Using GIS to create public transport travel time isochrones for the Glasgow area

David B. O'Sullivan MA (Cambridge 1988)

Dissertation submitted in partial fulfilment of the requirements for a Master of Science degree in Topographic Science (Cartography and Geoinformation Technology strand)

12 September 1997

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

0. ABSTRACT

Current off-the-shelf geo-information systems (GIS) handle free movement over networks reasonably well (for example, cars on the road network), but cannot usually cope with the timetabled services typical of public transport networks. The application of a GIS to one method of examining public transport and accessibility issues in the Glasgow area has been attempted and is described in detail. The method used is the construction of public transport travel time *isochrones*. These are lines which enclose locations which can be reached from a specified starting location inside some specified elapsed time. The relationship between isochrones and other accessibility measures is described. A set of 'GIS isochrone tools' has been designed and built for *ArcView,* a popular desktop GIS. Their operation is described in detail. Whilst the desktop GIS used has proved adequate for this task, execution is not fast enough for interactive investigation of isochrones, so a number of ideas for improvements in implementation are presented. Some of these ideas would speed up the current implementation; others would improve the accuracy of the results. A number of output maps from the current work are presented and assessed. A simple example of more complex manipulation of isochrones is presented to demonstrate why isochrones might be a particularly appropriate general tool for transport planners. It is argued that GIS is the ideal platform for a general tool of this sort, thanks to its well developed data handling and presentation capabilities.

1. INTRODUCTION

1.1 Aims

The work described in this dissertation was started with high ambitions of producing a program or suite of programs, in the form of additions to a standard GIS (in this case *ArcView 3.0*), which would allow a user to rapidly produce accurate isochrone maps of the Glasgow region from any specified starting point.

This aim has been only partially fulfilled, for a number of reasons, mostly related to the scale of this ambition. Glasgow is a large conurbation with a correspondingly large and complex public transport network. If we take the area covered by the Ordnance Survey 1:50 000 *Landranger* map, then as a boundary for 'Greater Glasgow', then that 1600 square kilometre area contains 122 conventional overground railway stations, and a 15 stop underground railway system. The bus services in the region are deregulated and legion. It is difficult even to guess at their numbers. Something of the order of three to four hundred distinct bus routes run into and through the city centre, and still more serve outlying areas. These services are run by two major companies and many smaller ones, particularly in the outlying areas.

In addition public transport is a complex and varied phenomenon in its own right. Express routes, standard bus routes, shorter 'hopper' routes, minor variations in routing with time of day (routes 6, 6A, 6B, 6C and the like), are all aspects of the problem of predicting the average speed of travel on public transport which require consideration. Small details of the 'street' network - subways, footbridges,

pedestrianised road schemes, footpaths, and parks all add to the complexity of estimating journey times by the most pervasive means of public transport - walking.

As a result there are many aspects of the work described which are not entirely satisfactory, but which could not be addressed in the allotted time. Much of this dissertation focuses on inadequacies in the current approach, and outlines 'better' solutions. This work can therefore be thought of as a 'pilot scheme' for what would have to be a larger scale project if a more complete solution were to be reached.

Note Throughout the remainder of this dissertation, programs are referred to as 'scripts' which is the *ArcView* term for programs written in its macro-language *AVENUE*.

1.2 Dissertation Outline

This dissertation is written in three parts: background information, a detailed description and critical review of the work done, and a presentation of some typical outputs.

Chapter 2 covers some general background material, starting with the reasons for the author's personal interest in the work outlined, and a brief consideration of potential applications. A discussion of isochrones and their relevance to transport accessibility studies follows. The transport features provided by typical off-the-shelf desktop GIS packages are then described.

Chapter 3 gives an overview of the approach used to produce isochrone maps. Included are a brief consideration of the reasons for using *ArcView 3.0* for this work,

together with a brief outline of the peculiarities of the *AVENUE* macro programming (or 'scripting') language. Some aspects of the language have had a strong effect on the approach taken. The later sections of this chapter explain the programming approach in outline.

Chapter 4 is the major part of this dissertation and includes detailed explanations and reviews of each of the *AVENUE* scripts written for this work. The explanation of the scripts is integrated with the review/criticism material to make it easier to see where exactly improvements could be made or alternative approaches could be adopted. Chapter 5 presents a summary of these criticisms in the form of a more general and wide-ranging discussion.

Chapter 6 presents some examples of outputs from the tools described, and where appropriate includes information which explains how these results could be useful.

Finally, chapter 7 draws some conclusions and presents some thoughts on possible extensions and further work.

Further relevant information appears in the Appendices, including data sources, full script listings and a 'User Manual' for the *ArcView* tools which have been built.

2. BACKGROUND

2.1 Personal interest and potential applications

One of the most frustrating experiences of life in our cities is the feeling waiting at a bus stop that you should be somewhere else. Should you have walked five minutes further up the road to the big junction where there is a wider choice of buses; or should you have walked in the other direction and caught a train instead? Many years living in Britain's two biggest cities, yet never owning a car, have given the author plenty of time to think about such matters. One of his more persistent thoughts on the subject was that it would be interesting to construct a map showing just where a traveller *ought* to be able to get to given complete knowledge of the transport system. Although it would not be much comfort to the hapless commuter getting wet at the bus stop it would be interesting to see such a map. Such a map is referred to in the following pages as an *isochrone map*.

So much for personal reasons for attempting the work described in this dissertation, there are also a number of potential real applications. Recent changes in the political climate make it more likely that policy makers will wish to assess where resources should be directed to effect improvements in the transport network. The same changes in attitude (and possibly taxation) may make it incumbent on businesses to pay more attention than previously to the accessibility of their facilities via public transport. As will be demonstrated, reasonable tools for assessing such access via private road transport are widely available already. Social researchers may be interested in using the methods outlined here to assess the variation in access to facilities - retail outlets, hospitals, leisure centres etc - which the existing networks provide, and to assess whether such variations are systematically related to other

measures of social welfare. It is imagined that the tools described would most likely be of interest to these groups - transport planners, geographers, social scientists, corporate personnel managers - and it is they who are referred to as 'users' in subsequent chapters. Nevertheless the hapless commuter mentioned above would doubtless be interested in the results!

2.2 Accessibility measures

Isochrones are an example of a much wider range of measures used by planners and theorists to measure *accessibility* in a region. There follows a discussion of some of the issues in this field (by no means exhaustive), which should give an idea of where an isochrone approach fits into the wider theoretical picture.

Accessibility is generally agreed to be hard to define but critical to any serious understanding of transport issues (Hanson 1986, p11ff). A related concept is *mobility* which describes the overall ability of people to travel over distances. A simple approach to measuring mobility is to count numbers and lengths of journeys, and express the results in distance per person per unit time. Mobility on this measure has increased consistently over time as technology has produced faster forms of transport (see for example Table 1-1 in Hanson p8). However, it is clear that much of this increase in mobility is due to necessity - homes, places of employment, retail outlets, leisure facilities are more geographically dispersed. In fact there is a suggestion in the literature that as mobility has increased travel times have remained more or less constant, and may even have risen somewhat (Tolley & Turton 1995 p16).

The goal of a transport system is not mobility for its own sake, but access to facilities. The demand for transport is a derived demand, arising out of travellers' needs for food, work, and other necessities. So, to evaluate a transport system we need some measure of its effectiveness in delivering people to the facilities they wish to use. Koenig (1980) refers to a definition proposed by Dalvi (1978), where accessibility "denotes the ease with which any land-use activity can be reached from a particular location, using a particular transport system". Such a definition makes it clear that accessibility relates to a given origin, a given transport system, and to a given land-use activity. For example, the accessibility via public transport of employment opportunities to residents of a particular housing estate might be of interest. In Koenig's article this definition is the prelude to an extended investigation of various measures based on economic concepts such as utility and consumer surplus. He presents the formulation adopted by most writers for the accessibility A_i of a location i of the form:

$$A_i = \sum_j O_j \, f(C_{ij})$$

where O_j is the set of opportunities (or land-use activities) available at location j, C_{ij} is the distance or time or cost of the journey from i to j, and $f(C_{ij})$ is some function of the cost. Usually the function f has the characteristic that as the cost increases, the value of the function falls, so this element of the formula represents a weighting of the sum so that more distant opportunities are rated lower in the total. This is intuitively sensible - otherwise job opportunities in Aberdeen might be included in accessibility measures for Glasgow, which would not make much sense. There is an extensive literature exploring the merits of different variants of the function f and exponential, Gaussian, and various other forms have been investigated (see for example Hansen 1959, Wachs & Kumagai 1973, Doling 1975, Weibull 1976, Frost & Spence 1995). It is clear that different approaches give different results. It is not

clear which might be appropriate in different situations. Guy (1983) illustrates how the precise method used can lead to different conclusions. Al-Sahili and Aboul-Ella (1992) show how a much simpler approach can still be useful. In any case, it can be seen that accessibility is a more subtle measure than mobility. Mobility may increase but if everything is further apart than before accessibility is likely to be not much changed, or even reduced.

An isochrone analysis is a special case of the above equation, where f is a step function, with value 1 where the cost (in my work expressed in units of time) is less than or equal to the isochrone limit, and value 0 where the cost of travel exceeds the limit. In other words only those opportunities reachable from i within the cost limit of the isochrone are included in the accessibility calculation. Koenig notes (p158) that an isochrone approach to the investigation of accessibility issues is "intrinsically sound" (this is in the context of an argument for the use of economic utility-based methods, rather than the distance weighted sum in the equation).

2.3 Constrained accessibility

Another view on accessibility measures contends that any realistic assessment of the situation must take account of typical subjects' time-budgets. A location i may afford access within one hour to a range of facilities, but if typical residents are not living to schedules or *time-budgets* which give them two or more hours to spare to make use of those facilities then they have no effective access. This *space-time* approach to accessibility studies was pioneered by Hägerstrand and others at the University of Lund. Robertson (1981) and Forer & Kivell (1981) are good examples of this approach. A common presentation consists of diagrams such as figure 2.1. These diagrams are two-dimensional with space represented as a single dimension only.

For an individual resident at location A, leaving home at (say) 8.00am, the shaded area in figure 2.1(a) indicates the range of accessible locations, when there is no constraint on that individual's movements other than the transport system. The slope of the lines diverging from A will vary with the speed at which the transport system allows movement through space. Now, if this individual must be at work at location B by 9.00am, then this places a constraint on movement which effectively limits accessibility to the shaded area in figure 2.1(b), since the parallelogram is the intersection of all the points which can be reached from A starting at 8.00am, with all the points from which B can be reached by 9.00am. This shaded area is often referred to as a *space-time prism* since it actually exists in three dimensions - two of space and one of time.

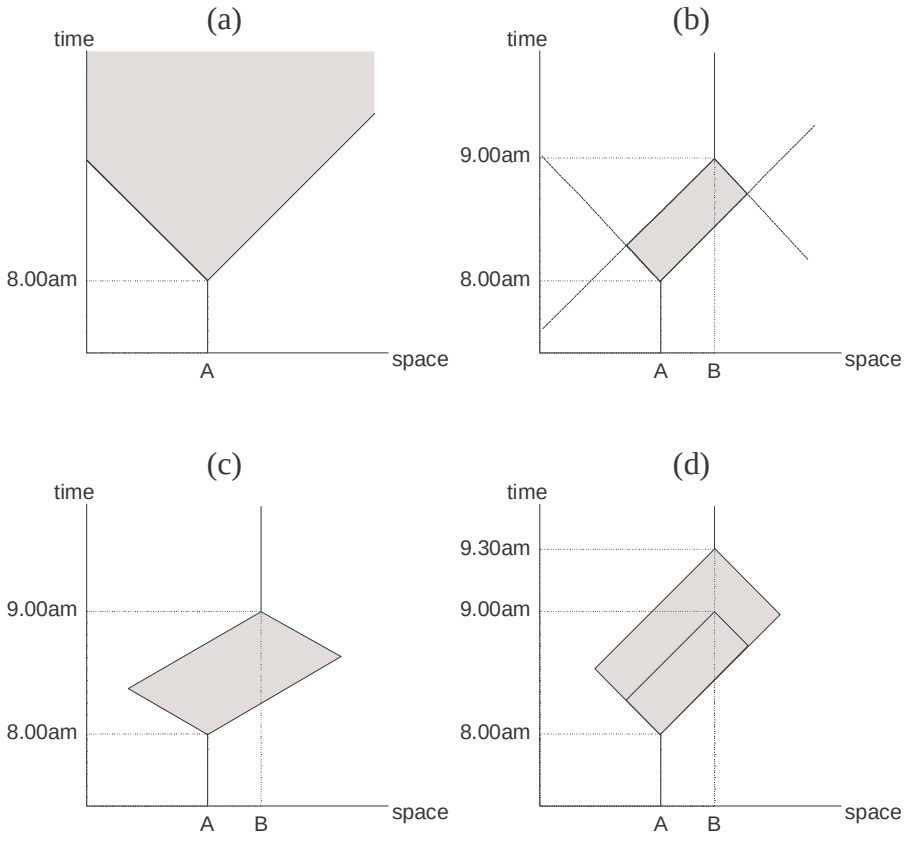


Figure 2.1 Space-time prisms showing travel with constraints

Figure 2.1(c) shows the effect of a better transport system, whereas, 2.1(d) shows the advantage of flexi-time arrangements. The space-time approach to accessibility is much more satisfying intellectually than the earlier overall accessibility indices, although it does suffer from problems of ease of application - specific cases must be considered, and it is difficult to summarise results. In practice both approaches are useful, and complementary.

2.4 Isochrones

Since using isochrones to analyse accessibility is sound, and readily understood, it is surprising that it appears so rarely in the literature. I have found few examples of isochrone maps in any of the materials I consulted. Glasgow PTE's study in 1969 did include some isochrone maps (pp 72, 92, 100), and I would not be surprised if there are similar examples hidden in similar transport planning documents elsewhere.

Possible reasons for the lack of academic interest compared to the more mathematical approaches referred to in the section 2.2 could be:

(i) separate maps must be produced for each origin location using isochrone analysis. Comparison of the accessibility at different origin locations is difficult. A natural solution is to convert the isochrones to some sort of index of opportunity - which is what the more mathematical measures are;

(ii) prior to the development of large scale computing resources in general, and GIS in particular, the construction of isochrones would have been tedious, difficult and slow;

(iii) there are interesting academic questions to be answered about what is the most appropriate exact form for the equation presented in the previous section; and

(iv) until recently there has been something of a bias against presenting material in a graphic rather than numeric form. The advent of the computing resources referred to in (ii) above, has started to change this.

Forer and Kivell's work (1981) uses the space-time approach to deduce sensible isochrones to plot for New Zealand housewives. This throws isochrones into an interesting light, in that if the diagrams of figure 2.1 are redrawn in three dimensions so that space has two dimensions (as it effectively has on a map, or in a GIS), then an isochrone is a horizontal slice through the unconstrained space time prism or 'cone'. Figure 2.2(a) illustrates this idea. Constrained accessibility can be deduced for the start and end destinations A and B, by intersecting various of the isochrones for A and B, depending on how much time is available for the journey (figure 2.2(b) gives an idea of what is involved). Points on the surface of the shaded volume can be reached with no time to spare; points inside the volume leave time free for other activities. A more detailed description of this process is provided in section 6.3 where I have examined the feasibility of carrying out this process in a GIS setting.

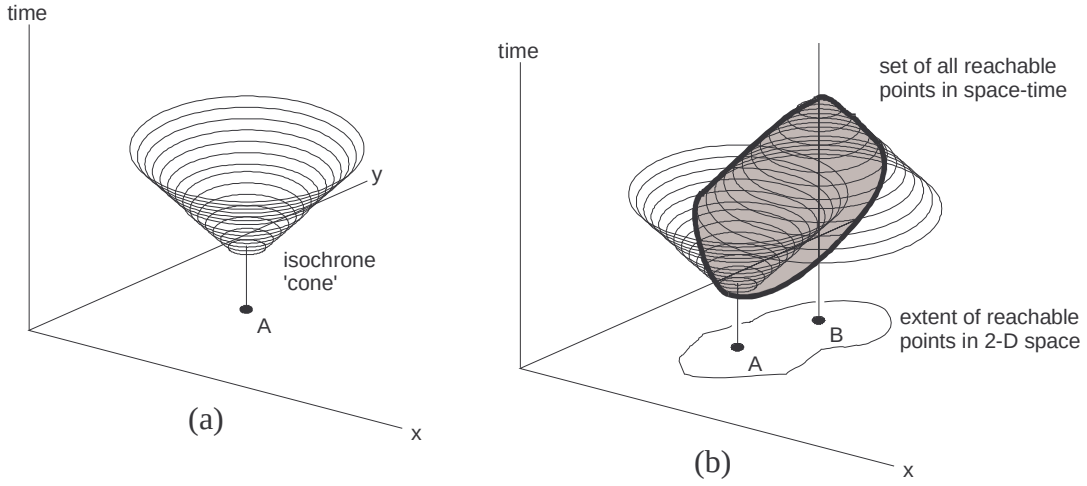


Figure 2.2 Three dimensional space-time prisms

Point (i) above is a valid and powerful criticism of isochrone techniques, and it is difficult to argue that they should replace more generalised mathematical or numerical measures, entirely. However, developments in GIS and visualization systems have lessened this problem, making it possible to compare complex spatial distributions. There is also a strong case to be made that isochrones provide "a more realistic measure of the way people perceive their opportunities: so many within such and such a journey time" (Doling 1975). The readiness with which an isochrone analysis of accessibility can be understood ought to make it a useful method for public consideration of transport options. A GIS platform is also the ideal setting for using isochrones since it allows re-use of the shapes generated in various analyses, including the constrained space-time approach. It is in this context, that an effort to produce isochrones in a GIS can be seen as useful.

2.5 Transport features in typical current GIS

Currently, standard GIS packages do not so much provide transport analysis tools as *network analysis* tools. This is because GIS is usually aimed at general users and transport applications are a relatively small niche market. Accessibility measures are not standard facilities in GIS.

Network analysis is based on well understood mathematical algorithms in planar graph theory. Dijkstra (1959) is usually cited as the primary source, although the brevity of that article indicates that much had gone before!. Texts in computer science present many of the relevant algorithms (see for example Dolan & Aldous 1993, Sedgewick 1988, or Ahuja *et al* 1993). The classic problem is to determine the shortest path between two nodes in a network. Consider the simple network presented in figure 2.3. The length of each link in the network is indicated by the

number printed alongside. Length may be expressed in any suitable units - usually of time or distance, although monetary cost is also possible. The problem is to find the shortest (least cost) path along links in the network between any two specified nodes, say X and Y.

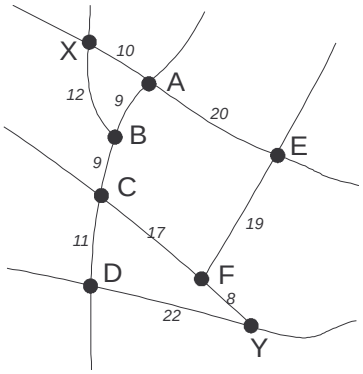


Figure 2.3 A simple shortest path problem

To solve this problem we consider all possible single link 'moves' from the start node X and assign the journey time along these links to the node at the end of that link (figure 2.4(a)). Now we find the node which has been reached by the shortest path, in this case A, and consider all the single link moves from here. Where the total journey length for a node is found to be shorter than any previously determined value for that node it is set to the shorter value. After a node has been considered in this way, the next node (in order of journey length) is considered. The shortest path from X to Y has been found when Y has the shortest remaining journey length. Figure 2.4 illustrates the steps in the process. In the diagram as each node has a journey length assigned this value is indicated on the circle. When a journey length has been finally determined a double circle is shown. Note how, in step (g), the value assigned to Y is reduced, since it can be reached via a shorter route from F than the previously assigned value from D in step (f).

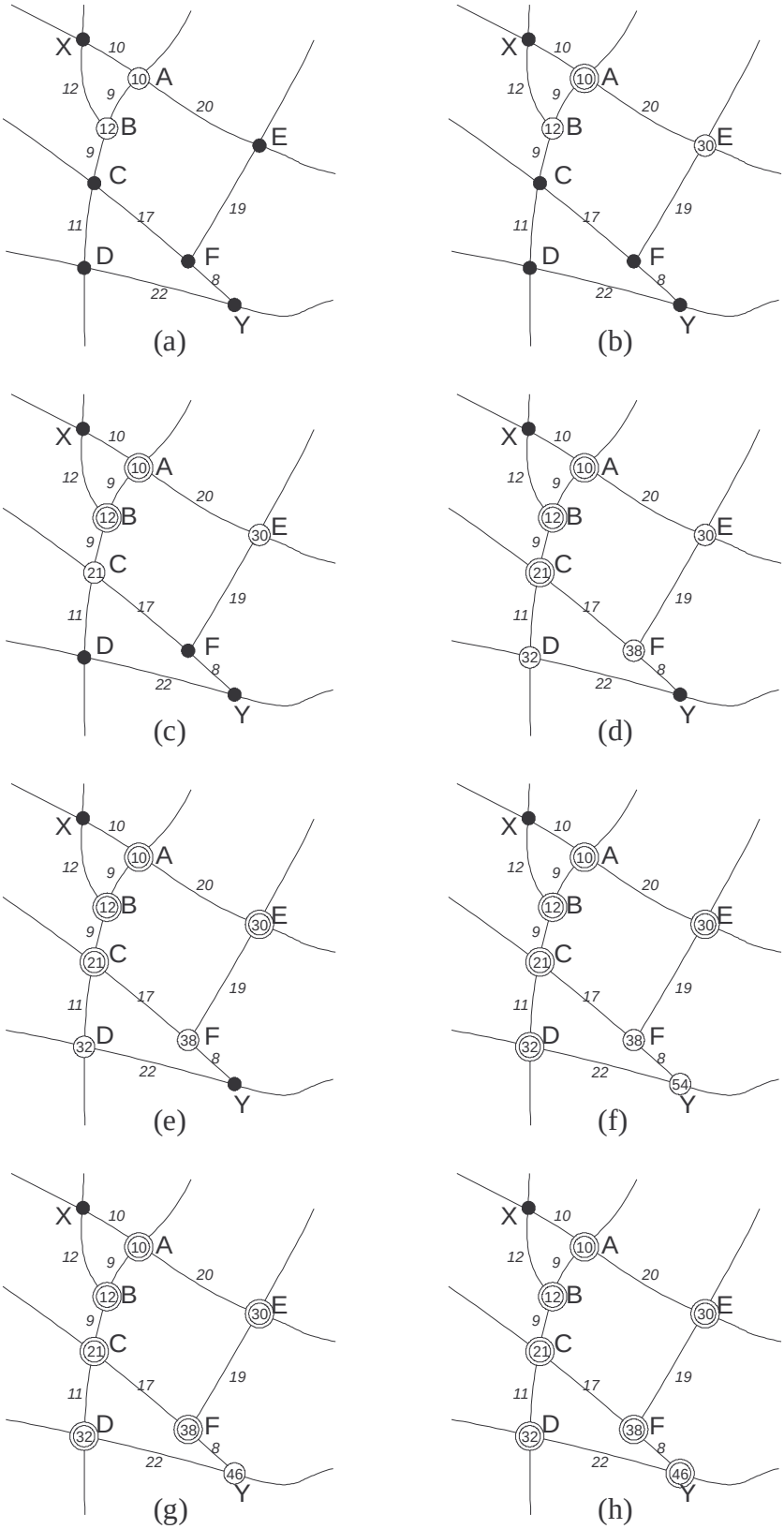


Figure 2.4 Steps in determining a shortest path

This example is trivially simple. Nevertheless, essentially the same algorithm is used as the basis for tackling a wide range of route determination problems. With large networks modifications are necessary to make operation more efficient, for example only considering links which are heading in approximately the right direction. Street networks in the real world can make matters appear more complex, but the problem remains the same - just bigger. We may be interested in travel times instead of link lengths, and travel speeds may be different for different links in the network. However, these modifications only involve measuring the 'length' of links in a different way.

The algorithm outlined is clearly immediately applicable to the analysis of private motor vehicle travel over road networks. Since it is so well understood and implemented in many GISs, this has the side-effect of making GIS systems particularly suited to analysis of transport applications involving motor vehicles and road networks. This bias towards road network analysis tools is reinforced by the fact that much of the extensive theory of transport planning has been aimed at solving transport problems by extending road networks (Hanson 1996, Taafe *et al* 1996, Tolley & Turton 1995). Indeed when writers talk about 'comprehensive transport models' they are often talking about models which, until recently, have been largely geared towards predicting, and catering for the demand for private road transport. Partly as a result when GIS systems provide more specifically *transport* analysis tools (as Caliper Corp's *TransCad* system does) these too are geared towards solving road transport problems. The aspects of the work described here which deal directly with journey times using timetabled public transport make no use of the network analysis tools in the GIS. Network analysis tools are only used for journey segments completed by walking over the street network.

A feature which *is* commonly available in GIS, and which *does* relate to the accessibility measures described, is the creation of a 'service network', or 'service area' around a selected location. For a network, the user specifies a point, and a time or distance. The GIS can then construct a 'sub-network' and a notional area around this sub-network which shows all the points on the network reachable within the specified time, from the specified point. This is a feature available in *ArcView 3.0* which has been used in constructing isochrones. More details are presented in section 4.9.

3. OVERVIEW OF APPROACH

3.1 Choice of *ArcView*

[More information about *ArcView* and its various extensions can be found in ESRI 1996a,b,c,d]. Choosing a GIS is a complex task in itself. *MapInfo, GIS+* and *PC-ARC/INFO* - A*rcView* were all considered for use in this project. *MapInfo*'s network analysis tools are very limited and it was eliminated from consideration at an early stage. *PC-ARC/INFO* was also eliminated since its interface is old-fashioned command line style and it is extremely frustrating to use. Either of Caliper Corp's *GIS+* or ESRI's *ArcView* could have been used. They have similar network analysis tools, powerful macro languages, and a wide range of spatial analysis tools. Eventually the object-based nature of *ArcView*'s *AVENUE* macro language, and its raster based spatial analyst extension led to the decision to use *ArcView*. As it turned out raster based analysis has not been used, and *GIS+* might have been a better choice. It provides for matrix data structures which would certainly be useful in optimising the scripts - allowing a compact record to be kept of which interconnections have been attempted. However, it seems certain that *AVENUE*'s object-based style has been important in allowing complex scripts to be written quickly.

In fact all of the GIS packages mentioned were used at some stage: *GIS+* has extremely useful features for converting data formats; *PC-ARC/INFO* is essential for getting Ordnance Survey data into a form useable by *ArcView*; and *MapInfo* has an easy-to-use module for generating proximity polygons which was also used.

3.2 Overview of Avenue

The following chapter examines the operation of a number of scripts in some detail. Full listings are presented in Appendix B. Before looking at these a few words about *AVENUE*, the language in which they are written, are in order.

AVENUE is object-based but not object-oriented. It allows a programmer (scriptwriter?!) to instantiate *ArcView* objects such as tables, shapes, lists and so on, which are then manipulated to carry out the desired functions. So, if an *ArcView* project contains a theme or layer called 'Hospitals', then the command

```
myTheme = theView.FindTheme( "Hospitals" )
```

will create a new object called `myTheme` which is a theme object containing all the data in the Hospitals theme, and which can be manipulated using requests recognised by that object class. Requests are preceded by the dot operator common in object oriented programming. So

```
myTable = myTheme.GetFTab
```

gets the feature table associated with the theme. `myTable` can be manipulated using any request recognised by feature table (FTab) objects. For example the lines

```
for each rec in myTable
    myTable.SetValue( fldID, rec, 0 )
end
```

run a loop which goes through each record in the table, and sets the value of the field referenced by object `fldID` to 0.

AVENUE is easy to learn, and once the objects available are understood it is fairly natural. However *AVENUE* is not properly object-oriented, because the programmer cannot create custom objects. Indeed the language is not even properly structured, since there is no provision for functions as in *Pascal* or *C*. Instead a script may call

another script (not itself - so there is no recursion). This process is a bit ungainly, requiring the programmer to use a `self` object to decode the parameters passed to the script. This discourages the development of elegantly structured programs, which is unfortunate. With more time it would probably be better to write programs in a language such as *C*, *C++*, or *Java*, and integrate these into *ArcView*. However, this work is not an exercise in computer science, and the intricacies of 'dynamic link libraries' seemed to belong to that discipline, more than to GIS and spatial processing.

Having said all that, the scripts do work, and *AVENUE* has proved its worth (to this scriptwriter at least). Particular commands and object classes are explained as required in the more detailed explanations of the scripts in the next chapter.

