Model documentation for the Australian Competition and Consumer Commission

Description of the Visual Basic used in the fixed LRIC model – Version 2.0

August 2009
9995-203

Analysys Consulting Limited
St Giles Court, 24 Castle Street
Cambridge, CB3 0AJ, UK
Tel: +44 (0)1223 460600
Fax: +44 (0)1223 460866
consulting@analysys.com
www.analysys.com

Contents

1 Introduction

Visual Basic code has been developed, using the compiler in Excel, for certain calculations to dimension both the access and core networks in the Analysys cost model.

For the access network, the Visual Basic is used to calculate the asset volumes required to deploy an access network to serve all locations within the Exchange Service Area (ESA). The algorithm considers a number of technologies to link locations back to the local exchange (LE) location in the ESA, namely fibre, copper, wireless or satellite (which would link customers to an earth-station in the core network). This code is stored in the workbook *Access – CODE.xls*.

For the core network, the Visual Basic purpose determines the efficient backhaul routes, using a spur and ring topology, from the local exchanges (Les) to the local access switches (or their next generation network (NGN) equivalents).

This document outlines the structure underlying both sections of code, each of which are significant. In addition, the code itself has been annotated should users want to inspect it within the Visual Basic Editor. A treatment of the specific principles underlying the algorithms can be found within the main documentation for the Analysys cost model, entitled *Fixed LRIC model documentation*.

The remainder of this document is laid out as follows:

- Section 2 describes the Visual Basic used for the access network deployment.
- Section 3 describes the Visual Basic used for the core network deployment.

2 Script for the access network algorithms

2.1 Introduction

The Visual Basic for the access network determines an efficient access network deployment for an ESA, given the location of the LE and a set of locations in the ESA with associated levels of demand. The algorithms includes two basic forms of deployment methodologies:

- The *urban* deployment uses a combination of copper and fibre and assumes that locations are grouped into clusters served by a distribution point (DP) which are themselves served by a pillar. This is currently used in Bands 1 and 2 and the more densely populated geotypes of Bands 3 and 4.

- The *rural* deployment uses a cost-based decision to determine whether it is more appropriate to serve each location in the ESA by wireline or wireless solutions. Locations served by wireline are served with either copper or fibre, whilst the remaining locations are served by either a wireless BTS network or satellite.

This document outlines the structure of the underlying code, which is located in the workbook *Access – CODE.xls*, which is part of the geoanalysis and access network module of the Analysys cost model.

The code follows two main paths depending on whether an urban or rural deployment is required. Both paths begin with a *start-up* phase, described in section 2.2, where constants and assumptions are read into the code.

For an urban deployment, there are then eight phases, summarised below and described in more detail in section 2.3:

- copper clustering phase
- copper DP cluster spanning tree phase
- copper DP cluster connection phase
- copper pillar connection phase
- fibre determination phase
- backhaul determination phase
- result storage phase
- assumption storage phase.

Rural deployments require more phases, since this type of deployment is considering more technologies. The phases are summarised below and described in more detail in section 2.4:

- initial copper clustering phase
- copper or wireless determination phase

- copper clustering phase
- copper pillar cluster spanning tree phase
- copper cluster connection phase
- backhaul determination
- fibre determination
- copper result storage
- wireless clustering
- satellite determination
- wireless backhaul determination
- copper and fibre result storage
- assumption storage.

2.2 Start-up phase

This stage is completed at the start of the calculation for any ESA. The following subroutines are used. For each sampled ESA to be calculated, the relevant Access DATA workbook is opened if needed, old assumptions are deleted and input arrays are populated.

Two subroutines are run at the start of the process, *SetupPermanentConstants* and *ReadInGeotypeData*, which read in inputs relevant to all ESAs.

For each ESA to be processed by the algorithms, three subroutines are used to read in input data specific to the ESA: *SetupConstantsForThisESA*, *DeleteOldESAOutputs* and *Initialise*.

All five of these subroutines are described in sections 2.2.1–2.2.5 below.

2.2.1 SetupPermanentConstants

Location:	Found in the *CommonCode* module
Purpose:	Reads in various assumptions and constants that are fixed regardless of geotype. This includes: • directory paths • cable sizes • array of network deployment assumptions

2.2.2 ReadInGeotypeData

Location:	Found in the *CommonCode* module
Purpose:	Reads in the ESA indices as stored on the ‘Summary’ worksheet i.e.

the ESA geotype, index and index within the sample of the geotype.

For each sampled ESA to be calculated, the relevant Access DATA workbook is opened if it is not already open. Old assumptions from previous calculations are then deleted and input arrays are populated. Throughout the process, the time taken at major stages in the process is stored.

2.2.3 SetupConstantsForThisESA

Location: Found in the *CommonCode* module

Purpose: This resets all of the global variables and reads in the assumptions that vary by geotype on the 'Inputs' worksheet. These include the:

- clustering capacity and distance constraints
- p-function coefficients
- assumptions for the copper versus wireless algorithm
- proxy cost function coefficients
- number of locations and the identity of the one that is the remote access unit (RAU) / local exchange(LE).

2.2.4 DeleteOldESAOutputs

Location: Found in the *CommonCode* module

Purpose: Deletes the contents of all of the cells which were written to in the last calculation of this ESA

2.2.5 Initialise

Location: Found in the *CommonCode* module

Purpose: Reads in the location coordinates and demand requirements for the relevant ESA from its worksheet in the Access DATA workbook: this includes the Geocoded National Address file (G–NAF) locations for rural ESAs.

In particular, the array gobjInputPoints() is populated with the co-ordinates and demand at each location.

The path of the code then diverges, depending on whether the ESA is to be processed with the urban or rural deployments. The code for urban deployments is described in section 2.3, whilst the code for rural deployments is described in section 2.4.

2.3 Urban deployment path

There are eight phases to the urban deployment:

- copper clustering phase, which is described in section 2.3.1
- copper DP cluster spanning tree phase, which is described in section 2.3.2
- copper DP cluster connection phase, which is described in section 2.3.3
- copper pillar connection phase, which is described in section 2.3.4
- fibre determination phase, which is described in section 2.3.5
- backhaul determination, which is described in section 2.3.6
- result storage phase, which is described in section 2.3.7
- assumption storage phase, which is described in section 2.3.8.

2.3.1 Copper clustering phase

This is run through the subroutine *AllClusteringMethods*, which is found in the *MainMacros* module. The urban deployment executes seven subroutines using *AllClusteringMethods*:

- *IdentifyRAU*
- *DivisiveClustering*
- *WriteClusterResults*
- *ClusterToPillarClusterLevel*
- *CalculateFibreUnitsOfDemand*
- *IdentifyPillars*
- *IdentifyDistPoints*.

