking process, along with different data files. Therefore, any changes you make to your main.cpp file will be overwritten. Your software **MUST** work with the provided main.cpp file otherwise your code will not compile with the replacement main.cpp file.

The main program uses a library called ACW_Wrapper. This library provides the timing functions. These times are generated automatically and output to the command prompt when you execute your program. Note all times are in microseconds.

The library also creates a file "log.txt". This contains a unique timestamp for your program, which you will use at the end of the trimester during the online test, when you will test your program.

"Commands.txt", is a file that you may edit. This is a list of command that your program is to execute.

Navigation class

The provided Navigation class must **NOT** be renamed otherwise the replaced main.cpp file will not be able to find it. You may, however, change the way the method parameters are passed (to satisfy Parasoft rules) but you must not change the type (i.e. string) – to reiterate **DO NOT** change any code inside of main.cpp. Do **NOT** output to the 'cout' in your code.

BuildNetwork(...) method will be used to (a) read in the network definition data (Places and Links files); and (b) construct your internal data structure(s). This method **MUST** return true if the build is successful (e.g. files have been correctly read) or false if the build is unsuccessful. You MUST not change either the name or the parameters of this method.

ProcessCommand(...) is used to process each navigation command in turn. This method **MUST** return true if it processes the command successfully or false if it does not process the command successfully. You **MUST NOT** change either the name or the parameters of this method.

The LLtoUTM(...) function has been provided to convert latitude and longitude into x and y coordinates, called eastings and northings. It is recommended that you make use of this function when calculating distances. The body of the function is contained within the ACW_Wrapper library.

The output file stream _outFile has already been created for you. Therefore, you **MUST** use this to output your results. **DO NOT** change the output filename.

Data

Data will be supplied in csv format (comma-separated variables) in two files: *Places.csv; Links.csv*. Command formats are as described later in this report.

For the network nodes (Places) file, each line of text will comprise a place name string (which may contain space(s)), followed by three numbers, being an integer reference code and two decimal numbers giving the location as a pair of coordinates for latitude and longitude in that conventional order. For example lines of the *Places* file might read:

```
Cottingham Rail,15931781,53.781,-0.407
Beverley Rail,15761842,53.842,-0.424
```

Data for each arc (Link) will comprise two reference numbers for the nodes that the arc joins, followed by a string giving the transport mode of the arc. For example, a line of the *Links* file that describes a Rail link between Cottingham Rail and Beverley Rail might read:

```
15931781,15761842,Rail
```

Transport Modes are Foot, Bike, Car, Bus, Rail, Ship. When journeys are being investigated, only valid arcs may be considered according to the required mode and the following rules of hierarchy:

1. *a rail or ship journey may only use arcs of the corresponding mode;*
2. *a bus journey may use bus and ship arcs, while a car journey may use car, bus and ship arcs;*
3. *a bike journey may use bike arcs and arcs defined in 1 and 2;*
4. *a foot journey may use any arc*.

These rules recognise for example that roadways generally have footpaths alongside (for the purposes of this exercise, motorways are excluded), and foot passengers may board all public transport modes.

Data Structures

You must define suitable C++ classes to hold the network data, preferably using dynamic data structure techniques to build a single structure which will allow processes to follow arcs (route segments) in sequence from a starting location towards and ultimately reaching a destination (see example diagram in fig 1). The location data may be held in a linked list or an array of node objects, with a suitable access mechanism so nodes can be accessed according to their reference number.

Arc data should then be held in a manner which allows all the arcs from a given node (with their mode type) to be accessed from that node, and so that each arc then gives access to the node at its other end. This will allow a journey finder/follower process to traverse the network smoothly by stepping along each consecutive arc in turn from the origin node to the destination node.

Note that the input data will show each arc only once, whereas routes may seek to traverse an arc from either end, so your data structure should allow for this. The diagram in fig 1 shows that node F has arcs to nodes C, G and P, and vice-versa, plus nodes C and E are linked. These eight arc objects would be built from only four data lines in the input data.

You may wish to calculate each arc distance during your input process, and store it within the arc node object, based on the coordinates of the two place nodes the arc joins. Alternatively, distances must be calculated when and if needed later during command processing.

