UNIVERSITY OF HULL

# ACW Tutorial

UNIVERSITY OF HULL

# ACW

- 50% of the module marks
- Marked out of 100
  - 80 marks for the implementation
  - 20 marks for a report
  - **Detailed marking scheme on Canvas for you to download and use to track your progress**
- Deadline 14:00, 30 April 2019
  - Submit code
  - Submit report
- **Make sure you read the whole ACW description!**

UNIVERSITY OF HULL

# ACW Marking Scheme

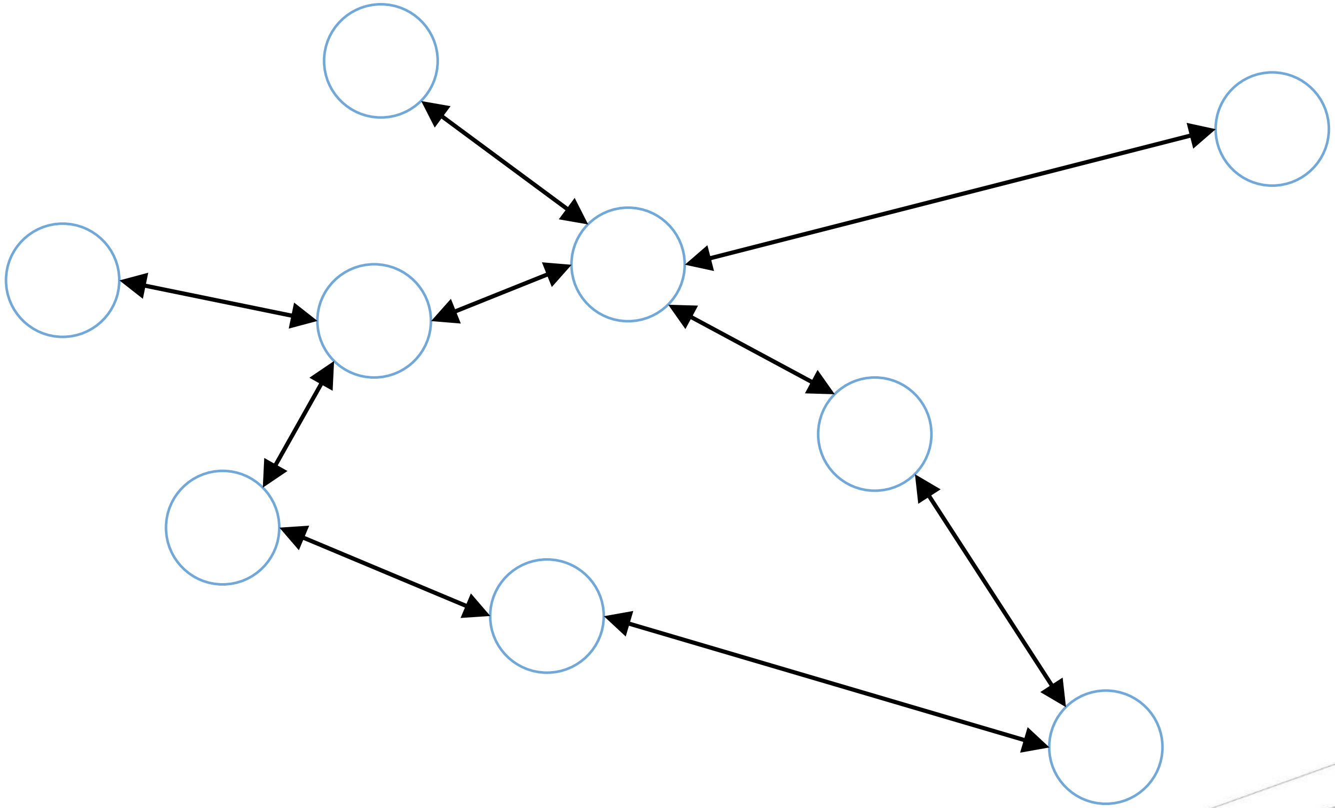
| All marks out of 10 (except Parasoft) | | |
|---|---|---|
| | **Weighting** | |
| **Operation** | | **0%** |
| maxDist - correctness | 1 | |
| maxLink - correctness | 1 | |
| FindDist - correctness | 1 | |
| FindNeighbour - correctness | 1 | |
| Check - correctness | 2 | |
| FindRoute - correctness | 2 | |
| FindShortestRoute - correctness | 2 | |
| maxDist - performance | 1 | |
| maxLink - performance | 1 | |
| FindDist - performance | 1 | |
| FindNeighbour - performance | 1 | |
| Check - performance | 2 | |
| FindRoute - performance | 2 | |
| FindShortestRoute - performance | 2 | |
| | | |
| **Implementation (using Parasoft C++ Test)** | | **100%** |
| Severity 1 - Marks lost per rule broken | 6 | |
| Severity 2 - Marks lost per rule broken | 4 | |
| Severity 3 - Marks lost per rule broken | 2 | |
| | | |
| **Report** | | **0%** |
| Report presentation and writing quality | 1 | |
| Timing and activity results | 1 | |
| Discussion of results | 2 | |
| Conclusion | 1 | |
| | | |
| **Software Implementation** | 0.8 | **0%** |
| **Report** | 0.2 | **0%** |
| | | **0%** |

