ADVANCED PROGRAMMING

TRANSPORT NETWORK DATA STRUCTURE AND PROCESS REPORT

ONAYO MORO

201707183 University of Hull

Word Count (Excluding Tables, Contents Page, Appendices and Title Page): 2528

Table of Contents

Introduction

This project's objective was to identify and apply advanced programming concepts. These concepts will be implemented into an exercise written in C++ where assessments on the performance of data structures that represent a transport network and support route-finding can be made and evaluated. A robust and efficient real-time application was needed in order to achieve this. The network that will be created is a collection of nodes and arcs. The nodes represent data about a place such as its location in terms of latitude and longitude, the place's reference number, and its name. On the other hand, the arcs contained which nodes, which were identified using their reference numbers, can be travelled to directly and via what transport mode. The efficiency of the application will be assessed by performing diagnostics on its functions were data, such as its runtime in microseconds and number of iterations, will be captured.

Build Network

As the build network's function was responsible for reading in and organising data, it was necessary to make sure to make the most of the two 'while loops' needed for constructing the arcs from the links file, and the nodes from the places file. The location data for the nodes were formatted as latitude and longitude in the places file, however, x and y coordinates are necessary in order to compare the distance between nodes. Instead of carrying out the conversion every time the coordinates were needed, the method for converting latitude and longitude to x and y was called and the data was read in and then pushed back into a global arcs vector. This way the x and y coordinates are replaced with the latitude and longitude avoiding wasting processing time invoking the method every time the coordinates of a node were needed.

After constructing the nodes, the arcs are constructed in the second 'while loop'. This presented an opportunity to link the arcs and nodes whilst the arcs are being constructed. After each arc was constructed, the arcs were run through a 'for loop' that went through the nodes. If a nodes' reference matched either the start or end reference of an arc, that arc would be added to that nodes' arcs vector. This vector would then contain every node the current node has an arc with as well as the transport method used to get there.

In conclusion, having the nodes link to arcs that contain information on each nodes neighbours proved advantageous in quickly finding what nodes were related to each other. Despite this improvement, if was necessary to view the reference data of an arc, the reference number would need to run through the nodes vector to identify the node. To improve this, sorting the data inside

the arc so that the destination node object within the arcs would have two destination node, one for each reference in the arc.

MaxDist

Input Command	Output	Iterations	Microsecond Runtime
MaxDist	Malton Rail,Zeebrugge Harbour,411.279	23436	162.0

For the max distance method, two nested 'for loops' are used. One was nested in the other, however they both search within the nodes. The difference between the loops was that one starts at the beginning of the vector (iterator1 = 0) and the other starts at the second value (iterator2 = iterator1 + 1). When the outer 'for loop' was looking at a node value, the inner 'for loop' was looking at the node in-front of it and continue incrementing till the end of its loop. It would do this whilst comparing the distance between the node from the first loop and the node from the second. The distance function was made inline inside the 'navigation.h' file. This decision allowed the Pythagoras function to be called every time it was needed throughout the program instead of re-writing the same calculation every time.

When the second 'for loop' reached the end of the vector, the first 'for loop' would iterate and the second loop's iterator would be the iterator of the first loop plus one. This was because the previous node from the first loop already had its distance compared to the rest of the nodes, therefore, it was no longer be needed.

A mathematical expression for the way the loops iterated can be expressed using Gauss' equation.[1]

$$Sum = \frac{n(n+1)}{2} - n \qquad 23436 = \frac{217(217+1)}{2} - 217$$

This equation not only explains the number of iterations shown in the first table, but it also signifies that the loops in the method are set up to be as efficient as possible.

[1] http://mathcentral.uregina.ca/QQ/database/QQ.02.06/jo1.html

MaxLink

Input Command	Output	Iterations	Microsecond Runtime
MaxLink	51889340,17191741,356.309	40071	73.0

The functionality of the max link method was similar to the max distance in that it attempted to find the largest distance between two arcs instead of two nodes. It was also similar in that it used two 'for loops'. Although, this time the outer loop looked through the arcs and the inner loop looked through the nodes. Therefore, for each instance of an arc, the method was looping through the scope of the nodes until it found the nodes for both the references in the arc. The distance calculation was then carried out, the arc loop incremented, then the node loop was reset.

