

Smart Ticket

Russell Coxon - D1701

Trinity Term 2000

Abstract

A software suite for finding a route across a network. *Smart Ticket* is specific to the London Underground rail network although some aspects would apply equally to a general network. This report contains a brief introduction to graphs and graph searching, traversal algorithms, and a detailed discussion of Dijkstra's algorithm. There follows a description of the *Smart Ticket* implementation of Dijkstra and its application to the London Underground network. The report is intended to be read by a person with a reasonable background in computing and the *Smart Ticket* report by Simon Mack will give further explanation of the user interfaces mentioned in this report. A list of references is provided at the end.

Contents

Acknowledgements

Project Information

Introduction	1
1 Graphs and Graph Searching	3
1.1 Undirected and Directed Graphs	3
1.2 Graph Traversal Algorithms	4
1.3 Dijkstra's Algorithm	5
1.4 Graph Representations	8
2 The <i>Smart Ticket</i> Data Structure	11
2.1 Designing the Data Structure	11
2.2 Building the Data Structure	12
2.3 Weighting the Graph	15
2.4 Examining the Data Structure	17

3	Route Finding	19
3.1	Specifying a route	19
3.2	A Few Complications	20
3.3	The Recommended Route	21
	Conclusion	23
	Bibliography	25

Acknowledgements

I would like to thank several people for their help and assistance at various stages throughout this project. Firstly I must thank my project partner, Simon Mack, for all his hard work in developing a stable suite of programmes in which to embed the route-finding software. Also, the project supervisor, Dr. Stephen Cameron for ensuring the project stayed on course and to schedule. I would also like to thank Mark White for answering all my obscure questions about Linux, C++, Python and Latex. His patience is remarkable.

Project Information

The *Smart Ticket* route-finding software was developed on a 233MHz Intel Pentium. The operating system was Red Hat Linux 6.0. GNU Emacs 20.3 was used as the text editor to write the source code and also this report. The C++ code was compiled with the GNU C++ compiler. The Python 5.2 interpreter and Idle 0.4 interface were used for prototyping and testing. This report was typeset with L^AT_EX version 2 ϵ and the figures were prepared with Xfig 3.2. Images of the user interface were taken from Netscape Communicator 4.72 and edited with the Gimp 1.0.4. Dire Straits was the preferred auditory stimulus.

Copyright in this document is vested in Russell Coxon, all rights are reserved. No part of this document may be reproduced by any method without the prior permission of the author.

Introduction

Road and rail networks are becoming increasingly complicated and there exist several software packages to help the traveller navigate around them. The London Underground is just one example of a network that can be baffling to somebody who does not use it regularly. The aim of this project is to design and implement a software package that will help the uninitiated traveller work out the best way to get to their desired destination. Through a simple graphical user interface a traveller is asked to choose the destination station and a recommended route across the network is then returned.

Smart ticket is a software package that allows access to the route-finding software via the World Wide Web. The route finding itself is performed by a server at a remote location. A client-server approach allows several users to make route requests simultaneously from terminals with a variety of architectures and operating systems. Administration tools are provided to allow for a closed station or defective section of track. The network may be altered at any time, with the changes being implemented immediately, thus ensuring that users will always be provided with correct routes to their destination.

The project was split into two parts as shown in figure 1. My partner, Simon Mack, chose to write the network code to handle the client-server communications, the client applications and the administration tools. A set of Java applets allow the user to specify the network, apply a weighting function and request a route across the network. The applets run within a conventional web browser and so are available to anyone with access to the internet. The user simply follows the on-screen instructions to specify the required start and end stations. The recommended route returned by the route engine is overlaid on the network map.

My responsibility was for the region in figure 1 inside the dashed box. A parser converts the data provided by the user into a manageable form, allowing a data structure containing the entire

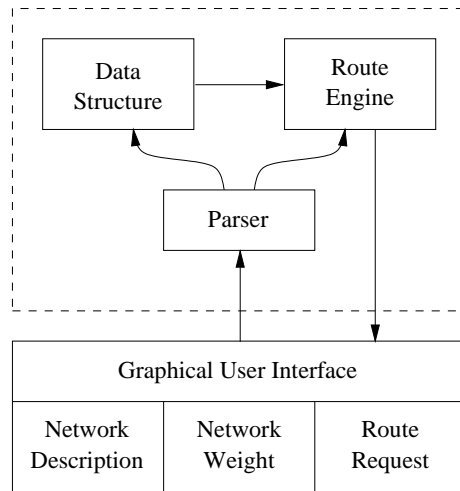


Figure 1: Project overview

network description to be built. The route-finding engine takes route requests from the parser and interrogates the data structure in order to calculate a route. The chosen route is returned direct to the user via the graphical interface shown in figure 2.