3.3 Overview of the glas1_0 project

All of the scripts described in the following chapter are part of an *ArcView* project called `glas1_0`. The project is stored in a file called `glas1_0.apr`. A project in *ArcView* consists of various documents - *views*, *tables* and *scripts* in this case.

A *view* is a window on the data in the project and appears as a map to the user. A view consists of layers, referred to as *themes*. Each theme contains different data, and is stored in a number of files - in particular the geospatial data is in a 'shapefile', suffix `.shp`, and the attribute data is in a database table, suffix `.dbf`.

`glas1_0` has one view containing themes for various features in the Glasgow region as set out in Table 3.1 (sources, full descriptions and the exact extent of the layers are set out in appendix A).

Table 3.1 The themes in the glas1_0 project view

Theme name	Filenames	Contents	Comments
area*.shp	`area*.*`	User created themes indicating region of interest for a series of analyses.	Intersected with stats2.shp to produce stns*.shp. Also used to clip final isochrones.
Bus Routes	`buses.*`	Line features each representing a bus route	Used extensively by scripts. Attribute table includes route number, duration, and frequency.
Centre.shp	`centre.*`	A rectangle representing the central zone of the city.	Used to ensure that the vital central stations are always selected in analyses.
Iso*.shp	`iso*.*`	Shape of isochrone output from scripts.	Each theme represents one time limit for one starting point.
Motorways	`mwaysnth.*`	Line data representing motorway sections of road network	Data is locational only - not used by scripts
north.shp, south.shp	`north.*`, `south.*`	Merged proximity polygons of vorsth.shp north and south of the Clyde. Each is a single area incorporating its side of the Clyde and adjacent land reachable on foot.	Used to mask isochrones so that travellers do not 'walk on water'
Railway lines	`railnth.*`	Line data representing approximate overground rail routes	Data is locational only - not used by scripts
River Clyde	`clyde.*`	Area feature representing impassable length of the River Clyde	The Clyde is used to mask isochrones created so that travellers cannot 'walk on water'
Stations	`stats2.*`	Point locations of all over- and under-ground stations	Attribute table also contains names and a station ID number
stns*.shp	`stns*.*`	User created themes containing subsets of stats2.*	Used to store the calculated arrival times for each station. Stns* may grow by addition of 'bus stops' - points determined by the scripts as points where a bus can be boarded to advantage.
vorsth.shp	`vorsth.*`	Proximity (Voronoi) polygons for the point features in stats2.shp (not including the underground stations)	Used to find appropriate stations in the neighbourhood of current location of traveller
walksnth.shp	`walksnth.*`	Line features representing all the A, B and minor road segments	Used extensively by scripts for walking portions of journeys, and also to locate bus routes.

There is one table in the project which is not associated with a theme: `table1.txt`. This table contains the weekday off-peak timetable data for the region's railway network. The bus frequency and route duration information contained in `buses.dbf` is also for the off-peak timetable. This restriction on the applicability of this work is

intended to limit the quantities of data involved. There would be no problem, in principle, with extending the system to include data for the whole day - or week.

3.4 Overview of scripts

3.4.1 Approach to programming

Rather than attempt to write one script which guides a user through a complex series of steps to produce a set of isochrones it was decided at an early stage to produce a set of simpler scripts and allow the user to organise the resulting outputs in whatever way seems appropriate. This has made the scripts themselves manageable, and allows them to be more easily understood than would have been the case with a single script. This approach also fits into *ArcView*'s customisation scheme somewhat better than a single large script would.

Given the time available for this work and the complexities involved, a very pragmatic approach to programming was adopted. Two factors have been prime concerns: first, getting something working, before worrying too much about refinement; second minimising the data requirements. It will become evident in chapter 4 that there is considerable scope for optimisation of the scripts, and also for improving or extending the coverage of the area by increasing the data in the GIS.

3.4.2 Outline of method of isochrone construction

There are three major steps in creating an isochrone using these scripts:

1. The user specifies an area of interest and starting point for subsequent analyses. These functions are performed by the SpecifyArea and SpecifyStPt scripts which create `area*` and `stns*` themes.

2. A set of ‘arrival times’ is generated for the railway stations and points along bus routes in a specified area of interest. This operation makes use of the timetable and bus route data in the GIS to determine how long it would take a person to arrive at the stations and at points along bus routes in the area of interest. The arrival times determined are stored in the `stns*` theme.
3. Finally the arrival times data from the previous step are used as a set of starting points from which a person could walk to any final destination. A set of areas each originating from a station or points along a bus route is built up. Each area is constrained by the arrival time associated with its origin and the time limit for the isochrone which has been requested. The small areas so created are merged into one larger shape which is the final isochrone outline required.

The scripts and their various functions are summarised in Table 3.2. overleaf. This table should make the more detailed descriptions in the next chapter easier to follow. The user guide in Appendix C may also be helpful in understanding the mechanics of this process.

Table 3.2 *ArcView* scripts written for isochrone construction

Script name	Summary of functions
SpecifyArea	Allows user to specify a rectangular area of interest for subsequent steps in analysis. Script creates new themes `stns*` and `area*` containing stations in the specified area and the outline of the area respectively. These are the input to subsequent steps.
SpecifyStPt	Asks user to specify the `area*` and `stns*` theme and accepts a location (specified by mouse click) which is start point for further analyses. The specified start point is added to the `stns*` theme specified.
GenerateArrivalTimes (GAT)	The main loop for the calculation of traveller arrival times at the stations in the specified `stns*` theme. During execution suitable points for boarding bus routes are added to `stns*` and their arrival times are also determined. Much of the code creates various 'dictionaries' - data structures which can be more efficiently accessed than the tables which provide persistent data storage. The main loop calls the scripts below to complete its task.
CloseStations	Called by GAT. Finds stations near to the current station and determines the time taken to walk to them, updating the arrival time for them if it is an improvement on previous value.
CloseBuses	Called by GAT. Finds bus routes near to the current station and determines the time taken to walk to them. A bus waiting time is added, and `AddBoardPtToBusRoute` is run to determine if this *bus boarding point* should be added to the `stns*` theme.
GetEarliestArrival	Called by GAT. Finds the station or bus boarding point in `stns*` which has the next earliest arrival time. This becomes the current location.
TryTrainsFromThisStation	Called by GAT. For the current station tests whether connections by train provide earlier arrival times at any stations and updates `stns*` accordingly.
TryBusesFromHere	Called by GAT. For the current bus boarding point tests whether other bus routes or stations can be reached and if so determines a suitable alighting point for this route. The potential arrival time at the 'target' station or bus route is determined and if it is sooner than currently stored value `stns*` is updated.
AddBoardPtToBusRoute	Determines whether candidate boarding points for a specified bus route provide any advantage over any currently stored for that route. Updates `stns*` theme accordingly.
IsochroneMainLoop (IML)	Requests user to specify `stns*` and `area*` themes from which isochrones are to be constructed. Creates a new set of dictionary data structures, which are then used by the specified isochrone building script to create a new theme containing the outline of an isochrone for a specified time limit.
Construct	Called by IML. Builds isochrone using *ArcView* service area tools.
QuickBuses	Called by IML. Builds isochrone using *ArcView* service area tools for stations, and circles for bus routes.
QuickBoth	Called by IML. Build isochrones using circles.

4. DETAILED DESCRIPTION AND CRITICAL REVIEW OF SCRIPTS

This chapter includes a full description of the scripts written for this work. Full listings with line numbers (not a feature of *AVENUE* - these are for reference) are included in Appendix B. The order of presentation, here and in the appendix, is as a user would use the scripts in producing an isochrone map, as outlined above and in Appendix C. This delays the examination of the more complex scripts until section 4.3, by which stage a reasonable idea of the workings of *AVENUE* should have been formed. More details of *AVENUE* are to be found in ESRI (1996d).

4.1 SpecifyArea and SpecifyStPt

These scripts allow the user to specify, respectively, a rectangular area of interest in the Glasgow area, and a starting location for subsequent determination of isochrones.

Figure 4.1 shows flow charts for `SpecifyArea` and `SpecifyStPt`. These are simple scripts - the decision boxes represent validity checking only. Note that here, as elsewhere the flow charts are deliberately simplified to save space.

The most notable aspect of these scripts is that they are a fairly crude attempt to reduce the work load in subsequent processing of isochrones. Calculations for the whole region are slow and a user interested in routes into central Glasgow from one outlying area will specify a rectangular area enclosing that area and the city centre only. It would be easy to alter the specification of the required area to accept a circle or polygon, rather than a rectangle - the `ReturnUserRect` request in line 11 of `SpecifyArea` need only be changed accordingly.

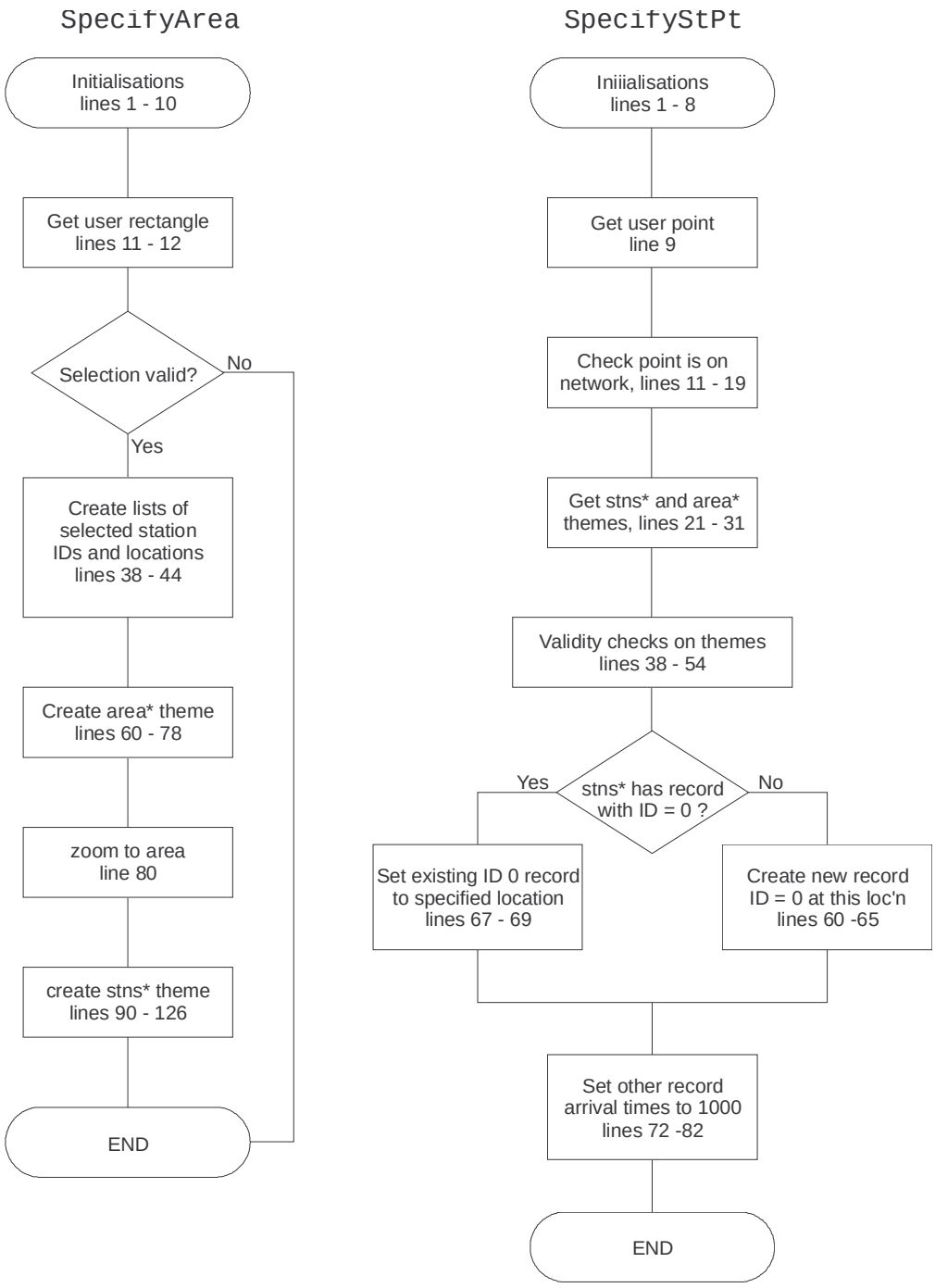


Figure 4.1 Flow charts of `SpecifyArea` and `SpecifyStPt` scripts

Of course, this sort of approach is not entirely satisfactory, since it is easy for a user to specify an area which does not include vital elements of the transport network from the specified starting point. Again, a crude solution to this (which would have to be reworked for a more polished implementation) has been adopted - insisting that

the selected area include an arbitrarily defined central zone. A better approach would be to define the useable stations and the area of interest separately - in fact this whole aspect of the interface requires some careful thought to work out all the permutations for analyses which users might wish to make. The pragmatic approach of producing *something* which works has been adopted. Were this project a 'real-world' application, users could then comment, and modifications could be made. This approach is common in developing and prototyping large systems.

Another aspect of these scripts which deserves comment is the way in which the user specified start point is added into the new `stns*` theme created by the `SpecifyStPt` script, as a 'station' with ID = 0. A similar mechanism is used when bus routes are introduced later, where points at which bus routes may be boarded are added into `stns*` as 'stations' with IDs in the 1000s. `stns*` is not really a stations theme at all. Rather it is collection of point locations for which the traveller's arrival time is known and which are used for isochrone construction. `stns*` also functions as persistent storage for these locations and times, between steps in the calculations.

4.2 GenerateArrivalTimes

`GenerateArrivalTimes` determines the times at which a traveller can arrive at various points in the area of interest. These arrival times are used by the `MainIsochroneLoop` script to construct the isochrone shapes which are the final output. `GenerateArrivalTimes` is a long script (almost 400 lines) but not particularly complicated - its flow chart is shown in figure 4.2. Much of the script is taken up with first creating various *dictionaries*, and at the end writing the data in one of these - the 'stations' dictionary - back into the `stns*` theme. A *dictionary* in *AVENUE* is an object which can store other objects which can be

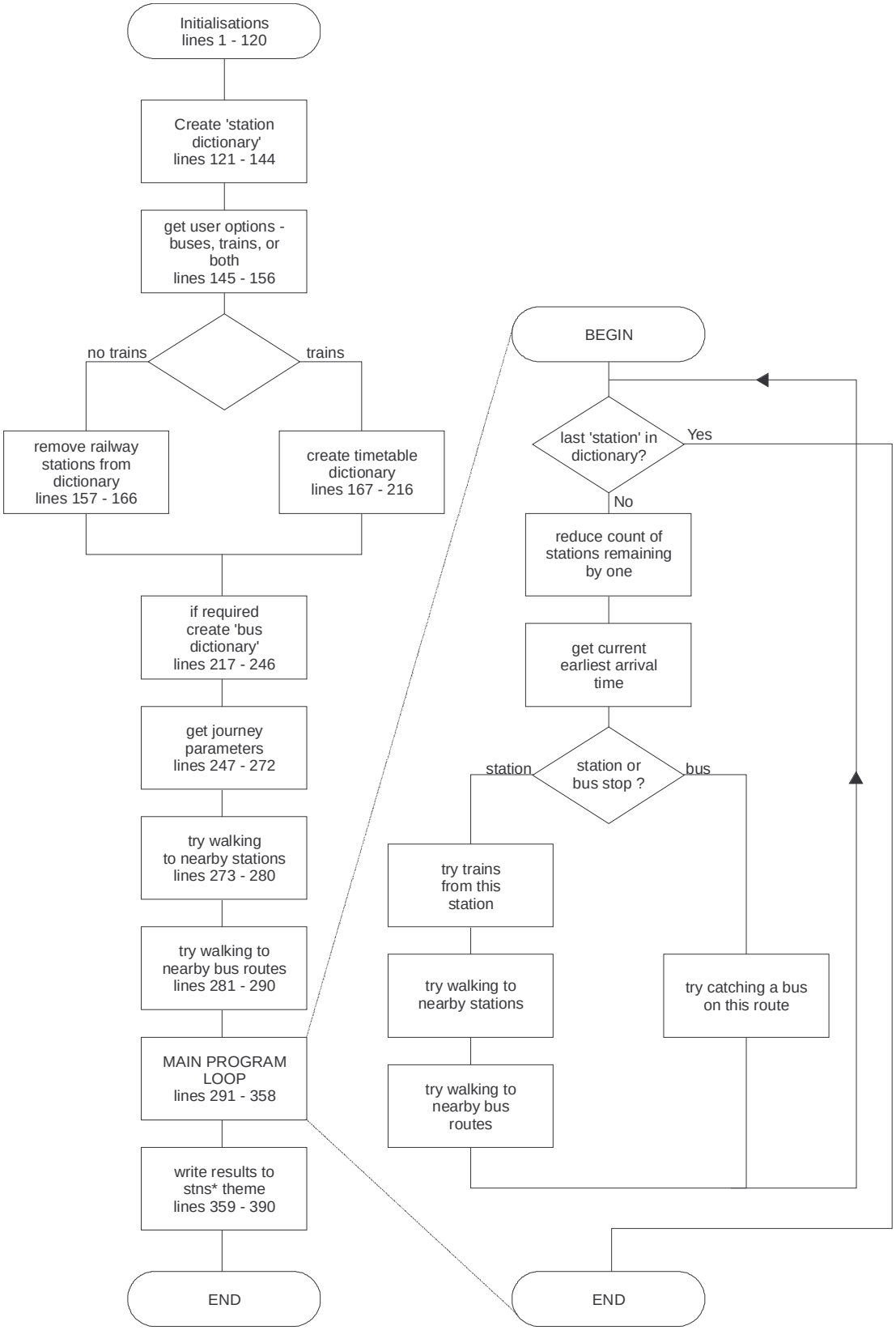


Figure 4.2 Flow chart of `GenerateArrivalTimes` script

accessed by a *key*, which may itself be any kind of object. Dictionaries can be used very much like multi-dimensional arrays to store large amounts of inter-related data. The contents of each of the station, timetable, and bus dictionaries created in the first part of `GenerateArrivalTimes` are explained in the comments in the program listings. The reason for creating these dictionaries is that they can be more rapidly manipulated and accessed than the database table data which is the form in which the data is stored.

The approach behind determining arrival times at stations and at points along bus routes in the area is essentially the same as the algorithm outlined in section 2.5 for determining shortest paths. The only difference is that scripts are required for determining the journey times at each stage by different transport modes. These scripts are called as required in the main loop of `GenerateArrivalTimes`. As this process proceeds the variable `unusedStns` keeps track of how many points in the station dictionary remain to have their arrival times determined. Once this number falls to one the script is finished, and the main loop terminates. Finally, arrival times stored in the station dictionary are copied back to the `stns*` theme, for later use in isochrone construction.

Lines 247 - 272 require the user to enter some journey parameters. These are all limits to the distance over which travellers will consider making connections on foot to, and between, transport services. This allows potentially important distinctions to be made between different travellers - since older or less mobile travellers are less likely to put up with long walk connections between services.

4.3 CloseStations and CloseBuses

These scripts are similar, and called from `GenerateArrivalTimes` when the current location is either the starting point or a rail station and the next stage of a traveller's journey will be to walk to another station, or to a bus route. If the current location has been reached on foot, then these scripts are skipped, since otherwise walking connections might be chained together and the whole region visited on foot! Flow charts are in figure 4.3. The station to station connection is particularly important in the context of Glasgow because this is the simplest connection between Queen Street and Central Station which are the most important nodes in the network. A particularly important aspect here is that both stations have a 'surface' and a 'low-level' station and connections must be made from one to the other by walking.

These scripts use quite different strategies for searching for nearby stations or bus routes which might be relevant. `CloseStations` makes use of the proximity polygons of the `vorsth.shp` theme. This theme was produced by modifying a simple set of proximity polygons based on the overground stations in the region. A major modification was removal of the outline of the River Clyde from all polygons which it intersected. Having achieved this, parts of polygons 'stranded' on the wrong side of the river with no station, were merged into neighbouring polygons on the same side of the river. The resulting polygons provide a reasonable way of predicting sensible stations for a traveller to go to from their current location. As it is implemented in `CloseStations`, a traveller will consider using any station in 'her own' polygon,

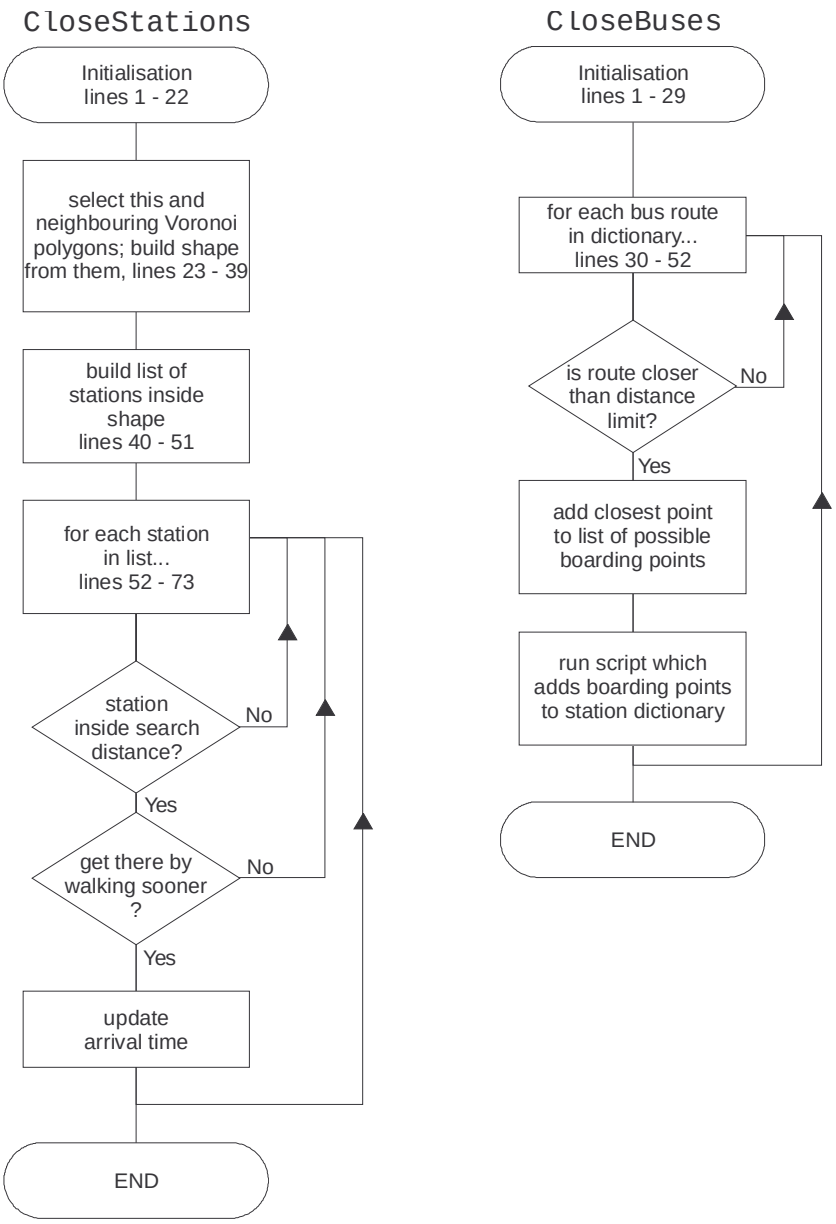


Figure 4.3 Flow charts of `CloseStations` and `CloseBuses`

and also any station in neighbouring polygons. This results in reasonably intelligent choices in most cases without too much effort. Figure 4.4(i) shows how this works - the search area is shaded. The traveller location is indicated by a star. A better approach might involve creating the proximity polygons using the traveller's location and the stations. Then nearby stations are found by selecting those in neighbouring proximity polygons to that of the traveller's location. The search area by this method is shown in figure 4.4(ii). This approach is likely to lead to fewer anomalies than the

current one, but would involve altering the proximity polygons during execution which would be slow. In some cases both approaches will produce the same result (although not in figure 4.4). The current method seems a reasonable compromise.

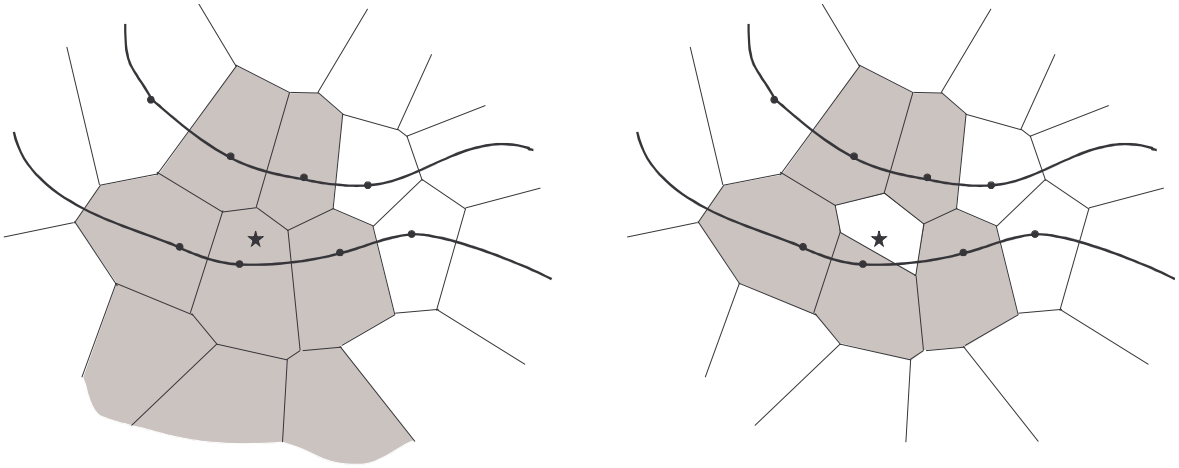


Figure 4.4 Searching for nearby stations by two possible methods

A simpler technique is used in `CloseBuses`, largely because the 'neighbourhood' approach is not obviously applicable. Were all bus stop locations known, a similar technique might work - although there would clearly be many more polygons. Since bus stop locations are not known - in this system - a simple distance criterion is used. Line 35 of `CloseBuses` determines the closest approach of the bus route to the current position using the `Distance` request. The boarding point for the traveller is then taken to be approximately the point on the bus route at which the closest approach occurs. This is intuitively sensible, although all sorts of special cases may be envisaged. Figure 4.5(i) when the 'target' bus route is fairly direct is a case where this approach is sensible - if the bus will take a traveller where she wants to go then going straight to a nearby stop is sensible. Figure 4.5(ii) is an example of where a poor decision is made by this method: either of the unfilled 'flags' could be a better point to board the route, depending on where the final destination is. However,

programming this sort of sophistication into the system would be a major task in itself. Perhaps a more serious flaw in this approach is that whilst nearest points are found 'as the crow flies' the walk to the point is over the street network. This may result in anomalies especially near significant obstacles like railway lines and motorways. The only protection against this problem is that if the journey turns out very long it will not result in an improved arrival time and will be ignored.

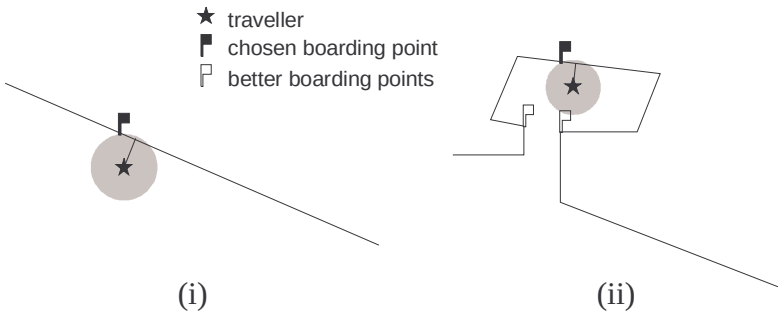


Figure 4.5 Two examples of choosing a bus route boarding point

The boarding point calculated is only approximate, because the `PointIntersection` request of line 40 returns a `Nil` value if a circle of slightly larger radius than the closest approach distance is not used. Once this point is determined, arrival of a bus at this point is estimated using the walk time over the street network (for the traveller to get there) and half the frequency of the bus route (waiting for a bus to appear). This assumption about waiting times for buses is another simplification, which is used wherever buses are involved. It is difficult to come up with any other method which is as simple, and which also gives repeatable and reasonably realistic results. A more detailed discussion of the pros and cons of this assumption can be found in section 5.3.