These are explained in more detail in the following sub-sections.

IdentifyRAU

Location:	Found in the *Clustering* module
Purpose:	This only calculates a RAU location for an ESA if no RAU location is stated in the 'Inputs' worksheet. Currently, each ESA uses the first location in the list as the location of the RAU, which has been extracted from ExchangeInfo. If a location is not stated for the RAU in the Access DATA workbooks, then the location closest to the demand-weighted centre

of the locations in the ESA is used. There are three sets of objects that can be used for this calculation by *IdentifyRAU*: locations, DPs and pillars. The urban deployment uses individual locations.

DivisiveClustering

This subroutine groups final drop points (FDPs) into clusters which are served by DPs. This clustering is based on a capacity and a distance constraint and is top-down in design. Specifically, a single parent cluster is created containing all of the locations and 'child' clusters are created from the parent, causing it to shrink in size. This ceases when the parent cluster satisfies both the capacity and distance constraints.

The following subroutines create the parent cluster:

► *InitialiseClusterAllocations*

Location:	Found in the *Clustering* module
Purpose:	Assigns each point to a single 'parent' cluster

► *InitialiseParentCapacity*

Location:	Found in the *Clustering* module
Purpose:	Calculates the total demand within the parent cluster

► *CalWeightedCentre*

Location:	Found in the *Clustering* module
Purpose:	By default, this calculates the demand-weighted centre of the parent cluster. If it is supplied with points that all have zero demand, then it will calculate the geometric centre of all the points.

► *CalSquareMaxDInP*

Location: Found in the *Clustering* module

Purpose: This identify the point furthest from the centre of the parent cluster in order to generate a child cluster. Having identified the point, it sets:

- the point as the first point in a new child cluster
- the demand of the child cluster as the demand at that point.

We note that we do allow points with demands greater than the limit of the capacity to be clustered as clusters of one point, although this fails the capacity limit criterion.

The main loop of the algorithm creates new child clusters until the parent cluster satisfies the demand and distance constraint. New child clusters are created by:

- Selecting the point in the parent cluster that is furthest from its demand-weighted centre, using the subroutine *ChooseFirstChildMember*, which is found in the *Clustering* module.
- Expanding the cluster by adding points from the parent. Each time a point is added:
  - all points in the parent cluster are shortlisted to those who are within twice the maximum permitted distance from the current child cluster demand-weighted centre and with a capacity that would not overfill the cluster capacity were it added
  - points that are found during this process to have too much demand for the cluster are not considered again for the cluster at all, but are kept within the parent cluster and are restored for consideration for the next child cluster
  - the point amongst these that is the closest to this child cluster demand-weighted centre is selected and provisionally re-allocated
  - the child cluster centre is re-calculated using *CalUnweightedCentre*, which is found in the *Clustering* module and calculates a geometric centres for the cluster
  - the re-allocation is finalised, unless the cluster no longer satisfies the distance criterion with the re-calculated cluster centre, in which case it is rejected.

This loop uses the subroutine *AllocatePointToCluster* to move points between the parent and child clusters.

► *AllocatePointToCluster*

Location:	Found in the *Clustering* module
Purpose:	Allocates a selected point to a given child cluster provided it doesn't violate distance constraints. If the allocation is accepted, then the new cluster centres of both the parent and the child are also re-calculated.

When the parent cluster satisfies the demand and distance constraint, it is re-written as the final child cluster. A series of subroutines are then used to improve the quality of these clusters. These subroutines are:

- *SimpleReassignment*
- *Swap*
- *FullOptimisation*
- *HighDemandSimpleReassignment*
- *HighDemandSwap*

- *SingleDemandSwap*.

A dictionary of points is used to accelerate the refinement processes using a Scripting.Dictionary object. This object is populated before the refinement begins.

These subroutines are reused frequently throughout the code for different clustering requirements and are each explained below. They can move any locations between cluster except for the RAU location when an ESA is using the fibre ring deployment. This very minor constraint allows the subroutines to be reused for clustering pillars into fibre rings, which requires that every cluster contains the RAU.

These subroutines require particularly intensive use of distance calculations, particularly for distance comparisons (e.g. identifying whether a point P1 is closer to a point P2 or a point P3). We use a quicker distance measure in each of these cases with the subroutine *CalcPfunctionDistanceComparisonOnly*.

► *CalcPfunctionDistanceComparisonOnly*

Location:	Found in the *CommonCode* module
Purpose:	For two points with co-ordinates (x_1, y_1) and (x_2, x_2) and a p-function with coefficients k and p, this subroutine outputs:

$$|x_1-x_2|^p + |y_1-y_2|^p$$

This output does not include taking the *p*th root that would be required in deriving the actual p–function distance and therefore requires less time. Comparing two measures calculated with this form will give the same result as comparing two actual p–function distances, as functions of the form x^p are increasing functions for positive x.

In contrast, the subroutine *CalcPfunctionDistance* calculates the actual p–function distance between two points, executing the final stage of taking the pth root.

► *CalcPfunctionDistance*

Location:	Found in the *CommonCode* module
Purpose:	For two points with co-ordinates (x_1, y_1) and (x_2, x_2) and a p-function with coefficients k and p, this subroutine outputs:

$$k(|x_1-x_2|^p + |y_1-y_2|^p)^{1/p}$$

This function can also be used for the normal straight-line distance measure, by using k=1 and p=2.

► *SimpleReassignment*

Location:	Found in the *Clustering* module

Purpose:

This subroutine will move a point from one cluster to another under certain conditions.

To start with, the demand-weighted centres for each cluster is calculated and clusters with spare capacity are flagged.

The main Do…Loop continues moving through reach point in turn until no more can be re-assigned. This can move through all of the points more than once. For each point:

- The cluster with the closest demand-weighted centre is identified.
- It is moved to this identified cluster if and only if:
  - – the point is closer to this new cluster's demand-weighted centre than its current one
  - – the new cluster has sufficient spare capacity
  - – all points in the new cluster obey the distance constraint with respect to a cluster centre re-calculated using this new point.
- Its old cluster is flagged to have spare capacity and the new cluster is checked to see whether it no longer has spare capacity.

► *Swap*

Location:	Found in the *Clustering* module

Purpose:

This subroutine will swap a point in one cluster with a point in another under certain conditions. To start with, the demand-weighted centres for each cluster is calculated.