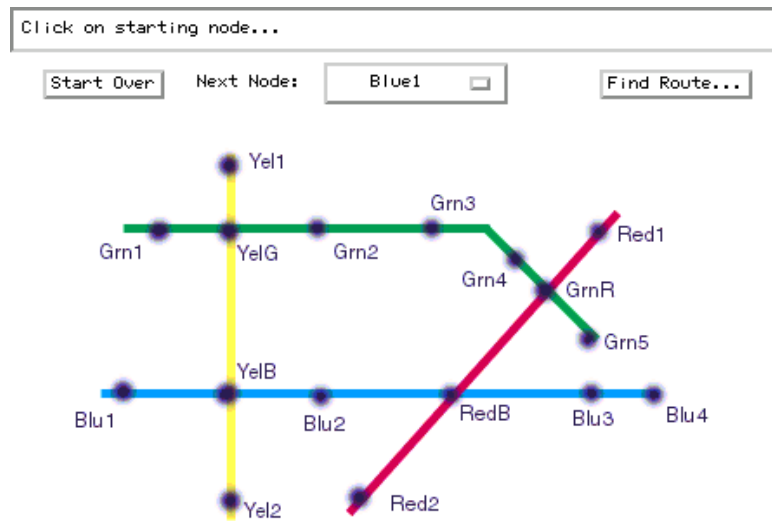


Figure 2: Graphical route-finding applet

Chapter 1

Graphs and Graph Searching

Any network may be described in mathematical terms as a graph. A graph is an abstraction that contains only the necessary data to allow searching and route finding. It is specified by a set of vertices, V , and a set of edges, E . The vertices are the nodes of the graph and the edges are the links between the nodes. In the case of the London Underground, the vertices could represent the stations, and the edges the sections of track between them. With this information it is possible to completely specify the network in the abstract form $G = (V, E)$. As a simple example consider the network shown in figure 1.1.

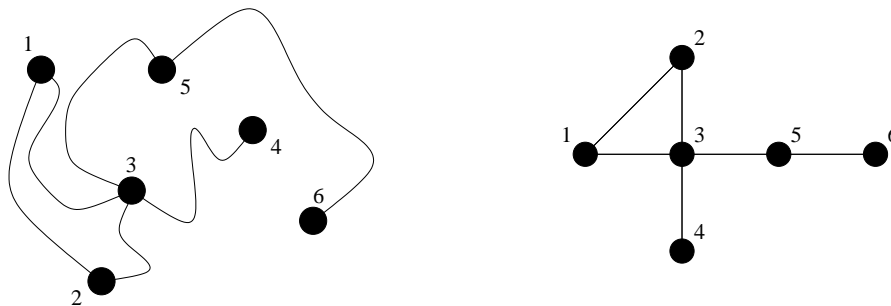


Figure 1.1: An undirected network and equivalent graph

1.1 Undirected and Directed Graphs

For the network shown in figure 1.1, the stations can be represented by the set of vertices $V = \{1, 2, 3, 4, 5, 6\}$. There is a line from vertex 1 to vertex 2 and so the edge $(1, 2)$ is an element of the set E . The complete set of edges is therefore $E = \{(1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (5, 6)\}$.

In this case no distinction has been made between the edge $(1, 2)$ and the edge $(2, 1)$; there is no sense of direction. This is an example of an *undirected* graph. If an edge (a, b) is specified, where $a, b \in V$, then there exists an edge from a to b , and an edge from b to a .

The edges of a *directed* graph have a specific direction, as in figure 1.2. If $(a, b) \in E$ then it is possible to travel from a to b , but there is not necessarily an edge from b to a . In figure 1.2, edge $(1, 2) \in E$ since an edge exists from vertex 1 to vertex 2. However there is no edge from vertex 2 to vertex 1 and so $(2, 1)$ is not in E . Two vertices are said to be *adjacent* if there exists an edge from one vertex to the other. In figure 1.2, vertex 2 is adjacent to vertex 1, but vertex 1 is not adjacent to vertex 2.

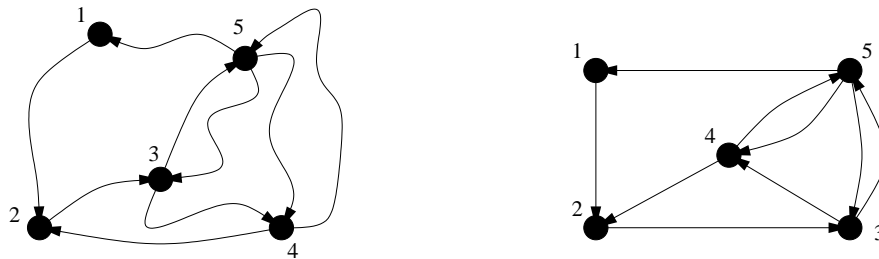


Figure 1.2: A directed network and equivalent graph

A *path* between two vertices is a sequence of ordered vertices that connect the ends of the path. For example, a path from vertex 2 to vertex 6 in figure 1.1 could be written as the sequence $\langle 2, 3, 5, 6 \rangle$. $\langle 2, 1, 3, 5, 6 \rangle$ would also be a valid path. The *length* of a path is the number of edges contained in the path. The path $\langle 2, 3, 5, 6 \rangle$ contains the edges $\{(2, 3), (3, 5), (5, 6)\}$ and so has length 3. If there exists a path from a to b , where $a, b \in V$ then b is said to be *reachable* from a . If the graph is undirected then a will also be reachable from b . Finding a route across a graph of any sort is called a *graph traversal* and this is the subject of the next section.