Having identified a suitable point for boarding any nearby bus routes, `CloseBuses` calls `AddBoardPtToBusRoute`, which determines whether newly identified points on a bus route result in a traveller arriving earlier anywhere along the route. This is explained in detail in section 4.7

4.4 GetEarliestArrival

This script is unremarkable and no flow chart is provided. Its only function is to find the 'station' (that is, station or bus route boarding point) which currently has the earliest arrival time. The functioning of the script should be obvious.

4.5 TryTrainsFromThisStation

This script accesses the timetable dictionary built by `GenerateArrivalTimes` to determine which stations in the rail network can be reached by direct trains from the current station. A flow chart is shown in figure 4.6. Much of the apparent complexity of this script is due to the format in which the timetable data has been stored. Only the off-peak timetable is used, and all train arrival times are expressed in minutes past the hour. This significantly reduces storage requirements, and - more to the point - initial data capture requirements, since all data was entered into a spreadsheet by hand, an error-prone and tedious process. However, as only the minutes part of any train's arrival time is known this must be adjusted to reflect both the time at which the traveller has arrived at the current station (`hourOffset` in lines 38 - 40) and whether the sequence of stations under consideration 'straddles the hour' - a sequence like 53, 57, 59, 02, for example (`clockOffset` in lines 44 - 46). Apart from that, the script simply determines at what times a traveller could get to other stations from the current one, and determines if these times are sooner than any time currently stored for those stations.

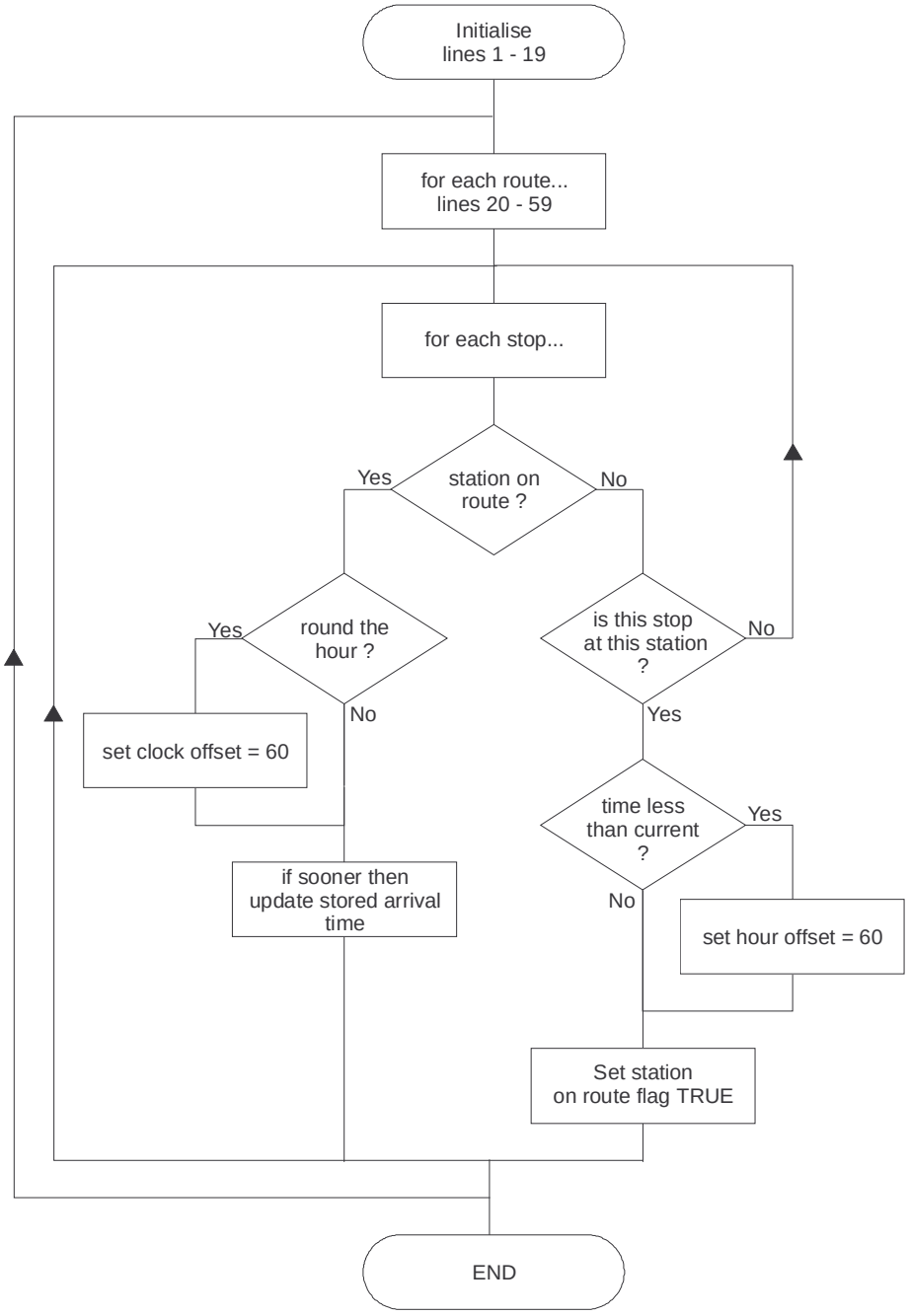


Figure 4.6 Flow chart for `TryTrainsFromThisStation`

This script has not been optimised, and one obvious improvement would be only to consider journeys in any available direction to the next station along. This would avoid unnecessary duplication of effort when subsequent stations are considered.

This modification would be easy to make, and would be high on any list of performance improvements.

A more interesting theoretical problem is raised by the use of the timetable. This makes the choice of starting time (entered by the user in response to lines 89 - 109 of `GenerateArrivalTimes`) critical to the end result. Fair comparisons between locations can only be made by running the script a number of times using different starting times and either aggregating the results, or choosing an appropriate start time by some means. It might be argued that a better approach would be to use a 'frequency' measure as has been adopted for the bus routes, and estimate a waiting time for trains in any particular direction. This approach does not seem entirely satisfactory however, since it does not reflect differences in the way travellers use these two modes. In the author's own experience, train travellers tend to know when the service they need runs, and to set off in time to catch it. Buses, on the other hand, unless they are express services, or very infrequent, are perceived as unpredictable, essentially random in their running, and users tend to 'just turn up'.

4.6 TryBusesFromHere

It is in handling bus routes that the most difficulties were encountered. The intricacies of this script and `AddBoardPtToBusRoute` are the result. `TryBusesFromHere` is an attempt at the same function for buses that `TryTrainsFromThisStation`, `CloseStations`, *and* `CloseBuses` carry out for the train services. `TryBusesFromHere` includes calculation of the walking segments of journeys using the bus services. This helps to account for its complexity. A flow chart is presented in figure 4.7. It can be seen from this that the script is really two

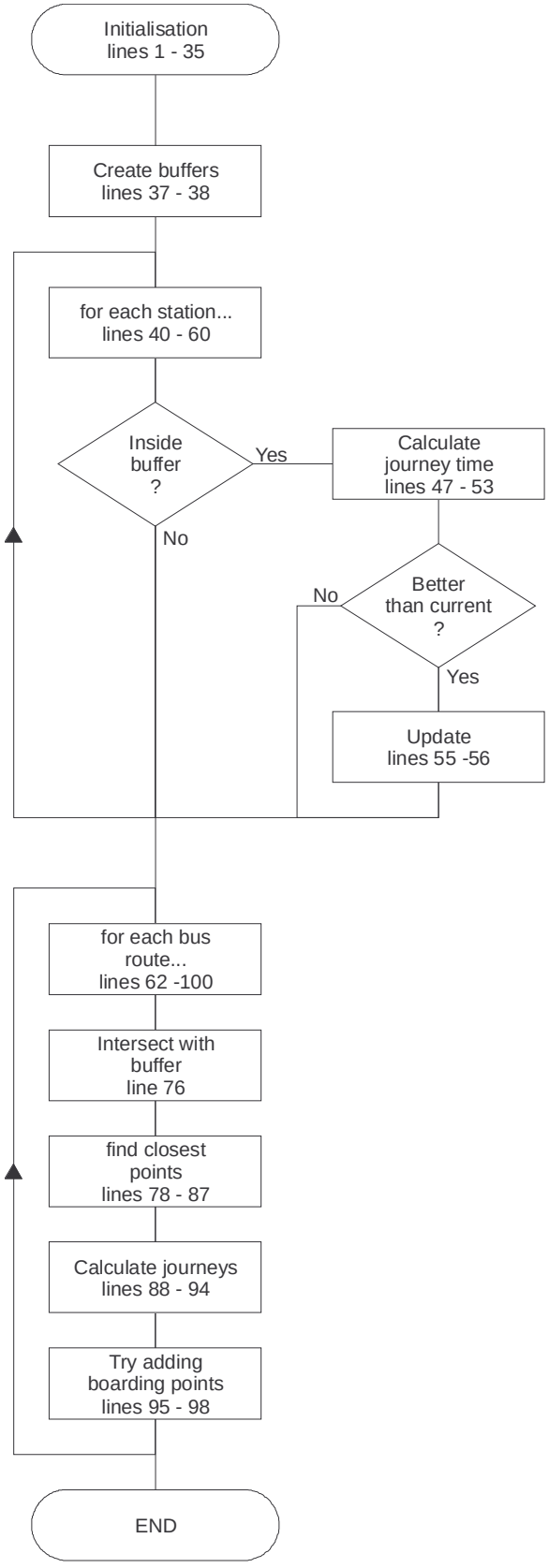


Figure 4.7 Flow chart for `TryBusesFromHere`

scripts - one dealing with connections to railway stations, the other with connections to other bus routes. If any of the improvements discussed in this section were to be introduced, it would make sense to split the script into two, to avoid creating one very large, unwieldy and hard to follow script.

Connections to either railway stations or bus routes both make use of a buffer zone constructed around the current bus route. The extent of the buffer is according to the numbers entered by the user in response to lines 247 - 267 of `GenerateArrivalTimes`. The size of the buffers may thus be different between stations and bus routes. This makes sense since it seems likely that travellers will tolerate longer walks to connect with the rail network than they would to connect to other bus routes - given that they can expect to make faster progress via the rail network in return for the extra 'investment' of time in getting to it.

Connection to railway stations is achieved simply by considering any stations which fall inside the buffer zone. Figure 4.8 illustrates the approach. The closest point on the bus route is used as an alighting point and the walk from the alighting point to the station is determined using *ArcView*'s shortest path network analysis tools. Criticisms of this sort of approach in `CloseBuses` outlined in section 4.3, and illustrated in figure 4.5, are equally valid here. The duration of the bus journey to the alighting point is determined by calculating what fraction of the whole bus route it represents, and assuming that the bus route is traversed at constant speed. This is obviously a great simplification, and a more sophisticated approach would take account of the likely variations, especially between intra-urban and inter-urban speeds. Neither *PC-ARC/INFO* nor *ArcView* provide any obvious method of doing this, although speed zone areas could easily be created, and calculations done during

execution. During consideration of *GIS+* in preparation for this work, its route facilities, which allow for storage of mileposts with linear features, looked like a promising tool for this sort of problem, and operator's timetable information could probably be used. Such an approach would require considerably more data about each bus route to be stored. For the current work, the improvement in accuracy which this would yield did not seem to justify the extra effort involved.

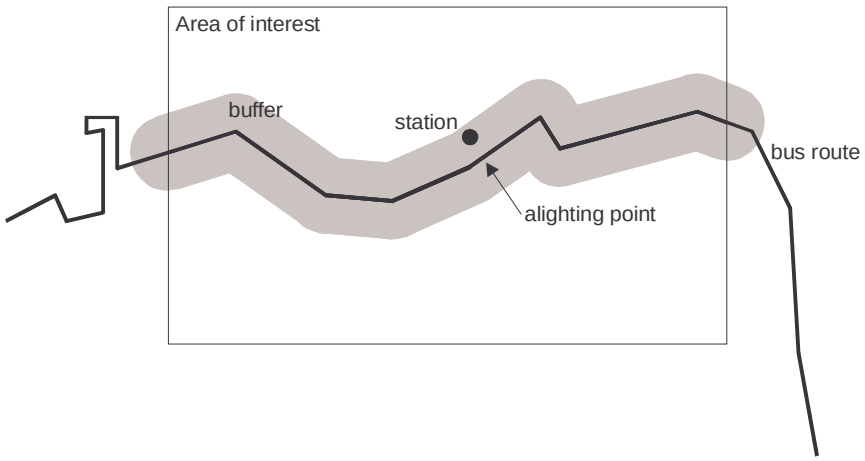


Figure 4.8 Connecting from bus to a railway station

Connections between bus routes are handled in a similar way. Figure 4.9 illustrates the method. Segments of 'target' bus routes which intersect the buffer zone around the current route are traversed until the closest point is found. This is a candidate boarding point for the target route, and the journey time to this point is calculated in a similar way to the journey in the station connection case. Candidate boarding points are retained if they improve on existing ones when `AddBoardPtToBusRoute` is run. Initial problems with this algorithm have caused great simplifications to be made. One of these is probably a 'simplification too far'. Simple intersections such as *target route 1* in figure 4.9 present no problem. However, where the current and target routes run parallel over significant lengths (*target route 2*) the script *does not*

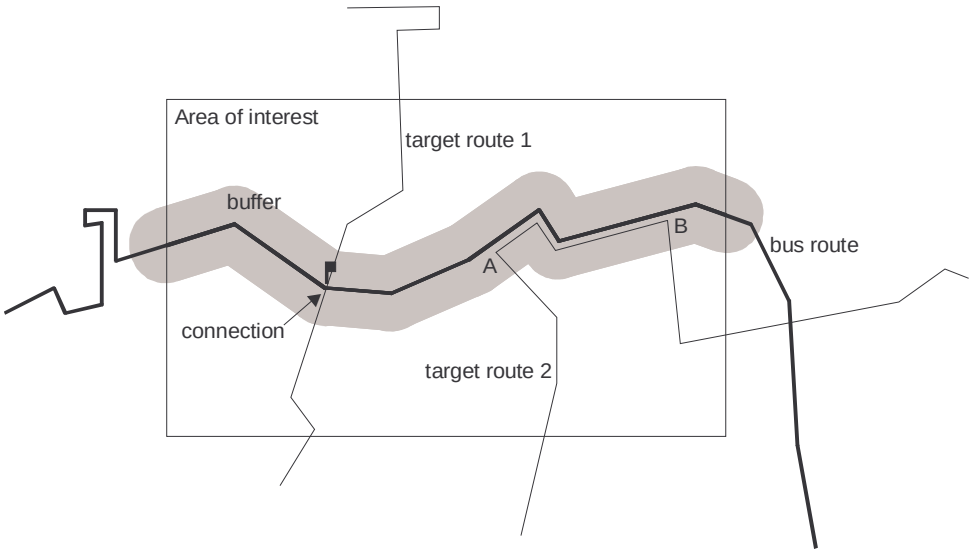


Figure 4.9 Connecting from one bus route to others

attempt to determine which of the two end points *A* or *B* would be a better point at which the traveller should change buses. A little thought soon reveals that whichever of the two is closer to the current location will *usually* be the preferred choice. *Usually* is the operative word, however, and there are many special cases, for example, where a target route diverges from the current route to meander around a residential district before rejoining the current route. In figure 4.10 it is not clear, and will depend on the current location which of *A*, *B*, *C* or *D* is a suitable point at which a traveller would switch between these bus routes.

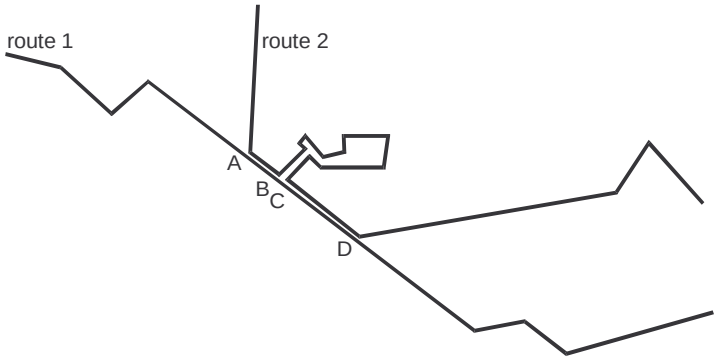


Figure 4.10 Complications in bus route connections

The general problem of determining suitable paths through a complex of public transport routes is difficult. Many strategies could be adopted. Constructing a network to represent all the possible interchanges, using timetable data, is one option, but this was rejected from the outset as requiring too much data. An obvious improvement over the method presented here would involve simply keeping track of which connections had already been attempted using an 'interconnection matrix'. This sort of change would not be difficult to implement (although there are complications arising out of the possibility of boarding bus routes at more than one point), and would produce significant performance improvements by avoiding needless rechecking for connections.

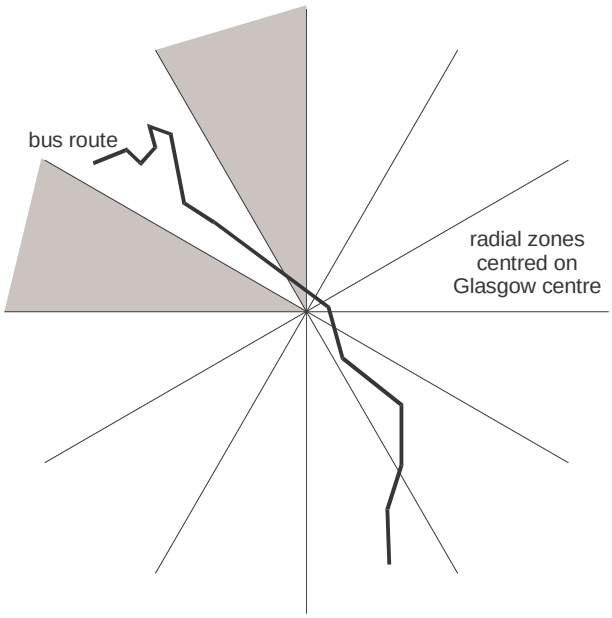


Figure 4.11 A possibility for improving the bus interconnection algorithm

Other approaches would involve more complex 'spatial reasoning' tasks - the sort of decisions which come naturally to humans deciding routes but which are difficult to program. An obvious example is where two routes run along the same streets to approximately the same destination. No traveller would be likely to change from the

number 6 bus to the number 6A, if they both go to the city centre. Building this sort of decision making into this system would be a large task. Ideas like creating radial sectors or zones around the city centre and only considering connections to routes exiting the current zone (see figure 4.11) might work. However, this sort of approach becomes very specific to the exact location or region, and is not entirely satisfactory. The techniques required are by and large still beyond the scope of GIS. Frank(1991) and Luo & Jones(1995) are just two examples from a large body of literature discussing some of the possibilities for future enhancements to GIS tools which could help with this sort of problem.

4.7 AddBoardPtToBusRoute

This script determines which of a list of candidate boarding points on a bus route are an improvement on boarding points already determined, that is which will allow a traveller to reach at least some destinations sooner. A flow chart is shown in figure 4.12.

There are three possibilities when a new boarding point (at a known time) is under consideration, in light of any previously determined boarding points - clearly, the first candidate boarding point is always added. Consider the diagram in figure 4.13. A is a previously determined boarding point, B is a new boarding point. Each has an associated time, t_A and t_B. The journey time from A to B is T. Now, if

$$t_A + T < t_B,$$

then clearly boarding point B is redundant, since it provides no improvement in access to points further along the route.

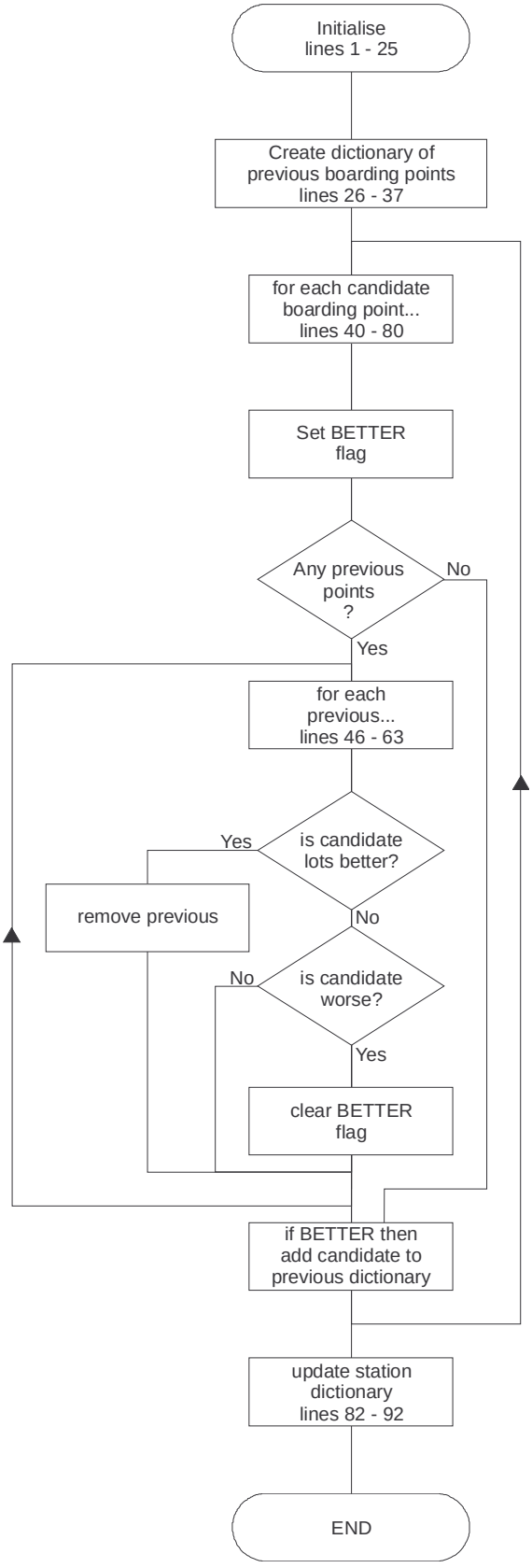


Figure 4.12 Flow chart for `AddBoardPtToBusRoute`

The mirror-image case, where

$$t_B + T < t_A,$$

means that B supersedes A, which is now redundant. The intermediate case where

$$T < |t_B - t_A|$$

means that both boarding points should be retained, since one provides earlier access to its segment of the route, while the other provides better access to its segment.

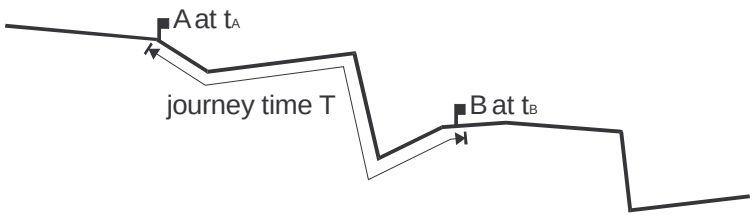


Figure 4.13 Comparing boarding points A and B

This is the logic which is implemented in the body of `AddBoardPtToBusRoute`. A possible refinement would involve discarding cases where T was below some threshold. This would not be difficult to implement. Note that this script uses the same method to determine bus journey times as has been used elsewhere, so that the same reservations and comments apply.

4.8 IsochroneMainLoop

`IsochroneMainLoop`, as its name is intended to suggest, simply runs the script which builds user specified isochrones using a set of previously determined arrival times. Much of the script is taken up with initialisations and user dialogues and will not be considered in any detail. One point of interest is that it proved impossible to make the *Avenue* request `msgBox.MultiInputAsList` work properly. This would allow the user to select a set of isochrones from options presented, and the script would then run several times around the subsequent loop. This has left operation of

the isochrone construction a little tedious since generating a set of isochrones from the same `stns*` data requires the user to run the script several times, each time answering the same sequence of questions. This problem may be simply due to a misunderstanding of the workings of this particular *AVENUE* request.

4.9 Construct, QuickBuses and QuickBoth

These scripts do the hard work of isochrone construction based on a set of points (stations, the starting point, and bus boarding points) each of which has an arrival time. A flow chart fragment is shown in figure 4.14.

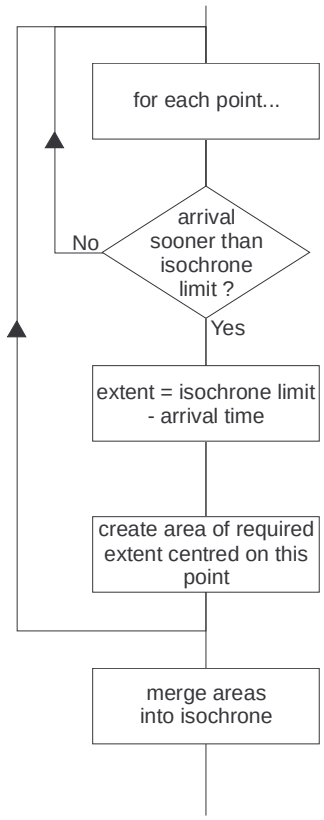


Figure 4.14 Flow chart fragment for the isochrone construction scripts

The method is really the same for all of these scripts, only the command used to construct each individual area in the isochrone changes. The 'quick' versions

simplify matters (and speed things up considerably) by assuming that the walking portion of journeys can be represented by 'steady progress in all directions'. In other words isochrones are constructed from circles of various radii. This assumption is not unreasonable in built up areas with a fine street grid. Clearly it falls down outside built up areas, and when dealing with major natural barriers such as the River Clyde. The Clyde has been handled by creating two shapes `north.shp` and `south.shp` which represent (overlapping) regions north and south of the river. The overlap is in areas where the Clyde does not represent a significant barrier to movement. Whichever of these themes apply is intersected with isochrones as they are constructed to remove inaccessible areas. This technique could be extended to allow for progress at different speeds across different classes of ground, but it is debatable how realistic such an approach would be, and also whether it could be made to run efficiently.

The slower approach using *ArcView*'s `FindServiceArea` request arguably produces better results, based as it is on the street network. The street network has been set up with a constant walking speed of 100 metres per minute assumed, and no allowance made for junctions and other complications. Much more work could be done refining details of the street network, introducing different speeds and turn penalties at complex junctions. However, all these effects are second order, and improvements are likely to be marginal, considering how much detailed editing and refinement of the network would be required. A more serious improvement would be to write a service area script from scratch, although this is not a trivial task. However, it would provide an opportunity of avoiding one of the more problematic aspects of the *ArcView* standard request. This always constructs a service area *and* a service network, *and* writes them to the hard drive of the PC. This is probably the

main reason why constructing isochrones by this method runs so slowly. Of course it is quite correct for *ArcView* to construct a service network, since that is actually what is being produced by this request. Figure 4.15 shows the distinction between the service network and the service area. The service network is represented by the white lines, and the service area by the grey shading. How *ArcView* determines when to join points around the perimeter of a service network to create the service area is not clear and this is another reason for preferring the faster circle-based methods.

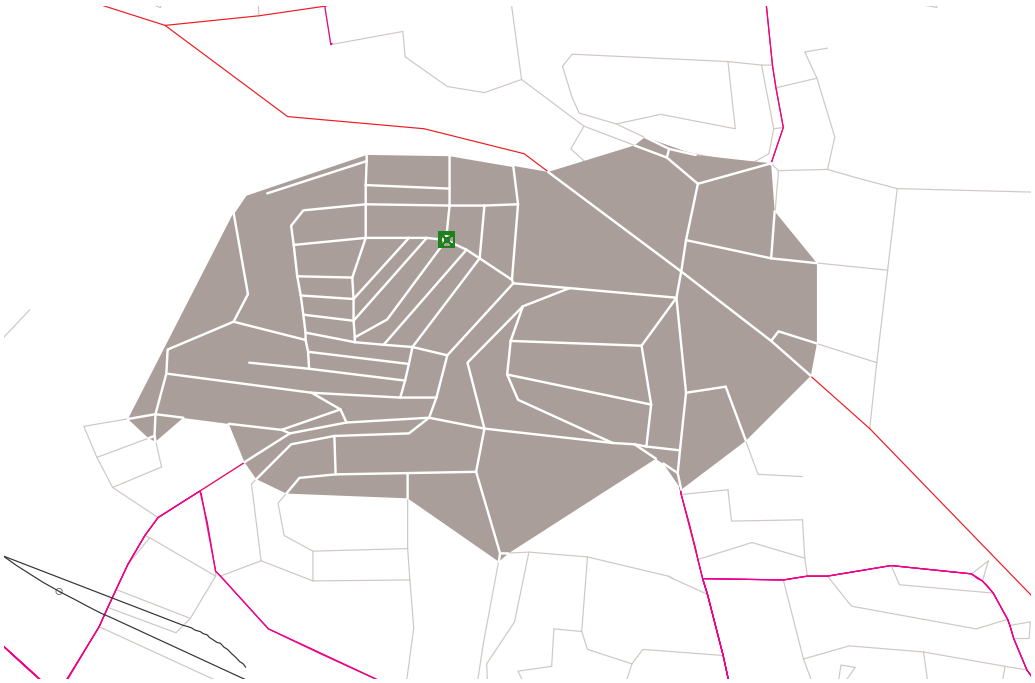


Figure 4.15 *ArcView* service network and service area

Confirmation that the quicker method produces acceptable results is provided by figure 4.16. In and around the urban area, and at a 'resolution' of say 5 minutes, the circle based method produces isochrones which are a good approximation to the more 'accurate' service area method.