# ACW Objective

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

UNIVERSITY
OF HULL

# Network - example



# Initial Code

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

# main.cpp

- The main.cpp file will be replaced by a different main.cpp file during the marking process, along with different data files
- Therefore, **DO NOT** make any changes to your main.cpp file
- Your software **MUST** work with the provided main.cpp file otherwise your code will not compile with the replacement main.cpp file
- "Commands.txt", is a file that you **should edit**
  - This is a list of commands that your program is to execute
  - **You need to add many more commands to test your software**

# ACW_Wrapper

- 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
  - 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 final online test, when you will test your program
- The library contains the LLtoUTM(...) function 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
  - You can use the x and y coordinates and Pythagoras' theorem to calculate the distance between two nodes (do not worry about the units)

# Navigation Class

- The provided Navigation class must **NOT** be renamed otherwise the replaced main.cpp file will not be able to find it
- Do **NOT** output to the ‘cout’ in your code
- The provided BuildNetwork(...) method will be used to:
  - read in the network definition data (Places and Links files);
  - construct the 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

# Navigation Class (cont.)

- Each Command will be processed using the provided ProcessCommand(...) method
  - This method **MUST** return true if it processes the command successfully or false if it does not process the command successfully
- 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

# Transport Modes

- 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.

# Places Data

- Data has be supplied in **csv** format (comma-separated variables)
- 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
```

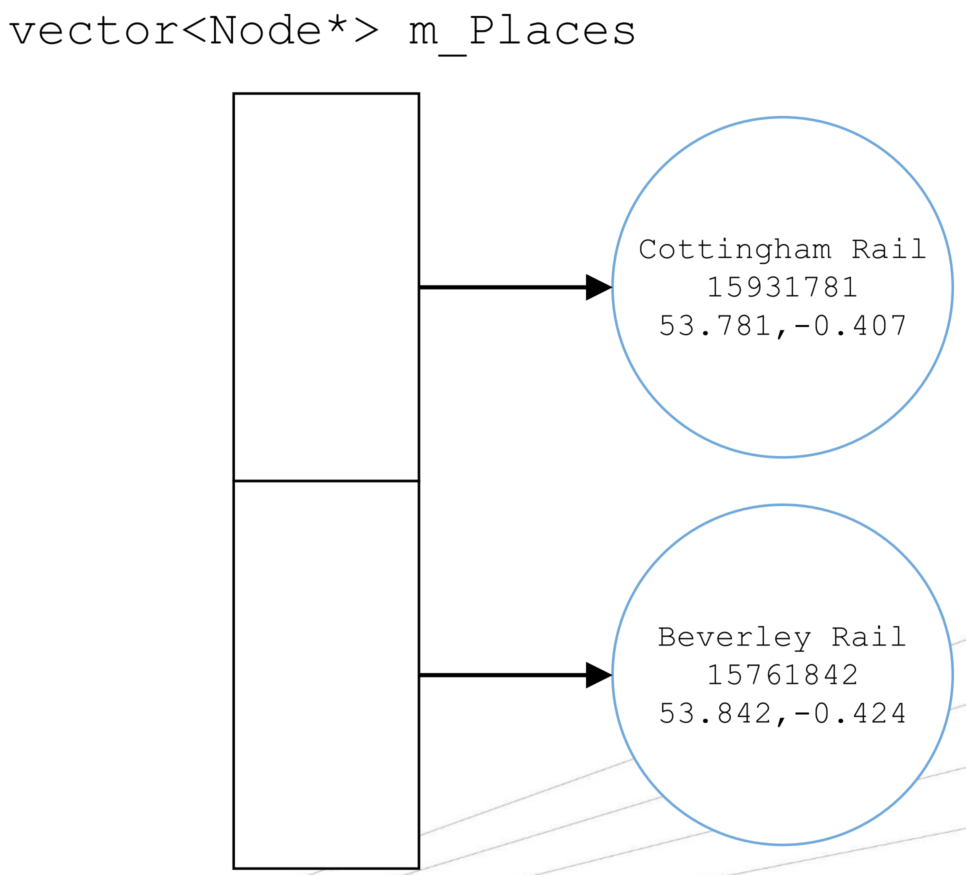
- You need to store the Nodes in a suitable data-structure so that you can search for Nodes for when processing commands
  - Suggest you add each Node to a list, e.g. `std::vector<Node*>`

UNIVERSITY OF HULL

# Network - example

- Nodes (Places):

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



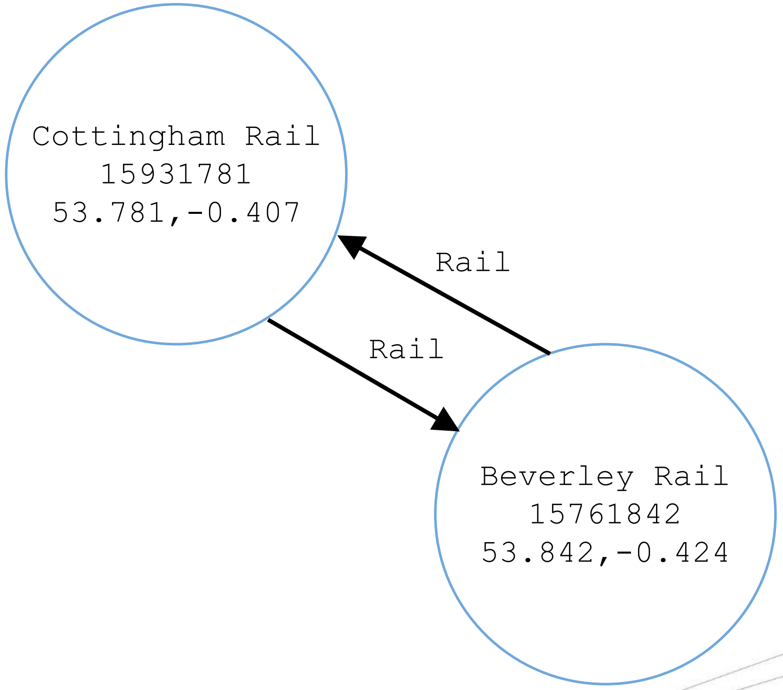
# Link Data

- 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
```