1.2 Graph Traversal Algorithms

Graphs are encountered regularly in computing and there are several well-known traversal algorithms. Two of the simplest are *depth-first search* and *breadth-first search*. Depth-first search works by recursively exploring deeper into the graph until it can proceed no further. The algorithm then backtracks until it finds unexplored vertices of the graph which are subsequently explored. Breadth-first search works in the opposite manner by exploring all the vertices adjacent to the start vertex, only once these have been explored does it search deeper into the graph.

Both of these algorithms have low orders of complexity, $\Theta(V + E)$ and $O(V + E)$ respectively, however, neither algorithm is optimal. While they are used frequently to find routes across networks, there is no guarantee that the routes found will be the shortest possible. In some applications this may not be important, but in the case of a rail network it would be unacceptable to send passengers on a longer journey than necessary.

The London Underground is an example of a network where there may be a large number of equally acceptable routes, but there is always a chance that a non-optimal algorithm will return a completely impractical route. Not many passengers would relish the idea of travelling from Leicester Square to Covent Garden via Heathrow Airport. For this reason it is necessary to consider *shortest path* algorithms in selecting the most appropriate for *Smart ticket*.

In most applications of graph traversal it is necessary to apply a *weight* to some or all of the edges in E . Consider as an example a communications network, where it is necessary to find the shortest path for a telephone call. In this case, the ideal route, as far as the telephone company is concerned, would be the cheapest and not necessarily the shortest. So if the edges of the graph represent routes between exchanges, then a weighting function should assign a higher weight to the more expensive telecommunication media such as satellite, with a lower weight for land based routes. This may mean that instead of sending a call by satellite, a shortest path algorithm, working on a properly-weighted graph, may choose to send the call via a more convoluted, and possibly longer route, using cheaper land-based media.

Algorithms for finding the shortest path therefore need to be able to take into account the weighting function of the graph to ensure that the shortest path, as far as the algorithm is concerned, is in fact the ideal path based on the user's criteria. The particular type of traversal that is required for *Smart Ticket* is a solution to the *single-source shortest path* problem. From a given source vertex, s , it is necessary to find the shortest path to every other vertex in the set V . The best known algorithm for solving the single-source problem is Dijkstra's algorithm. A proof of Dijkstra is beyond the scope of this discussion but is well documented in many texts.

1.3 Dijkstra's Algorithm

Dijkstra's algorithm may be used to find the shortest paths between a specified source vertex and every other vertex in a directed graph $G = (V, E)$, see figure 1.3. A weighting function $w(a, b)$

gives the weight of the edge between vertex a and vertex b . The edge weights are restricted to non-negative real numbers.

```

1  for each vertex  $a \in V$            initialise all variables
2       $d[v] = \infty$ 
3       $p[v] = \text{NULL}$ 
4   $d[s] = 0$                          set start vertex distance to zero
5   $U = V$                              all vertices are initially unexplored
6  while  $U \neq \emptyset$              loop until all vertices have been explored
7       $a = \min(U)$                    choose vertex with minimum distance field
8       $U = U - \{a\}$ 
9      for each vertex  $b \in \text{adj}[a]$  loop until all adjacent edges have been relaxed
10         if  $d[b] > d[a] + w(a, b)$ 
11              $d[b] = d[a] + w(a, b)$ 
12              $p[b] = a$ 
13   $c = t$                              begin backtrack to find shortest path
14   $P = \langle c \rangle$ 
15  while  $p[c] \neq \text{NULL}$            continue until source is reached
16       $P = P \cup p[c]$ 
17       $c = p[c]$ 

```

Figure 1.3: Dijkstra's algorithm and backtrack routine

Each vertex in the set V is assigned a distance variable. This is a measure of the current estimate of the shortest path from the source vertex to the vertex in question. So for a given vertex $c \in V$, and source vertex s , where $c \neq s$, then the shortest path estimate from s to c is $d[c]$. If c is not reachable from s then the value of $d[s]$ will not be modified by the algorithm. Initially the distance variables of all the vertices in the graph are set to infinity (lines 1-2, figure 1.3). The distances are represented as two byte numbers in the *Smart Ticket* implementation and so the distance fields are initially set to $2^{16} - 1 = 65535$. The distance estimate $d[s]$ of the source vertex is set to zero (line 4).