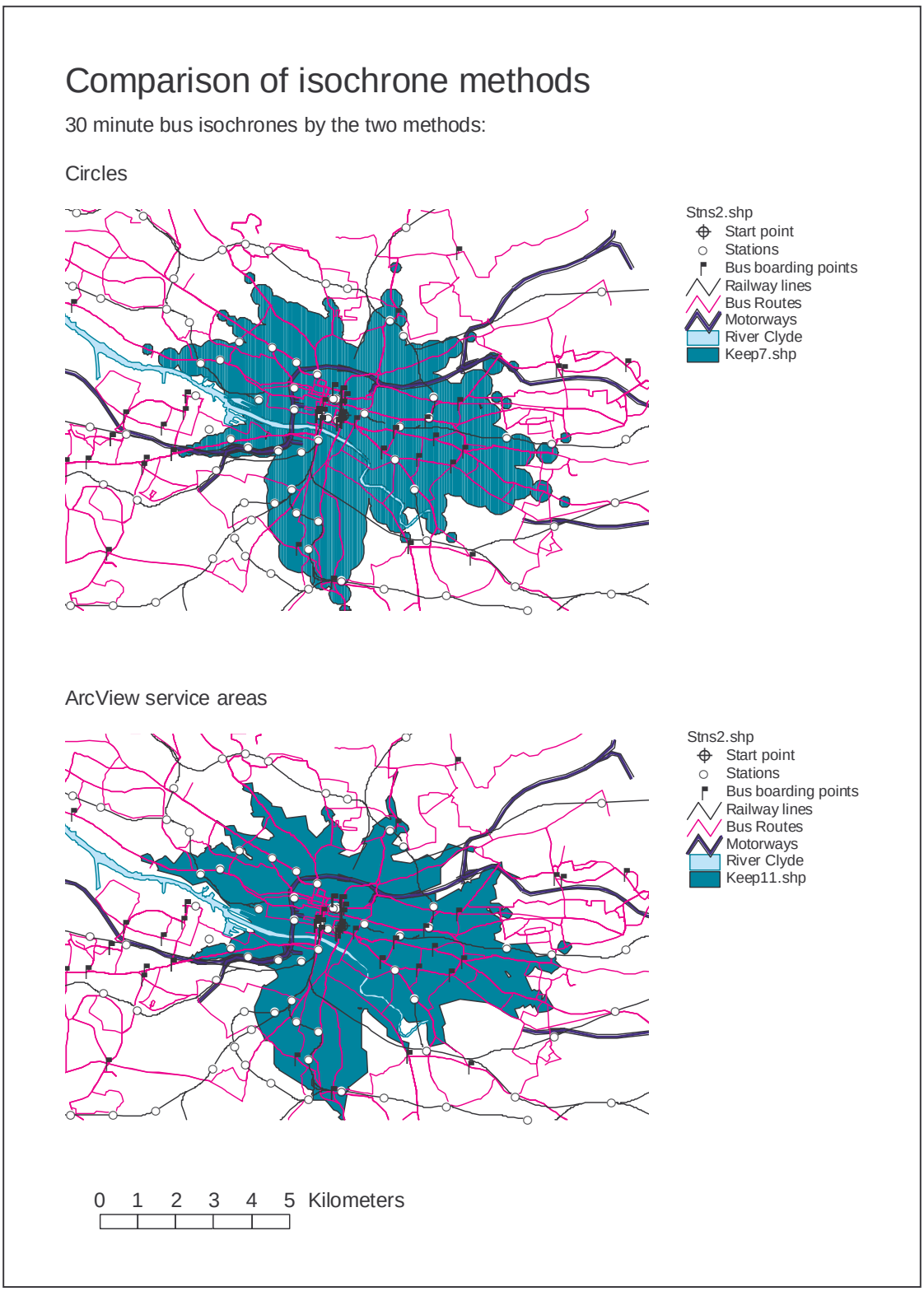


Figure 4.16 Comparison of two methods of isochrone generation

5. OVERALL REVIEW OF SCRIPTS

It should be clear from the foregoing that this work is far from a fully realised set of tools for constructing isochrones. However, it could form the basis of such a set of tools, albeit with enhancements. Many of these have been mentioned in the previous chapter, but it seems worthwhile reviewing them now that a full description of the work has been provided.

5.1 Performance optimisation

Urgent attention to optimising all the existing scripts' performance (that is speed of operation) is required. This would involve a number of changes. The most difficult of these is adapting the way in which bus routes and their interconnections are handled, although there is also room for improvement elsewhere. A good start for the bus routes would be to maintain an 'interconnection matrix' which would record for each connection made between bus routes that such a connection has been attempted (whether resulting in new boarding points or not). This would speed up operation as the `GenerateArrivalTimes` script moves through each route, since currently it considers all possible connections for all routes. This is particularly inefficient in the Glasgow example because interconnections between almost *all* the routes are feasible in the compact central zone of the city.

A refinement of the matrix might involve a pre-processing step where the bus network is considered as a whole and all connections which make very little sense are 'pre-set' to ensure they are never considered. This would remove connections between routes running in broadly similar directions. Such a pre-processing step

would be dependent on the defined area of interest, and would probably involve some complex spatial processing.

As was noted in section 4.5, there is similar scope for speeding up `TryTrainsFromThisStation` by only considering travel to the next station along at each step. In contrast to the bus problem, this would be a simple change to make.

5.2 Refinement of area of interest definition

At the moment the mechanism for defining the user's area of interest is somewhat clumsy. It would be easy to provide variants of the tool which allow definition of circular, elliptical, and polygon areas, rather than a rectangle.

This does not really address the thornier problems of organising subsets of the stations information and making this intelligible to the user. There is a tendency for themes to proliferate during execution and keeping track of them all becomes difficult. One possibility is to use *object tags*. All objects, including themes, have a tag which can be set to any other kind of object. This method could be used to store the station dictionary associated with a user defined area. Additional tools could then be built to display the data in the area theme's object tag as required, or to create a new theme from it for subsequent use. Thus when a user defined an area, only a new `area*` theme would be created - not two new themes as in the current implementation. Running the `GenerateArrivalTimes` script would result in a prompt for a starting point and time (which is a more logical approach than the current one). The main outcome of this approach would be the avoidance of a proliferation of new themes as the user investigates different problems.

Whilst this approach might be more elegant than the current one, it can not possibly avoid the problem of transport facilities which lie just outside the area of interest. This difficulty is one that can only be handled by acknowledging that the further the starting point of any set of results is from the centre of the study area, the less valid are the results, particularly if the user is interested in similarly peripheral destinations (on the same side of the study area - that is, not through the centre).

A completely different approach would be to drop areas of interest entirely, and to instead have the `GenerateArrivalTimes` script curtail its operation as soon as some specified elapsed travel time limit had been reached. This exacerbates the problem of execution times since a set of times generated with (say) a 30 minute time limit in force would subsequently be no use for generation of 60 minute isochrones. On the other hand it would probably be possible to rework the existing scripts to use an existing set of arrival times for 'expansion' further afield. The problems here illustrate some of the difficulties of working in the 'prototyping' manner, and this area would require careful thought about user requirements for a satisfactory solution to be devised.

5.3 Changes to waiting time calculation

The difference in calculation of waiting times for trains and buses has been touched on earlier: timetables are used for train times, whereas a simple frequency value is used for buses. This has the side-effect of making output maps sensitive to the chosen starting time, which is especially problematic where many trains are available (see section 6.1). The chosen method for buses may also be somewhat pessimistic in that it seems unlikely that users of infrequent buses 'just turn up' - they are likely to allow for the timetable. Thus a preferable approach might be to determine both train

and bus waiting times using the 'half-the-timetable-frequency' method where frequency is high (say 10 minutes or less) but some fixed delay where services are less frequent, say 5 minutes. Implementing this change in the current system would be difficult since the train timetable data would require alteration to a format more like that of the bus routes. Additionally, there are problems with calculating connection delays between services, and, in theory at least, the current system allows some investigation of likely variability in travel times which may be considered an advantage.

5.4 Better estimation of bus route journey times

Currently bus journey times are calculated as if buses progress at constant speed along their route. This is a convenient simplification, and one which may have significant effects on the perception of accessibility which outputs from this work gives. This is particularly important since much of the concern about accessibility in the Glasgow area relates to perceptions of poorer access to the city centre for residents of peripheral estates, served mostly by buses. Much more data on traffic speeds in the area is required before anything which was not an equally or more distorting simplification could be made. Clearly a better estimation method would be a significant enhancement.

It is hard to see how this can be achieved without a *lot* more data. The most obvious approach is to use the street network (already in use for walking journeys) and attach appropriate impedances to each street segment and intersection. If this task is too daunting (and it certainly scares the author!), some consideration of methods of adding mileposts with timings to the `Bus Routes` theme might offer a less data intensive solution. It should be borne in mind that any attempt to grapple with the

full complexity of a bus network as comprehensive as that in Glasgow is likely to run into difficulties. Conversations with Strathclyde Passenger Transport Executive (SPTE) reveal that they currently hold details of around 12000 'bus profiles', that is distinct variations in routes, followed by buses in the region. Around one third of these are in Glasgow city. In that light comprehensive approaches are probably best left to the SPTE itself!

5.5 Refinement of walk portion of journeys

Notwithstanding arguments that the circle-based approximation method for isochrone construction is adequate, it would be desirable to tidy up some of the loose ends in the current street network, used for the walked portion of journeys, and also by the slower isochrone construction scripts. This work would mostly involve using more detailed information, possibly just a larger dataset to include walkways, footpaths, subways, pedestrian precincts and so on. A more ambitious set of improvements would involve writing a new version of the *ArcView* 'service area' generation requests. This could be better matched to the exact requirements of this task. It could probably also run more quickly than `FindServiceArea` since it would not create intermediate files written to the hard drive.

6. EXAMPLES OF OUTPUT

This chapter contains examples of output produced using the tools described in this dissertation. Each example is accompanied by commentary, highlighting interesting features of the output.

6.1 Centre of town outwards

Figure 6.1, 6.2 and 6.3 show isochrones from a central Glasgow location, in this case the author's home at 60 Wilson Street, approximate National Grid coordinates NS 594 652. The maps show respectively the isochrones for train only, bus only, and both modes. All have been generated using a start time of 15 minutes past the hour. This was chosen as the time which gives rapid access to the largest number of services from the two central stations. This is clearly not an entirely satisfactory approach in the central area where many services are available, and lends weight to a preference for the method of waiting time calculation proposed in section 5.3.

Returning to the maps, the way in which train travel allows the traveller to 'leap-frog' from one location to another is clearly visible in figure 6.1. The continuous movement outwards from the centre associated with bus travel is also clear in figure 6.2. This map also shows the major routes well in particular Paisley Road to the west of the centre, and various corridors to the east. Figure 6.3 shows how using both modes allows a traveller to fill in gaps in the coverage associated with train travel. This is particularly evident along the north bank of the Clyde, and in the east. It should be clear from all maps that there is an east-west bias in the transport network as against north-south movement.

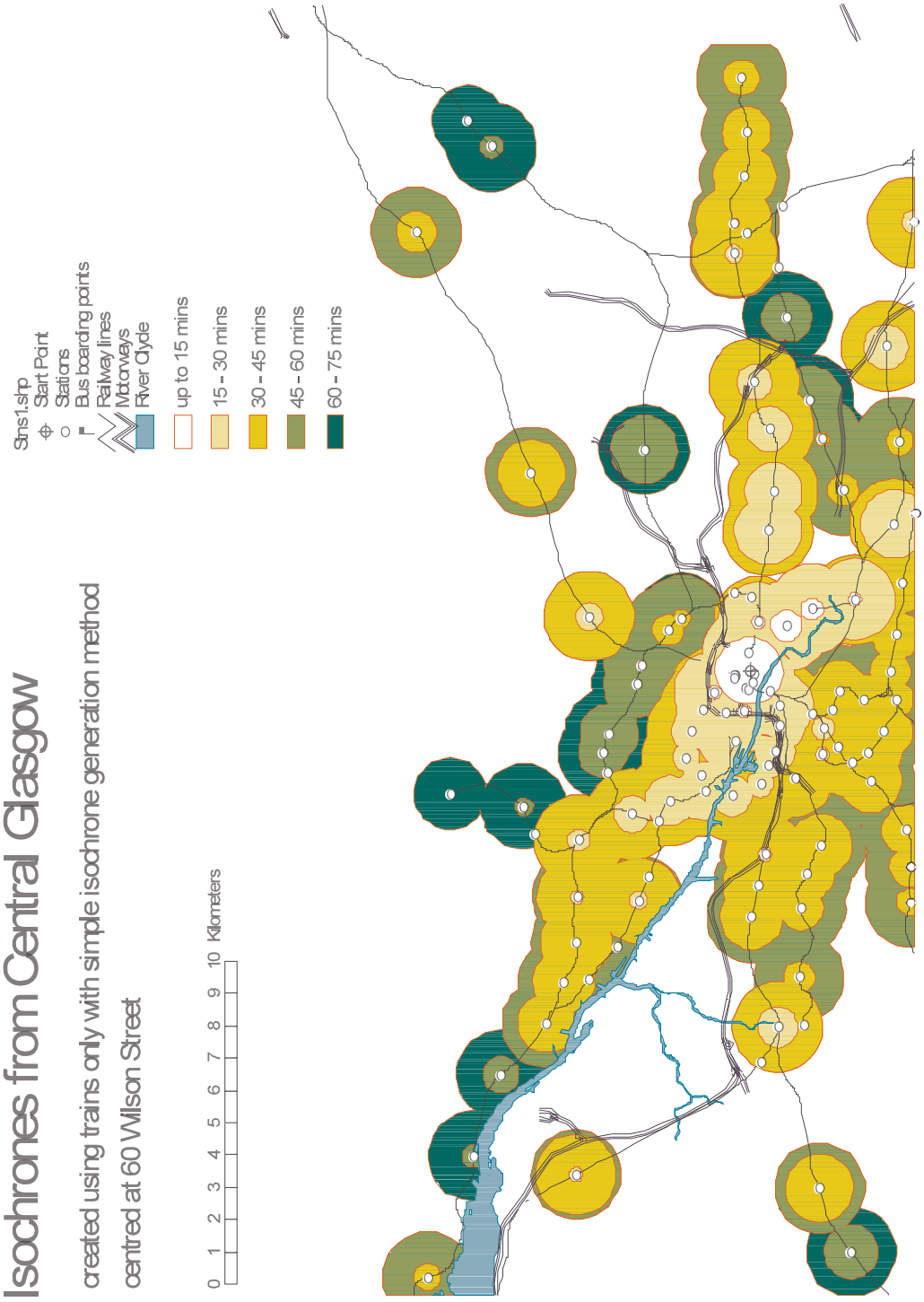


Figure 6.1 Isochrones for whole study area (trains only)

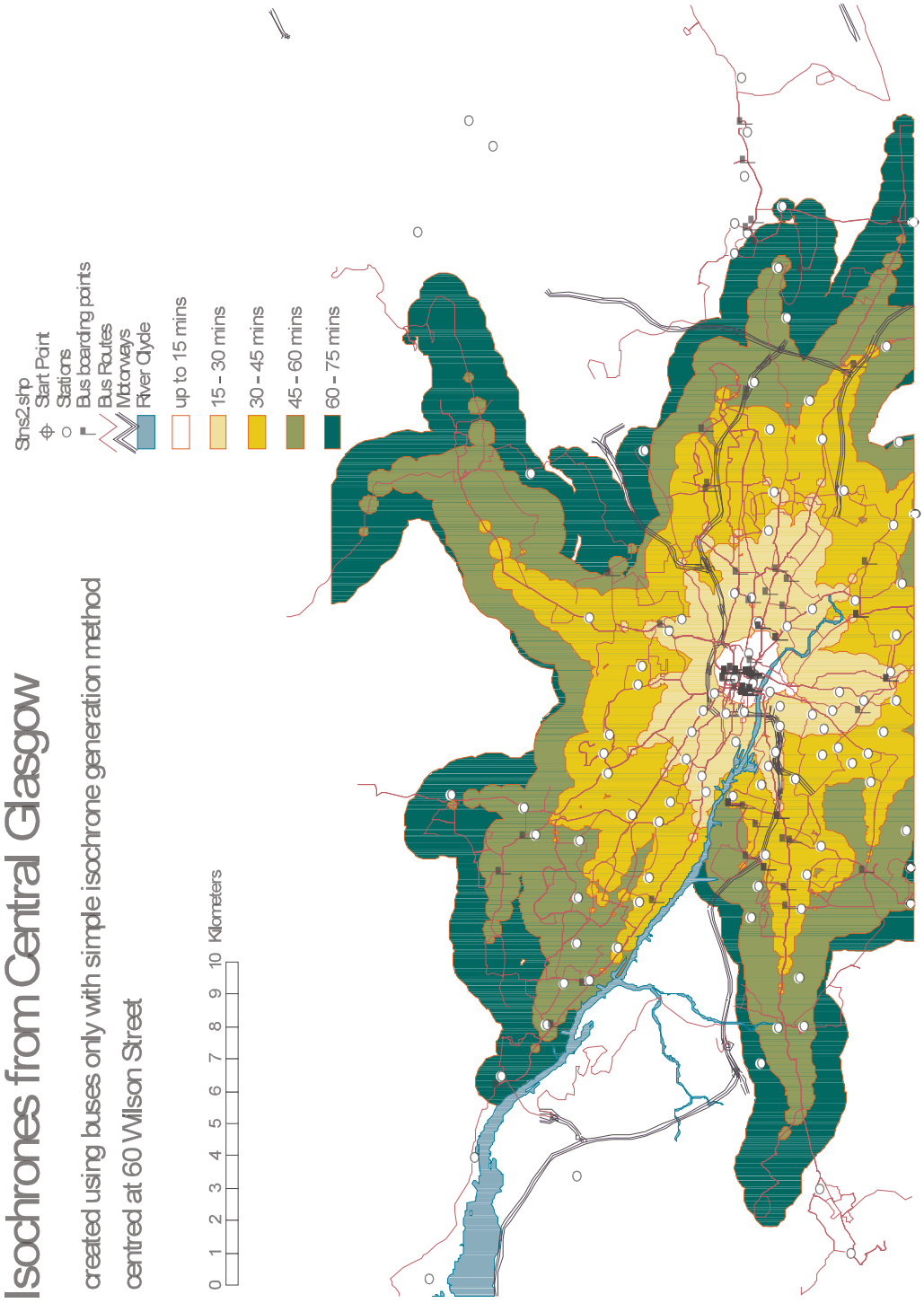


Figure 6.2 Isochrones for whole study area (buses only)

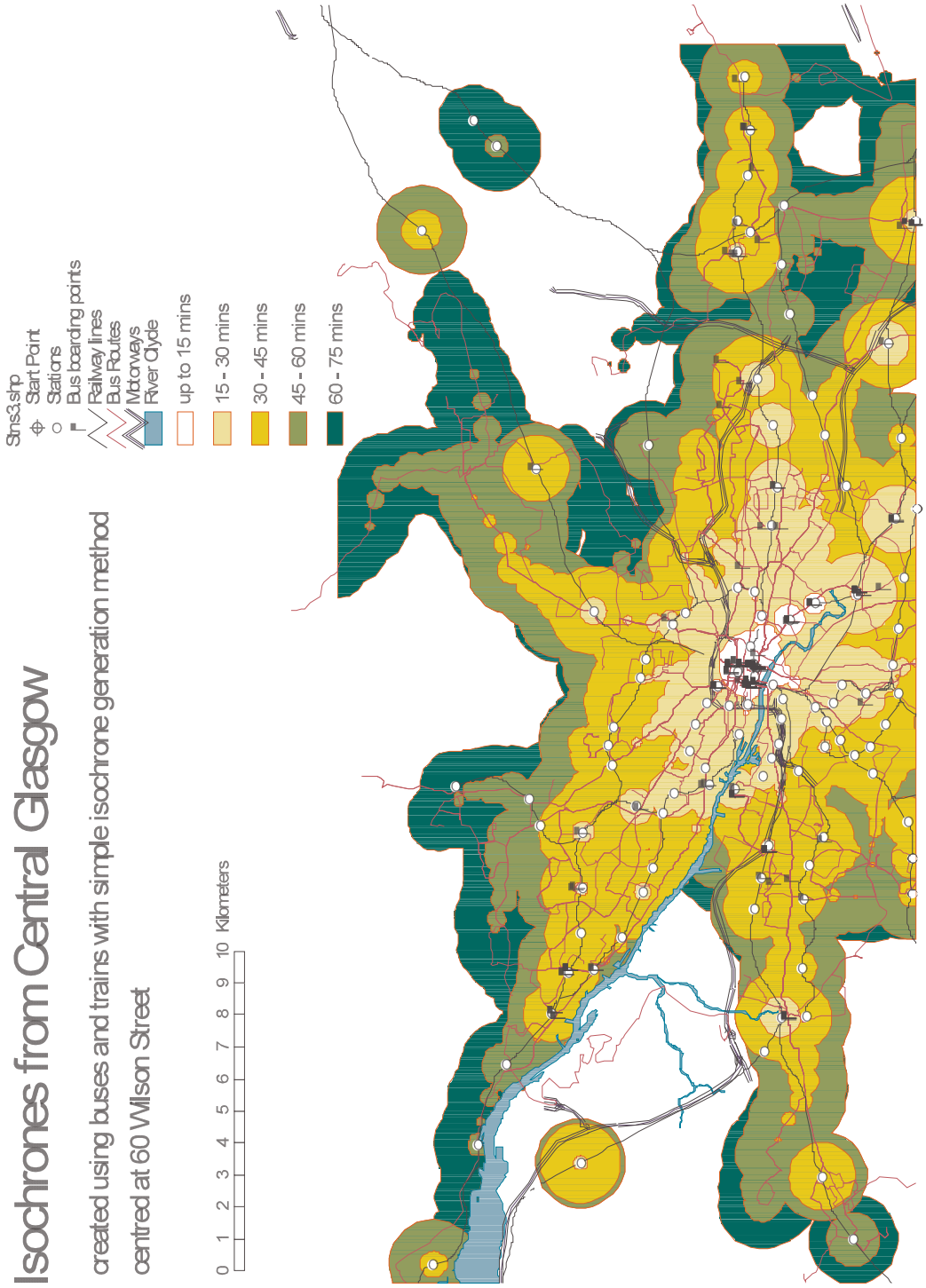


Figure 6.3 Isochrones for whole study area (both modes)

Note that these maps were all produced with very little information about bus routes around Paisley to the west of the city centre, and the gap in coverage is evident. Note also that the sharp cut-offs on the edges of the map area are because the Ordnance Survey data used runs out at these points.

6.2 Comparison of Easterhouse and Milngavie

Easterhouse is an outlying estate on the eastern edge of Glasgow which is generally regarded as deprived with low employment, poor housing and low incomes. Milngavie, by contrast, is an outlying town to the north of the city with high levels of owner-occupation, and generally regarded as well-to-do. These two locations have been chosen to demonstrate the sort of comparisons which could be made using the isochrone tools. Since the comparison is purely qualitative, no socio-economic data are presented to back up the above assertions. However, comparisons between areas of contrasting socio-economic character could be a primary focus of any governmental body concerned with ensuring that public transport services and subsidies are directed to appropriate areas. This might be from an equity standpoint, but also potentially from the point of view that the impact of public transport provision on car usage may be of interest.

Figure 6.4 shows the two study areas, which are of approximately equal area, both including the centre of the city. Figures 6.5 and 6.6 illustrate the Easterhouse isochrones, while figures 6.7 and 6.8 show the corresponding data for Milngavie. All these maps have been constructed using a starting point about 15 minutes walk from the nearest station. These diagrams point up the difficulties in using isochrone maps to assess comparative accessibility. It is hard to discern any strong differences in accessibility between the two locations. Many bus routes run between Easterhouse

and the centre, and arguably the traveller who uses buses only would be better served living in Easterhouse.

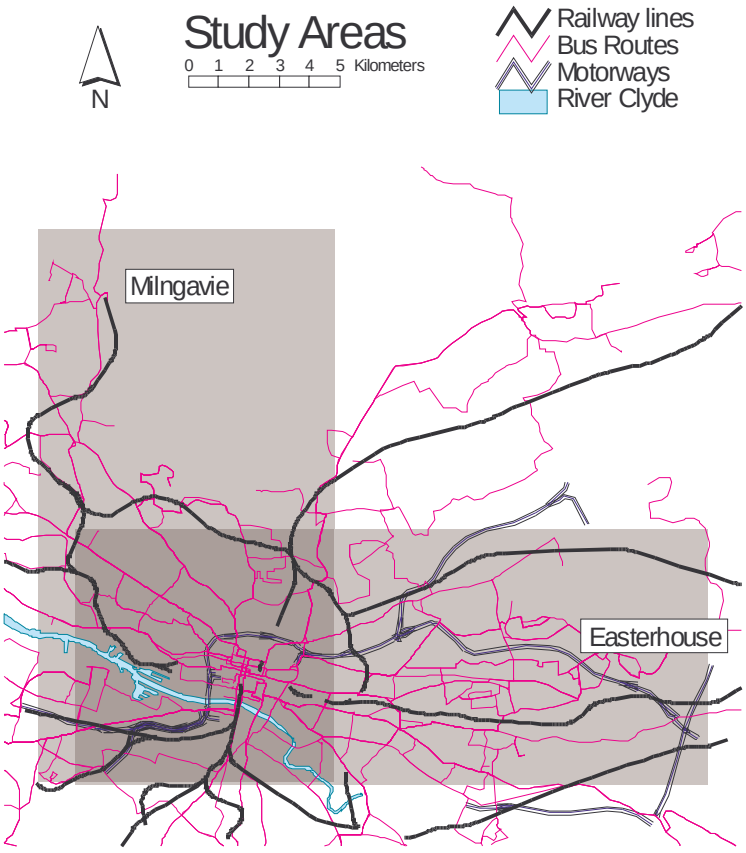


Figure 6.4 The Easterhouse and Milngavie study areas

Perhaps a more interesting result is shown in figure 6.9. This shows isochrones for both modes from a location in Easterhouse, but over a wider study area which extends in all directions from Easterhouse itself. The start time for this map was deliberately chosen to favour trains *out of town*, nevertheless it is clear that residents of Easterhouse are likely to experience difficulties travelling anywhere other than the city centre on public transport. This is probably what we would expect. It does raise questions about how feasible it would be for residents of Easterhouse with no access to a car to take jobs at the increasing number of peripheral industrial and retail locations. Note that the same is likely to be true for car-less residents of Milngavie.