The main Do…Loop cycles through all points in turn, possibly multiple times, until no more points can be swapped. For each point P:

- The cluster with the closest demand-weighted centre is identified.
- If the identified cluster is not P's current cluster and, if moving P to the new cluster violates the maximum capacity constraints, then try to find a point Q in the new cluster which can be swapped with P so that all the following are satisfied:
  - – the two new clusters both satisfy the cluster capacity

constraint
- the sum of the two distances between the points and the cluster centres of their new clusters is lower than the sum of the distances between the points and the cluster centres of their original clusters
- both clusters obey the distance constraint with respect to their new cluster demand-weighted centre.

- If such a Q is found, then:
  - temporarily revise the two clusters
  - re-calculate their demand-weighted centres
  - re-calculate the sum of the two distances between the points and the *re-calculated* cluster centres of their new clusters
  - check if this new total distance is lower than the sum of the two distances between the points and the *original* cluster centres of their original clusters.

- If this test is also successful, then make the swap permanent. Otherwise, restore P and Q to their original clusters.

► *FullOptimisation*

Location: Found in the *Clustering* module

Purpose: This subroutine will move a point from one cluster to another if certain criteria are satisfied, though these criteria are different to those in *SimpleReassignment*. Specifically, it tries to minimise the total distance from the points in a cluster to its cluster centre.

Firstly, the demand-weighted centres for each cluster is calculated.

Then, for each cluster, the sum of the distances between the points in a cluster and its demand-weighted cluster centre are calculated. This uses the subroutine *CalcTotalDist*, which can be found in the *Clustering* module. Clusters with spare capacity are also flagged.

The main Do…Loop cycles through all points in turn, possibly multiple times, until no more can be moved. For each point P:

- The cluster containing P is identified.
- The total distance (d1) between all points in this cluster and its cluster centre is stored.
- P is temporarily removed from its cluster and both the demand-weighted cluster centre and the total distance (d2) between the

cluster points and the new cluster centre are re-calculated.

- P is then restored to its cluster.
- For each cluster with sufficient spare capacity to accommodate P, the total distance (d3) between all points in this cluster and its current cluster centre is stored.
- P is then added into this cluster and the demand-weighted cluster centre and the total distance (d4) between the DP locations and the new cluster centre are calculated.
- The cluster which gives the largest reduction in total distance (i.e. which maximises ([d1-d2]-[d4-d3]) is identified.
- If no clusters give a reduction, then proceed to the next point.
- Otherwise, P is moved to the identified cluster provided that it would also satisfy the normal distance constraint using its new demand-weighted centre.
- The original cluster is flagged as now having spare capacity.
- The cluster that has received P is also checked to see if it still has spare capacity.

► *HighDemandSimpleReassignment*

Location: Found in the *Clustering* module

Purpose: This is similar to *SimpleReassignment*, except that it only considers points with high demand (more than one unit of demand). Firstly, the demand-weighted centres for each cluster is calculated. Points with a high demand (more than 1 and at most the absolute maximum cluster capacity are then identified.

The main Do…Loop cycles through all points in turn, possibly multiple times, until no more can be re-assigned. For each high-demand point P in turn:

- Identify the cluster whose demand-weighted centre is closest to P.
- P is moved to this cluster if all the following are satisfied
  - – P is closer to this new cluster's demand-weighted centre than its current one
  - – the new cluster has sufficient spare capacity (using the absolute maximum capacity limit, not the normal cluster capacity limit)
  - – all points in the new cluster obey the distance constraint with respect to the new cluster centre.
- If these are satisfied, then the cluster centres for both clusters involved are re-calculated.

► *HighDemandSwap*

Location: Found in the *Clustering* module

Purpose: This is similar to *NormalSwap*, except that it only considers points with high demand (more than one unit of demand).The demand-weighted centres for each cluster is calculated and high-demand points are identified.

The main Do…Loop cycles through all points in turn, possibly multiple times, until no more can be swapped. For each high-demand point P in turn:

- The cluster whose demand-weighted centre is closest to P is identified.
- If the cluster is not P's current cluster and, if moving the point to the new cluster violates the maximum capacity constraints, then try to find a point in the new cluster which can be swapped with P so that all the following are satisfied:
  - the two new clusters both satisfy the cluster demand constraint, but using the absolute maximum capacity as the limit
  - the sum of the two distances between the points and their DP cluster centres is improved compared with before
  - both clusters obey the distance constraint with respect to their new cluster demand-weighted centre.
- If such a point is found, then revise the two clusters and re-calculate their demand-weighted centres.
- Otherwise, return the points to their original clusters.

► *SingleDemandSwap*

Location: Found in the *Clustering* module

Purpose: This is identical to *HighDemandSwap*, except that it only considers points with one unit of demand.

WriteClusterResults

The first copper clustering phase for the urban deployment, which allocates locations to clusters served by a DP, uses the following sequence of refinement subroutines:

- *SimpleReassignment*
- *Swap*
- *FullOptimisation*
- *HighDemandSimpleReassignment*

- *HighDemandSwap*
- *SingleDemandSwap*
- *SimpleReassignment*
- *Swap*.

Having completed the clustering of the points, the cluster indices are printed on the output worksheet for the ESA.

Location:	Found in the *Clustering* module
Purpose:	Writes the cluster indices for each location in column H of the output worksheet for the ESA, beside its co-ordinates.

ClusterToPillarClusterLevel

Following the clustering of locations into DP clusters, a second clustering phase occurs by grouping DP clusters into pillar clusters. In order to reuse the clustering subroutines described above, local copies of the arrays containing the DP clustering data are taken, allowing the original arrays to be reused for the second level of clustering. This is accomplished by a subroutine called *CopyClusteringArrays*.

► *CopyClusteringArrays*

Location: Found in the *CommonCode* module

Purpose: Stores the values contained in:

- glNumPoints, which is the number of points to be clustered
- gNumChildClusters, which is the number of child clusters created
- glClusterAssignedTo(), which contains the cluster index for each point
- glNumVerticesInCluster(), which contains the number of locations in each cluster
- glClusterCapacity(), which contains the number of units of demand within each cluster
- gobjInputPoints(), which contains data specific to each location, including the co-ordinates and number of units of demand.

In particular, the DP cluster for each location is stored in glDPCluster().

The location of the DP that serves each DP cluster is defined as the location in the cluster closest to its demand-weighted centre. The identity of this point is stored in the array glClusterMainPoints(). The array gobjInputPoints() is then re-populated with the data for all the DP clusters.

The subroutine *DivisiveClustering* is then reused to derive pillar clusters for the DP clusters. These clusters are derived using only the points that are used as the DPs for the DP clusters. This implicitly ensures that locations in the same DP cluster are in the same pillar cluster.

In this case, DPs are grouped into clusters served by pillars, using the subroutine *DivisiveClustering.* This clusters a set of locations based on a capacity and a distance constraint specific to pillars. These constraints are obviously larger than those used for clustering into DP clusters.