The number of iterations showed similarities to the Gauss equation, however, with a few adjustments. Changing the value of n inside (n+1) to the scope of the inner loop (217+1) instead of using the scope of the outer loop for the arcs (360) gave a sum of 38880. This value was similar to the number of iterations in the table above. This suggests that the difference in value may be attributed to the nodes having a variable number of neighbouring nodes and therefore a varying number of arcs. Due to this, the increment going through the nodes could have stopped at any point in the vector.

The reason the max link method had a lower microsecond runtime compared to the max distance was due to the fact that the max link was only calling the Pythagoras function 360 times which was the size of the arcs vector. On the other hand, the max distance was calling the Pythagoras function every single time the loops iterated resulting in a higher microsecond time. That resulted in 23436 distance calculations for the max distance method.

In conclusion, whilst the microsecond time was relatively low, the amount of iterations could have been greatly reduced if alterations to the build network were made. Whilst the nodes linked to the arc objects, the same was not true in the other direction. If the arcs linked to the node objects that related to the references in my arc, one 'for loop' could simply be used to run through the arcs and look inside the node objects for their location data to carry out the distance calculation. Despite needing one more function call to look at the node within the arc adding to the microseconds, the iterations would have only been equal to the size of the loop (360). Thus the overall time would have seen a decrease.

FindDist

Input Command	Output	Iterations	Microsecond Runtime
FindDist 9361783 11391765	Selby Rail,Howden Rail,13.5306	18	27.0

Only one loop was used for the find distance as the method was quite simple. The method would simply loop through the nodes vector and find the nodes it was searching for and extract their location data for the distance calculation. This meant that the number of iterations depended on where the data of the last node was in the nodes vector. In the case of the results above the number of iterations was only 18 as the reference number 11391765 was the 17th value in the nodes vector. A linear search was used for this function as it would have been faster than a binary search due to the size of the vector not being large enough to benefit from a binary search. A binary search would have needed to spend a certain amount of time reducing the scope of the search. The search would start by halving the vector and using the half where the reference number was greater or smaller than the reference number of the node located in the halfway point of the nodes vector. This process would have repeated for the halved scope until the node with the matching reference was found.[2]

For the binary search method, the size of the search would need to be much larger, therefore, I decided that a more optimal method was the linear search method.

FindNeighbour

Input Command	Output	Iterations	Microsecond Runtime
FindNeighbour 8611522	8631524 11251704 9361783 12321385 13491586	5	21.0
FindNeighbour 13531780	12991762 13881767 15221843	3	26.0

[2] https://github.com/gibsjose/cpp-cheat-sheet/blob/master/Data%20Structures%20and%20Algorithms.md#23-binary-search

Find neighbour is one of the most efficient methods due to the nature of its request and the support implemented by the data structure created in the build network. A simple 'for loop' going through the nodes was all that was needed to identify the node matching the reference. This meant that the method had a Big-O complexity of $0(n)$.

After the node was identified, a function call was used to look into the arcs associated with the node in question. Another vector was then responsible for saving the associated arcs, then a loop identified the nodes neighbours. Consequently, the number of iterations was equal to the number of elements in the current nodes vector of arcs. For example, if the node had 6 neighbours, there would be 6 iterations.

Check

Input Command	Output	Iterations	Microsecond Runtime
Check Rail 14601225 12321385 8611522 9361783	14601225,12321385,PASS 12321385,8611522,PASS 8611522,9361783,PASS	336	29.0
Check Ship 14601225 12321385 8611522 9361783	14601225,12321385,FAIL	360	26.0
Check Car 12271393 13461591 14431547 15041554 15861586	12271393,13461591,PASS 13461591,14431547,PASS 14431547,15041554,PASS 15041554,15861586,PASS	745	41.0

The check function was essentially a look-up to see if the arcs in a route existed. Furthermore, we needed to check if the necessary transport method existed for those arcs. As certain transport methods could travel using other transport methods, an inline function was created to look at the current transport method inputted from the command and return the transport methods it was allowed to use as strings into a 'transport' vector.