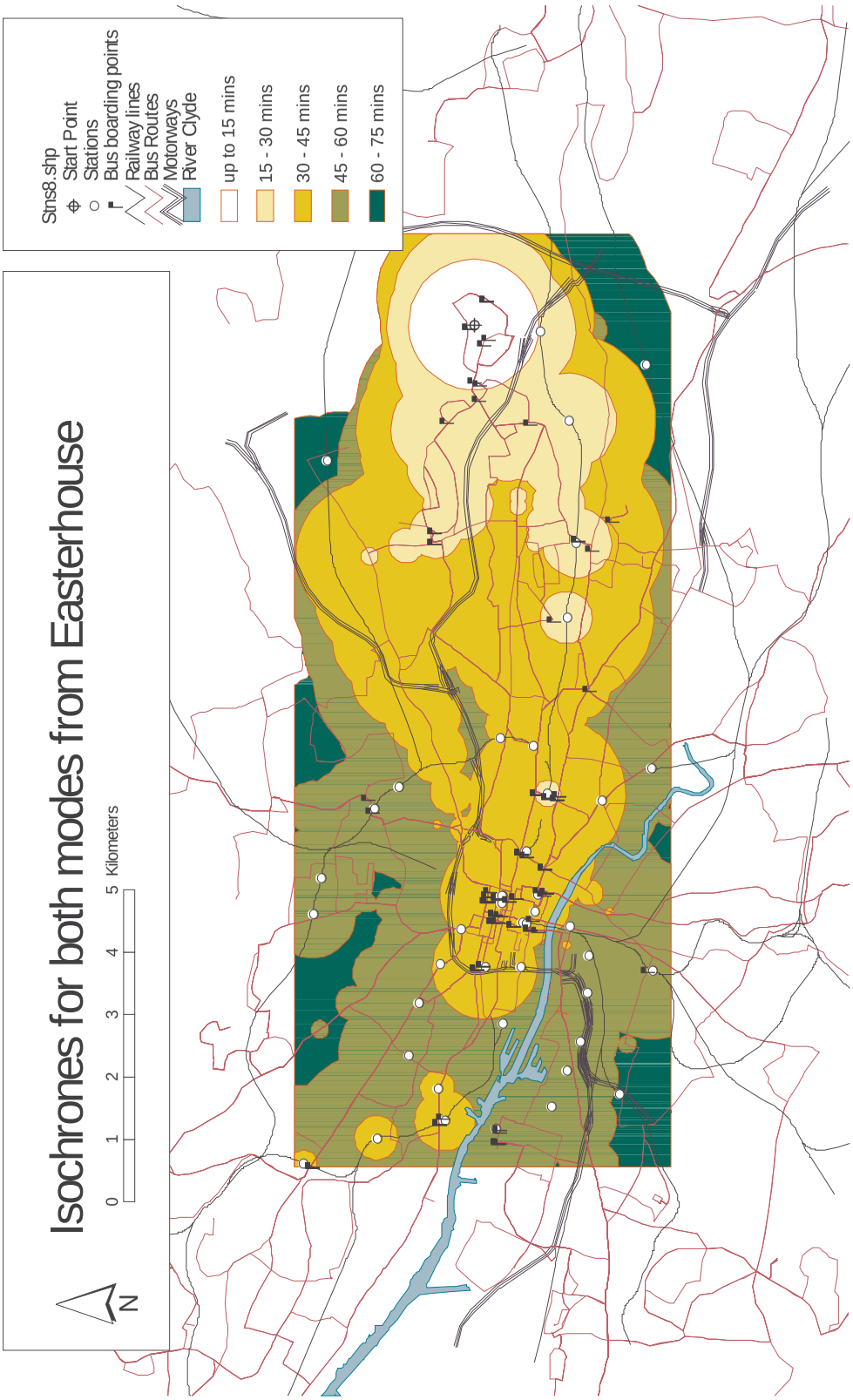


Figure 6.5 Easterhouse - both modes

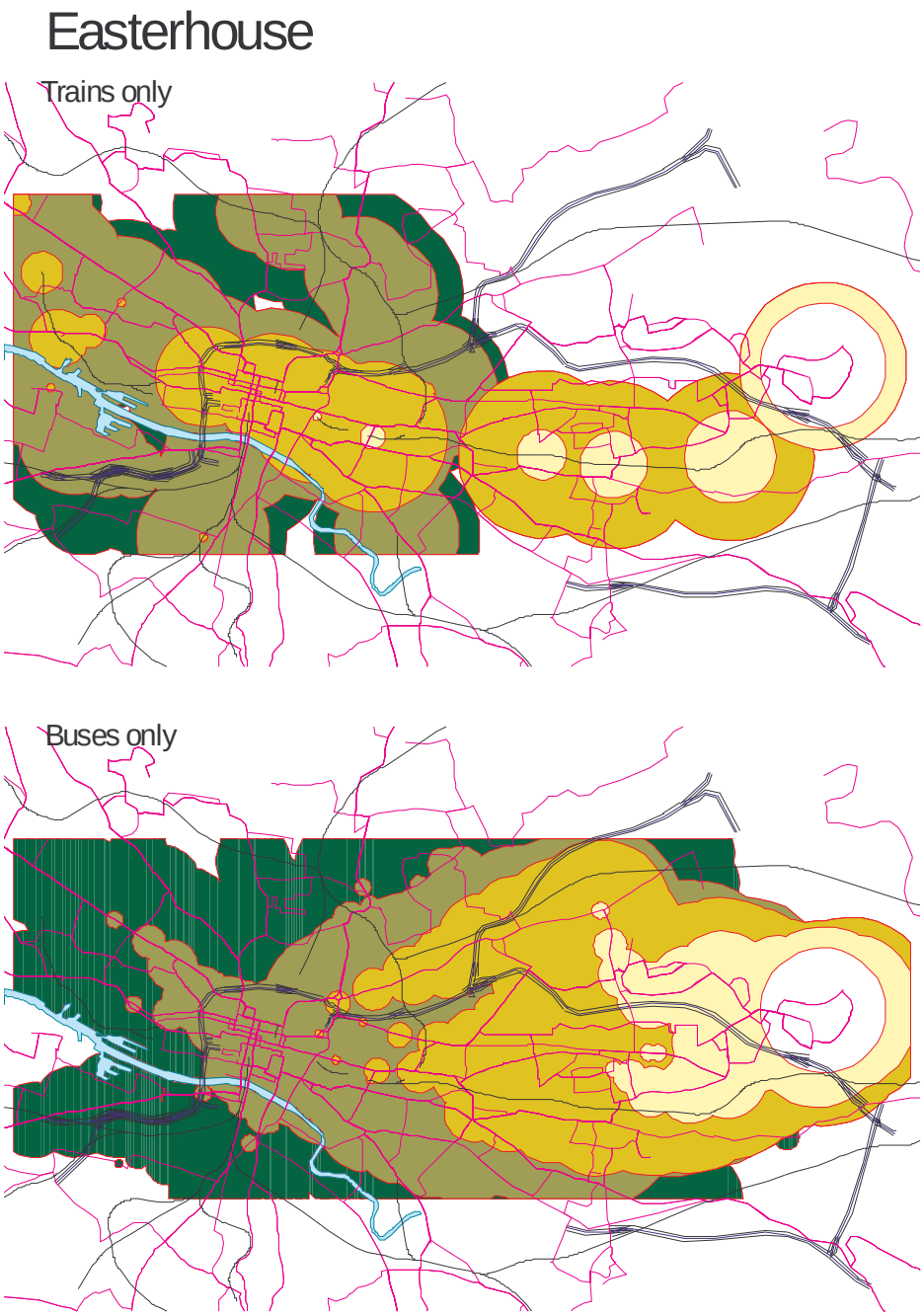


Figure 6.6 Easterhouse - each mode separately

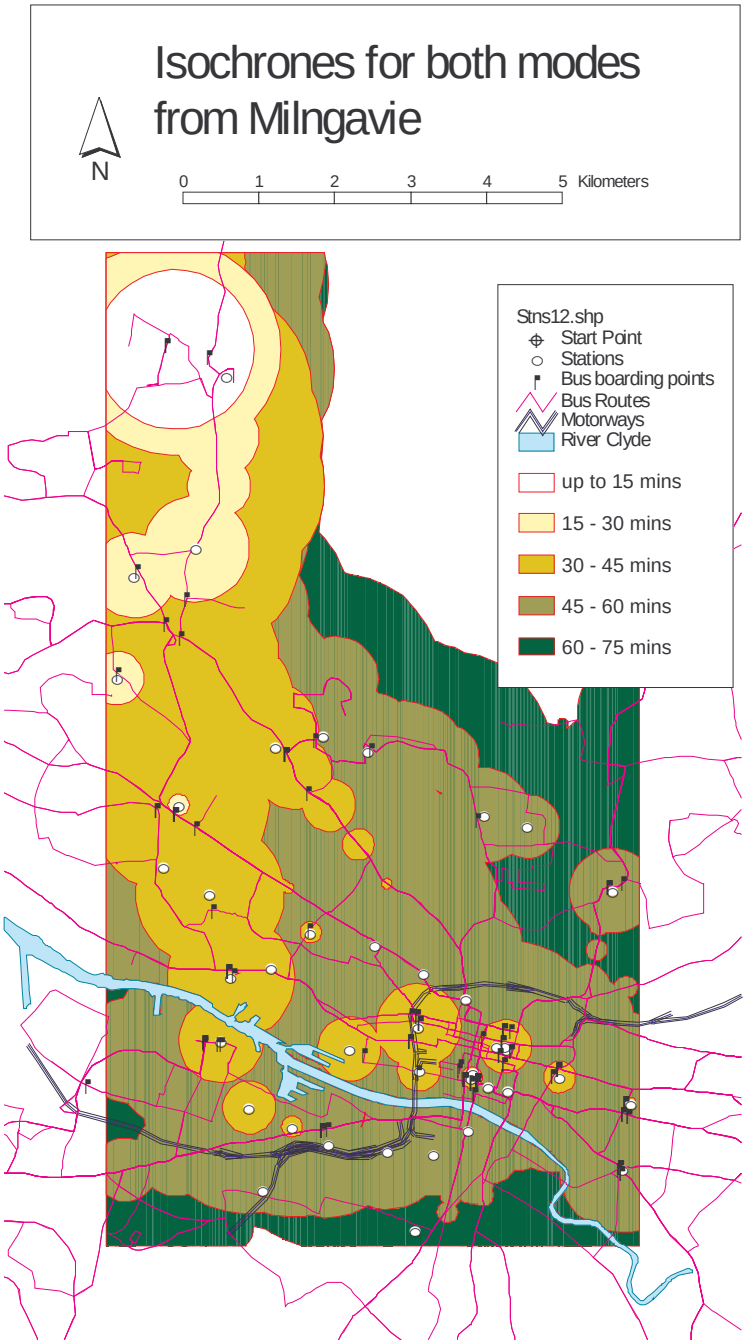


Figure 6.7 Milngavie - both modes

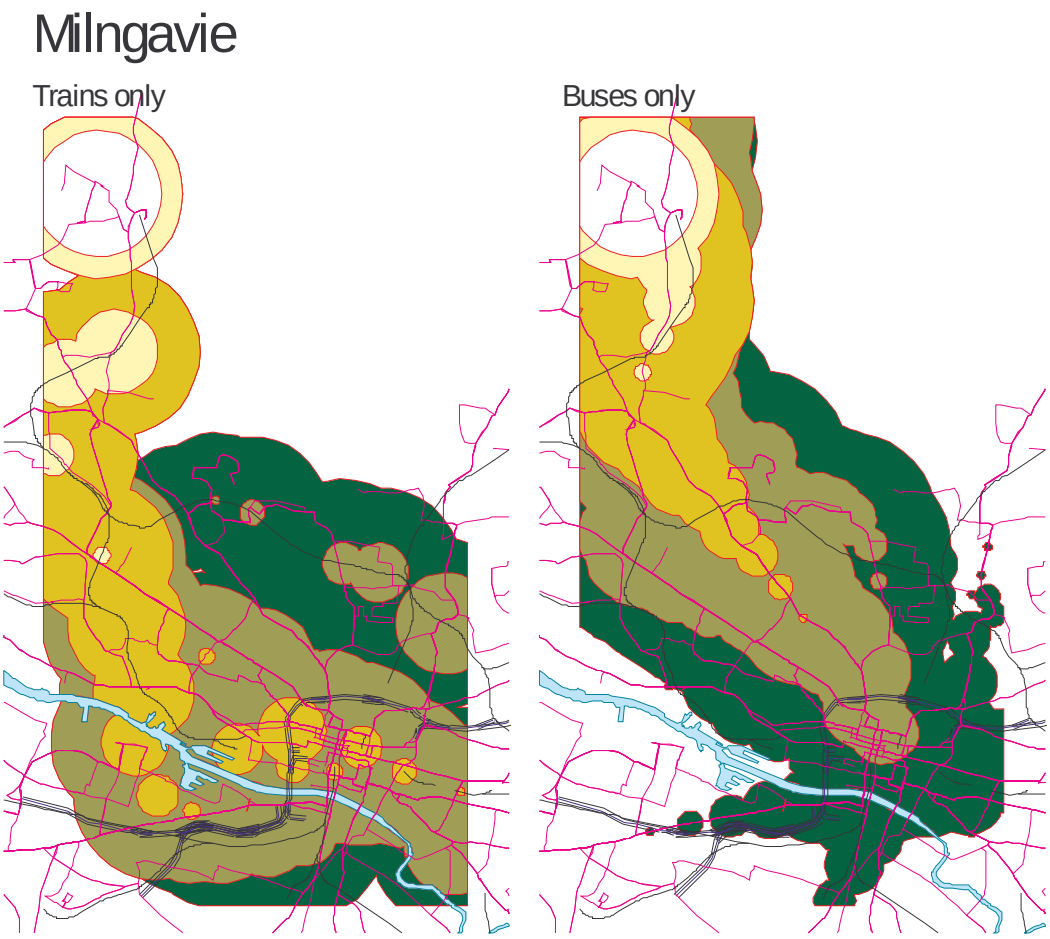


Figure 6.8 Milngavie - each mode separately

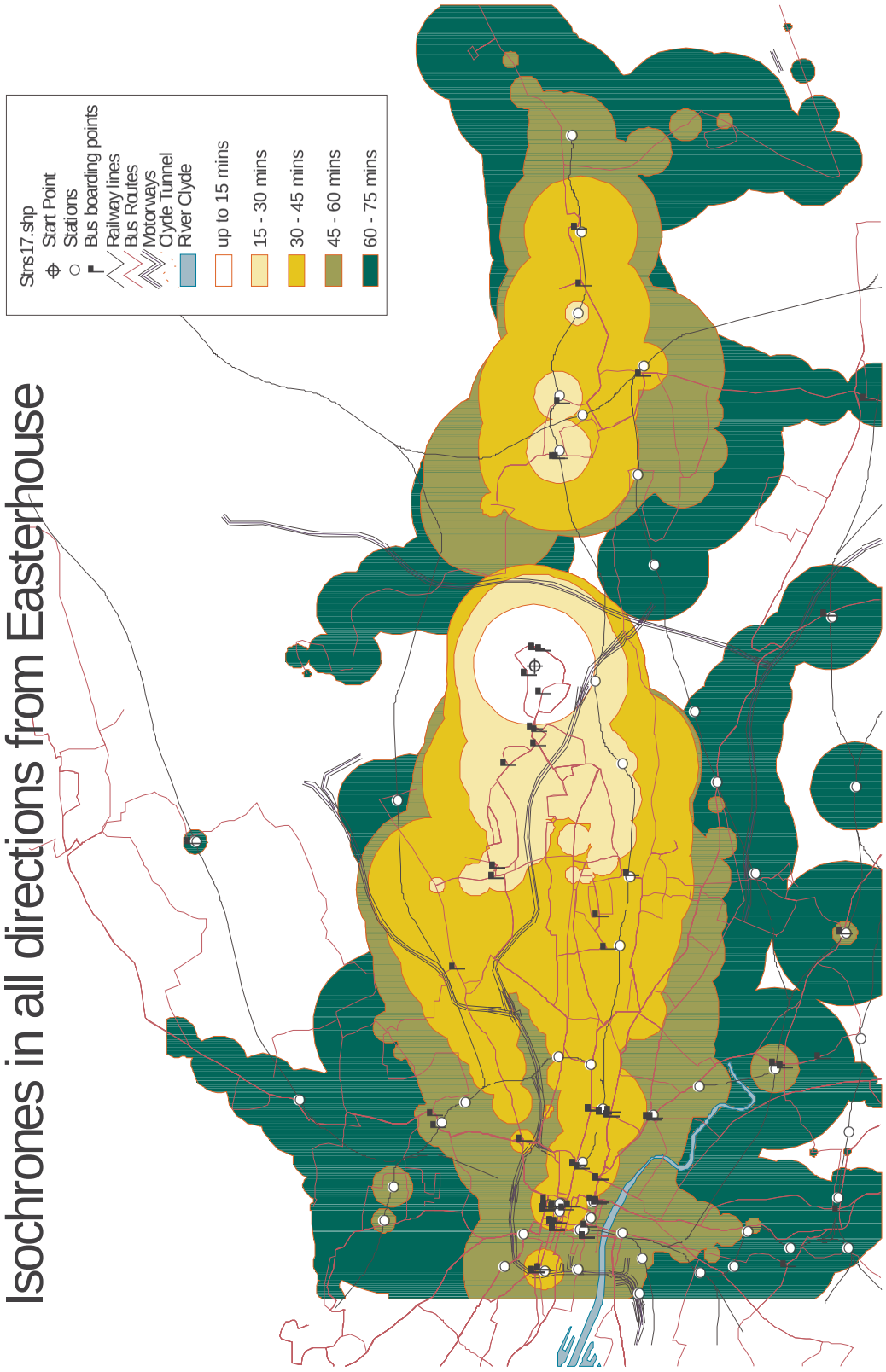


Figure 6.9 Isochrones in any direction from Easterhouse

6.3 Constrained accessibility

The final output presented here shows how isochrones from two locations could be merged to produce accessibility maps for individual traveller's situations. These have been produced 'by hand' in a drawing package, using isochrones produced by the isochrone scripts, but the process could clearly be automated in the GIS. This work relates to the ideas discussed in section 2.3. Two sets of isochrones have been produced. Figure 6.10 shows isochrones at 5 minute intervals from a work place A (indicated by the 'cross hairs'). The isochrones have been constructed using the scripts described in this dissertation. Only buses have been used in the calculation.

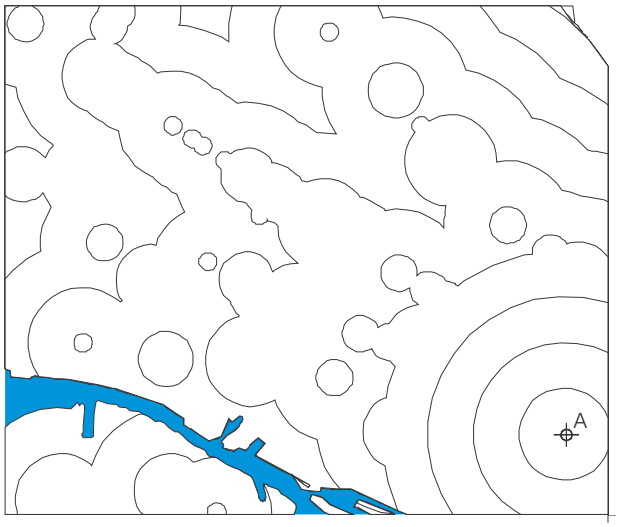


Figure 6.10 Isochrones from place of employment

Figure 6.11 shows a similar set of isochrones for a school B, also at 5 minute intervals. Now if we consider a situation where a parent working part time at A must pick up children from school at B one hour later, we can produce a set of shapes showing areas which can be visited *en route*, and the time available for doing other things (shopping, visiting post office *etc*) at those locations. The resulting figure is

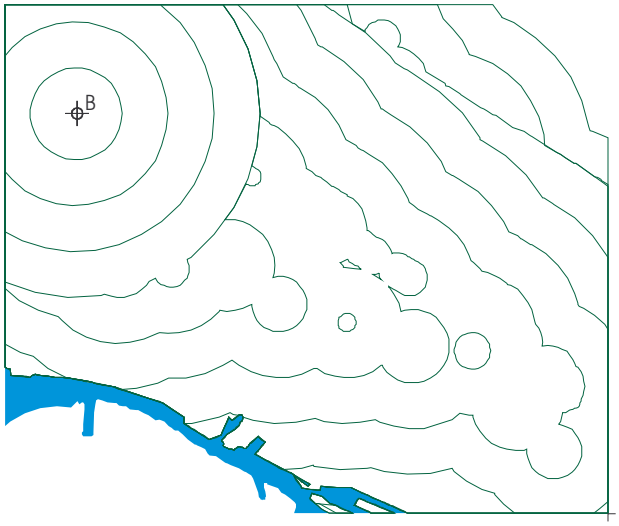


Figure 6.11 Isochrones from a school

shown in figure 6.12 together with the bus routes through this area. Darker regions can be reached in the available time, but would leave little time for other activities. Lighter regions can be reached with up to 25 or more minutes spare time. As can be seen from the diagram and as might be expected these areas lie close to the bus routes along which travel from the place of employment to the school takes place.

Figure 6.12 Merged isochrones

The method of producing these merged isochrones is to find the intersection area between pairs of isochrones (one from the work place, one from the school) whose combined journey time leaves a given amount of available time at the destination. For brevity, let us label a pair of isochrones (x, y), where x is the time in minutes from the work place, and y is the time in minutes from the school. Thus with 60 minutes available, the '5 minutes spare time' region consists of the merged intersection areas of pairs of isochrones whose combined journey time is 55 minutes: (5, 50), (10, 45), (15, 40) and so on. Producing this example has been rather arduous, but a relatively small amount of programming would produce the same output automatically. There would, of course, be difficulties - for example, where isochrones overlap in several places.

Another problem, which is skipped over here, is that the destination actually requires a set of *travel-to* isochrones. Since bus waiting times have been simplified by ignoring the timetable, the *bus* travel-from isochrones used here are allowable. However, the way in which train waiting times are calculated could not be easily fitted into this approach, since it uses timetable data which is intrinsically unidirectional. However, producing travel-to isochrones would not be a complex modification of the scripts described.

Hopefully, this example gives some idea of the potential for further investigation of accessibility problems using isochrones.

7. SUGGESTIONS FOR FURTHER WORK AND CONCLUSIONS

7.1 Further work

Aside from optimisation of the current tools, there are various possibilities for further work. An intriguing problem would be to use this framework, to produce travel *monetary cost* isochrones. These would represent areas which could be reached at equal expense to the traveller. Arguably, these could give a truer - certainly equally valid - picture of accessibility. Expressing such costs relative to average incomes of the starting point census district, would also be an interesting extension. A related development could be to build 'traveller profiles' which characterise different sorts of travellers, in terms of their inclination to walk, their walking speed, and their income. Analyses of this sort could be carried out with the current tools, but the process would be arduous. Another interesting direction would be to develop some scripts which manipulate input sets of isochrone themes and facilities data to calculate accessibility indices like those discussed in chapter 2. Yet another development could be to use different ways of estimating bus waiting times, and train waiting times, with a view to examining the variations in service experienced by travellers which can often be as great a deterrent to using public transport as time and cost. Another possibility has been described in section 6.3 where isochrones are used to produce a constrained accessibility map.

7.2 Conclusions

Isochrones as presented and used here provide an easily understood method for examining transport accessibility issues. They can also be the basis of very much more complex and sophisticated analysis techniques. Their clarity of meaning and interpretation which matches well with people's perception of the transport options

available to them is a further advantage. Given these advantages it is perhaps surprising that they are used so little in the literature. Presumably this is due to the difficulties and intricacies of construction by manual methods, and to limitations in computing power. This work demonstrates that even given only a few weeks, an inexperienced GIS programmer can make a passable attempt at producing a useable set of GIS tools for automated production of isochrone maps. Furthermore this has been achieved in the context of a very large city, with a complex public transport network. This is a testament to the power and flexibility of current desktop GIS systems.

Nevertheless, as is clear from the foregoing, many simplifications have been made to make this work feasible. The resulting system remains slow, so that any kind of interactive investigation (of say proposed new bus routes) would still be a tedious and time-consuming task. Suggestions have been made for improvements in the current scripts, but it seems clear that dramatic improvements would require a different approach. This could involve 'intelligent spatial processing' akin to the methods by which humans navigate through a complex transport system, and bringing in techniques from artificial intelligence and expert systems. An alternative approach would be that taken by suppliers to the transport analysis profession, who have concentrated on specialist software which is based mainly on extensions of the network analysis techniques described in section 2.5. (see for example *www.mva-group.com*) AI techniques appear to be some way off for mainstream GIS. Specialist transport software is available now, which suggests that the best approach might involve interfacing those packages to GIS. The technique used here of feeding a set of locations with arrival times to an isochrone building routine is simple and provides the required information relatively quickly since the methods used, such as

buffering are ideally suited to GIS. Thus transport analysis software could be used to generate the arrival times, and a GIS to build the isochrones.

Where GIS really comes into its own is in immediately locating the output iscohrone shapes in relation to any other available data in the region of interest. This means that numerous accessibility problems are immediately capable of straightforward analysis using standard overlay techniques. More complex spatial manipulations can also be applied easily. The constrained accessibility analysis in section 6.3. is an example of what could be done. In short, it seems likely that, for the foreseeable future, GIS will be an important technology to transport planners, providing a flexible and efficient way to integrate transport analysis with other aspects of the planning process.

SELECTED BIBLIOGRAPHY

Ahuja, R K, Magnanti, T L, Orlin, J B (1993), *Network Flows: theory, algorithms, and applications*, Prentice Hall - Eaglewood Cliffs

Al-Sahili, K; Aboul-Ella, M (1992), 'Accessibility of public services in Irbid, Jordan', *Journal of Urban Planning & Development - ASCE*, v118 n1, pp 1- 12

Dijsktra, E W (1959), 'A Note on Two Problems in Connection with Graphs', *Numerische Mathematik I*, pp269-71

Dolan, A, Aldous, J (1993), *Networks and Algorithms: An introductory approach*, John Wiley & Sons - Chichester

Doling, John (1979), *Accessibility and strategic planning*, Centre for Urban and regional Studies, University of Birmingham

ESRI (1996a), *Using ArcView GIS*, Environmental Systems Research Institute Manual

ESRI (1996b), *Using the ArcView Network Analyst*, Environmental Systems Research Institute Manual

ESRI (1996c), *Using the ArcView Spatial Analyst*, Environmental Systems Research Institute Manual

ESRI (1996d), *Using AVENUE*, Environmental Systems Research Institute Manual

Forer, P C, Kivell, H (1981), 'Space-time budgets, public transport, and spatial choice', *Environment and Planning A*, v13 pp497-509

Frank, A U (1991), 'Qualitative spatial reasoning about cardinal directions', in *Auto-carto 10: Proceedings of ACSM-ASPRS Annual Convention*, Baltimore 1991, pp148-167

Frost M E, Spence N A (1995), 'Rediscovery of accesibility and economic potential: the critical issue of self-potential', *Environment and Planning A*, v27 pp1833-1848

Glasgow Passenger Transport Executive (1969), *Greater Glasgow Transportation Study*, in two volumes

Guy, C M (1983), 'The assessment of access to local shopping opportunities: a comparison of accessibility measures (Reading, UK)', *Environment & Planning B*, v10 n2, pp 219-237

Hansen, W G (1959), 'How accessibility shapes land use', *Journal of the American Institute of Planners*, v25 pp73-76

Hanson, S (Editor) (1986), *The geography of urban transportation*, Guilford Press: New York - London

Keechoo Choi, Tschangho John Kim (1994), 'Integrating transportation planning models with GIS: issues and prospects', *Journal of Planning Education & Research*, v13 n3, pp 199-207

Koenig, J G (1980), 'Indicators of urban accessibility: Theory and application', *Transportation*, v9 pp145-172

Lewis, S, McNeil, S (1986), 'Developments in microcomputer network analysis tools within GIS', *ITE Journal*, v56 n10, pp31-35

Lewis, S(1990), 'Use of geographical information systems in transportation modeling', *ITE Journal*, v60 n3, pp 34-38

Luo, L Q, Jones, C B (1995), 'A deductive database model for GIS' in *Innovations in GIS 2*, ed Fisher, P, pp33-42, Taylor & Francis - London

Robertson, I M L (1981), *Accessibility to social facilities in a peripheral housing estate: Drumchapel, Glasgow*, Centre for Urban & Regional Research, University of Glasgow

Sedgewick, R (1988), *Algorithms*, Addison-Wesley

Taafe, E J, Gauthier, H L, O'Kelly, M E (1996), 'Geography of Transportation', Prentice-Hall

Tolley, R, Turton, B (1995), *Transport systems, policy and planning*, Longman Scientific & Technical - Harlow

Wachs, M, Kumagai, T G (1973), 'Physical accessibility as a socio-economic indicator', *Socio-economic Planning Sciences*, v7 pp437-456

Weibull, J W (1976), 'An axiomatic approach to the measurement of accessibility', *Regional Science and Urban Economics*, v6 pp357-379

APPENDICES

A. Data sources and data preparation

A brief description of the various data layers or *themes* in the `glas1_0` *ArcView* project was given in chapter 3. This appendix provides more details of the data sources used and the data preparation steps carried out to assemble the project. Details of the various attribute tables are also supplied.

Geographic and temporal extent of data

All data layers in the project are complete for the 40km (EW) by 20km (NS) area whose south west corner has National Grid reference NS 400 600.