InitialiseClusterAllocations, *InitialiseParentCapacity*, *CalWeightedCentre*, *CalSquareMaxDInP* and the main Do..Loop follow as previously described. However, a different order of refinement algorithms are used in this case, namely:

- *SimpleReassignment*
- *Swap*
- *FullOptimisation*
- *SimpleReassignment*
- *Swap*.

The resulting clustering data is then stored in separate arrays and the location-specific clustering data is restored to its original arrays using *CopyClusteringArrays*. In particular, the:

- number of DPs in each pillar cluster is stored in glNumClustersInPillarCluster()
- total demand served by each pillar is stored in glPillarClusterCapacity()
- parent pillar of each DP is stored in glClusterToPillarCluster().

Having completed the first pass of the pillar clustering, a cleaning subroutine called *ConsolidatePillars* is used to check whether any pillars can be merged without breaking the pillar capacity and distance constraints. This is used in the urban deployment to merge points that are isolated and are effectively served by their own pillar. Using this subroutine, these points are reassigned to be served by their closest pillar that can accommodate the additional capacity.

► *ConsolidatePillars*

Location:	Found in the *Clustering* module
Purpose:	The subroutine *IdentifyPillars* is first used to define a pillar for each pillar cluster. Local copies are then made of the following arrays: • glPillarClusterCapacity() • glNumClustersInPillarCluster() • glPillarClusterMainPoints() • glClusterToPillarCluster(). An array of the original pillar indices is also populated. This is to

track the merging of the pillar clusters.

The pillars are then sorted into descending order of capacity, using the subroutine *NearestPointsQuickSort*. An array mapping the sorted pillar indices to the original indices, lngCopyPillarClusterIndexesINVERSE(), is then populated.

The main loop then considers pillar clusters P in descending order of capacity until no more consolidation is possible. For each pillar cluster P, every other pillar cluster Q is considered in turn for potential merging:

- If the total capacity of P and Q is less than the absolute maximum pillar cluster capacity, then temporarily merge the locations from P and Q into a single cluster (R) and re-calculate the pillar location for R using *ReIdentifyPillar*.
- A merge is stated to be possible if:
  - R would satisfy the (absolute) pillar capacity constraint and distance constraint with itsnew pillar location
  - (only for urban deployments) if the capacity of P is smaller than some critical value, regardless of the result of the distance constraint test.
- If the merge is possible, then Q is labelled as a possible merge with P in lng_tempClusterwhichcanmerge() and the distance between the cluster centres of P and Q is stored in dbl_tempClusterCentre_distance().
- Having checked through all possible Q, if the list of candidate pillar clusters for merging with P is empty, then flag that P cannot be merged with any other clusters. It is then not considered for further consolidation in the rest of the algorithm.
- If the list is not empty, then merge P with the pillar cluster Q in the list whose cluster centre is closest to P's pillar and use *ReIdentifyPillar* to calculate the new pillar location.

The subroutines *IdentifyPillars* and *ReIdentifyPillar* are described in more detail below.

IdentifyPillars

Location: Found in the *Clustering* module

Purpose:	For the urban deployment, pillar locations are identified as one of the DP locations in the cluster. The location defined as the pillar for each pillar cluster is stored in glPillarClusterMainPoints(). For the case of the pillar cluster containing the RAU, the pillar location is defined to be the RAU. Otherwise, the demand-unweighted centre of all the locations in the pillar locations using the subroutine *CalUnWeightedPillarclusterCentre*, found in the *Clustering* module. The DP location that is closest to this demand-unweighted centre is then defined as the pillar location for that pillar cluster. The subroutine *ReIdentifyPillar* is identical to *IdentifyPillar* in every respect, except that it recalculates the pillar location for a single pillar cluster, rather than every pillar cluster.

CalculateFibreUnitsOfDemand

Location:	Found in the *Clustering* module
Purpose:	Calculates the demand served by fibre in each pillar cluster, by identifying those points that are served by fibre. This is stored in the array glDemandServedByFibreByPillar ().

IdentifyDistPoints

Location:	Found in the *Clustering* module
Purpose:	This subroutine can only be used after the pillar locations have been defined. It redefines the DP location as the location in the DP cluster that is closest to the pillar location, rather than the demand-weighted centre. The only exceptions are when the DP is also a pillar location, in which case the previous DP location is retained. The new identities of the DP locations are used to update the array glClusterMainPoints().

2.3.2 Copper DP cluster spanning tree phase

Having defined the clusters and locations of DPs and pillars, the subroutine *ConstructTreeFollowingClusterMainPointIdentification* derives the minimum spanning trees for each cluster. It begins with the subroutines *GetMaxPointsInCluster* and *SetupArraysForSpanningTree*, which requires information from across all the spanning trees.

GetMaxPointsInCluster

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Identifies the maximum number of points in a DP cluster across all DP clusters.

SetupArraysForSpanningTree

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose: Dimensions the key arrays for the minimum spanning tree process:

- glUnattachedPoint() – the list of locations in the cluster that are unattached at any point in the algorithm
- glAttachedPoints() – the list of locations in the cluster that are attached at any point in the algorithm
- glVertexRoute() – for any location, the location that it passes through in order to get back to the node location
- gdDistanceMatrix() – stores the distance between any two points in the cluster
- gobjEdges() – stores the vertices and lengths associated with each edge in the spanning tree.

Following these two subroutines, each DP cluster is then treated in turn. The following subroutines are used in order to create and store the calculations:

- *SetupPointsInCluster* – identifies the central point in the cluster
- *SetupGdDistanceMatrix* – calculates the required distances
- *ConstructTree* – constructs the minimum spanning tree for the cluster
- *StoreRoutes* – for each point P, identifies the point P passes through to get back to the node
- *GetTotalDistance* – calculates the total trench within the tree
- *GetSheathLength* – calculates the total copper sheath within the tree
- *GetCopperLength* – calculates the total copper pair length within the tree
- *WriteNetworkResults* – writes the list of edges in the spanning tree onto the output worksheet.

SetupPointsInCluster

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose:	• Calculates the number of points in the cluster • Identifies the node for the cluster i.e. the DP location for the DP cluster and states this location as the central point *cp.*

SetupGdDistanceMatrix

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For the urban deployment, calculates the straight-line distance between any two pairs of points and stores these distances in the array gdDistanceMatrix().