The parent fields of each vertex are set to null (line 3). The parent of a vertex is the one that precedes it in the shortest path from s to the current vertex. For example, if the shortest path from source s to vertex n is $\langle s, l, m, k, n \rangle$ then the parent of n , $p[n]$, is k . Similarly $p[k] = m$, $p[m] = l$ and $p[l] = s$.

A set of unexplored vertices U is maintained and once this set is empty, the algorithm terminates. Initially all the vertices are unexplored and so U contains all the vertices in the set V (line 5). The vertex with the minimum distance value is chosen from the set U (line 7). In the first instance $d[s] = 0$ and all the other vertex distances are infinity, or at least extremely large. So the first time through the loop the source vertex is chosen to be explored. The chosen vertex

is removed from the set of unexplored vertices (line 8), ensuring that it will not be explored more than once.

The next stage of the algorithm in figure 1.3 is a *relaxation* process. Each vertex has associated with it a set of adjacent vertices. These are the vertices that can be reached from the vertex in question by traversing just one edge. For example, in figure 1.1 the set of vertices adjacent to vertex 3 would be $adj[3] = \{1, 2, 4, 5\}$. A vertex b which is adjacent to the vertex being explored is chosen (line 9). If the shortest distance estimate $d[b]$ is less than the sum of the shortest distance estimate to a , $d[a]$, and the weight of the edge between a and b , $w(a, b)$, then the shortest path to b is modified to go through vertex a . This is best demonstrated by example in figure 1.4.

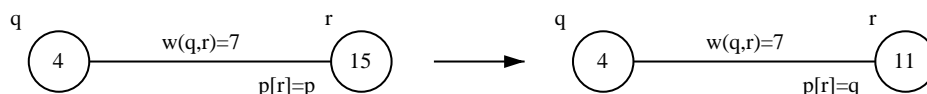


Figure 1.4: Relaxation of an edge

Two vertices, q and r , of an undirected graph are connected by the edge $(q, r) \in E$. Before the relaxation process (lines 10-12), the shortest distance estimates are $d[q] = 4$ and $d[r] = 15$. The parent of vertex r is some other vertex p , where $p \neq q$. If the weight of the edge is $w(q, r) = 7$ then it can be seen that $d[r] > d[q] + w(q, r)$, and so the shortest distance estimate to r can be improved by modifying the shortest path to go through q . After relaxing the edge (q, r) , $d[r] = d[q] + w(q, r) = 11$, the parent of r is set to q indicating that the shortest path from the source s to r is likely to go through vertex q .

Returning to the general case, the algorithm in figure 1.3 will continue until every edge in the graph has been relaxed. On termination of the algorithm the distance field of every vertex will be the length of the shortest path from the source to that vertex. The shortest path itself may be found by following the parent fields from the destination t back to the source s which is the process of *backtracking*. Line 13 sets the variable c to the destination vertex t . The sequence P , is the path from destination to source, which is simply the reverse of the required path. Initially P contains only the destination vertex t (line 14). The parent of the current vertex, $p[c]$ is examined in line 15. If it is NULL then either the source has been reached (recall that the parent of the source vertex is set to NULL in line 3 and never modified), or c is not reachable from the source. If $p[c] \neq \text{NULL}$ then the parent of c is added to the sequence P (line 16), and c is set to its own parent (line 17). Backtracking along the path from the destination continues until the source is reached when the routine will terminate (line 15). Finding the path from s to t is now the trivial

case of reversing the sequence P , not shown in figure 1.3.

1.4 Graph Representations

In the case of *Smart ticket*, the computer needs to have knowledge of the layout of the London Underground network. The stations, segments of track between them, and the different lines must all be stored in a *data structure*. The previous section has shown how an arbitrary network may be specified in abstract form as a graph $G = (V, E)$, essentially a set of vertices and a set of edges. This chapter considers how to represent a graph within the memory of a computer.

There are two main methods of representing a graph; an *adjacency matrix*, or a set of *adjacency lists*. An adjacency matrix is simply a square matrix of boolean values which has the same dimension as the number of vertices in the graph. A true value in position i, j indicates that an edge exists from vertex i to vertex j . Conversely a false value indicates that edge (i, j) is not an element of the set of graph edges E . Figure 1.5 shows the adjacency matrix for the directed graph given earlier in figure 1.2. The adjacency matrix representation is favoured for undirected graphs since it is symmetric. This means that only the upper or lower triangle must be stored. The disadvantage of an adjacency matrix representation is that it contains many false values as well as true ones. It would be more efficient to only store the edges of a graph that exist, rather than also recording those that do not.