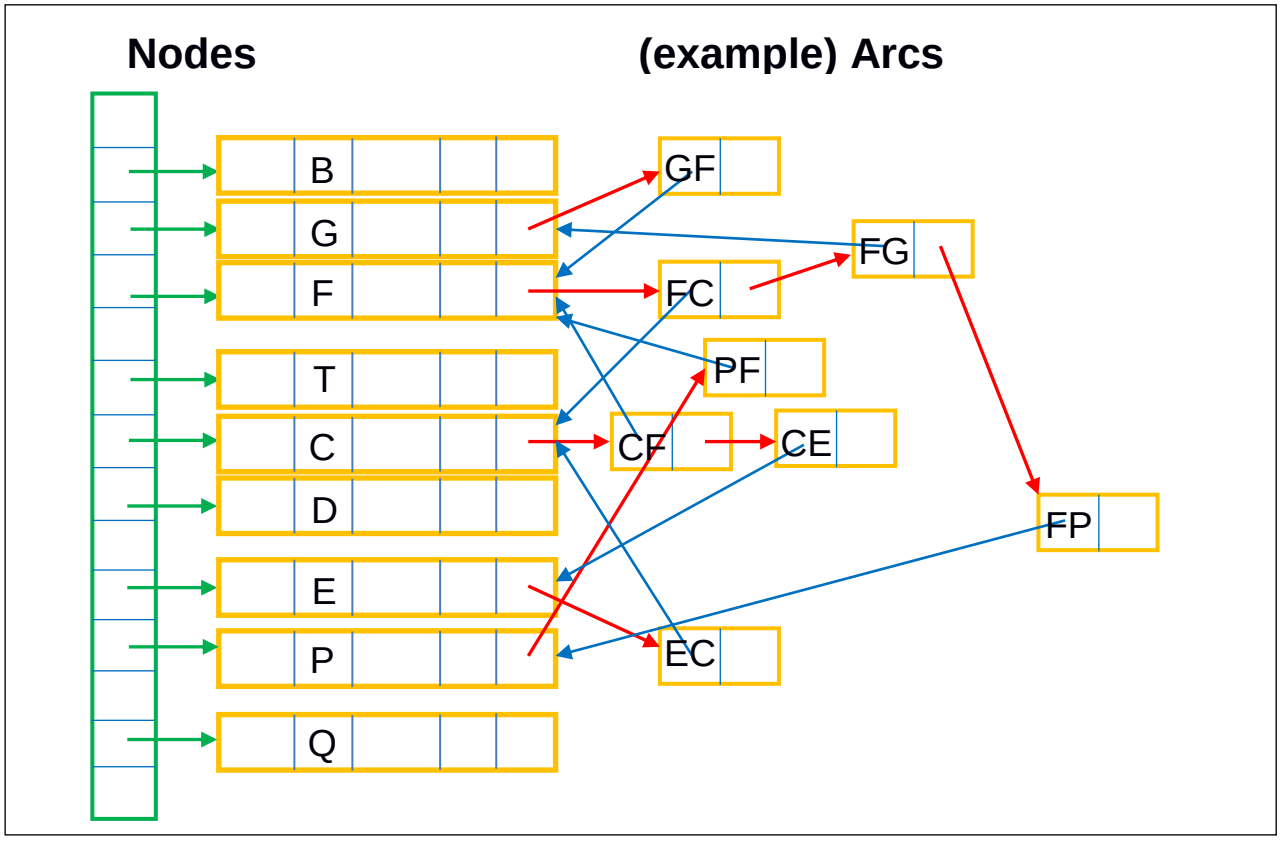

Fig 1. Example of a suitable dynamic network data structure

Processes, Commands and Output Format Definitions

The provided BuildNetwork(...) method will be used to

1. Read in the network definition data (Places and Links files); and
2. Construct the internal data structure(s).

This method **MUST** return true if the build is successful or false if the build is unsuccessful.

Each Command will be passed by the main method directly for processing using the provided ProcessCommand(...) method. This method will

1. Invoke appropriate process code; and
2. Produce the required output of journey sequence(s) and performance diagnostic information (see below).

This method **MUST** return true if it processes the command successfully or false if it does not process the command successfully.

The Commands will contain certain formats (notice white spaces rather than commas). A list of Commands with example data can be found below. For testing purposes, you should write your own example data.

Each Command should result in an output comprising

1. The text of the original command on one line;

2. Required output on subsequent line(s);
3. Followed by a single blank line.

All distance values will be outputted to 3 d.p.

MaxDist

Command *MaxDist* will find the furthest-separated places and calculate the distance between them. It will output the starting place name, the end place name, and the direct distance between the two.