ConstructTree

In order to derive the spanning tree for the cluster, the algorithm begins with the central point *cp* identified in *SetupPointsInCluster*. All other locations in the cluster are assumed to be unattached. Locations are then added to the tree incrementally. Each time a location is linked to the tree, it becomes attached and the lists of attached and unattached locations are updated using the function *IdentifyAttachedAndUnattachedPoints.*

► *IdentifyAttachedAndUnattachedPoints*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Moves through the entire set of points and enumerates them using two arrays, glUnattachedPoint() and glAttachedPoints(). glUnattachedPoint(*j*)=*k* means that point *k* is the *j*th point in the list that is unattached. Similarly, glAttachedPoint(*j*)=*k* means that point *k* is the *j*th point in the list that is attached.

In order to determine which location to join to the tree, the algorithm calculates the average cost per unit of demand of linking an unattached point P to an attached point Q using a trench and a cable on the existing tree. This is determined by the function *AverageCostPerLine*.

► *AverageCostPerLine*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For the urban deployment, it determines the: • Extra capacity and the copper pair requirements (*c*) needed to serve the unattached point.

- Length of extra trench distance (*d*) to link the attached and unattached point.

The cost of the new link is then calculated using the proxy cost expression:

$$k_1*d + k_2*c + k_3*d*c + k_4*\sqrt{c}$$

and calculates the new cost per unit of demand for the entire tree.

For each unattached point P, the edge to connect each attached point Q to the tree is considered. The edge that gives the lowest new average cost per unit of demand for the whole tree is stored in the array objEdgeList() using *AddToEdgeList*.

► *AddToEdgeList*

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose: Having identified the best edge by which to join a particular point to the existing tree, this subroutine stores the vertices of this edge and its average cost per unit of demand.

Having stored the best edge to link each unattached point P to the tree in objEdgeList(), the edge that gives the overall lowest new average cost per line is then permanently added to the tree using *AddCheapestEdgeInListToObjEdges*.

► *AddCheapestEdgeInListToObjEdges*

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose:

- Adds the best edge in objEdgeList(), in terms of average cost per line, to the list of edges for the minimum spanning tree.
- Updates the array glVertexRoute() for this new edge in the tree, defined by:

  glVertexRoute(unattached location on new edge) = attached locations on new edge

- Updates the total copper length required to link the location all the way back to the node.

The number of unattached points is reduced by 1 and the lists of attached and unattached points updated using *IdentifyAttachedAndUnattachedPoints*. The loop in *ConstructTree* continues until all locations in the cluster are part of the spanning tree.

StoreRoutes

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For each point P in the cluster, this stores the point Q that it passes back through in order to reach the DP, with the array defined as: glRouteToCentre(P) = Q

GetTotalDistance

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total trench in the spanning tree, by cycling through all points P in the cluster and calculating the distance between P and its predecessor on its way back to the DP.

GetSheathLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total cable sheath in the spanning tree, by cycling through all points in the cluster and calculating the distance between P and its predecessor back to the DP. It then multiplies this by the sheath requirements for the link given the demand at P.

GetCopperLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total copper cable length in the spanning tree, by cycling through all points P in the cluster and calculating the distance between P and its predecessor back to the DP. It then multiplies this by the copper pair requirements given the demand at P.

WriteNetworkResults

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• Writes the points and their co-ordinates in the output worksheet for the ESA (in rows BF–BM) that define every edge in the spanning trees for the DP clusters.

- Write the number of ducts needed, by type, for each edge.
- Determines how many extra pits are required along these edges.
- Writes the DP locations, their co-ordinates and their parent pillar in the output worksheet for the ESA (in rows BX–CD).

2.3.3 Copper DP cluster connection phase

This phase is run through the *ConnectClusters* subroutine, which is found in the *ClusterToCluster* module. This joins up all the DP locations within a pillar cluster back to the pillar using the subroutine *RunAtClusterLevel*, which also lies in *ClusterToCluster* module. This subroutine is only used if there is more than one DP location in the pillar cluster.

Firstly, the largest number of DP locations to be found in any pillar is calculated. Then, *RunAtClusterLevel* executes several subroutines:

- *SetUpClusterPairIndex*
- *IndexClusterWithinPillarCluster*
- *SortPairsOfClusters*
- *RoutePointsForCluster*
- *ApplyDijkstra* (contained within *RoutePointsForCluster*).

SetUpClusterPairIndex

Location:	Found in the *ClusterToCluster* module
Purpose:	Indexes pairs of DPs in a pillar area so that each unordered pair occurs exactly once. The index uses triangular numbers: e.g. for four DPs, (DP1,DP2) → 1, (DP1,DP3) → 2, (DP1,DP4) → 3, (DP2,DP3) → 4, (DP2,DP4) → 5, (DP3,DP4) → 6

IndexClusterWithinPillarCluster

Location:	Found in the *ClusterToCluster* module
Purpose:	This creates a new indexing ClusterIndex() of DP clusters in a pillar cluster so that they are numbered from 1 to *n*, where n is the number. For example, if we are looking at the second pillar cluster and the first DP cluster within this pillar cluster is DP cluster 100, then ClusterIndex(1)=100. An inverse mapping InverseClusterIndex() is

also stored, so that we can move between the two indices.

Following *IndexClusterWithinPillarCluster*, the DP that is the pillar location is identified.

SortPairsOfClusters

Location:	Found in the *ClusterToCluster* module
Purpose:	• For each pair of DP clusters in this pillar cluster: – calculate how many unique pairs of points there are which have one point from each DP cluster – explicitly identify these pairs of points and, for each pair, calculate the distance between the two points – sort the pairs of points in order of this distance, with the closest pair of points listed first.

RoutePointsForCluster

This subroutine is found in the *ClusterToCluster* module. Its purpose is to calculate, for each pair of DP locations, the connection, possibly via other DP locations, that has the lowest cost (according to our proxy cost function $k_1*d + k_2*c + k_3*d*c + k_4*\sqrt{c}$).

A linkage between two DPs can be split into two components: a component that uses only additional trench and a component that uses only existing trench.

The proxy cost function for costing up a part of a link which requires new trench assumes additional trench cost. This is calculated using M_1 * new trench length L_N, where the cost multiplier $M_1 = k_1 + (k_3 * \text{cabling capacity } C_D) + (k_4 * \sqrt{C_D})$.

The proxy cost function for costing up a part of a link which uses only existing trench assumes no additional trench cost. The cost of this is calculated by using M_2 * existing trench length L_E, where the cost multiplier $M_2 = (k_3 * \text{cabling capacity } C_D) + (k_4 * \sqrt{C_D})$.

These separate cost multipliers M_1 and M_2 are calculated for each pair of DPs and stored in the arrays dCostMultiplier1() and dCostMultiplier2() respectively. AS shown above, these multipliers depend on the value of C_D, which itself depends on whether this part of the network is assumed to be tapered or non-tapered, as shown below in Table 2.1.