In order to confirm an arc, the first two references needed to run through the arcs vector to see if they connected. Then, using the 'count' function, the program would see if the arcs transport method exists within the transport vector. If an arc exists with the right transport method, it would compare the last reference to the reference in-front and run the arc verification process again. If an arc did not exist, the iterations would only be equal to the size of the arcs vector as the loop would

need to search the whole vector for a match. Furthermore, the microsecond time for a failed arc would be approximately the same as it was the amount of time it took for the loop to iterate to the end. For the number of iterations for the passes, it was simply the number of iterations it took to find each arc added together. Therefore the number of iterations would have depended on the location of the arcs within the vector. This would explain the variation and similarities in microsecond times as well as iterations in the tabulated results.

FindRoute / FindShortestRoute

Input Command	Output	Iterations	Microsecond Runtime
FindRoute Bike 8441694 51889340	8441694 10381699 10351609 13461591 14431547 15041554 16191675 17161614 19151566 51889340	29979	344.0
FindRoute Bus 16701778 16161770	16701778 16561780 16411782 16361787 16231787 16131779 16161770	5695	163.0
FindRoute Rail 16021756 14521393	FAIL	4	18.0
FindShortestRoute Foot 8441694 51889340	8441694 8631524 8611522 13491586 15911575 17311606 19131564 19151566 51889340	392913	1654.0
FindShortestRoute Rail 51889340 12032132	FAIL	2	28.0
FindShortestRoute Car 15491853 9791429	15491853 14671957 12201929 9201956 9611790 9061761 8441694 8631524 9791429	651586	2094.0

In the find route method, full use of the inline transport method to return the relevant transport strings to a transport vector was used. The first step was to find the neighbours of the start node so that every possible starting position was made clear.

Breadth first search was used as opposed to a depth first search to transverse a tree structured set of data. It was necessary to find each nodes' neighbour from the starting positions whilst also checking if the starting positions happened to arc with the end node. A scenario in which a depth first search would be advantageous is if recursion was used to build the route by calling the method in an 'if statement'. The reason this technique wasn't used was due to the limit of recursion set by the CPU buffer. With a large enough data set the CPU buffer could overflow, reaching its limit and causing the program to crash. Whilst the data set in this case was not large enough to cause such an error, writing a program robust enough to handle larger data sets using vectors offered a better contingency plan. In addition to this, vectors can hold much more data compared to the buffer size of a CPU which is dependent on the hardware the program is running on at that time.

The neighbours for the starting positions neighbours are then found and the process is repeated for all new neighbours with a valid transport method until the end node is found. If the end node is not found, then it would be clear that a route with the transport method could not be found and the route has failed. As soon as the end node is found, a loop will go through the previously discovered neighbours as well as the vector of arcs to see if there is an arc between the last node in the final route vector, which at the start just contains the end node, and the vector of neighbours. As the arcs are discovered, the references from the vector of neighbours will be added to the final route vector. Thus, the route will be constructed.

The number of iterations is dictated by many things, most notably the loop going through the nodes, the nodes arcs , the vector of visited nodes from the breadth search and the arcs vectors. If the start and end nodes are separated far from each other, the breadth search would prove to struggle to in terms of efficiency as it would need to search for each nodes neighbours as well as their neighbours. This would increase the number of iterations. In addition to this, if the number of available transport methods are common, then the breadth search may see a substantial increase the width of its search. However, an increase in the width of a breadth search would be beneficial and result in reduced iterations if the destination node was relatively closer to the start node as the breadth search would have an opportunity to end the search sooner and in fewer microseconds.

The tabulated results for FindRoute signifies how a start and end position that is closer to each other "16701778, 16161770" with a limited transport root "Bus", can have less iterations (5695) due to there being less valid nodes to arcs with. On the other hand, "8441694" and "51889340" are quite

distant from each other and use the “Bike” which can access many transport methods. Therefore, the number of iterations was much higher (29979).

The difference between the FindRoute and the FindShortesRoute is the way they construct the route. FindRoute is designed to return the first route it finds as fast as possible. Whilst FindRoute adds the first valid node it sees to its route, FindShortestRoute will continue looking for a node that is closer up the tree data structure and subsequently closer to the start node. This ensures that the shortest route will always be found.

Conclusion

I believe that what I learned during this project has strengthened my understanding of how data structure influences problem solving. Moreover, I have come to a better understanding of how to set up data structures and use those structures to advantageous effect in problem solving. Through the use of debugging tools and Parasoft, I have been able to achieve greater efficiency by refactoring my code with advanced programming techniques. Because of this, my ability to apply advanced programming techniques and concepts has increased in maturity.