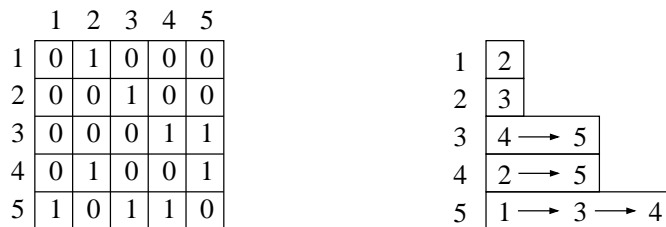


Figure 1.5: Adjacency matrix and list representations of the directed graph in figure 1.2

An adjacency list representation is a set of lists, one list for each vertex in the graph. Each list, indexed by vertex number, contains references to the vertices that are adjacent to the vertex given by the list number. For example, figure 1.5 shows the adjacency list representation of the directed graph in figure 1.2. There are five vertices, and consequently five lists. Vertex 1 has only one vertex which is adjacent to it, vertex 2, and so list 1 contains only vertex 2. Similarly vertex 2 only has vertex 3 adjacent to it, so list 2 contains only vertex 3. The number of vertex numbers

that must be stored to represent a directed graph is equal to the number of edges in the set E . This is an efficient storage method for sparse graphs where an adjacency matrix would contain many false values. It is the preferred method of representing a directed graph.

Chapter 2

The *Smart Ticket* Data Structure

The London Underground could be treated as an undirected network; if it is possible to travel between two stations in one direction, then it is also possible to travel between the same stations in the opposite direction. However, while the current network is unidirectional, new parts of the network may be directional, it would be impossible to incorporate directed travel into an undirected graph. Considering the network as directed allows for lines between stations in opposite directions to be weighted differently. If, for example, the section of the Northern Line between Waterloo and Embankment in the northbound direction was closed, then the edge of the graph representing that section of track in the graph could be assigned an extremely large weight. This would mean that any passengers who would normally expect to use that part of the line would be sent via an alternative route. The weighting of the southbound section of the same line would not be changed so passengers travelling south would not be affected.

2.1 Designing the Data Structure

It has already been shown that an adjacency list representation is the most appropriate for directed graphs, so the next problem to consider is how to represent the London Underground in the form of an adjacency list. Figure 1.5 gives an example of how to represent a simple directed graph in adjacency list form. The Underground network is significantly more complex. The major complication that arises is that the network is divided into twelve distinct lines, with travel between lines permitted at only a few stations. Each line may be considered as a separate network but the traversal algorithm must consider the network as a whole in order to find a route across it.

Vertices are defined to be *any instance of a station on any line*, so there will be significantly more vertices in the graph than there are stations in the network. As an example consider Bond Street Station, figure 2.1. Bond Street is connected to four other stations by the Central and Jubilee lines. By the definition of vertices above, Bond Street is represented by two vertices, one on the Central Line and another on the Jubilee Line. A *virtual line* between the vertices represents the action of changing lines. A weight applied to the virtual line is used to impose a penalty for changing lines. This will be referred to as the *line change penalty*. The line change penalty ensures that passengers are not made to change lines too frequently, instead they may be sent by a slightly longer route with fewer changes in order to minimise the journey time.

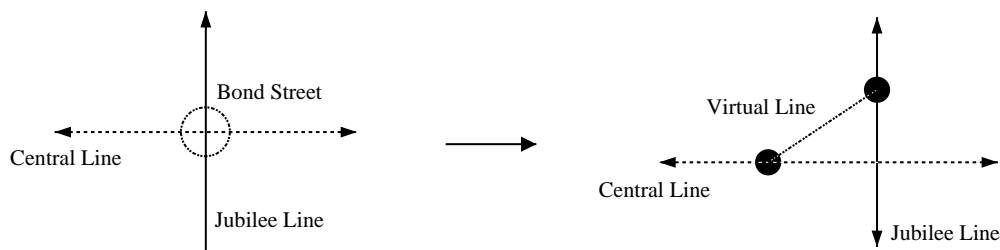


Figure 2.1: Bond Street station represented by two vertices and a *virtual line* between them

2.2 Building the Data Structure

Having discussed the method of representing a station by one or more vertices, it is now necessary to consider how the Underground network itself is entered into the data structure. An administration tool is provided as part of the *Smart Ticket* suite. A simple graphical user interface, by Simon Mack, lets a manager specify the network. Stations and sections of track may be added, modified or removed using the applet. The Applet itself does not change the data structure, but instead generates a string according to an agreed specification that is passed to an external function, see figure 2.2

All the functions to build and maintain the data structure, as well as to find routes across the network, are contained in the class, `routeEngine`. The function, `setNetwork`, a member function of `routeEngine`, creates the data structure that is used by the graph traversal algorithm.

Each of the lines and stations in the network is assigned a unique reference number. It was decided, for convenience of compatibility between C++ and Java data types, to restrict all reference numbers to a single byte, so only numbers in the range $[0, 255]$ are available. This set is

divided by specifying the numbers in the range [8, 31] to represent lines and [32, 255] for stations. The remaining values [0, 7] are retained as reserved characters. This imposes a restriction on the size of the network to 24 lines and 224 stations, enough for a prototype system such as *Smart Ticket*.