Assumption of distribution network	*C_D*	*Joints required*
Fully tapered	Smallest cable size in the distribution network that accommodates the larger of the demands served by the two DPs	Smallest cable size in the distribution network that accommodates the larger of the demands served by the two DPs
Non-tapered	Number of pairs in the main cable size used in the distribution network	Equal to the larger of the demands served by the two DPs

Table 2.1: Calculation of jointing and C_N [Source: Analysys]

The remainder of *RoutePointsForCluster* then proceeds as follows, for each pair of DP clusters:

- Identify the pair of points, 1 from each DP cluster, which are the closest (at a distance *d* apart, calculated using the p–function).
- Calculate the sum of the distances D_T of each point in the pair back to its respective DPs.
- The proxy cost of linking the two DP clusters together through these points is then assumed to be $(M_1 * d) + (M_2 * D_T)$.
- For each pair of DP clusters, identify the pair of points which give the lowest proxy cost: this gives a fully meshed set of linkages between all DPs.
- We then apply a version of the Dijkstra algorithm using the subroutine *ApplyDijkstra*. This identifies a subset of these linkages that can link all DPs back to the RAU at the lowest cost.

ApplyDijkstra

Location:	Found in the *ClusterToCluster* module

Purpose:

- Apply the Dijkstra algorithm to derive a least proxy cost route between any DP cluster and the pillar node, using the calculated mesh of linkages.
- Set lIncoming() for each DP, by default, to be its own demand.
- Assume, provisionally, that all DPs are connected directly to the pillar.
- Start the algorithm with the pillar.
- For every other DP, recall the requirements for linking it to the pillar:
  - extra trench
  - cost of linking the two DPs
  - cabling cost of linking the two DPs (i.e. excluding

    trench cost)
  - joints required, using *GetJointingCosts*
  - total sheath length between them.

- Execute the following loop whilst there are still unconnected DPs, starting at the pillar:
  - for a given connected DP i, look through the unconnected DPs and determine which DP (*j*)is the most cost effective to link directly to DP i
  - update the array lIncoming(i), which is defined as the total demand passing through the DP on the way back to the pillar, to include the demand at *j*
  - for each unconnected DP, test to see if its currently identified path to the pillar has a lower proxy cost than if it were to go through i. If the proxy cost is lower, then set its provisional route back to the pillar to be via i and update lIncoming() accordingly
  - for each unconnected DP *k*, test this by:
    - calculating all the extra jointing costs of going through DP i (possibly via other DPs) back to the pillar, rather than direct
    - if ([total cabling cost of linking *k* to pillar via i] +[extra jointing costs of linking *k* to pillar via i] + [the cost of linking *k* to DP i]) < cost of linking *k* to the pillar directly, then set the provisional link for *k* to be via DP i
  - set i to be the DP that was just connected (*j*) and return to the start of the loop.

- When all DPs have been connected back to the RAU, aggregate:
  - extra trench to join all DP clusters in the pillar cluster to the pillar location
  - demand-related jointing required
  - number of branching kits required
  - distance related jointing required (non-tapered case only)
  - incremental copper sheath required
  - incremental copper sheath by cable size
  - incremental copper km required
  - the routes that DPs take back to their parent pillar, in glPreviousClusterBackToPillar().

This subroutines refers to several other functions within it, which are explained below. These are *GetJointingCosts* and *CostOfEdgesCountedTwice*.

► *GetJointingCosts*

Location:	Found in the *ClusterToCluster* module
Purpose:	This identifies the jointing required for a particular link from a DP i back to the pillar. The calculation depends on whether the network is assumed to be fully tapered or non-tapered:

In the tapered case, we revise the cabling required on each link required on the path from i back to the pillar based on the cable sizes available to us. The jointing is then aggregated at each DP on the path that i takes back to the pillar, each time adding on the demand at the DP and the downstream cable capacity.

In the non-tapered case, we revise the cabling required on each link required on the path from i back to the pillar based on the main and minor cable sizes assumed. The jointing is then aggregated at each DP on the path that i takes back to the pillar, each time adding on the demand at the DP. A full joint of the cable is only included at regular distance intervals, rather than at every DP.

The final step in *ApplyDijkstra* is to calculate the sheath requirements for each link within the pillar cluster, using the subroutine *CalculateDuctByType*.

► *CalculateDuctByType*

Location:	Found in the *CalculateDuct* module
Purpose:	Identifies the number of sheaths by cabling type within each DP–DP link in the pillar clusters, as preparation for the derivation of the number of ducts within each link in the subroutine *WriteDuctOutputs*.

If the cabling within the DP–pillar network is assumed to be tapered, then 1 sheath is assumed to be required within each link.

If the cabling within the DP–pillar network is assumed to be non-tapered, then the sheath requirements are derived with *GetNonTaperedSheath*.

For each DP–DP link and each edge within these links, the number of sheaths required is calculated and stored as intra-pillar (copper) duct.

2.3.4 Copper pillar connection phase

The latter part of *ConnectClusters* is also used to drive the connection of pillars to RAU. Firstly, the largest number of DP locations to be found in any pillar is calculated and the arrays lClusterIndex(), lInverseClusterIndex(), lClusterPairIndex() are resized.

RunAtPillarClusterLevel, which is found in the *PillarClusterToPillarcluster* module, then runs the following subroutines:

- *SetUpClusterPairIndex*
- *SortPairsOfPillarClusters*
- *RoutePointsForPillarCluster*
- *ApplyDijkstraForPillarClusters* (within *RoutePointsForPillarCluster*).

SetUpClusterPairIndex

Location:	Found in the *PillarClusterToPillarcluster* module
Purpose:	This sets up an index for pairs of pillar clusters, so that each unordered pair occurs exactly once. The index uses triangular numbers: e.g. for four pillars, (P1,P2) → 1, (P1,P3) → 2, (P1,P4) → 3, (P2,P3) → 4, (P2,P4) → 5, (P3,P4) → 6

SortPairsOfPillarClusters

Location: Found in the *PillarClusterToPillarcluster* module

Purpose:
- For each pair of pillar clusters in the ESA:
  - calculate how many unique pairs of DPs there are which have one DP from each pillar cluster
  - explicitly identify these pairs of DPs and, for each pair, calculate the distance between the two DPs using the p-function
  - sort the pairs of DPs in order of this distance, with the closest pair of DPs listed first.

RoutePointsForPillarCluster

Location: Found in the *PillarClusterToPillarcluster* module

Purpose:
- Finds the least proxy cost route for connections between pillar clusters and then connections to the RAU.