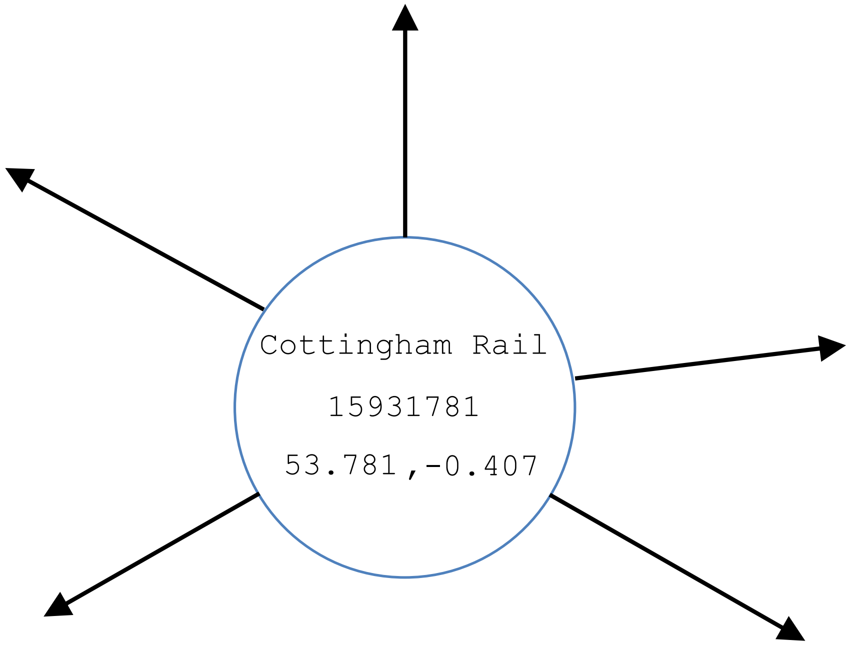
# Network - example

- Arc (Link): `15931781,15761842,Rail`
- Note: bi-directional Arc (link) between Nodes



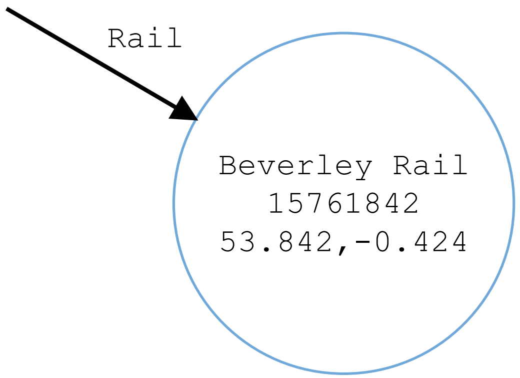
# Node class data members?

- Name (e.g. Cottingham Rail) – `string`
- Reference number (e.g. 15931781) – `int`
- Latitude (e.g. 53.781) – `float`
- Longitude (e.g. ,-0.407) – `float`
- Any number of Arcs (links) – `Arc*` array or `std::vector<Arc*>`?



# Arc class data members?

- Destination Node – `Node*`
- Mode (e.g. Rail) – `string or int or enum?`



# C++ Pre-processor

- We have specified that:
  - **Node** class has **Arc** data members
  - **Arc** class has a **Node** data member

```cpp
// Node.h
#pragma once
#include <string>
#include <vector>
#include "Arc.h"
using namespace std;

class Node
{
private:
   vector<Arc*> m_arcs;
// Methods, Other Data, etc.
};
```

```cpp
// Arc.h
#pragma once
#include <string>
#include "Node.h"
using namespace std;


class Arc
{
private:
   Node* m_destination;
// Methods, Other Data, etc.
};
```

UNIVERSITY
OF HULL

# C++ Pre-processor

- We have specified that:
  - **Node** class has **Arc** data members
  - **Arc** class has a **Node** data member

```
// Node.h
#pragma o
#include
#include
#include
using namespace

class Node
{
private:
   vecto
// Method
};
```

```
// Arc.h
#pragma once
#include
#incl

p
   Nod
// Methods,
};
```