A string is used to pass the network description from the server to the `setNetwork` routine. It consists of a line number followed by a series of station numbers on that line, followed by another line number and its associated stations and so on. Figure 2.2 shows a simple network and the string that would be generated to describe it by the administration tool.

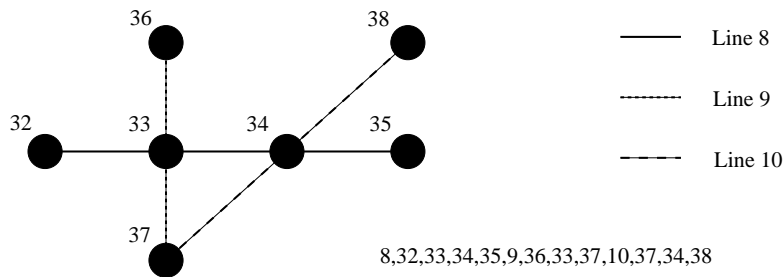


Figure 2.2: A simple network and its network description string

A simple network may be described by this method but there are two situations that may occur in more complex networks, including the London Underground, that are not allowed for. The first is that some lines are branched, for example the District Line has four different branches. The solution is to give each branch a different reference number and treat it as a separate line, the problem with this is highlighted later. The second situation to be considered is continuous lines. The Circle Line, as its name suggests, is a good example. By choosing an arbitrary starting station, and making it the first and last entry in the list of stations on the line, then a continuous line can be specified.

```
class routeEngine {
:
    int setNetwork ( unsigned char *network, int length );
:
}
```

`setNetwork` takes a pointer to the network description string, `*network`, and an integer, `length`, giving the string length, as its arguments. The task of `setNetwork` is to parse the string passed to it from the administration tool, and create the data structure.

By considering the range of each number in the string, it is possible to distinguish between lines, stations and reserved characters. Scanning through the string, `setNetwork` can identify any new instances of a station on a line and call the private member function `add_graph_vertex` to generate a new vertex. By examining each station in turn, `SetNetwork` can build the adjacency list for each vertex, the vertices being stored as a singly linked list. This means that the storage for the vertices may be allocated dynamically, essential since the number of vertices can not be known at the time of compilation.

```
struct graph_vertex {  
  
    unsigned short int    index;  
    unsigned short int    distance;  
    explored_state        colour;  
    graph_vertex          *parent;  
    unsigned char         *weights;  
    graph_vertex          **adjacents;  
    unsigned char         array_length;  
    graph_vertex          *next_graph_vertex;  
  
};
```

Each vertex is a C++ `struct`, containing a pointer, `next_graph_vertex` to the next element in the list. The next element also has a pointer to the following element and so on. The `next_graph_vertex` pointer of the last element in the list is `NULL`. The reason for using a `struct` instead of a `class` is that the data type `graph_vertex` contains data only and not methods. The functions operating on the `graph_vertex` elements require access to most of the `struct` elements and so there would be no advantage in making some of the data elements private in a `class` implementation.

Each `graph_vertex` contains a number of data fields. Since the same station may occur on more than one line, it is necessary to specify both the line and station numbers in order to find a particular vertex. To make the indexing of vertices easier, a function `make_index` is used to create a double-byte number that uniquely defines each vertex. The `distance` field is used to record the current estimate of the length of the shortest path to the vertex in question. So for a given vertex a , the `distance` field would contain the value of $d[a]$, see figure 1.3. The `parent` field maintains a pointer, $p[a]$ to the parent of the vertex being considered.

As the algorithm runs it is important to maintain a record of the vertices that have been explored and those that have not. A useful analogy that is commonly used is colour. Unexplored

vertices are coloured white, while explored vertices are black. Once no more white vertices remain, the algorithm terminates. In this implementation, vertices have an `explored_state` which may take either of the values `WHITE` or `BLACK`.

```
enum explored_state { WHITE, BLACK };
```

The `**adjacents` field of each `graph_vertex` element contains a pointer to an array of pointers. The array `*adjacents` contains pointers to all the graph vertices that are adjacent to the current vertex. `array_length` is the number of vertices in the array, the length of `*adjacents`.

The weights of each edge must also be stored in the data structure, this is done using the `weights` array, which also has length `array_length`. Consider a vertex a with some other vertex b adjacent to it, $a, b \in V$. If a pointer to b is stored in `adjacents[i]`, then the weight of the edge from a to b , $w(a, b)$, is stored in `weights[i]`. A value of 255 in any element of a `weights` array indicates that the line change penalty should be used as the weight for that edge. Storing the line change penalty separately from the main data structure means that it can be changed without having to scan through the data structure. A value of 224 causes the *same line penalty* to be used. The same line penalty is used on edges that do not require any special weighting. All weights are restricted to the range $[1, 100]$.