All timetable data (train and bus) is for published weekday off-peak services.

Map layers derived from Ordnance Survey *Meridian* data

Many of the layers were generated from Ordnance Survey National Transfer Format (OS NTF) data supplied as two *Meridian* digital product 'tiles' reference NS46 and NS66. A Glasgow University authored utility program `ntf2ung` was used to convert many of these layers to *ARC*/*INFO* ungenerate format prior to further processing. *ARC*/*INFO* can convert ungenerate to 'coverages' which *ArcView* can import, although if further preparation is required they must then be converted to shapefiles. The various data manipulations performed to arrive at the final set of themes are summarised in the diagram overleaf.

Additional notes:

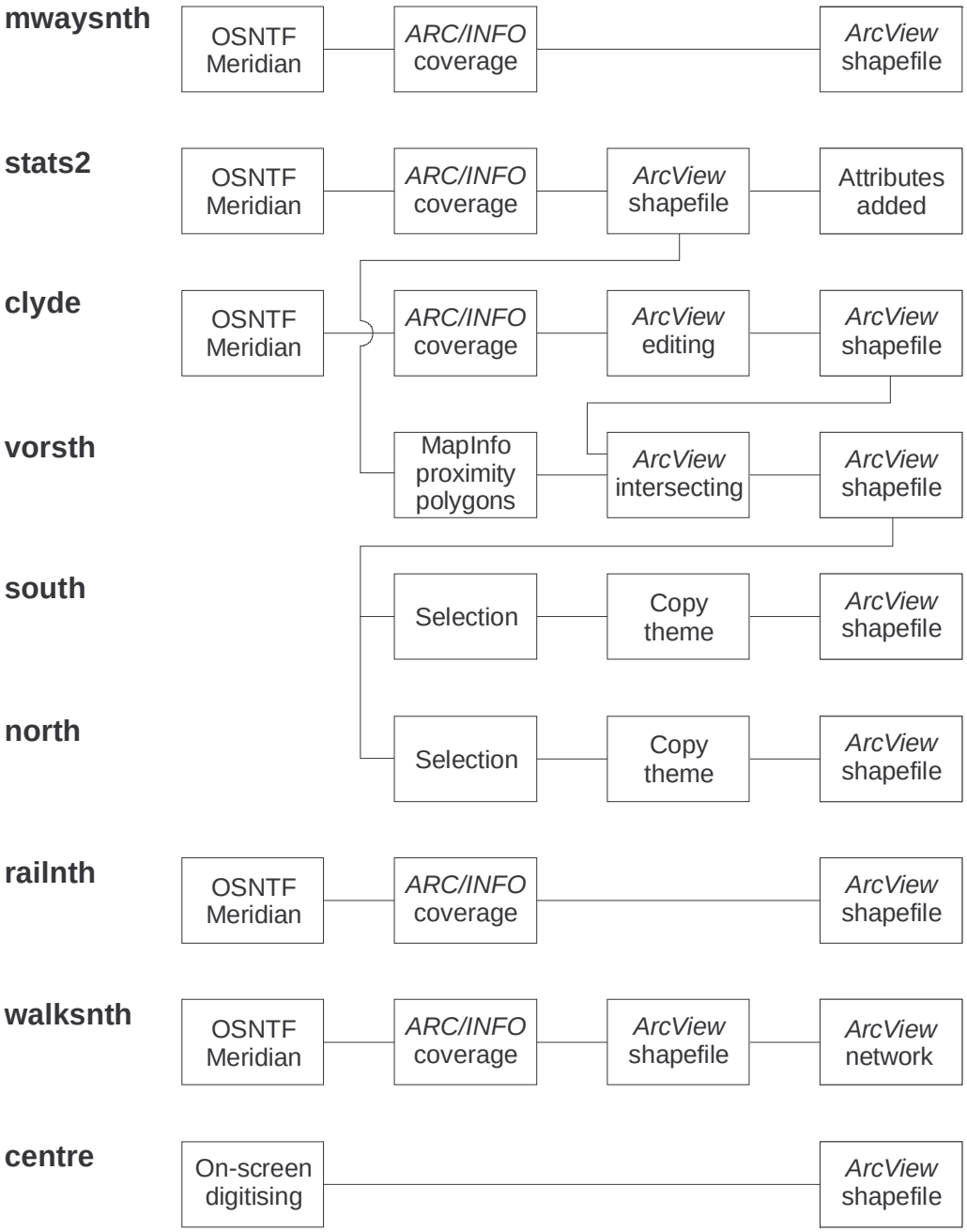
The **stats2** theme contains point locations of all the over and underground stations in the region. Three overground and all the underground stations were not included in the *Meridian* dataset and were added by on-screen digitising. Additionally, second locations were added for Central and Queen Street stations, to account for the low-level stations at these locations. The feature table associated with this data has fields for each station record, as follows:

"ID" is an integer value, unique to each station. Overground stations are numbered from 1 to 123, and underground stations from 200 to 213. When copies of subsets of **stats2** are created by specifying an area of interest (**stns*** themes) the specified start point has ID 0, and any added bus route boarding points have ID numbers greater than or equal to 1000.

"Name" is the station name as specified in published timetables.

Other fields in **stats2** are unused. In **stns*** themes additional fields are as follows:

"Visited" is a Boolean flag indicating that this station has been 'reached' in the running of `GenerateArrivaltimes`.



"Arrival" is an integer value which stores the elapsed journey time for reaching this location from the specified start point.

"On_foot" is a Boolean flag set when a location is reached on foot during the running of `GenerateArrivalTimes`. This is used when constructing isochrones so that such locations are *not* used as centre points for the walk segments of journeys.

The **vorsth** proximity polygons were created from overground stations only. The intersection with the **clyde** theme ensures that polygons which touch but are on opposite banks of unbridged parts of the Clyde, are not neighbours. This prevents 'walking on water' during script execution.

The **north** and **south** themes each include all the proximity polygons on the named bank of the Clyde, and adjacent polygons where the river is bridged.

The **walksnth** theme has additional attributes for segment lengths, and walk time in minutes (calculated with an assumed walking speed of 100m/min). These attributes are required by *ArcView* for the creation of a network. This walk speed is probably somewhat fast, and modifications to scripts to adjust speed could be worthwhile.

The **centre** theme is a simple rectangle which encompasses all the stations in an area bounded by the M8 motorway to the north and west, by High Street to the east, and by the River Clyde to the south.

Bus Routes theme

This theme contains selected bus routes in the region. It is a line theme with one line feature for each bus route. Each line follows the course of a bus route along segments in the **walksnth** theme as closely as could be determined from published bus timetables. The digitising method adopted was to run the *ArcView* Shortest Path tool having input a sequence of points along the route. The resulting route theme was added into the **Bus Routes** theme by running a specially written script `AddBusRoute` which prompts the user for values for the following attributes:

"Route" is a decimal value with one decimal place allowed. This is a crude method for coping with multiple versions of the same base route number such as 12, 12A, 12B and so on, which would be represented as 12.0, 12.1, 12.2. This approach avoids constructing a very complex database of bus routes which seemed to be beyond the scope of this project. It also allows a simple method of generating IDs for bus route boarding points which are required for inclusion in **stns*** themes: the bus route number is multiplied by 1000. If more than one boarding point is found the resulting value is incremented. Thus ID numbers like 12000, 12001, 12002 are generated. Obviously this places a 100 boarding point limit on each route, but this is unlikely to be exceeded.

"Duration" is an integer value representing the bus route's timetabled running time in minutes.

"Frequency" is an integer value representing the weekday off-peak mean time in minutes between scheduled buses.

"Length" is a value indicating the length of each route in metres. This is unused.

Timetables published by FirstBus, Clydeside and Strathclyde Buses were used in the digitising. The *AZ Street Atlas* of *Glasgow Hamilton Motherwell Paisley* was invaluable in identifying streets along which buses run.

Train timetable data

The off-peak railway timetables are stored in `table1.txt` which has the following fields for each record:

"Route" is an integer ID number for each route. These numbers have meaning only in the context of this project. A route is a one-way sequence of station stops which runs during weekday off-peak hours.

"Stop" is an integer number indicating a particular station's position along a route - 1 for the first stop, 2, for the second and so on.

"Station" is an integer number which matches the ID number in the **stats2** theme. This is the link between the timetable (time) data and the station (location) data.

"Time" is an integer number between 0 and 59 which is the timetabled time in minutes past the hour at which this station is reached on this route.

"Last" is a Boolean flag indicating when a station is the last stop on this route. This flag is not used by the *ArcView* scripts described here.

"Used" is a flag indicating when a station stop has been checked to see if it provides an earlier arrival time at the station in question.

Note that the flags referred to above are not altered by the *ArcView* scripts - since the data is actually copied into a `Dictionary` object at the start of the `GenerateArrivalTimes` script.

B. Printouts of scripts

SpecifyArea

```
'-------------------------------------------------------------------------
' Script to accept rectangular area input from user and create two themes:
' 1. stns?.shp containg the stations in the specified area
' 2. area?.shp conttaining the rectangle
'-------------------------------------------------------------------------
theView = av.GetActiveDoc
theWDir = av.GetProject.getWorkDir

themeList = theView.GetThemes
theDisplay = theView.GetDisplay
areaRect = theDisplay.ReturnUserRect
area = areaRect.asPolygon

centre = theView.FindTheme( “centre.shp” )
centreZone = centre.ReturnExtent.asPolygon

while ( True )
  if ( area.Contains( centreZone ).Not ) then
    msgBox.Info( “Specified area must contain central zone”, “Invalid area” )
    break
  end

  statsTheme = theView.FindTheme( “stats2.shp” )

  statsTab = statsTheme.GetFTab

  statsTheme.SelectByPolygon( area, #VTAB_SELTYPE_NEW )
  scope = statsTab.GetSelection

  if ( scope.Count = 0 ) then
    msgBox.Info( “No features in selection area”, “” )
    break
  end

  fldOrigID = statsTab.FindField( “ID” )
  fldOrigShp = statsTab.FindField( “Shape” )

  idList = {}
  ptList = {}

  for each rec in statsTab.getSelection
    idList.Add( statsTab.ReturnValue( fldOrigID, rec ) )
    ptList.Add( statsTab.ReturnValue( fldOrigShp, rec ) )
  end

  fName = FileName.Make( theWDir.AsString ).MakeTmp( “stns”, “shp” )
  outName1 = FileDialog.Put( fName, “*.shp”, “Specify Stations theme File Name” )

  if ( outName1 = NIL ) then
    break
  end

  fName = FileName.Make( theWDir.AsString ).MakeTmp( “area”, “shp” )
  outName2 = FileDialog.Put( fName, “*.shp”, “Specify Area theme File Name” )

  if ( outName2 = NIL ) then
    break
  end

  selAreaTab = FTab.MakeNew( outName2, Polygon )

  if ( selAreaTab.StartEditingWithRecovery ) then
    fieldList = {}
    fieldList.Add( Field.Make( “Area”, #FIELD_DOUBLE, 16, 2 ) )
    selAreaTab.AddFields( fieldList )
    fldArea = selAreaTab.FindField( “Area” )
    fldRect = selAreaTab.FindField( “Shape” )

```

```
      rec = selAreaTab.AddRecord
      selAreaTab.SetValue( fldRect, rec, area )
      selAreaTab.SetValue( fldArea, rec, area.ReturnArea )
    end

    selAreaTab.StopEditingWithRecovery( True )

    selAreaTheme = FTheme.Make( selAreaTab )
    theView.AddTheme( selAreaTheme )
    selAreaTheme.setVisible( True )

    theDisplay.ZoomToRect( areaRect )

    posn = 0
    themeList = theView.GetThemes
    for each i in 0..( themeList.count - 1 )
      if ( themeList.Get( i ) = centre ) then
        posn = i + 1
      end
    end

    themeList.Shuffle( selAreaTheme, posn )
    theView.SetEditableTheme( NIL )
    selStnsTab = FTab.MakeNew( outName1, point )

    if ( selStnsTab.StartEditingWithRecovery ) then
      fieldList = {}
      fieldList.Add( Field.Make( "ID", #FIELD_LONG, 8, 0 ) )
      fieldList.Add( Field.Make( "Arrival", #FIELD_FLOAT, 8, 2 ) )
      fieldList.Add( Field.Make( "On_foot", #FIELD_LOGICAL, 8, 0 ) )
      selStnsTab.AddFields( fieldList )
      fldShp = selStnsTab.FindField( "Shape" )
      fldID = selStnsTab.FindField( "ID" )
      fldArr = selStnsTab.FindField( "Arrival" )
      fldOnFoot = selStnsTab.FindField( "On_foot" )

      for each i in 0..( idList.Count - 1 )
        id = idList.Get( i )
        pt = ptList.Get( i )

        recn = selStnsTab.AddRecord
        selStnsTab.SetValue( fldShp, recn, pt )
        selStnsTab.SetValue( fldID, recn, id )
        selStnsTab.SetValue( fldArr, recn, 1000.0 )
        selStnsTab.SetValue( fldOnFoot, recn, False )
      end
    end

    selStnsTab.StopEditingWithRecovery( True )

    selStnsTheme = FTheme.Make( selStnsTab )
    theView.AddTheme( selStnsTheme )
    selStnsTheme.setVisible( True )

    theView.SetEditableTheme( NIL )

    av.GetProject.SetModified( True )
    theDisplay.Invalidate( True )

    break
  end
```

SpecifyStPt

```
'-------------------------------------------------------------------------
' Script to accept point input from user and add it to a specified stns?.shp theme
'-------------------------------------------------------------------------
while ( True )
  theView = av.GetActiveDoc
  theWDir = av.GetProject.getWorkDir

  theDisplay = theView.GetDisplay
  startPoint = theDisplay.ReturnUserPoint

  _walkingRoutes = theView.FindTheme( “walksnth.shp” ).GetFTab
  _walkNetDef = NetDef.Make( _walkingRoutes )
  _walkNetwork = Network.Make( _walkNetDef )

  ' ------ Check point is near the street network
  if ( _walkNetwork.IsPointOnNetwork( startPoint ).Not ) then
    msgBox.Info( “Specified point is too far from road network”, “” )
    break
  end

  themeList = theView.GetThemes
  statsTheme = msgBox.Choice( themeList, “Specify the theme containing the stations:”,
   “Stations” )

  if ( statsTheme = NIL ) then
    break
  else
    areaTheme = msgBox.Choice( themeList, “Specify the area theme:”, “Area of
   interest” )

    if ( areaTheme = NIL ) then
      break
    else
      statsTab = statsTheme.GetFTab
      fldShp = statsTab.FindField( "Shape" )
      fldID = statsTab.FindField( "ID" )
      fldArr = statsTab.FindField( "Arrival" )
      fldOnFoot = statsTab.FindField( "On_foot" )

      if ( statsTab.GetShapeClass.GetClassName <> "Point" ) then
        msgBox.Info( "Specified stations theme must contain point features", "Invalid
           theme" )
        break
      else
        areaTab = areaTheme.GetFTab

        if ( areaTab.GetShapeClass.GetClassName <> "Polygon" ) then
          msgBox.Info( "Specified area theme must contain a rectangle", "Invalid
            theme" )
          break
        else
          areaTheme.SelectByShapes( { startPoint }, #VTAB_SELTYPE_NEW )
          inside = areaTab.GetSelection

          if ( inside.count <> 1 ) then
            msgBox.Info( "Specified point is not inside specified area theme", "" )
            break
          else
            areaTheme.ClearSelection

            if ( statsTab.StartEditingWithRecovery ) then
              statsTab.Query( "([ID] = 0)", statsTab.GetSelection, #VTAB_SELTYPE_NEW )

              if ( statsTab.GetSelection.Count = 0 ) then
                rec = statsTab.AddRecord
                statsTab.SetValue( fldShp, rec, startPoint )
                statsTab.SetValue( fldID, rec, 0 )
                statsTab.SetValue( fldArr, rec, 0.0 )
                statsTab.SetValue( fldOnFoot, rec, False )
              else
                for each rec in statsTab.GetSelection
```

```
                    statsTab.SetValue( fldShp, rec, startPoint )
                  end
                end

                statsTab.Query( "([ID] <> 0)", statsTab.GetSelection, #VTAB_SELTYPE_NEW
    )

                for each rec in statsTab.GetSelection
                  statsTab.SetValue( fldArr, rec, 1000.0 )
                  statstab.SetValue( fldOnFoot, rec, False )
                end
              end

              statsTab.StopEditingWithRecovery( True )
              theView.SetEditabletheme( NIL )
              statsTheme.ClearSelection
            end

            break
          end
        end
      end
    end
  end
end
```

GenerateArrivalTimes

```
1     ' ------ GenerateArrivalTimes
2     ' ------ Last changed 5 August 1997 version: Alpha1.1
3     ' ------ Script to determine arrival times at stations and critical points
4     ' ------ on bus routes for a chosen pre-selected part of the region
5
6     while ( True ) 'Whole script runs in a while loop to facilitate break conditions
7
8       ' ------ Initialisations
9       numLoops = 0
10      carryOn = True
11      cancelled = False
12
13      curStn = 0
14
15      useTrains = True
16      useBuses = True
17
18      unusedStns = 0
19
20      NUMSTATIONS = 100
21      NUMROUTES = 82
22      NUMBUSES = 100
23
24      theView = av.GetActiveDoc
25      themeList = theView.GetThemes
26
27      ' ------ Following are globals as network is large and passing it seems like
28      ' ------ asking for trouble!
29
30      ' ------ Set up walksnth.shp as the basis for the network analysis
31      _walkingRoutes = theView.FindTheme( "walksnth.shp" ).GetFTab
32      _walkNetDef = NetDef.Make( _walkingRoutes )
33      _walkNetwork = Network.Make( _walkNetDef )
34
35      costField = _walkNetDef.GetCostFields.Get( 1 ) 'uses MINUTES for the network costs
36      _walkNetwork.SetCostField( costField )
37
38      ' ------ Set vorsth.shp to be the station neighbourhoods theme
39      _neighbourhoods = theView.FindTheme( "vorsth.shp" )
40      _neighboursTab = _neighbourhoods.GetFTab
41
42      ' ------ Get the area of interest. This has been defined by running
         SpecifyAreaOfInterest
43      areaTheme = msgBox.Choice( themeList, "Specify the AREA theme:", "Area of interest"
         )
44
45      if ( areaTheme = NIL ) then
46        break
47      end
48
49      areaTab = areaTheme.GetFTab
50      if ( areaTab.GetShapeClass.GetClassName <> "Polygon" ) then
51        msgBox.Info( "Specified area theme must contain a rectangle", "Invalid theme" )
52        break
53      end
54
55      outline = areaTheme.ReturnExtent.asPolygon
56      buslimit = outline '.ReturnBuffered( 500 )(life's too short!) 'bus connections up to
         500m outside will be considered
57
58      ' ------ Set up user specified theme as the source stations theme and table
59      stations = msgBox.Choice( themeList, "Specify the theme containing the STATIONS:",
         "Stations" )
60      if ( stations = NIL ) then
61        break
62      end
63
64      stationsTab = stations.GetFTab
65      if ( stationsTab.GetShapeClass.GetClassName <> "Point" ) then
66        msgBox.Info( "Specified stations theme must contain points", "Invalid theme" )
67        break
```

```
68      end
69
70      fldShp = stationsTab.FindField( "Shape" )
71      fldID = stationsTab.FindField( "ID" )
72      fldArr = stationsTab.FindField( "Arrival" )
73      fldOnFoot = stationsTab.FindField( "On_foot" )
74
75      ' ------ Check that a start point is defined
76      startPtquery = stationsTab.GetSelection
77      stationsTab.Query( "([ID] = 0)", startPtquery, #VTAB_SELTYPE_NEW )
78      if ( startPtQuery.Count <> 1 ) then
79        msgBox.Info( "There is a problem with the start point in the stations theme you
         chose.", "Error" )
80        break
81      end
82
83      ' ------ warn that previous times will be erased if this script is run!
84      carryOn = msgBox.MiniYesNo( "Script will erase arrival times for these stations. Go
         ahead anyway?", True )
85      if ( carryOn.Not ) then
86        break
87      end
88
89      ' ------ Get a startTime from the user
90      while ( True )
91        strStartTime = msgBox.Input( "Enter a start time (in minutes past the hour):",
            "Start time", "0.0" )
92        if ( strStartTime = NIL ) then
93          cancelled = True
94          break
95        end
96        if ( strStartTime.IsNumber ) then
97          startTime = strStartTime.asNumber
98          if ( ( startTime < 0 ) or ( startTime > 59 ) ) then
99            msgBox.Info( "Value should be between 0 and 59", "" )
100         else
101           break
102         end
103       else
104         msgBox.Info( "Invalid Entry", "" )
105       end
106     end
107     if ( cancelled ) then
108       break
109     end
110     startTime = strStartTime.asNumber
111
112     ' ------ remove any bus boarding points added by previous runs NB bus boarding pts
                have
113     ' ------ IDs in the thousands - eg 12000, 12001, 12002 etc would be those on route
                12
114     busPts = stationsTab.getSelection
115     stationsTab.Query( "[ID] >= 1000", stationsTab.GetSelection, #VTAB_SELTYPE_NEW )
116
117     if ( stationsTab.StartEditingWithRecovery ) then
118       stationsTab.RemoveRecords( busPts )
119     end
120     stationsTab.StopEditingWithRecovery( True )
121
122     ' ------ Create a dictionary of lists each representing a station
123     ' ------ Each list is { pt, id, arr, used, onfoot } where:
124     ' ------    pt is the point location
125     ' ------    id is a unique identification number
126     ' ------         ( 0 for the start point, 1-999 for stations, 1000+ for bus boards )
127     ' ------    arr is the current best-estimate of the arrival time
128     ' ------    used flags that this station arrival time has now been calculated
129     ' ------    onfoot flags that arrival was on foot
130     ' ------  The station ID number is the dictionary key
131     stnDict = Dictionary.Make( NUMSTATIONS + 1 )
132
133     for each rec in stationsTab
134       unusedStns = unusedStns + 1 'the progress counter used in the main program loop
135
```

```
136       stnID = stationsTab.ReturnValue( fldID, rec )
137       pt = stationsTab.ReturnValue( fldShp, rec )
138
139       if ( stnID = 0 ) then 'arrival time already known and = startTime
140         stnDict.Add( stnID, { pt, stnID, startTime, False, False } )
141       else 'set arrival time to some high value
142         stnDict.Add( stnID, { pt, stnID, 1000.0, False, False } )
143       end
144     end
145
146     ' some users will be interested in buses or trains only access, so...
147     options = msgBox.ChoiceAsString( { "Buses only", "Trains only", "Both modes" },
          "Choose the combination of modes required", "" )
148     if ( options = NIL ) then
149       break
150     end
151     if ( options = "Buses only" ) then
152       useTrains = False
153     elseif ( options = "Trains only" ) then
154       useBuses = False
155     end
156
157     if ( useTrains.Not ) then
158       for each s in stnDict
159         if ( s.Get( 1 ) = 0 ) then 'start point is still relevant...
160           continue
161         else '...but other stations are not so set the used and onfoot flags
162           s.Set( 3, True )
163           s.Set( 4, True )
164           unusedStns = unusedStns - 1 'so main loop is executed correct number of times
165         end
166       end
167     else
168       ' ------ Set up table1.txt as the source of timetable data
169       timeTab = VTab.Make( ( "table1.txt" ).asFileName, False, False )
170       timeTab.Activate
171
172       if ( timeTab.HasError ) then 'this is irrecoverable so watch out for it!
173         msgBox.Info( "Error in timetable source table.", "Problem" )
174         break
175       end
176
177       fldRoute = timeTab.FindField( "Route" )
178       fldStop = timeTab.FindField( "Stop" )
179       fldTime = timeTab.FindField( "Time" )
180       fldStn_ID = timeTab.FindField( "Station_id" )
181       fldUsed = timeTab.FindField( "Visited" )
182
183       ' ------ Create a dictionary of lists representing train routes in the timetable
184       ' ------ Note that this data has no spatial content.
185       ' ------ Each route is indexed by a number, and represented by a list of stops
186       ' ------ each element of the stops list is itself a list as follows:
187       ' ------ { stnID, time, used }
188       ' ------    stnID is the station ID number
189       ' ------    time is the minutes past the hour of the arrival at that station
190       ' ------    used is a flag not used at present
191       ttDict = Dictionary.Make( NUMROUTES )
192
193       ' First create empty entries for all the stops in the timetable
194       for each rec in timeTab
195         routeNum = timeTab.ReturnValue( fldRoute, rec )
196         if ( ttDict.Get( routeNum ) = NIL ) then  ' If no entry for this route create
              one
197           ttDict.Add( routeNum, { routeNum, { -1, -1, False } } )
198
199         else  'otherwise add an item (stop) to the relevant list
200           ttDict.Get( routeNum ).Add( { -1, -1, False } )
201         end
202       end
203
204       for each rec in timeTab  'this time through the table set each entry to the stored
                                     value
205                               'but only for those stations in the selection - this
```

```
                                leaves an
206                           'ID of -1 for unselected stations, which is used
                                elsewhere
207       routeNum = timeTab.ReturnValue( fldRoute, rec )
208       stopNum = timeTab.ReturnValue( fldStop, rec )
209       stnID = timeTab.ReturnValue( fldStn_ID, rec )
210       time = timeTab.ReturnValue( fldTime, rec )
211       if ( stnDict.Get( stnID ) <> NIL ) then
212         ttDict.Get( routeNum ).Set( stopNum, { stnID, time, False } )
213       end
214     end
215   end
216
217   if ( useBuses ) then
218     ' ------ Another dictionary, this time for buses. Again each entry
219     ' ------ is a list: { routePath, rtNum, length, durn, freq, used } where:
220     ' ------     routePath is the shape representing the actual route over the street
                      network
221     ' ------     rtNum is the route number. This is a decimal, 12.0 is 12, 12.1 is 12A
                      etc
222     ' ------     length is the length of routePath in metres ... unused
223     ' ------     durn is the timetabled route duration in minutes
224     ' ------     freq is the timetabled time between buses in minutes
225     ' ------     used flags that the route has been used
226     busTheme = theView.FindTheme( "Bus Routes" )
227     busTab = busTheme.GetFTab
228     fldBusShp = busTab.FindField( "Shape" )
229     fldBusRoute = busTab.FindField( "Route" )
230     fldBusLength = busTab.FindField( "Length" )
231     fldBusDurn = busTab.FindField( "Duration" )
232     fldBusFreq = busTab.FindField( "Frequency" )
233
234     busDict = Dictionary.Make( NUMBUSES )
235
236     for each rec in busTab
237       routePath = busTab.ReturnValue( fldBusShp, rec )
238       rtNum = busTab.ReturnValue( fldBusRoute, rec )
239       length = busTab.ReturnValue( fldBusLength, rec )
240       durn = busTab.ReturnValue( fldBusDurn, rec )
241       freq = busTab.ReturnValue( fldBusFreq, rec )
242
243       busDict.Add( rtNum, { routePath, rtNum, length, durn, freq, False } )"
244     end
245   end
246
247   ' ------ Now get some journey parameters from the user.
248   while ( True )
249     mwtList = msgBox.MultiInput( "Maximum distances", "", { "to stations", "between
       stations", "to buses", "between buses" }, { "2000", "1500", "1000", "500" } )
250     if ( mwtList.Count = 0 ) then
251       cancelled = True
252       break
253     end
254     strToStnMax = mwtList.Get( 0 )
255     strBetweenStnMax = mwtList.Get( 1 )
256     strToBusMax = mwtList.Get( 2 )
257     strBetweenBusMax = mwtList.Get( 3 )
258     if ( strToStnMax.IsNumber and strBetweenStnMax.IsNumber and strToBusMax.IsNumber
                and strBetweenBusMax.IsNumber ) then
259       break
260     else
261       msgBox.Info( "Invalid entry", "" )
262     end
263   end
264   if ( cancelled ) then
265     break
266   end
267
268   toStnMax = strToStnMax.asNumber
269   betweenStnMax = strBetweenStnMax.asNumber
270   toBusMax = strToBusMax.asNumber
271   betweenBusMax = strBetweenBusMax.asNumber
272
```

```
' ------ Find and walk to stations near to start point
if ( useTrains ) then
  carryOn = av.Run( "CloseStations", { curStn, stnDict, toStnMax  } )
end
if ( carryOn.Not ) then 'Error encountered
  break
end

' ------ Find and walk to bus routes near to start
if ( useBuses ) then
  busOut = av.Run( "CloseBuses", { curStn, stnDict, toBusMax, busDict, unusedStns }
    )
  carryOn = busOut.Get( 0 )
  if ( carryOn.Not ) then 'Error encountered
    break
  end
  unusedStns = busOut.Get( 1 ) 'which changes to reflect bus board points added
end

' ------------------------
'        Main Program loop
' ------------------------

while( numLoops < ( NUMSTATIONS * 4 ) ) 'guards against infinite loop if something
 goes wrong
  numLoops = numLoops + 1
  av.SetStatus( ( numLoops / ( unusedStns + numLoops ) ) * 100 )

  if ( unusedStns = 1 ) then ' This is the correct terminating condition
    break
  end

  stnDict.Get( curStn ).Set( 3, True )
  unusedStns = unusedStns - 1

  ' ------ Find current earliest arrival time
  curStn = av.Run( "GetEarliestArrival", { curStn, stnDict } )
  if ( curStn = -1 ) then 'Error condition
    break
  end

  ' ------ Now for this point proceed by bus or train...

  if ( curStn >= 1000 ) then 'do the bus thing
    if ( useBuses ) then 'If trains only shouldn't get here, but just in case...
      busOut = av.Run( "TryBusesFromHere", { curStn, busDict, stnDict,
        betweenBusMax, toStnMax, busLimit, unusedStns } )
      carryOn = busOut.Get( 0 )
      if( carryOn.Not ) then 'Error encountered
        break
      end
      unusedStns = busOut.Get( 1 ) 'which changes to reflect added or superseded
                                      boarding pts
    end

  else
    if ( useTrains ) then 'If buses only shouldn't get here but just in case...

      ' ------ Get on trains that pass through here
      carryOn = av.Run( "TryTrainsFromThisStation", { curStn, stnDict, ttDict } )
      if ( carryOn.Not ) then 'Error encountered
        break
      end

      ' If didn't get here on foot then try walking to nearby buses and trains
      if ( stnDict.Get( curStn ).Get( 4 ).Not ) then

        ' ------ Walk to nearby stations
        carryOn = av.Run( "CloseStations", { curStn, stnDict, betweenStnMax } )
        if ( carryOn.Not ) then 'Error encountered
          break
        end