References

Citations

Gibson, J. and Chen, D. (2019). *gibsjose/cpp-cheat-sheet*. [online] GitHub. Available at: https://github.com/gibsjose/cpp-cheat-sheet/blob/master/Data%20Structures%20and%20Algorithms.md.

Cplusplus.com. (2019). *vector - C++ Reference*. [online] Available at: http://www.cplusplus.com/reference/vector/vector/ [Accessed 30 Apr. 2019].

Hacker Noon. (2019). *What does the time complexity O(log n) actually mean?*. [online] Available at: https://hackernoon.com/what-does-the-time-complexity-o-log-n-actually-mean-45f94bb5bfbf.

Appendices

```
const inline std::vector<string> TransType(const std::string& tr)const {
    vector<string> method;
    if (tr == "Rail") {
        method.push_back("Rail");
    }
    else if (tr == "Ship") {
        method.push_back("Ship");
    }
    else if (tr == "Bus") {
        method.push_back("Bus");
        method.push_back("Ship");
    }
    else if (tr == "Car") {
        method.push_back("Car");
        method.push_back("Bus");
        method.push_back("Ship");
    }
    else if (tr == "Bike") {
        method.push_back("Car");
        method.push_back("Bus");
        method.push_back("Ship");
        method.push_back("Rail");
        method.push_back("Bike");
    }
    else if (tr == "Foot") {
        method.push_back("Car");
        method.push_back("Bus");
        method.push_back("Ship");
        method.push_back("Rail");
        method.push_back("Bike");
        method.push_back("Foot");
    }
    return method;
}
```

Figure 1: Inline transport method, returns vector of transport methods.

```
const inline double Distance(const double x1, const double y1, const double x2, const double y2)const {
    double dist; return dist = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
}
```

Figure 2: Inline distance calculation

```
bool Navigation::ProcessCommand(const string& commandString)
{
    istringstream inString(commandString);
    string command;
    inString >> command;

    if (command == "MaxDist") {
        return MaxDist();
    }
    else if (command == "MaxLink") {
        return MaxLink();
    }
    else if (command == "FindDist") {
        int start, end;
        inString >> start;
        inString >> end;
        return FindDist(start, end);
    }
    else if (command == "FindNeighbour") {
        int ref;
        inString >> ref;
        return FindNeighbour(ref);
    }
    else if (command == "Check") {
        int ref = 0;
        string tr;
        inString >> tr;
        vector<int> references;
        while (inString >> ref)
            references.push_back(ref);
        Check(tr, references);
        return true;
    }
    else if (command == "FindRoute") {
        int start = 0, end = 0; string tr; const string& FR = "FR";
        inString >> tr;
        inString >> start;
        inString >> end;
        _outFile << "FindRoute " << tr << " " << start << " " << end;
        return FindRoute(start, end, tr, FR);
    }
    else if (command == "FindShortestRoute") {
        int start = 0, end = 0; string tr; const string& FSR = "FSR";
        inString >> tr;
        inString >> start;
        inString >> end;
        _outFile << "FindShortestRoute " << tr << " " << start << " " << end;
        return FindRoute(start, end, tr, FSR);
    }return false;
}
```