2.3 Weighting the Graph

As shown earlier, the edges of the graph represent sections of track between stations, or passage between lines. So, edge weights can be used to promote or deter travel through a particular part of the network. The function `setWeight`, a public member function of the `routeEngine` class, can be used to set the weight of a particular edge of the graph. It takes as its arguments the `start` and `end` stations, and the `line` number of the edge whose `weight` is to be set.

`setWeight` does not itself make any change to the data structure. Instead it records the weight request in a text file, `weights.txt`. Another member function of `routeEngine`, `restart`, must then be called in order for the route changes to be implemented. The reason for this seemingly obscure method of operation is to do with the way in which multiple instances of *Smart Ticket* may be running at any one time. When a user makes a route request, a copy of the base class is made for that particular user. Any changes made to the data structure, such as setting global

obtained by invoking `getWeight` with `start` equal to one or two respectively.

The user interface, written by Simon Mack, that allows a manager to change or inspect weights or penalties is shown in figure 2.3. The nodes at the end of the edge whose weight is to be set can be selected from a drop down list. Edge weights can be set with the `update` button or read from the data structure by clicking `refresh`. The global line change penalty may be handled in a similar manner.

The screenshot shows a graphical user interface for controlling edge weights and global penalties. At the top, there are three dropdown menus for selecting nodes: 'Blue2', 'Blue Line', and 'Yellow Blue'. Below these are three rows of controls, each with a text input field, an 'Update' button, and a 'Refresh' button. The first row is labeled 'From first node:' with a value of '20'. The second row is labeled 'Opposite direction:' with a value of '20'. The third row is labeled 'Global Line Change Penalty:' with a value of '25'. At the bottom, there is a text box containing the message ':TCP1NetClient: Data received from host www.smartticket.dhs.org port 9735'.

Figure 2.3: Edge weight and global penalty control applet

Future versions of *Smart Ticket* could make use of the edge weight capability to implement, for example, timetabling. A separate programme could make calls to `setWeight` in order to encourage passengers to use trains which have spare capacity. By keeping a record of previous routes requested, and interfacing the system to ticket machines and barriers, it would, in theory at least, be possible to know the position of any passenger in the network. Congestion at a particular station could be avoided by increasing the weight of the edges in the vicinity of the busy station.

2.4 Examining the Data Structure

Another of the six functions defined in the `routeEngine` class is `getNetwork`. `getNetwork` collapses the data structure into the string that originally created it. The function has no practical purpose in the current version of *Smart ticket*. It proved very useful in debugging the `SetNetwork` function by comparing the returned string with the network description string that was originally passed from the network definition applet.

Two further functions, which have been removed from the final implementation, `test_all` and

```
class routeEngine {  
:  
    int getNetwork ( unsigned char *network, int length );  
:  
}
```

`test_vertex` were used to debug the data structure. The functions printed either the whole data structure, or a selected part of it respectively, to the screen in a readable manner. Although this method of debugging is slow it does allow thorough examination of the data structure.

Chapter 3

Route Finding

Once `setNetwork` has built the network data structure, with edge weights and global penalties set as required, the remaining function in the `routeEngine` class is used to find a route across the network. `getRoute` takes two arguments; a string, `route` and an integer `length`.

```
class routeEngine {  
    :  
    int getRoute ( unsigned char *route, int length );  
    :  
};
```

3.1 Specifying a route

Through an intuitive graphical user interface, see figure 2, a passenger may specify their journey requirements. The passenger must enter a start and destination station, and also any stations that they wish to stop at in between. A string, `route`, is generated that contains the station numbers chosen by the passenger. In the simplest case a journey from a to b would be represented in the string as $\#(a), \#(b)$, where $\#()$ is the unique reference number assigned to each station as described earlier. If a passenger wishes to stop at station c on the way, then `route` would read $\#(a), \#(c), \#(b)$. Since `route` is passed to `getroute` by pointer it is also necessary to specify the length of the string. If this was not done then `getRoute` would be likely to try to access out of bounds memory, leading to a segmentation fault.

In the current version of *Smart Ticket*, *via* requests have not been implemented, so `getRoute`

takes `route[0]` to be the start station and `route[length-1]` as the destination station. A via request could be implemented in a future version by splitting the route into `length-1` parts and making successive calls to `GetRoute` with each part of the required journey.

3.2 A Few Complications

One major difficulty arose due to the method of defining vertices in the abstract data structure, see figure 2.1. The start and destination stations which the passenger has specified are passed to `getRoute`, but in the data structure several vertices may represent the same station. This means that there is likely to be a choice of starting and destination vertices. The difficulty is choosing which vertices should begin and end the journey. It was found during testing that often the route recommended across the network by `getRoute` involved a line change at the starting and/or destination stations. It clearly does not matter to the passenger which line they begin their journey on, nor does it matter which line takes them to their destination station.

The solution to this problem is to find all the vertices that represent the start and destination stations requested by the passenger. The function `Dijkstra`, an implementation of the Dijkstra algorithm, is then invoked on each pair of start and destination vertices, and the shortest path distance recorded. The pair of vertices that have the shortest path between them is chosen, and the route returned to the passenger.