# C++ Pre-processor

- Correct implementation (forward class declaration)

```cpp
// Node.h
#pragma once

#include <string>
#include <vector>
using namespace std;

class Arc;

class Node
{
private:
    vector<Arc*> m_arcs;
// Methods, Other Data, etc.
};
```

```cpp
// Arc.h
#pragma once

#include <string>
using namespace std;


class Node;

class Arc
{
private:
    Node* m_destination;
// Methods, Other Data, etc.
};
```

# C++ Pre-processor

- Correct implementation

```cpp
// Node.cpp
#include "Node.h"
#include "Arc.h"

// Methods, etc.
```

```cpp
// Arc.cpp
#include "Arc.h"
#include "Node.h"

// Methods, etc.
```

UNIVERSITY OF HULL

# Commands

- Each Command will be passed by the main method for processing using the ProcessCommand(...) method
- This method will:
  1. invoke appropriate process code;
  2. produce the required output of journey sequence(s) and performance diagnostic information
- This method **MUST** return true if it processes the command successfully or false if it does not process the command successfully

# Commands

- The Commands will contain certain formats (notice white spaces rather than commas)
- A list of Commands with example data can be found on the following slides
  - **For testing purposes, you should write your own example data and commands!**
- Each Command **MUST** 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.

# Command – MaxDist (8 marks)

- 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 them
- Example:

```
MaxDist
```

Output:

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

UNIVERSITY OF HULL

# Command – MaxLink (8 marks)

- 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 them
- Example:

```
MaxLink
```

Output:

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

# Command – FindDist (8 marks)

- 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 them
- Example:

```
FindDist 9361783 11391765
```

Output:

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

# Command – FindNeighbour (8 marks)

- 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>
```

# Command – Check (16 marks)

- 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
```

1. Check valid route by the stated mode from 14601225 to 12321385, i.e. first step of journey sequence
2. Check valid route by the stated mode from 12321385 to 8611522, i.e. stage step of journey sequence
3. Check valid route by the stated mode from 8611522 to 9361783, i.e. final step of journey sequence

- 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.

# Command – Check (cont.)

- 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>
```

# Command – Check (cont.)

- Example of an **incorrect** route:

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

Output:

```
Check Ship 14601225 12321385 8611522 9361783
14601225,12321385,FAIL
<blank line>
```

# Command – FindRoute (16 marks)

- 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
- **This can find ANY valid route, and does not have to find the shortest route!**

UNIVERSITY OF HULL

# Command – FindRoute (cont.)

- Example of a **correct** route:

```
FindRoute Rail 9081958 15832241
```

Output:

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

# Command – FindRoute (cont.)

- Example of an **incorrect** route:

```
FindRoute Ship 9081958 15832241
```

Output:

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

# Command – FindRoute (cont.)

- You are navigating from a starting Node to a destination Node
- Algorithm:
  - This involves looking through the Arc links of the starting Node to find Neighbour Nodes
  - Then you look through the Arc links of the Neighbour Nodes for their Neighbour Nodes
  - This continues until you either find the destination Node or the search ends

# Command – FindRoute (cont.)

- Strategies:
  - This algorithm is recursive
  - A recursive algorithm will start to return information after it has found the destination Node
    - Therefore it may be a good idea to start the route finder algorithm at the destination Node looking for the starting Node
  - A network data-structure has bi-directional links (Arcs) between Nodes, and it can have circular routes
    - That is, Node A may link to Node B, that may link to Node C, that may link to Node A
    - Therefore, you must record the previous Nodes that have been visited and check those against the current Node to make sure that you have not visited it yet

UNIVERSITY OF HULL

# Command – FindShortestRoute (16 marks)

- 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
- **This is the same as FindRoute, only it finds the shortest route rather than any route!**

# Manipulation of strings and streams

- Manipulating CSV files

```
ifstream finPlaces(fileNamePlaces);
char line[255];
finPlaces.getline(line, 255, ',');
string place = string(line);
```

- Consider utilising input streams

```
// input stream inStream = "FindDist 9361783 11391765"
string command;
int startNodeID, endNodeID;
inStream >> command;
inStream >> startNodeID;
inStream >> endNodeID;
```