Example:

```
MaxDist
```

Output:

```
MaxDist
York Rail,Rotterdam Harbour,416.543
<blank line>
```

MaxLink

Command *MaxLink* will find the longest single arc (as two node references). It will output the starting place reference, the end place reference, and the direct distance between the two.

Example:

```
MaxLink
```

Output:

```
MaxLink
17191741,61279944,358.402
<blank line>
```

FindDist

Command *FindDist* will calculate the distance between specified places. It will output the starting place name, the end place name, and the direct distance between the two.

Example:

```
FindDist 9361783 11391765
```

Output:

```
FindDist 9361783 11391765
Selby Rail,Howden Rail,13.531
<blank line>
```

FindNeighbour

Command *FindNeighbour* will list all neighbours of specified place. It will output the references of all nodes that are connected to a given node.

Example:

```
FindNeighbour 8611522
```

Output:

```
FindNeighbour 8611522
9361783
11251704
12321385
13491586
<blank line>
```

Check

Command *Check* will verify a proposed route between given places by the stated mode (e.g. Rail, Car, etc.) over the given stage connections. Example:

```
Check <mode> 14601225 12321385 8611522 9361783
```

Check valid route by the stated mode from 14601225 to 12321385, i.e. first step of journey sequence

Check valid route by the stated mode from 12321385 to 8611522, i.e. stage step of journey sequence

Check valid route by the stated mode from 8611522 to 9361783, i.e. final step of journey sequence

Note that there may be any number of stage connections.

It will output the references as it verifies the proposed route between given places by the stated mode (e.g. Rail, Car, etc.) over the given stage connections. Each connection will be outputted as PASS is the connection is valid and FAIL if the connection is not valid. If a FAIL is found then the process will stop:

Example of a correct route:

```
Check Rail 14601225 12321385 8611522 9361783
```

Output:

```
Check Rail 14601225 12321385 8611522 9361783
14601225,12321385,PASS
12321385,8611522,PASS
8611522,9361783,PASS
<blank line>
```

Example of an incorrect route:

```
Check Ship 14601225 12321385 8611522 9361783
```

Output:

```
Check Ship 14601225 12321385 8611522 9361783
14601225,12321385,FAIL
```

```
<blank line>
```

FindRoute

Command *FindRoute* will find a journey sequence of nodes between first (start) and destination (second) places by the stated mode. It will output the references of a route from the starting node to the end node by the stated mode (e.g. Rail, Car, etc.). If there is no valid route then output FAIL:

Example of a correct route:

```
FindRoute Rail 9081958 15832241
```

Output:

```
FindRoute Rail 9081958 15832241
9081958
12032132
15832241
<blank line>
```

Example of an incorrect route:

```
FindRoute Ship 9081958 15832241
```

Output:

```
FindRoute Ship 9081958 15832241
FAIL
<blank line>
```

FindShortestRoute

Command *FindShortestRoute* will find the shortest journey sequence of nodes between first (start) and destination (second) places by the stated mode. It will output the references of a route from the starting node to the end node by the stated mode (e.g. Rail, Car, etc.). If there is no valid route then output FAIL. The shortest route is one defined as requiring the least number of nodes.

Example of a correct route:

```
FindShortestRoute Rail 9081958 15832241
```

Output:

```
FindShortestRoute Rail 9081958 15832241
9081958
12032132
15832241
<blank line>
```

Example of an incorrect route:

```
FindShortestRoute Ship 9081958 15832241
```

Output:

```
FindShortestRoute Ship 9081958 15832241
FAIL
```

```
<blank line>
```

Marking Scheme

A detailed marking scheme has been published. This marking scheme will contain a breakdown of all of the marks and will give you the ability to mark yourself as you develop your software and write your report.

Deliverables

Code

You are to submit all of your code to the module site.

BEFORE you submit your code, make sure that you:

- Open your project in Visual Studio and then select **Build → Clean Solution**.

Parasoft results

- Parasoft C++ Static Test results for your source code (in the form of an auto-generated HTML report)

Report

You are also to submit a separate report, that will contain the following sections:

- Appropriate report introduction
- Timing and activity results (suitably tabulated) for each process operated (data set-up and command operations in turn)
- Discussion of results
- Conclusion

Your discussion of results and conclusion should address the following issues:

- A brief review of the data structures you have implemented: how are they organised and how are they operated?
- For each process and command operation, how does the timing performance relate to the number of data structure nodes accessed (this discussion should make reference to your timing and activity results);
- Discuss any changes or alternatives to the data structure design that might simplify the code implementation, or improve the process or command operations.