Figure 3: ProcessCommand

```
const bool Navigation::BuildNetwork(const string &fileNamePlaces, const string &fileNameLinks)const
{
    fstream finPlaces(fileNamePlaces), finLinks(fileNameLinks);
    char line[255];
    double latitude, longitude;
    int startRef, endRef, ref;

    if (finPlaces.fail() || finLinks.fail()) {
        return false;
    }
    else
    {
        int i = 0; double Northings, Eastings;
        while (finPlaces.good())
        {
            finPlaces.getline(line, 255, ',');
            string placeName = string(line);

            finPlaces.getline(line, 255, ',');
            istringstream sin(line);
            sin >> ref;

            finPlaces.getline(line, 255, ',');
            istringstream sin1(line);
            sin1 >> latitude;

            finPlaces.getline(line, 255, '\n');
            istringstream sin2(line);
            sin2 >> longitude;

            LLtoUTM(latitude, longitude, Northings, Eastings);

            Nodes* const n = new Nodes(placeName, ref, Northings, Eastings);
            nodes.push_back(n);
        }
        while (finLinks.good())
        {
            finLinks.getline(line, 255, ',');
            istringstream sin(line);
            sin >> startRef;

            finLinks.getline(line, 255, ',');
            istringstream sin1(line);
            sin1 >> endRef;

            finLinks.getline(line, 255, '\n');
            string transport = string(line);

            Arcs* const a = new Arcs(startRef, endRef, transport);
            arcs.push_back(a);

            for (Nodes* const n : nodes) {
                if (n->g_ref() == startRef) {
                    arcs[i]->s_destination(n);
                    Arcs* const ar = new Arcs(startRef, endRef, transport);
                    n->add_arcs(ar);
                    for (Nodes* const no : nodes)
                        if (endRef == no->g_ref())
                            ar->s_destination(no);
                }
                else if (n->g_ref() == endRef) {
                    Arcs* const ar = new Arcs(endRef, startRef, transport);
                    n->add_arcs(ar);
                    for (Nodes* const no : nodes)
                        if (startRef == no->g_ref())
                            ar->s_destination(no);
                }
            }i++;
        }
    }return true;
}
```

Figure 4: BuildNetwork

```
bool Navigation::MaxDist() {
    string start, end; double dist, max = 0;

    for (int i = 0; i < nodes.size(); i++)
    {
        const double x1 = nodes[i]->g_lat();
        const double y1 = nodes[i]->g_longitude();
        for (int p = i + 1; p < nodes.size(); p++)
        {
            const double x2 = nodes[p]->g_lat();
            const double y2 = nodes[p]->g_longitude();
            dist = Distance(x1, y1, x2, y2);
            if (dist / 1000 > max)
            {
                start = nodes[i]->g_place();
                end = nodes[p]->g_place();
                max = dist / 1000;
            }
        }
    }
    _outFile << "MaxDist \n" << start << "," << end << "," << fixed << setprecision(3) << max << "\n" << endl;
    return true;
}
```

Figure 5: MaxDist method

```
bool Navigation::MaxLink() {
    int maxstart = 0, maxend = 0, bothrefs = 0;
    double x1, y1, x2, y2, max = 0; int counter = 0;

    for (Arcs* const a : arcs)
    {
        for (Nodes* const n : nodes)
        {
            counter++;
            if (a->g_startref() == n->g_ref()) {
                bothrefs += 1;
                x1 = n->g_lat();
                y1 = n->g_longitude();
            }
            else if (a->g_endref() == n->g_ref()) {
                bothrefs += 1;
                x2 = n->g_lat();
                y2 = n->g_longitude();
            }
            if (bothrefs == 2) {
                bothrefs = 0;
                const double dist = Distance(x1, y1, x2, y2);
                if (dist / 1000 > max)
                {
                    max = dist / 1000;
                    maxstart = a->g_startref();
                    maxend = a->g_endref();
                }
                break;
            }
        }
    }_outFile << "MaxLink \n" << maxstart << "," << maxend << "," << fixed << setprecision(3) << max << "\n" << endl;
    return true;
}
```

Figure 6: MaxLink method

```
bool Navigation::FindDist(const int start, const int end) {
    string startplace, endplace;
    double x1, y1, x2, y2; int bothrefs = 0;

    for (Nodes* const n : nodes)
    {
        if (start == n->g_ref()) {
            bothrefs += 1;
            x1 = n->g_lat();
            y1 = n->g_longitude();
            startplace = n->g_place();
        }
        else if (end == n->g_ref()) {
            bothrefs += 1;
            x2 = n->g_lat();
            y2 = n->g_longitude();
            endplace = n->g_place();
        }
        if (bothrefs == 2) {
            const double dist = Distance(x1, y1, x2, y2) / 1000;
            _outFile << "FindDist " << start << " " << end << "\n" << startplace << "," << endplace << "," << fixed << setpreci
            return true;
        }
    }
    return true;
}
```

Figure 7: FindDist method

```
bool Navigation::FindNeighbour(const int ref) {
    string neighb;
    _outFile << "FindNeighbour " << ref << endl;
    for (Nodes* const n : nodes) {
        if (ref == n->g_ref()) {
            vector<Arcs*> const arcs = n->g_arcs();
            for (Arcs* const a : arcs) {
                neighb = neighb + to_string(a->g_endref()) + "\n";
            }
            _outFile << neighb << endl; return true;
        }
    }return true;
}
```