- This assumes that:
  - if the route for a pillar location back to the RAU goes through other pillar clusters, then it passes through their pillar locations
  - it also assumed that each pillar–RAU link is a discrete cable.

- The proxy cost function when costing up linkages with new trench assume additional trench cost [uses a cost multiplier of $M_1 = k_1 * + (k_3 *$ pillar–RAU cabling capacity $C_D) + (k_4 * \sqrt{C_D})$, to multiply by the new trench length L_N].

- The proxy cost function when costing up linkages through existing trenches assume no additional trench cost [uses a cost multiplier of $M_2 = (k_3 *$ pillar–RAU cabling capacity $C_D) + (k_4 * \sqrt{C_D})$, to multiply by the existing trench length L_E].

- When costing the links between any two pillar locations, for each unique pair of pillar clusters:
  - identify the pair of DPs (with one from each pillar cluster) which are the closest (a distance *d* apart, calculated with the p–function)
  - calculate the sum of their distances back to their respective pillars, D_T
  - the proxy cost of linking the two pillars clusters together through these DPs is then assumed to be $(M_1 * d) + (M_2 * D_T)$

- For comparing the costs of linking two pillars, since the cable capacity between a pillar and the RAU is constant, jointing proxy costs are not included in our sum, so k_2 is not included above.

- For each pair of pillar clusters, identify the pair of DPs which give the lowest linking proxy cost.

- Add on the jointing cost for each of these best linkages.

- This gives us a fully meshed set of linkages between all pillars.

- These linkages are stored in the array C2CEdgePillar().

ApplyDijkstraForPillarClusters (within RoutePointsForPillarCluster).

Location:	Found in the *PillarClusterToPillarcluster* module

Purpose:

- Apply the Dijkstra algorithm to derive a least proxy cost route between any pillar and the RAU, using these linkages.
- For every other pillar, recall the requirements for linking it to the RAU:
  - extra trench
  - cost of linking the two pillars
  - cabling cost of linking the two pillars (i.e. excluding trench cost)
  - (effectively) total sheath length between them.
- Assume, provisionally, that all pillars are connected directly to the RAU.
- Start with the RAU.
- Then, execute the following loop whilst there are still unconnected pillars:
  - for a given (connected) pillar i, look through the remaining unconnected pillars and decide which pillar (*j*) is the most cost effective to link directly to pillar i
  - then, for each unconnected pillar *k*, test to see if there is a cheaper proxy cost in linking it back to the RAU by going through i, or via the current provisional path, by:
    - o calculating all the extra jointing costs of going through pillar i (possibly via other pillars) back to the RAU, rather than its existing path
    - o if ([total cabling cost of linking *k* to RAU via i] +[extra jointing costs of linking *k* to RAU via i] + [the cost of linking *k* to pillar i]) < cost of linking *k* to the RAU directly, then set the link for *k* to be via pillar i
  - set i to be the pillar that was just connected (*j*) and return to the start of the loop.
- When all pillars have been connected, calculate:
  - total extra trench required to join all pillars to the RAU
  - incremental copper sheath required to link the pillar location back to the RAU, for each pillar cluster
  - for each pillar, the previous pillar on its way back to the RAU, stored in glPreviousPillarCentre().

2.3.5 Fibre determination phase

Having completed the trench network for the copper network, the algorithm then seeks to overlay a fibre network to serve the locations of high demand. There are two paths to follow for this section depending on whether the fibre implementation uses a series of fibre rings or point-to-point connections. These are handled by the subroutines *IncludePillarsInFibreRingForHighDemand* and *LinkFibrePointsDirectlyToPillar* respectively, which are both found in the *BuildFibreRing* module.

IncludePillarsInFibreRingForHighDemand

For the urban deployment, this subroutine derives a series of fibre rings passing through some or all of the pillar locations. The extra trench and cables required to create this rings is derived. The incremental fibre sheath and cable required to link the fibre-fed locations back to their parent pillar and onto the fibre ring are then also calculated.

- For each pillar cluster, the number of DPs are identified.
- The pillar clusters that are to be included within the fibre rings are then identified: this may be all of them, or it may only be those that serve fibre-fed locations. Those pillars to be included are flagged using the array gBolPillarClusterInAFibreRing().
- The subroutine *ClusterNodesForFibreRing* groups the identified pillars into a set of clusters, each one to be served by a ring.

► *ClusterNodesForFibreRing*

Location: Found in the *BuildFibreRing* module

Purpose:

- The data contained within the arrays used in the clustering algorithms is first backed up into local arrays. In particular, the data contained within gobjInputPoints() is stored in objgIndividualPoints_FibreRing().
- The clustering arrays are then re-populated with the data for the pillars to be put into a fibre ring.
- Using the maximum number of nodes that can be in a fibre ring (glMaxNodesInFibreRing), the number of rings required is then calculated. This in turn sets the capacity constraint for the fibre ring clustering, by attempting to achieve a balance in the number of pillars in each ring.
- The subroutine *DivisiveClustering*, found in the *Clustering* module, is then used to cluster the pillars into fibre rings. An

effectively infinite distance constraint is used and the design of the clustering algorithms means that the RAU always lies in each of the clusters.

- The allocation of each pillar to a fibre ring is then stored in glPillarClusterToFibreRing() and the clustering arrays are restored to their original values.
- Each fibre ring is then mapped into the array glFibreRingToPillarMapping(), defined by:

  glFibreRingToPillarMapping(j,k) = *k*th pillar in the *i*th fibre ring

- The first pillar in each ring is always defined to be the RAU.

- For each ring, the total number of fibres that the ring serves across all the pillar (barring the RAU) is calculated. This assumes a fixed number of fibre (glFibrePairsDPToPillar) for each fibre-served location. This number of fibres is stored in glTotalFibreDemandOnRing().
- The subroutine *RunAtFibreRingLevel* then identifies the lowest cost linkages between each pair of pillar location, assuming the existing trench network for the copper served locations.

► *RunAtFibreRingLevel*

Location: Found in the *PillarClusterToPillarCluster* module

Purpose:

- This runs several subroutines similar to those run in the copper pillar connection phase. These subroutines are:
- *SetUpClusterPairIndex*, as explained in section 2.4.5
- *SortPairsOfPillarClusters*, as explained in section 2.3.4
- *RoutePointsForFibreRing*, as explained below.