```

```
342             ' ------ Walk to nearby bus routes
343             if ( useBuses ) then
344               busOut = av.Run( "CloseBuses", { curStn, stnDict, betweenBusMax, busDict,
                     unusedStns } )
345               carryOn = busOut.Get( 0 )
346               if ( carryOn.Not ) then 'Error encountered
347                 break
348               end
349               unusedStns = busOut.Get( 1 )
350             end
351           end
352         end
353       end
354     end
355
356     'Keep the next line around for convenience debugging
357     'msgBox.Info( "numLoops = "++numLoops.asString, "" )
358
359     ' ------ Now update the stations table with the arrival times determined
360     if ( stationsTab.StartEditingWithRecovery ) then
361       ' First all the actual stations
362       for each rec in stationsTab
363         id = stationsTab.ReturnValue( fldID, rec )
364         arrTime = stnDict.Get( id ).Get( 2 ) - startTime
365         onFoot = stnDict.Get( id ).Get( 4 )
366         stationsTab.SetValue( fldArr, rec, arrTime )
367         stationsTab.SetValue( fldOnFoot, rec, onFoot )
368       end
369
370       ' Then add points for bus route boarding points
371       if ( useBuses ) then
372       for each s in stnDict
373           if ( s.Get( 1 ) < 1000 ) then
374             continue
375           else
376             id = s.Get( 1 )
377             recn = stationsTab.AddRecord
378             stationsTab.SetValue( fldShp, recn, stnDict.Get( id ).Get( 0 ) )
379             stationsTab.SetValue( fldID, recn, id )
380             stationsTab.SetValue( fldArr, recn, ( stnDict.Get( id ).Get( 2 ) - startTime
                  ) )
381             stationsTab.SetValue( fldOnFoot, recn, stnDict.Get( id ).Get( 4 ) )
382           end
383         end
384       end
385     end
386
387     stationsTab.StopEditingWithRecovery( True )
388     theView.SetEditableTheme( NIL )
389
390     break 'Only run once! The while loop is to allow break for error conditions
391   end
```

CloseStations

```
1     ' ------ CloseStations
2     ' ------ Last changed 4 August 1997 version: Alpha1.01
3     ' ------ Script to find nearby stations and walk to them
4     ' ------ updating their arrival times if they are improved
5
6     ' ------ call looks like this: ( "CloseStations", { curStn, stnDict, toStnMax  } )
7
8     if( self.Is(List).Not ) then
9       msgBox.Info( "Bad call 1 to CloseStations", "" )
10      return False
11    elseif( self.Count <> 3 ) then
12      msgBox.Info( "Bad call 2 to CloseStations", "" )
13      return False
14    else
15      stn = self.Get( 0 )               'The current station ID
16      sDict = self.Get( 1 )             'The dictionary with all the station details
17      maxCrowDist = self.Get( 2 )       'maximum distance away walking will be attempted
18    end
19
20    start = sDict.Get( stn ).Get( 0 )    'station location
21    time = sDict.Get( stn ).Get( 2 )     'current arrival time
22
23    ' ------ _neighbourhoods is a set of Voronoi/Thiessen polygons each centred on a
                station.
24    ' ------ They are modified so that the Clyde is excluded from their combined coverage.
25    fldShp = _neighboursTab.FindField( "Shape" )
26
27    ' ------ Select the Voronoi pgon of the current station, and all its immediate
                neighbours
28    _neighbourhoods.SelectByShapes( { start }, #VTAB_SELTYPE_NEW )
29    _neighbourhoods.SelectByTheme( _neighbourhoods, #FTAB_RELTYPE_ISWITHINDISTANCEOF,
         0.0,#VTAB_SELTYPE_OR )
30
31    searchArea = Polygon.MakeNull
32
33    for each rec in _neighboursTab.GetSelection
34      vorPoly = _neighboursTab.ReturnValue( fldShp, rec )
35      searchArea = searchArea.ReturnUnion( vorPoly )
36    end
37
38    _neighbourhoods.ClearSelection
39
40    ' ------ Create a list of those stations which are in the area
41    closeStnsList = {}
42    for each possStn in sDict
43      stnPos = possStn.Get( 0 )
44      stnID = possStn.Get( 1 )
45      if ( stnID >= 1000 ) then 'its a bus stop not a station
46        continue
47      elseif ( searchArea.Contains( stnPos ) ) then
48        closeStnsList.Add( stnID )
49      end
50    end
51
52    ' ------ Go through the selected stations
53    for each id in closeStnsList
54
55      if ( sDict.Get( id ).Get( 4 ).Not ) then   ' only proceed if this station
56                                                 ' has not been reached on foot already
57        stnPos = sDict.Get( id ).Get( 0 )
58
59        ' Check its inside the allowable distance
60        dist = start.Distance( stnPos )
61        if ( dist < maxCrowDist ) then
62
63          ' calculate the potential arrival time
64          walkTime = _walkNetwork.FindPathCost( { start, stnPos }, False, False )
65          possArrTime = time + walkTime
66
67          if ( possArrTime < sDict.Get( id ).Get( 2 ) ) then 'sooner than the current
                                                                value
```

```
68            sDict.Get( id ).Set( 2, possArrTime )
69            sDict.Get( id ).Set( 4, True )
70          end
71        end
72      end
73    end
74
75    return True
```

CloseBuses

```
1     ' ------ CloseBuses
2     ' ------ Last changed 3 August 1997 version: Alpha1.0
3     ' ------ Script to find nearby bus routes and walk to them
4     ' ------ Boarding points are added to or removed from the
5     ' ------ stations dictionary as required
6
7     ' ------ call looks like this:
8     ' ------ ( "CloseBuses", { curStn, stnDict, toBusMax, busDict, unusedStns } )
9
10    if ( self.Is( List ).Not ) then
11      msgBox.Info( "Error 1 in CloseBuses", "" )
12      return { False, 1 }
13    elseif ( self.Count <> 5 ) then
14      msgBox.Info( "Error 2 in CloseBuses", "" )
15      return { False, 1 }
16    else
17      stn = self.Get( 0 )            ' ID of the boarding point
18      sDict = self.Get( 1 )          ' the station dictionary
19      distLimit = self.Get( 2 )      ' the furthest distance allowed walking to a bus
          route
20      bDict = self.Get( 3 )          ' the bus route dictionary
21      numStns = self.Get( 4 )        ' the number of stations (includes bus board pts)
                                         remaining
22    end
23
24    startPoint = sDict.Get( stn ).Get( 0 )    ' location of the current station
25    startTime = sDict.Get( stn ).Get( 2 )     ' time of arrival at current station
26
27    theView = av.GetActiveDoc
28    theDisplay = theView.GetDisplay
29
30    ' ------ Go through the bus route dictionary checking closest approaches against the
                limit
31    for each r in bDict
32      routePath = r.Get( 0 )
33      busRtNum = r.Get( 1 )
34
35      distToRoute = startPoint.Distance( routePath )
36      if ( distToRoute > distLimit ) then 'too far
37        continue
38      else 'find a point of intersection
39        walkArea = circle.Make( startPoint, ( distToRoute + 5 ) )
40        boardPt = walkArea.PointIntersection( routePath ).asList.Get( 0 )
41
42        ' ------ Work out the boarding time
43        atTime = startTime + ( 0.5 * r.Get( 4 ) ) + _walkNetwork.FindPathCost( { boardPt,
            startPoint }, False, False )
44
45        boardPtList = { boardPt, atTime }
46
47        ' ------ now check it against already stored boarding points, altering as required
48        numStns = av.Run( "AddBoardPtToBusRoute", { { boardPtList }, busRtNum, numStns,
            sDict, bDict } )
49
50      end
51    end
52
53    return { True, numStns }
```

GetEarliestArrival

```
1     ' ------ GetEarliestArrival
2     ' ------ Last changed 4 August 1997 version: Alpha1.01
3     ' ------ Script to return the ID of the station currently reached soonest
4
5     ' ------ call looks like this: ( "GetEarliestArrival", { curStn, stnDict } )
6
7     if ( self.Is(List).Not ) then
8       msgBox.Info( "Invalid argument passed 1 in GetEarliestArrival ", "" )
9       return -1
10    elseif ( self.Count <> 2 ) then
11      msgBox.Info( "Invalid argument passed 2 in GetEarliestArrival ", "" )
12      return -1
13    else
14      stn = self.Get( 0 )
15      sDict = self.Get( 1 )
16    end
17
18    first = 2000.0
19    time = sDict.Get( stn ).Get( 2 )
20
21    for each s in sDict
22      if ( s.Get( 3 ).Not ) then
23        thisStn = s.Get( 1 )
24        thisTime = s.Get( 2 )
25        if ( thisTime < first ) then
26          first = thisTime
27          firstStn = thisStn
28        end
29      end
30    end
31
32    return firstStn
```

TryTrainsFromThisStation

```
1     ' ------ TryTrainsFromThisStation
2     ' ------ Last changed 12 August 1997 version: Alpha1.2
3     ' ------ Script to try trains from here
4
5     ' ------ call looks like this:
6     ' ------ ( "TryTrainsFromThisStation", { curStn, stnDict, ttDict } )
7
8     if( self.Is(List).Not ) then
9       msgBox.Info( "Bad call 1 to TTFTS", "" )
10      return False
11    elseif( self.Count <> 3 ) then
12      msgBox.Info( "Bad call 2 to TTFTS", "" )
13      return False
14    else
15      stn = self.Get( 0 )
16      sDict = self.Get( 1 )
17      ttDict = self.get( 2 )
18    end
19
20    for each route in ttDict
21      numStops = route.Count - 1
22      stopsAtStn = False
23      hourOffset = 0
24      clockOffset = 0
25      prevArrTime = 0
26
27      for each stp in 1..numStops
28        thisStn = route.Get( stp ).Get( 0 )
29
30        if ( ( thisStn <> -1 ) ) then
31          ttArrTime = route.Get( stp ).Get( 1 )
32          curArrTime = sDict.Get( thisStn ).Get( 2 )
33
34          if ( stopsAtStn.Not ) then 'not found this stn on the route yet so check for it
35
36            if ( thisStn = stn ) then 'stn is on route so...
37              stopsAtStn = True '...flag the fact, and...
38              if ( ttArrTime < curArrTime ) then '...set an offset if necessary
39                hourOffset = 60
40              end
41            end
42
43          else 'stn is on this route
44            if ( ttArrTime < prevArrTime ) then 'rounded the hour, so another offset is
                                                    reqd
45              clockOffset = 60
46            end
47
48            possArrTime = hourOffset + clockOffSet + ttArrTime
49
50            if ( possArrTime < curArrTime ) then 'better than previous, so update
51              sDict.Get( thisStn ).Set( 2, possArrTime )
52              sDict.Get( thisStn ).Set( 4, False ) 'Not on foot
53            end
54          end
55
56          prevArrTime = ttArrTime 'for checking if hour has passed
57        end
58      end
59    end
60
61    return True
```

TryBusesFromHere

```
1     ' ------ TryBusesFromHere
2     ' ------ Last changed 4 August 1997 version: Alpha1.01
3     ' ------ Script to try connections to other bus routes and
4     ' ------ stations accessible to this bus route, modifying the
5     ' ------ stations dictionary as required
6
7     ' ------ call looks like this:
8     ' ------ ( "TryBusesFromHere", { curStn, busDict, stnDict, betweenBusMax, toStnMax,
                  busLimit, unusedStns } )
9
10    if ( self.Is( List ).Not ) then
11      msgBox.Info( "Error 1 in TryBusesFromHere", "" )
12      return { False, 1 }
13    elseif ( self.count <> 7 ) then
14      msgBox.Info( "Error 2 in TryBusesFromHere", "" )
15      return { False, 1 }
16    else
17      startID = self.Get( 0 )      ' id of bus stop in station dictionary
18      bDict = self.Get( 1 )        ' bus route dictionary
19      sDict = self.Get( 2 )        ' station dictionary
20      bBusMax = self.Get( 3 )      ' max dist between bus routes
21      tStnMax = self.Get( 4 )      ' max dist to a station
22      extent = self.Get( 5 )       ' buffered rectangle inside which buses will be
         considered
23      uuStns = self.Get( 6 )       ' number of unused stations remaining
24    end
25
26    stPt = sDict.Get( startID ).Get( 0 )
27    stTime = sDict.Get( startID ).Get( 2 )
28
29    busRtNum = ( startID - ( startID Mod 100 ) ) / 1000
30
31    routePath = bDict.Get( busRtNum ).Get( 0 )
32    routeDurn = bDict.Get( busRtNum ).Get( 3 )
33    routeFreq = bDict.Get( busRtNum ).Get( 4 )
34
35    startPercent = routePath.PointPosition( stPt )
36    routeInAOI = routePath.ReturnIntersection( extent )
37    stnSearch = routeInAOI.ReturnBuffered( tStnMax )
38    busSearch = routeInAOI.ReturnBuffered( bBusMax )
39
40    ' ------ first check for journeys to railway stations
41    for each pt in sDict
42      if ( pt.Get( 3 ) ) then 'already done forget about it
43        continue
44      elseif ( pt.Get( 1 ) < 1000 ) then 'only interested in stations not bus stops
45        stnPos = pt.Get( 0 )
46        if ( stnSearch.Contains( stnPos ) ) then 'possible journey
47          dist = routePath.Distance( stnPos ) 'closest approach distance
48          walkArea = circle.Make( stnPos, ( dist + 5 ) )
49          alightPt = routePath.PointIntersection( walkArea ).asList.Get( 0 ) 'where youd
                                                                                  get off the bus
50          alightPercent = routePath.PointPosition( alightPt )
51          diff = ( alightPercent - startPercent ).Abs
52          journeyTime = routeDurn * ( diff / 100 ) 'approximate journey time
53          possArr = stTime + journeyTime + _walkNetwork.FindPathCost( { alightPt, stnPos
               }, False, False )
54          if ( possArr < pt.Get( 2 ) ) then 'its earlier than current so update
55            pt.Set( 2, possArr )
56            pt.Set( 4, True ) 'Arrived on foot
57          end
58        end
59      end
60    end
61
62    ' ------ then check the connections with other bus routes
63    for each rt in bDict
64      if ( rt.Get( 1 ) = busRtNum ) then 'bus route different from this one
65        continue
66      end
67
```

```
68      targetRtNum = rt.Get( 1 )
69      tgtRtPath = rt.Get( 0 )
70
71      if ( busSearch.Intersects( tgtRtPath ) ) then
72
73        possBoards = {} 'start building a list of candidate connections
74
75        ' Get the segments of the target route which are near the current one
76        segsList = tgtRtPath.LineIntersection( busSearch ).asList
77
78        for each seg in segsList
79          ' Get the closest point - this should really be more sophisticated
80          minDist = bBusMax
81          for each pt in seg
82            dist = pt.Distance( routePath )
83            if ( dist < minDist ) then
84              minDist = dist
85              minPt = pt
86            end
87          end
88          walkArea = circle.Make( minPt, ( minDist + 10 ) )
89          alightPt = walkArea.PointIntersection( routePath ).asList.Get( 0 )
90          diff = ( startPercent - routePath.PointPosition( alightPt ) ).Abs
91          journeyTime = routeDurn * ( diff / 100 )
92          possArr = stTime + journeyTime + ( 0.5 * rt.Get( 4 ) ) +
               _walkNetwork.FindPathCost( { alightPt, minPt }, False, False )
93          possBoards.Add( { minPt, possArr } )
94        end
95        uuStns = av.Run( "AddBoardPtToBusRoute", { possBoards, targetRtNum, uuStns, sDict,
            bDict } )
96        if ( uuStns = -1 ) then
97          return { False, 1 }
98        end
99      end
100   end
101
102   return { True, uuStns }
```

AddBoardPtToBusRoute

```
1     ' ------ AddBoardPtToBusRoute
2     ' ------ Last changed 4 August 1997 version: Alpha1.01
3     ' ------ Script attempts to add a set of possible boarding points to
4     ' ------ those already stored for a route. Script checks if the new boarding
5     ' ------ points improve on the previous one and modifies the station dictionary
                 accordingly
6
7     ' ------ call looks like this:
8     ' ------ ( "AddBoardPtToBusRoute", { { boardPtList }, busRtNum, numStns, sDict, bDict
                 } )
9     if ( self.Is( List ).Not ) then
10      msgBox.Info( "Error 1 in AddBdPtToBusRoute", "" )
11      return -1
12    elseif ( self.Count <> 5 ) then
13      msgBox.Info( "Error 2 in AddBdPtToBusRoute", "" )
14      return -1
15    else
16      bPtList = self.Get( 0 )      ' list of lists { pt, time }
17      rtNum = self.Get( 1 )        ' the bus route number
18      uuStns = self.Get( 2 )       ' current number of unused stations
19      sDict = self.Get( 3 )        ' station dictionary
20      bDict = self.Get( 4 )        ' bus route dictionar
21    end
22
23    rtPath = bDict.Get( rtNum ).Get( 0 )
24    rtDurn = bDict.Get( rtNum ).Get( 3 )
25
26    ' ------ Make a dictionary from the cuurently store boarding points
27    prevBoards = Dictionary.Make( 10 )
28    firstID = rtNum * 1000
29    lastID = firstID + 99
30    for each id in firstID..lastID
31      if ( sDict.Get( id ) <> NIL ) then
32        bdPosn = sDict.Get( id ).Get( 0 )
33        bdTime = sDict.Get( id ).Get( 2 )
34        prevBoards.Add( id, { bdPosn, bdtime, id } )
35      end
36    end
37
38    dummyID = 1000000 'temporary ID for new boarding pt which won't interfere with real
                        ones
39    betterBoards = {} 'List which will contain the boarding points which improve on
                        existing ones
40
41    for each possBd in bPtList
42      better = True                       'Assume its better until proven otherwise
43      prevToRemove = {}                   'List of IDs of boarding points which will be
         replaced
44      if ( prevBoards.Count <> 0 ) then   'No previous, skip all this
45
46        for each prev in prevBoards 'Check against each of the previously stored boarding
                                       pts
47          id = prev.Get( 2 )
48          prevTime = prev.Get( 1 ) ' boarding time at previous point
49          prevPercent = rtPath.PointPosition( prev.Get( 0 ) )   ' its location along the
                                                                    route
50          thisPercent = rtPath.PointPosition( possBd.Get( 0 ) ) ' the new location along
                                                                    the route
51          diff = ( thisPercent - prevPercent ).Abs  ' ...the difference...
52          journeyTime = rtDurn * ( diff / 100 )  ' ...hence an approximate time to get
                                                     here from there
53          prevArrTime = prevTime + journeyTime ' so an arrival time here from there
54          possArrtime = possBd.Get( 1 ) ' the new arrival time
55
56          improvement = prevArrTime - possArrTime
57
58          if ( improvement > journeyTime ) then    'Much better so the previous superseded
59            prevToRemove.Add( id )
60          elseif ( improvement < 0 ) then          'worse, so falsify assumption that its
                                                       better
61            better = False
```

```
62          end
63        end
64      end
65
66      if ( better ) then 'boarding point currently under consideration to be added
67        betterBoards.Add( possBd )
68        newPoss = possBd.Add( dummyID )
69        prevBoards.Add( dummyID, newPoss ) 'also add it to the previous dictionary for
                                                comparison
70        dummyID = dummyID + 1 'and generate another dummy ID
71      end
72
73      for each item in prevToRemove 'remove any superseded boarding points
74        prevBoards.Remove( item ) 'from prevBoards...
75        if ( sDict.Get( item ) <> NIL ) then
76          sDict.Remove( item ) '...and from the stations Dictionary
77          uuStns = uuStns - 1
78        end
79      end
80    end
81
82    ' ------ Add all the better boarding points in after all have been considered
83    ' ------ This makes it easier to keep the IDs in reasonable shape
84    busStopID = rtNum * 1000 'start from x000
85    for each b in betterBoards
86      while ( sDict.Get( busStopID ) <> NIL ) 'find the first available
87        busStopID = busStopID + 1
88      end
89      sDict.Add( busStopID, { b.Get( 0 ), busStopID, b.Get( 1 ), False, True } )
90      uuStns = uuStns + 1
91    end
92
93    return uuStns
```

IsochroneMainLoop

```
1     ' ------ IsochroneMainLoop
2     ' ------ Last changed 6 August 1997 version: Alpha1.2
3     ' ------ Script to determine arrival times at stations and critical points
4     ' ------ on bus routes for a chosen pre-selected part of the region
5
6     while ( True )
7
8       ' ------Set the active view to the current one
9       curStn = 0
10      startTime = 0.0
11
12      NUMSTATIONS = 100
13      NUMBUSES = 100
14
15      theView = av.GetActiveDoc
16      themeList = theView.GetThemes
17      wDir = av.GetProject.GetWorkDir
18
19      ' ------ Set up walksnth.shp as the basis for the network analysis
20      _walkingRoutes = theView.FindTheme( "walksnth.shp" ).GetFTab
21      _walkNetDef = NetDef.Make( _walkingRoutes )
22      _walkNetwork = Network.Make( _walkNetDef )
23
24      costField = _walkNetDef.GetCostFields.Get( 1 )
25      _walkNetwork.SetCostField( costField )
26
27
28      statsTheme = msgBox.Choice( themeList, "Specify the theme containing the STATIONS:",
         "Stations" )
29      if ( statsTheme = NIL ) then
30        break
31      end
32      strStations = statsTheme.GetName
33
34      areaTheme = msgBox.Choice( themeList, "Specify the AREA theme:", "Area of interest"
         )
35      if ( areaTheme = NIL ) then
36        break
37      end
38      strArea = areaTheme.GetName
39
40      outline = areaTheme.ReturnExtent
41
42      ' -------------------------------------------------
43      ' Set up selected theme as the stations theme and table
44      ' -------------------------------------------------
45      statsTab = statsTheme.GetFTab
46      fldShp = statsTab.FindField( "Shape" )
47      fldArr = statsTab.FindField( "Arrival" )
48      fldID = statsTab.FindField( "ID" )
49      fldOnFoot = statsTab.FindField( "On_foot" )
50
51      ' ------ Create a dictionary of lists each representing a station
52      stnArrDict = Dictionary.Make( NUMSTATIONS + 1 )
53
54      for each rec in statsTab
55        arr = statsTab.ReturnValue( fldArr, rec )
56        pt = statsTab.ReturnValue( fldShp, rec )
57        onFoot = statsTab.ReturnValue( fldOnFoot, rec )
58        stnID = statsTab.ReturnValue( fldID, rec )
59        stnArrDict.Add( stnID, { Arr, pt, onFoot, stnID } )
60      end
61
62      busTheme = theView.FindTheme( "Bus Routes" )
63      busTab = busTheme.GetFTab
64      fldBusShp = busTab.FindField( "Shape" )
65      fldBusNum = busTab.FindField( "Route" )
66      fldBusDurn = busTab.FindField( "Duration" )
67
68      ' ------ Another dictionary for the bus routes
69      busDict = Dictionary.Make( NUMBUSES )
```

```
    for each rec in busTab
      rtNum = busTab.ReturnValue( fldBusNum, rec )
      busStopID = rtNum * 1000
      if ( stnArrDict.Get( busStopID ) <> NIL ) then
        busDict.Add( rtNum, { busTab.ReturnValue( fldBusShp, rec ), busTab.ReturnValue(
           fldBusDurn, rec ) } )
      end
    end

    strMethod = msgBox.ChoiceAsString( { "Quick for both modes", "Quick Bus method",
     "Standard method" }, "Choose method of evaluation", "" )
    if ( strMethod = NIL ) then
      break
    end

    cancelled = False
    while ( True )
      reqdIso = msgBox.Input( "Enter time in minutes for required isochrone", "Build
        Isochrone", "" )
      if ( reqdIso = NIL ) then
        cancelled = True
        break
      end
      if ( reqdIso.isNumber ) then
        break
      else
        msgBox.Info( "Please enter a number", "" )
      end
    end
    if ( cancelled ) then
      break
    end

    strComm = msgBox.Input( "Enter any comments for the theme properties dialogue:",
     "Comments", "Created by:  Purpose:" )
    if ( strComm = NIL ) then
      break
    end

    isoList = { reqdIso.asNumber }
    'isoList = { 75, 60, 45 }
    maxWalk = 20.0

    for each Isochrone in isoList
      if ( strMethod = "Quick Bus method" ) then
        av.Run( "QuickBusIsochrone", { isochrone, stnArrDict, maxWalk, outline, busDict,
           strStations, strArea, strComm } )
      elseif ( strMethod = "Standard method" ) then
        av.Run( "Construct", { isochrone, stnArrDict, maxWalk, outline, busDict,
           strStations, strArea, strComm } )
      else
        av.Run( "QuickBoth", { isochrone, stnArrDict, maxWalk, outline, busDict,
           strStations, strArea, strComm } )
      end
    end
    break
  end
```