Figure 8: FindNeighbour method

```
bool Navigation::Check(const std::string& transport, const std::vector<int>& references) {
    string refs; int passfail = 0; int ref1, ref2;
    for (const int r : references)
        refs = refs + to_string(r) + " ";

    _outFile << "Check " << transport << " " << refs << endl;

    for (int r = 0; r < references.size() - 1; r++)
    {
        ref1 = references[r]; ref2 = references[r + 1];
        passfail = Transport(transport, ref1, ref2);
        if (passfail == 1)
        {
            _outFile << ref1 << "," << ref2 << "," << "FAIL" << "\n" << endl; return true;
        }
    }
    _outFile << endl; return true;
}
```

Figure 9: Check method

```
int Navigation::Transport(const std::string& transport, const int ref1, const int ref2) {
    for (Arcs* const a : arcs)
    {
        if ((ref1 == a->g_startref() || ref1 == a->g_endref()) && (ref2 == a->g_startref() || ref2 == a->g_endref()))
        {
            vector<string> method; method = TransType(transport);
            int c = 0;
            if (count(method.begin(), method.end(), a->g_transport_method()))
            {
                _outFile << ref1 << "," << ref2 << "," << "PASS" << "\n"; c++;
            }
            if (c == 0)
                return 1;
            else
                return 0;
        }
    }return 0;
}
```

Figure 10: Transport method used for Check method

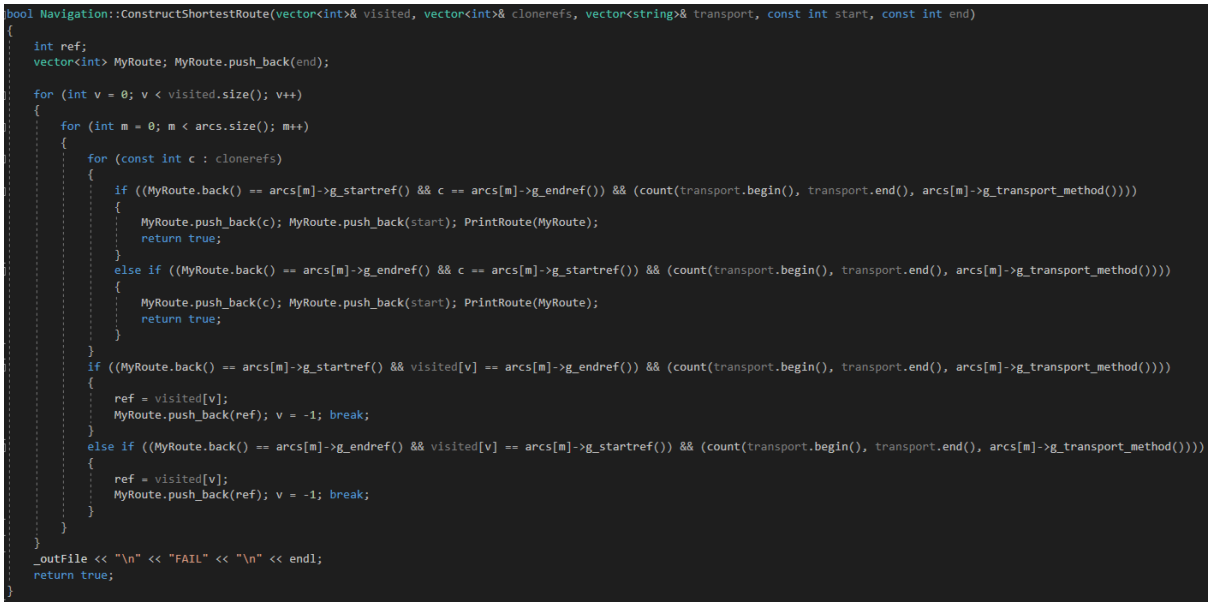
```
bool Navigation::FindRoute(const int start, const int end, const std::string& tr, const std::string& R)
{
    vector<string> transport; transport = TransType(tr);
    vector<int> currentrefs;
    for (int n = 0; n < nodes.size(); n++)
    {
        if (nodes[n]->g_ref() == start)
        {
            vector<Arcs*> myarcs = nodes[n]->g_arcs();
            for (int a = 0; a < myarcs.size(); a++)
            {
                if (end == myarcs[a]->g_endref() && (count(transport.begin(), transport.end(), myarcs[a]->g_transport_method()))
                {
                    _outFile << "\n" << start << "\n" << end << "\n" << endl;
                    return true;
                }
                else if (count(transport.begin(), transport.end(), myarcs[a]->g_transport_method()))
                {
                    currentrefs.push_back(myarcs[a]->g_endref());
                }
            }
            if (myarcs.size() != 0)
            {
                Route(currentrefs, start, end, tr, R);
                return true;
            }
            else if (myarcs.size() == 0)
            {
                _outFile << "\n" << "FAIL" << "\n" << endl;
                return true;
            }

        }
    }
    return true;
}
```