During the testing of the stand-alone version of *Smart Ticket*, a second data structure related problem was encountered. It was explained earlier that branched lines were considered as a series of separate lines that intersected at the branch junctions. This allowed for branches of lines to be represented in the same manner as ordinary lines in the data structure. The problem with this representation is that the route-finding algorithm has no knowledge that two separate lines in the data structure do in fact represent branches of the same line. The result of this is that the recommended route will sometimes indicate that a passenger must change line, when in fact no line change is required. A solution to this has not been implemented in *Smart Ticket*, but a suitable method would be to record in a separate file a list of which lines in the abstract data structure actually represent the same line in the real network. This information would then be available to the route-finding software when calculating the most efficient route.

3.3 The Recommended Route

The route returned from the function `Dijkstra` contains the index number of every vertex in the path from source to destination. This is more information than the passenger requires and so the returned string is edited before being passed back to the Java applet from which the passenger specified their journey. The string, `route` is edited to contain only the index numbers of stations at which the passenger must change line, as well as the start and destination stations. The route is presented to the passenger as a series of connected symbols, superimposed on a map of the network. This method of displaying the route is very simple, and indicates clearly the recommended route that the passenger should take.

Conclusion

The current version of *Smart Ticket* meets most of the original design specifications. A network may be described via a simple graphical user interface, edge weights and the line change penalty set, and a route requested. A route is returned which is the shortest possible from the given start to the destination. Although it is possible to request a journey via a particular station, this request is currently ignored but could be accommodated in a future version. The user interfaces are intuitive and easy to use, a demonstration is available on the world wide web at www.smartticket.dhs.org, although this page is currently password protected.

Due to the amount of time that would be required to specify the complete London Underground network, *Smart Ticket* has only been tested on small networks. However a standalone version has also been written, by Russell Coxon, which provides a very primitive text-based interface. The entire London Underground network has been included. By making use of a short programme written in Python, it is possible to convert real station names and lines into index numbers which can then be entered into the standalone interface. The standalone version only provides the recommended route in numerical form, and it is necessary to use the Python programme again to convert the index numbers back into real-world names. While the user interface is a little awkward, the standalone version demonstrates the capability of the route-finding algorithm and there is no problem representing a network of more than 350 stations in the data structure. On the few occasions that the recommended route has been unacceptable this has usually been due to extra knowledge not available to the system, such as avoiding certain stations which are known to be busy, or staying within certain zones where a ticket is valid.

During the development process, regular meetings were held to agree on firstly the broad specifications for *Smart Ticket*, and latterly the finer details of the project. It was essential to agree on specifications for passing data between the two parts of the project before beginning work on writing code. For this reason the definition of the class `routeEngine`, as well as the

prototypes of its member functions were written early in the development process. This allowed both designers to ensure that their own code met the agreed specifications. The separate halves of *Smart Ticket* were merged together five times. After each merge the whole system was tested and bugs reported back to the author of the troublesome piece of code. The merged code was then used as a basis from which the code for the next merge could be developed. This method of agreeing specifications before beginning to write code allowed the two parts of the project to be written entirely separately, with a guarantee that the resulting merged code would meet all the requirements.

Throughout the design and development stages, as much generality as possible has been maintained concerning the application of the software. Although the current version is based on the London Underground rail network, it is not hard to see that the software has other applications, some of which have been hinted at earlier in this report. Route finding is an extremely important technique and the methods demonstrated here could equally be applied to other fields. *Smart Ticket* could have applications in finding routes through communications networks, designing printed circuit boards, urban planning, and numerous other areas where an element of route finding is necessary.

In conclusion the *Smart Ticket* project has been very successful and I am extremely pleased with the result. *Smart Ticket* was only ever intended to be a prototype system; the current version, including the standalone, demonstrates very clearly that the system is suitable for use on a much larger scale network.

Bibliography

- Adamson, T. Iain, *Data Structures and Algorithms* (Springer, 1996).
- Brassard, Gilles & Bratley, Paul, *Fundamentals of Algorithmics* (Prentice-Hall, 1996).
- Cameron, Debra, *GNU Emacs Pocket Reference* (O'Reilly & Associates, 1999).
- Cormen, H. Thomas & Leiserson, E. Charles & Rivest, L. Ronald, *Introduction to Algorithms* (MIT Press/McGraw-Hill, 1990).
- Lamport, Leslie, \LaTeX : *A Document Preparation System* (Addison-Wesley, 1994).
- Loudon, Kyle, *Mastering Algorithms with C* (O'Reilly & Associates, 1999).
- Lutz, Mark & Ascher, David, *Learning Python* (O'Reilly & Associates, 1999).
- Oualline, Steve, *Practical C++ Programming* (O'Reilly & Associates, 1997).