Construct

```
1     ' ------ Construct
2     ' ------ Last changed 12 August 1997 version: Alpha1.1
3     ' ------ Script to build isochrone based on a set of arrival times
4     ' ------ in a "station" theme
5
6     ' ------ av.Run( "Construct", { isochrone, stnArrDict, maxWalk, outline, busDict,
                                      strStations, strArea, strComm } )
7
8     if ( self.Is( List ).Not ) then
9       msgBox.Info( "Error 1 in Construct", "Error" )
10      return False
11    elseif ( self.Count <> 8 ) then
12      msgBox.Info( "Error 2 in Construct", "Error" )
13      return False
14    else
15      time = self.Get( 0 )
16      arrTimes = self.Get( 1 )
17      maxWalk = self.Get( 2 )
18      extent = self.Get( 3 )
19      busRoutes = self.Get( 4 )
20      strS = self.Get( 5 )
21      strA = self.Get( 6 )
22      strC = self.Get( 7 )
23    end
24
25    theView = av.GetActiveDoc
26    theDisplay = theView.GetDisplay
27    wDir = av.GetProject.GetWorkDir
28    clyde = theView.FindTheme( "Clyde.shp" )
29
30    outName = FileName.Make( wDir.asString ).MakeTmp( "iso"++time.asString, "shp" )
31    isoShape = Ftab.MakeNew( outName, polygon )
32    isoTheme = FTheme.Make( isoShape )
33
34    isoTheme.SetComments( "This isochrone generated from "++strS++" and "++strA++" themes.
         Extent is "++time.asString++" minutes. " + strC )
35
36    isoThemeTab = isoTheme.GetFTab
37    fldList = { Field.Make( "Time", #FIELD_BYTE, 8, 0 ), Field.Make( "Area",
         #FIELD_DOUBLE, 16, 2 ) }
38    isoThemeTab.AddFields( fldList )
39
40    fldShp = isoThemeTab.FindField( "Shape" )
41    fldTime = isoThemeTab.FindField( "Time" )
42    fldArea = isoThemeTab.FindField( "Area" )
43
44    theView.AddTheme( isoTheme )
45
46    for each item in arrTimes
47      arr = item.Get( 0 )
48      if ( arr < time ) then
49
50        ptList = {}
51        areaList = {}
52        id = item.get( 3 )
53        if ( id < 1000 ) then
54
55          onFoot = item.Get( 2 )
56          if ( ( ( onFoot = False ) or ( id = 0 ) ) and ( arr < 5000 ) ) then
57            ptList.Add( item.Get( 1 ) )
58            areaExtent = time - arr
59            areaExtent = areaExtent.Min( maxWalk )
60            areaList.Add( { areaExtent } )
61          end
62        else
63
64          rtNum = ( id - ( id Mod 100 ) ) / 1000
65          route = busRoutes.Get( rtNum ).Get( 0 )
66          busStop = item.Get( 1 )
67          bsPercent = route.PointPosition( busStop )
68
```

```
69          durn = busRoutes.Get( rtNum ).Get( 1 )
70          distInOneMin = 100 / durn
71
72          pt1 = Point.MakeNull
73          pt2 = Point.MakeNull
74          arr = arr.Round
75          fromTime = arr.Ceiling - ( arr.Ceiling mod 2 )
76          toTime = time.Floor - ( time.Floor mod 2 )
77          for each t in fromTime..toTime by 2
78            pct1 = bsPercent + ( ( t - arr ) * distInOneMin )
79            pct2 = bsPercent - ( ( t - arr ) * distInOneMin )
80            pct1 = pct1.Min( 100 )
81            pct2 = pct2.Max( 0 )
82            pt1 = route.Along( pct1 )
83            pt2 = route.Along( pct2 )
84            ptList.Add( pt1 )
85            ptList.Add( pt2 )
86            areaExtent = time - t
87            areaExtent = areaExtent.Min( maxWalk )
88            areaList.Add( { areaExtent } )
89            areaList.Add( { areaExtent } )
90
91
92          end
93        end
94
95
96        _walkNetwork.FindServiceArea( ptList, areaList, False, True )
97        if ( _walkNetwork.HasServiceAreaResult ) then
98          isonet = ( "isonet.shp" ).asFileName
99          isoarea = ( "isoarea.shp" ).asFileName
100         _walkNetwork.WriteServiceArea( isonet, isoarea )
101         File.Delete( isonet )
102         tmpIsoSrc = SrcName.Make( isoarea.asString )
103         tmpIsoTheme = Theme.Make( tmpIsoSrc )
104         tmpIsoTheme2 = tmpIsoTheme.Clone
105
106         theView.AddTheme( tmpIsoTheme2 )
107         theView.SetEditableTheme( tmpIsoTheme2 )
108
109         isoSelRect = tmpIsoTheme2.ReturnExtent
110         tmpIsoTheme2.SelectByPolygon( isoSelRect.AsPolygon, #VTAB_SELTYPE_NEW )
111         tmpIsoTheme2.CopySelected
112
113         tmpIsoTheme2.StopEditing( False )
114
115         theView.SetEditableTheme( isoTheme )
116         isoTheme.Paste
117         isoTheme.StopEditing( True )
118
119         theView.DeleteTheme( tmpIsoTheme2 )
120         File.Delete( isoarea )
121       end
122
123
124     end
125   end
126
127   theView.SetEditableTheme( isoTheme )
128   selRect = isoTheme.ReturnExtent
129   isoTheme.SelectByPolygon( selRect.AsPolygon, #VTAB_SELTYPE_NEW )
130   isoTheme.UnionSelected
131   isoTheme.SetVisible( True )
132   for each rec in isoThemeTab
133     poly = isoThemeTab.ReturnValue( fldShp, rec)
134     poly = poly.ReturnClipped( extent )
135     isoThemeTab.SetValue( fldShp, rec, poly )
136     isoThemeTab.SetValue( fldTime, rec, time )
137     isoThemeTab.Setvalue( fldArea, rec, ( poly.ReturnArea / 10000 ) )
138   end
139
140   symIso = Symbol.Make( #SYMBOL_FILL )
141   symIso.SetColor( Color.GetGreen )
```

```
142   symIso.SetOLColor( Color.GetWhite )
143   symIso.SetOLWidth( 1 )
144   legIso = isoTheme.GetLegend
145   legIso.SingleSymbol
146   legIso.SetClassInfo( 0, { time.asString++" mins", time.asString, symIso, time, time }
         )
147
148   isoTheme.StopEditing( True )
149   isoTheme.ClearSelection
150
151   theDisplay.Invalidate( True )
152
153   theView.SetEditableTheme( NIL )
154
155   return True
```

QuickBoth

```
1     ' ------ QuickBoth
2     ' ------ Last changed 12 August 1997 version: Alpha1.1
3     ' ------ Script to build isochrone based on a set of arrival times
4     ' ------ in a "station" theme
5
6     ' ------ av.Run( "Construct", { isochrone, stnArrDict, maxWalk, outline, busDict,
                                      strStations, strArea, strComm } )
7
8     if ( self.Is( List ).Not ) then
9       msgBox.Info( "Error 1 in Construct", "Error" )
10      return False
11    elseif ( self.Count <> 8 ) then
12      msgBox.Info( "Error 2 in Construct", "Error" )
13      return False
14    else
15      time = self.Get( 0 )
16      arrTimes = self.Get( 1 )
17      maxWalk = self.Get( 2 )
18      extent = self.Get( 3 )
19      busRoutes = self.Get( 4 )
20      strS = self.Get( 5 )
21      strA = self.Get( 6 )
22      strC = self.Get( 7 )
23    end
24
25    theView = av.GetActiveDoc
26    theDisplay = theView.GetDisplay
27    wDir = av.GetProject.GetWorkDir
28    clyde = theView.FindTheme( "Clyde.shp" )
29    north = theView.FindTheme( "north.shp" )
30    south = theView.FindTheme( "south.shp" )
31    northTab = north.GetFTab
32    southTab = south.GetFTab
33    fldNorth = northTab.FindField( "Shape" )
34    fldSouth = southTab.FindField( "Shape" )
35    for each rec in northTab
36      polyNth = northTab.ReturnValue( fldNorth, rec )
37      polyNth = polyNth.ReturnClipped( extent )
38    end
39    for each rec in southTab
40      polySth = southTab.ReturnValue( fldSouth, rec )
41      polySth = polySth.ReturnClipped( extent )
42    end
43
44    outName = FileName.Make( wDir.asString ).MakeTmp( "iso"++time.asString, "shp" )
45    isoShape = Ftab.MakeNew( outName, polygon )
46    isoTheme = FTheme.Make( isoShape )
47
48    isoTheme.SetComments( "This isochrone generated from "++strS++" and "++strA++" themes.
                             Extent is "++time.asString++" minutes. " + strC )
49
50    isoThemeTab = isoTheme.GetFTab
51    fldList = { Field.Make( "Time", #FIELD_BYTE, 8, 0 ), Field.Make( "Area",
                                    #FIELD_DOUBLE, 16, 2 ) }
52    isoThemeTab.AddFields( fldList )
53
54    fldShp = isoThemeTab.FindField( "Shape" )
55    fldTime = isoThemeTab.FindField( "Time" )
56    fldArea = isoThemeTab.FindField( "Area" )
57
58    theView.AddTheme( isoTheme )
59
60    for each item in arrTimes
61      arr = item.Get( 0 )
62      if ( arr < time ) then
63
64        ptList = {}
65        areaList = {}
66        id = item.get( 3 )
67
68        if ( id < 1000 ) then
```

```
69
70          onFoot = item.Get( 2 )
71          if ( ( ( onFoot = False ) or ( id = 0 ) ) and ( arr < 500 ) ) then
72            ptList.Add( item.Get( 1 ) )
73            areaExtent = time - arr
74            areaExtent = areaExtent.Min( maxWalk )
75            areaExtAsDist = areaExtent * 70
76            areaList.Add( item.Get( 1 ).ReturnBuffered( areaExtAsDist ) )
77          end
78        else
79          buf = Polygon.MakeNull
80
81          rtNum = ( id - ( id Mod 100 ) ) / 1000
82          route = busRoutes.Get( rtNum ).Get( 0 )
83          busStop = item.Get( 1 )
84          bsPercent = route.PointPosition( busStop )
85
86          durn = busRoutes.Get( rtNum ).Get( 1 )
87          distInOneMin = 100 / durn
88
89          pt1 = Point.MakeNull
90          pt2 = Point.MakeNull
91          arr = arr.Round
92          fromTime = arr.Ceiling - ( arr.Ceiling mod 2 )
93          toTime = time.Floor - ( time.Floor mod 2 )
94          for each t in fromTime..toTime by 2
95            pct1 = bsPercent + ( ( t - arr ) * distInOneMin )
96            pct2 = bsPercent - ( ( t - arr ) * distInOneMin )
97            pct1 = pct1.Min( 100 )
98            pct2 = pct2.Max( 0 )
99            pt1 = route.Along( pct1 )
100           pt2 = route.Along( pct2 )
101           ptList.Add( pt1 )
102           ptList.Add( pt2 )
103           areaExtent = time - t
104           areaExtent = areaExtent.Min( maxWalk )
105           areaExtAsDist = 70 * areaExtent
106           areaList.Add( pt1.ReturnBuffered( areaExtAsDist ) )
107           areaList.Add( pt2.ReturnBuffered( areaExtAsDist ) )
108         end
109       end
110
111       if ( ptList.Count <> 0 ) then
112
113         if ( isoThemeTab.StartEditingWithRecovery ) then
114           for each i in 0..( ptList.Count - 1 )
115             area = Polygon.MakeNull
116             if ( polyNth.Contains( ptList.Get( i ) ) ) then
117               if ( polySth.Contains( ptList.Get( i ) ) ) then
118                 area = ( areaList.Get( i ).ReturnIntersection( polyNth ) ).ReturnUnion(
                                areaList.Get( i ).ReturnIntersection( polySth ) )
119               else
120                 area = areaList.Get( i ).ReturnIntersection( polyNth )
121               end
122             elseif ( polySth.Contains( ptList.Get( i ) ) ) then
123               area = areaList.Get( i ).ReturnIntersection( polySth )
124             end
125             recn = isoThemeTab.AddRecord
126             isoThemeTab.SetValue( fldShp, recn, area )
127           end
128         end
129         isoThemeTab.StopEditingWithRecovery( True )
130       end
131     end
132   end
133
134   theView.SetEditableTheme( isoTheme )
135   selRect = isoTheme.ReturnExtent
136   isoTheme.SelectByPolygon( selRect.AsPolygon, #VTAB_SELTYPE_NEW )
137   isoTheme.UnionSelected
138   isoTheme.SetVisible( True )
139   for each rec in isoThemeTab
140     poly = isoThemeTab.ReturnValue( fldShp, rec)
```

```
    poly = poly.ReturnClipped( extent )
    isoThemeTab.SetValue( fldShp, rec, poly )
    isoThemeTab.SetValue( fldTime, rec, time )
    isoThemeTab.Setvalue( fldArea, rec, ( poly.ReturnArea / 10000 ) )
  end

  symIso = Symbol.Make( #SYMBOL_FILL )
  symIso.SetColor( Color.GetRed )
  symIso.SetOLColor( Color.GetWhite )
  symIso.SetOLWidth( 1 )
  legIso = isoTheme.GetLegend
  legIso.SingleSymbol
  legIso.SetClassInfo( 0, { time.asString++" mins", time.asString, symIso, time, time }
     )

  isoTheme.StopEditing( True )
  isoTheme.ClearSelection

  theDisplay.Invalidate( True )

  theView.SetEditableTheme( NIL )

  return True
```

QuickBusIsochrone

```
1     ' ------ QuickBusIsochrone
2     ' ------ Last changed 12 August 1997 version: Alpha1.1
3     ' ------ Script to build isochrone based on a set of arrival times
4     ' ------ in a "station" theme
5
6     ' ------ av.Run( "Construct", { isochrone, stnArrDict, maxWalk, outline, busDict,
                                      strStations, strArea, strComm } )
7
8     if ( self.Is( List ).Not ) then
9       msgBox.Info( "Error 1 in Construct", "Error" )
10      return False
11    elseif ( self.Count <> 8 ) then
12      msgBox.Info( "Error 2 in Construct", "Error" )
13      return False
14    else
15      time = self.Get( 0 )
16      arrTimes = self.Get( 1 )
17      maxWalk = self.Get( 2 )
18      extent = self.Get( 3 )
19      busRoutes = self.Get( 4 )
20      strS = self.Get( 5 )
21      strA = self.Get( 6 )
22      strC = self.Get( 7 )
23    end
24
25    theView = av.GetActiveDoc
26    theDisplay = theView.GetDisplay
27    wDir = av.GetProject.GetWorkDir
28    clyde = theView.FindTheme( "Clyde.shp" )
29    north = theView.FindTheme( "north.shp" )
30    south = theView.FindTheme( "south.shp" )
31    northTab = north.GetFTab
32    southTab = south.GetFTab
33    fldNorth = northTab.FindField( "Shape" )
34    fldSouth = southTab.FindField( "Shape" )
35    for each rec in northTab
36      polyNth = northTab.ReturnValue( fldNorth, rec )
37      polyNth = polyNth.ReturnClipped( extent )
38    end
39    for each rec in southTab
40      polySth = southTab.ReturnValue( fldSouth, rec )
41      polySth = polySth.ReturnClipped( extent )
42    end
43
44    outName = FileName.Make( wDir.asString ).MakeTmp( "iso"++time.asString, "shp" )
45    isoShape = Ftab.MakeNew( outName, polygon )
46    isoTheme = FTheme.Make( isoShape )
47
48    isoTheme.SetComments( "This isochrone generated from "++strS++" and "++strA++" themes.
         Extent is "++time.asString++" minutes. " + strC )
49
50    isoThemeTab = isoTheme.GetFTab
51    fldList = { Field.Make( "Time", #FIELD_BYTE, 8, 0 ), Field.Make( "Area",
         #FIELD_DOUBLE, 16, 2 ) }
52    isoThemeTab.AddFields( fldList )
53
54    fldShp = isoThemeTab.FindField( "Shape" )
55    fldTime = isoThemeTab.FindField( "Time" )
56    fldArea = isoThemeTab.FindField( "Area" )
57
58    theView.AddTheme( isoTheme )
59
60    for each item in arrTimes
61      arr = item.Get( 0 )
62      if ( arr < time ) then
63
64        ptList = {}
65        areaList = {}
66        id = item.get( 3 )
67        if ( id < 1000 ) then
68
```

```
69          onFoot = item.Get( 2 )
70          if ( ( ( onFoot = False ) or ( id = 0 ) ) and ( arr < 500 ) ) then
71            ptList.Add( item.Get( 1 ) )
72            areaExtent = time - arr
73            areaExtent = areaExtent.Min( maxWalk )
74            areaList.Add( { areaExtent } )
75
76            _walkNetwork.FindServiceArea( ptList, areaList, False, True )
77            if ( _walkNetwork.HasServiceAreaResult ) then
78              isonet = ( "isonet.shp" ).asFileName
79              isoarea = ( "isoarea.shp" ).asFileName
80              _walkNetwork.WriteServiceArea( isonet, isoarea )
81              File.Delete( isonet )
82              tmpIsoSrc = SrcName.Make( isoarea.asString )
83              tmpIsoTheme = Theme.Make( tmpIsoSrc )
84              tmpIsoTheme2 = tmpIsoTheme.Clone
85
86              theView.AddTheme( tmpIsoTheme2 )
87              theView.SetEditableTheme( tmpIsoTheme2 )
88
89              isoSelRect = tmpIsoTheme2.ReturnExtent
90              tmpIsoTheme2.SelectByPolygon( isoSelRect.AsPolygon, #VTAB_SELTYPE_NEW )
91              tmpIsoTheme2.CopySelected
92
93              tmpIsoTheme2.StopEditing( False )
94
95              theView.SetEditableTheme( isoTheme )
96              isoTheme.Paste
97              isoTheme.StopEditing( True )
98
99              theView.DeleteTheme( tmpIsoTheme2 )
100             File.Delete( isoarea )
101           end
102         end
103       else
104         buf = Polygon.MakeNull
105
106         rtNum = ( id - ( id Mod 100 ) ) / 1000
107         route = busRoutes.Get( rtNum ).Get( 0 )
108         busStop = item.Get( 1 )
109         bsPercent = route.PointPosition( busStop )
110
111         durn = busRoutes.Get( rtNum ).Get( 1 )
112         distInOneMin = 100 / durn
113
114         pt1 = Point.MakeNull
115         pt2 = Point.MakeNull
116         arr = arr.Round
117         fromTime = arr.Ceiling - ( arr.Ceiling mod 2 )
118         toTime = time.Floor - ( time.Floor mod 2 )
119         for each t in fromTime..toTime by 2
120           pct1 = bsPercent + ( ( t - arr ) * distInOneMin )
121           pct2 = bsPercent - ( ( t - arr ) * distInOneMin )
122           pct1 = pct1.Min( 100 )
123           pct2 = pct2.Max( 0 )
124           pt1 = route.Along( pct1 )
125           pt2 = route.Along( pct2 )
126           ptList.Add( pt1 )
127           ptList.Add( pt2 )
128           areaExtent = time - t
129           areaExtent = areaExtent.Min( maxWalk )
130           areaExtAsDist = 70 * areaExtent
131           areaList.Add( pt1.ReturnBuffered( areaExtAsDist ) )
132           areaList.Add( pt2.ReturnBuffered( areaExtAsDist ) )
133         end
134
135
136         if ( isoThemeTab.StartEditingWithRecovery ) then
137           for each i in 0..( ptList.Count - 1 )
138             area = Polygon.MakeNull
139             if ( polyNth.Contains( ptList.Get( i ) ) ) then
140               if ( polySth.Contains( ptList.Get( i ) ) ) then
141                 area = ( areaList.Get( i ).ReturnIntersection( polyNth ) ).ReturnUnion(
```

```
                            areaList.Get( i ).ReturnIntersection( polySth ) )
              else
                area = areaList.Get( i ).ReturnIntersection( polyNth )
              end
            elseif ( polySth.Contains( ptList.Get( i ) ) ) then
              area = areaList.Get( i ).ReturnIntersection( polySth )
            end
            recn = isoThemeTab.AddRecord
            isoThemeTab.SetValue( fldShp, recn, area )
          end
        end
        isoThemeTab.StopEditingWithRecovery( True )
      end
    end
  end

  theView.SetEditableTheme( isoTheme )
  selRect = isoTheme.ReturnExtent
  isoTheme.SelectByPolygon( selRect.AsPolygon, #VTAB_SELTYPE_NEW )
  isoTheme.UnionSelected
  isoTheme.SetVisible( True )
  for each rec in isoThemeTab
    poly = isoThemeTab.ReturnValue( fldShp, rec)
    poly = poly.ReturnClipped( extent )
    isoThemeTab.SetValue( fldShp, rec, poly )
    isoThemeTab.SetValue( fldTime, rec, time )
    isoThemeTab.Setvalue( fldArea, rec, ( poly.ReturnArea / 10000 ) )
  end

  symIso = Symbol.Make( #SYMBOL_FILL )
  symIso.SetColor( Color.GetBlue )
  symIso.SetOLColor( Color.GetWhite )
  symIso.SetOLWidth( 1 )
  legIso = isoTheme.GetLegend
  legIso.SingleSymbol
  legIso.SetClassInfo( 0, { time.asString++" mins", time.asString, symIso, time, time }
     )

  isoTheme.StopEditing( True )
  isoTheme.ClearSelection

  theDisplay.Invalidate( True )

  theView.SetEditableTheme( NIL )

  return True
```

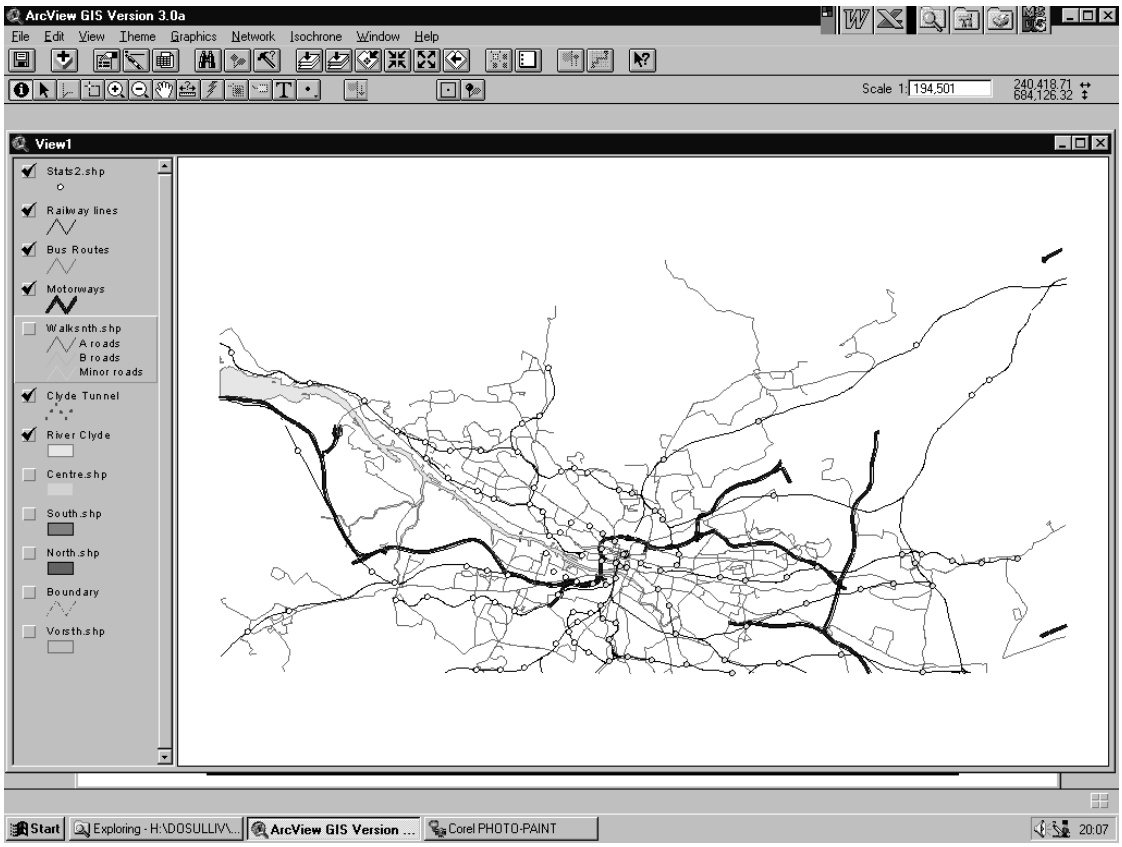
C. Description of use of isochrone tools

This appendix is intended to function as a brief 'User Manual' for the scripts described in the main body of the dissertation. It is presented in the order in which scripts should be run to produce a set of isochrones for a specified location.

Basics

This appendix assumes a working knowledge of *ArcView*. The tools described in the following sections are found in the `glas1_0` project. It is assumed that all the required data (shape files, *ARC/INFO* coverages, and so on) are in the same directory as the project `.apr` file.

The figure below shows what the `glas1_0` main view (**View1**) looks like. All isochrone tools are accessed from **View1**.



The tools referred to in the following sections are the two buttons at the right hand end of the lower tool bar. All menu-accessed tools are found in the **Isochrone** menu.

Defining the area of interest

The first task before constructing a set of isochrones is to define the area of interest. This is a rectangular shape which defines a sub-region of the Glasgow area. The area defined must contain all of the central zone of Glasgow. The extent of the central zone can be seen by making the `centre.shp` theme visible.

To define the area of interest:

- Select the tool marked with a box containing a dot (the left-hand of the two isochrone tools).
- Drag the mouse to create a rectangular area of interest.

ArcView will now prompt you for names for two new themes - an area theme and a stations theme. These will be used in subsequent steps. Names of the form `area*` and `stns*` will be suggested, and it is recommended that you accept these.

The `area*` theme will be shuffled down the table of contents so that it is behind other information, and the display will then zoom to the extent of the `area*` theme.

It is recommended that you retain the `stns*` and `area*` names for your working data, and copy results which you want to keep to specifically named files. This will facilitate housekeeping tasks in the project directory since all `stns*` and `area*` files can be periodically deleted without losing important information.

`Stns*` contains the railway stations inside the specified area. If you inspect the associated feature table you will see that each station has an [Arrival] time of 1000. This value indicates that no arrival time calculations have been done on this theme.

Defining the starting location for isochrones

The next step is to define a starting location in the stations theme. To do this:

- Select the tool marked with a 'lollipop' (the right-hand of the two isochrone tools).
- Click the mouse at the desired start location.

You will be prompted for the names of the `stns*` and `area*` themes to which your starting point belongs.

ArcView adds a new point to the selected `stns*` theme. If a starting location previously existed it is replaced with the new one. If you check the contents of the `stns*` feature table you will see that a new point has been added with [ID] set to 0 and [Arrival] set to 0.

You can now run the script which determines arrival times for the locations in the `stns*` theme.

Generating a set of arrival times

This script is run by selecting the **Arrival times...** option in the **Isochrone** menu.

When this menu option is selected a series of dialogue boxes appear requesting user input, as follows:

- You must specify the `area*` theme which applies.
- You must specify the `stns*` theme which applies.
- You will be warned that proceeding further will erase any arrival times already calculated for the specified `stns*` theme. If you want to retain these calculations, cancel, and make a copy of the `stns*` theme to a new shape file.
- You will be prompted to specify a start time, in minutes past the hour. This is used to determine travel times via the rail network.
- You will be prompted to select the travel modes which are to be used for the calculations.
- Finally, you will be prompted to enter some journey parameters. These are distances in metres either to, or between stations and bus routes. These are maximum connection distances *as the crow flies* over which the system will consider connections on foot between transport services.

The script will now run. Depending on the extent of the defined area of interest this may take some time. For a run on the whole region, using buses and trains, this could be up to 45 minutes. It is calculations for the bus routes which really slow things down, so it is recommended that preliminary exploration (to determine say the optimal starting time) be done using trains only. Any run including buses is likely to take some time.

Sensible settings for the journey parameters are:

To stations:	1500
Between stations:	1000
To buses:	500
Between buses:	250

If anything, you should consider using lower values than these.

When execution is complete, you can check that it has been successful by looking at the `stns*` feature table. The [Arrival] field should now contain various values for the different locations. If buses have been used in the calculations `stns*` will also contain new points with [ID] in the 1000s. These are bus boarding points determined by the script. You may want to set the legend to distinguish these. Preset legend `stns.avl` will do this. If trains were not selected, the actual station locations will still have [Arrival] set to a high value.

Constructing an isochrone

Having determined a set of arrival times (stored in a `stns*` theme), you can construct the isochrone you require. This is done by selecting the **Build isochrone...** option from the **Isochrone** menu.

Again you will be prompted for a series of inputs:

- You must specify the `stns*` theme which contains the points and arrival times on which the isochrone will be based.
- You must specify the `area*` theme associated with the specified `stns*` theme. This is used to clip the isochrone to size.
- You must specify the method of isochrone calculation. It is recommended that you choose **Quick for both modes**.
- You must specify the isochrone time in minutes. Normally round numbers should be selected, and you should bear in mind that the estimated resolution of results is no better than 5 minutes, so it does not make much sense to specify (say) 11 minutes.
- You will be prompted for additional comments. These are stored in the properties of the new theme which will be created. Note, however that comments are not copied when you create a copy of the theme.

Again this script may take some time to run. Isochrones up to about 45 minutes are generated reasonably quickly, even with bus boarding points in the specified `stns*` theme. Longer times than this will take longer. With a large `stns*` data set and a very long time limit (say 90 minutes), it is not impossible that an out of memory or similar crash could occur.

When complete, this script will have added a new theme named `iso##*.shp` to the table of contents. This is the isochrone you requested. ## indicates the time limit of the isochrone in minutes, and * is a sequence number. It is recommended that working data be left with the default name, whereas data you wish to keep should be copied to new shape files and given a more meaningful name. This facilitates housekeeping in the project directory.

Note that the isochrones are opaque by default so you may have to organise the order in which they appear in the table of contents to get a satisfactory map. Some suitable default legends are available called `15min`, `30min`, `45min`, `60min` and `75min.avl`.