Figure 11: FindRoute method for finding start nodes

```
bool Navigation::Route(vector<int>& currentrefs, const int start, const int end, const string& tr, const string& R)
{
    vector<int> clonerefs = currentrefs;
    vector<string> transport; transport = TransType(tr);
    vector<int> visited; visited.push_back(start);
    for (int c = 0; c < currentrefs.size(); c++)
    {
        for (int n = 0; n < nodes.size(); n++)
        {
            if (currentrefs.size() != 0 && nodes[n]->g_ref() == currentrefs[c])
            {
                if (!count(visited.begin(), visited.end(), currentrefs[c]))
                {
                    visited.push_back(currentrefs[c]);
                }
                vector<Arcs*> myarcs = nodes[n]->g_arcs();
                for (int a = 0; a < myarcs.size(); a++)
                {
                    if ((count(transport.begin(), transport.end(), myarcs[a]->g_transport_method())))
                    {
                        if (end == myarcs[a]->g_endref())
                        {
                            if (R == "FR")
                            {
                                visited.push_back(myarcs[a]->g_startref());
                                ConstructRoute(visited, clonerefs, transport, start, end);
                                return true;
                            }
                            else if (R == "FSR")
                            {
                                visited.push_back(myarcs[a]->g_startref());
                                ConstructShortestRoute(visited, clonerefs, transport, start, end);
                                return true;
                            }
                        }
                        else if (!count(visited.begin(), visited.end(), myarcs[a]->g_endref()))
                        {
                            currentrefs.push_back(myarcs[a]->g_endref());
                        }
                    }
                }
                currentrefs.erase(currentrefs.begin());
                n = -1; c = 0;
            }
        }
    }
    _outFile << "\n" << "FAIL" << "\n" << endl;
    return true;
}
```

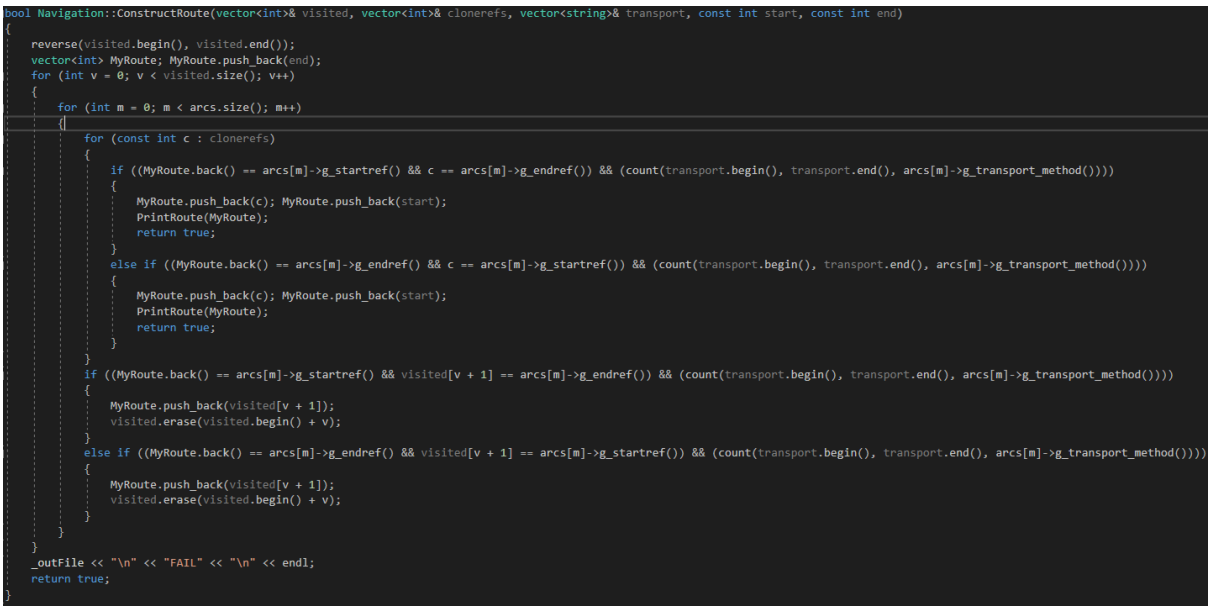
Figure 12: Route method for breadth search



```
bool Navigation::ConstructShortestRoute(vector<int>& visited, vector<int>& clonerefs, vector<string>& transport, const int start, const int end)
{
    int ref;
    vector<int> MyRoute; MyRoute.push_back(end);

    for (int v = 0; v < visited.size(); v++)
    {
        for (int m = 0; m < arcs.size(); m++)
        {
            for (const int c : clonerefs)
            {
                if ((MyRoute.back() == arcs[m]->g_startref() && c == arcs[m]->g_endref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
                {
                    MyRoute.push_back(c); MyRoute.push_back(start); PrintRoute(MyRoute);
                    return true;
                }
                else if ((MyRoute.back() == arcs[m]->g_endref() && c == arcs[m]->g_startref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
                {
                    MyRoute.push_back(c); MyRoute.push_back(start); PrintRoute(MyRoute);
                    return true;
                }
            }
            if ((MyRoute.back() == arcs[m]->g_startref() && visited[v] == arcs[m]->g_endref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
            {
                ref = visited[v];
                MyRoute.push_back(ref); v = -1; break;
            }
            else if ((MyRoute.back() == arcs[m]->g_endref() && visited[v] == arcs[m]->g_startref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
            {
                ref = visited[v];
                MyRoute.push_back(ref); v = -1; break;
            }
        }
    }
    _outFile << "\n" << "FAIL" << "\n" << endl;
    return true;
}
```

Figure 13: Method for constructing the shortest Route



```
bool Navigation::ConstructRoute(vector<int>& visited, vector<int>& clonerefs, vector<string>& transport, const int start, const int end)
{
    reverse(visited.begin(), visited.end());
    vector<int> MyRoute; MyRoute.push_back(end);
    for (int v = 0; v < visited.size(); v++)
    {
        for (int m = 0; m < arcs.size(); m++)
        {
            for (const int c : clonerefs)
            {
                if ((MyRoute.back() == arcs[m]->g_startref() && c == arcs[m]->g_endref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
                {
                    MyRoute.push_back(c); MyRoute.push_back(start);
                    PrintRoute(MyRoute);
                    return true;
                }
                else if ((MyRoute.back() == arcs[m]->g_endref() && c == arcs[m]->g_startref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
                {
                    MyRoute.push_back(c); MyRoute.push_back(start);
                    PrintRoute(MyRoute);
                    return true;
                }
            }
            if ((MyRoute.back() == arcs[m]->g_startref() && visited[v + 1] == arcs[m]->g_endref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
            {
                MyRoute.push_back(visited[v + 1]);
                visited.erase(visited.begin() + v);
            }
            else if ((MyRoute.back() == arcs[m]->g_endref() && visited[v + 1] == arcs[m]->g_startref()) && (count(transport.begin(), transport.end(), arcs[m]->g_transport_method())))
            {
                MyRoute.push_back(visited[v + 1]);
                visited.erase(visited.begin() + v);
            }
        }
    }
    _outFile << "\n" << "FAIL" << "\n" << endl;
    return true;
}
```

Figure 14: Construct first route method

```
bool Navigation::PrintRoute(vector<int>& MyRoute)
{
    string route; reverse(MyRoute.begin(), MyRoute.end());
    for (const int mr : MyRoute)
    {
        route = route + "\n" + to_string(mr);
    }
    _outFile << route << "\n" << endl;
    return true;
}
```

Figure 13: PrintRoute method for printing to the output file