► *RoutePointsForFibreRing*

Location: Found in the *PillarClusterToPillarCluster* module

Purpose:

- Finds the least proxy cost route for connections between any two pillar clusters.
- The proxy cost function when costing up linkages with new trench assume additional trench cost [uses a cost multiplier of $M_1 = k_1 * + (k_3 * \text{DP–pillar cabling capacity } C_D) + (k_4 * \sqrt{C_D})$, to multiply by the new trench length L_N].
- The proxy cost function when costing up linkages through

existing trenches assume no additional trench cost [uses a cost multiplier of $M_2 = (k_3 * \text{DP–pillar fibre cabling } C_D) + (k_4 * \sqrt{C_D})$, to multiply by the existing trench length L_E].

- A single estimate of the total fibre cable used in any pillar–pillar link is made by taking the maximum of the cable sizes required on the fibre rings, estimated using *ProxyCableToUseInFibreRing*. This is only used within the proxy cost functions.

- When costing the links between any two pillar locations, for each unique pair of pillar clusters:
  - identify the pair of DPs (with 1 from each pillar cluster) which are the closest (a distance *d* apart, calculated with the p–function)
  - calculate the sum of their distances back to their respective pillars, D_T.

- The proxy cost of linking the two pillars clusters together through these DPs is then assumed to be $(M_1 * d) + (M_2 * D_T)$.

- For each pair of pillar clusters, identify the pair of DPs which give the lowest linking proxy cost.

- This gives us a fully meshed set of linkages between all pillars.

- These linkages are stored in the array C2CFibreRing().

► *ProxyCableToUseInFibreRing*

Location: Found in the *BuildFibreRing* module

Purpose: This identifies a single fibre cable size to use for the proxy cost function used to determine the lowest cost paths between any pair of pillar locations.

- For each fibre ring, the smallest fibre cable size that can accommodate the total capacity on that ring is identified.

- If no single cable size has enough capacity, then the combination of the two largest cable sizes that gives enough capacity is identified.

- The largest cable requirement across all fibre rings is then used as the single estimate of the required cable size.

For each fibre ring, the following subroutines are then run to determine the best ring formation for the pillars in this ring:

- RunTheTSPAlgorithm
- UpdateFibreCableAndTrenchArrays
- TotalTrenchForThisFibreRing
- TotalCableForThisFibreRing.

► *RunTheTSPAlgorithm*

Location:	Found in the *RunTSP* module
Purpose:	This version of the algorithm is almost identical to that of the subroutine *RunTSP* in the core network algorithms, as described in section 3.3. It uses a Travelling Salesman Problem (TSP) algorithm to derive the most efficient ring structure to join a set of points. Versions of the relevant Visual Basic code are contained in: • the modules *InitialiseTSP*, *RunTSP* and *ShortestRing* • the class modules *clsClusterPair*, *clsRing*, *clsTSPData* and *clsTSPInputData*.

► *UpdateFibreCableAndTrenchArrays*

Location:	Found in the *BuildFibreRing* module
Purpose:	For a given fibre ring, this stores particular data about the fibre ring, including: • the next pillar in the ring for each given pillar • whether, for each link in the ring, the trench already exists from the copper deployment or not • the pillar that is first on the ring after the RAU.

► *TotalTrenchForThisFibreRing*

Location:	Found in the *BuildFibreRing* module
Purpose:	Calculates the extra trench required to connect the pillars in a given fibre ring.

► *TotalCableForThisFibreRing*

Location:	Found in the *BuildFibreRing* module

Purpose: Calculates the total fibre sheath and cable lengths required to connect the pillars in a given fibre ring

- For the given fibre ring, the smallest fibre cable size that can accommodate the total capacity on that ring is identified.
- If no single cable size has enough capacity, then the combination of the two largest cable sizes that gives enough capacity is identified.

Having calculated the trench and cable requirements for the fibre rings, *IncludePillarsInFibreRingForHighDemand* then calculates:

- The fibre sheath needed to join each fibre-fed location back to its parent pillar, by using the same path followed by the copper network (each fibre-fed location has a nominal unit of demand served by the copper network).
- The fibre cable length needed to join each fibre-fed location back to its parent pillar on a pillar-by-pillar basis, by multiplying the total FDP–DP and DP–pillar fibre sheath lengths by their respective assumed fibre cable sizes.
- The fibre-specific duct required, using *CalculateFibreDuctByType* for links back to the pillar and *CalculateDuctsForFibreBackFromPillar* for links from the pillar back to the RAU.

LinkFibrePointsDirectlyToPillar

Location: Found in the *BuildFibreRing* module

Purpose: This calculates the fibre sheath and cable length requirements for joining each fibre-fed location back to its parent RAU via its parent DP and pillar. This uses the path determined by the nominal unit of demand assigned to each fibre-served location in the copper network.

This function does not need incremental trench, since the existing trench network is assumed to be used as the path back to the RAU.

For the urban deployment:

- The number of DPs in each pillar cluster are calculated.
- Fibre-fed locations are flagged in the array bolPointFedByFibre().
- DPs served only by fibre are flagged by bolDPServedByFibre().

- The fibre sheath needed to join each fibre-fed location back to its parent RAU is calculated, by using the same path followed by the copper network.
- The fibre length needed to join each fibre-fed location back to its parent RAU is calculated for each pillar cluster, by multiplying the total FDP–DP, DP–pillar fibre sheath lengths by their respective assumed fibre cable sizes.
- The pillar-RAU fibre sheath length is assumed to be the same as that for the analogous connection in the copper network
- The pillar-RAU fibre length is assumed to be the fibre sheath length multiplied by the assumed fibre cable size in the DP–pillar part of the network.
- A discrete cable is assumed for each fibre-fed location all the way back to the RAU (point-to-point architecture).
- The amount of fibre-specific duct required is calculated using *CalculateFibreDuctByType* for links back to the pillar and *CalculateDuctsForFibreBackFromPillar* for the links from the pillar back to the RAU.

► *CalculateFibreDuctByType*

Location:	Found in the *CalculateDuct* module
Purpose:	This calculates the duct required for the entire route for each fibre cable from the fibre-fed FDPs back to the pillar.

► *CalculateDuctsForFibreBackFromPillar*

Location:	Found in the *CalculateDuct* module
Purpose:	Where point-to-point fibre is used, this calculates the duct required for the entire route for each fibre-fed FDP from its parent pillar back to the RAU. Where fibre rings are used, this calculates the duct required for the entire route for each pillar-pillar link in the ring(s).

2.3.6 Backhaul determination phase

Having calculated the copper and fibre networks for an urban deployment for the ESA, the backhaul requirements for each access node are then derived. For example, pillars may be too far from the RAU to be linked by copper, in which case a large pair gains system (LPGS) is installed and a fibre link replaces the copper link. This is accomplished by the subroutine *DetermineBackhaulForCopperServedAreas*.

DetermineBackhaulForCopperServedAreas

Location: Found in the *WirelessAndSatellite* module

Purpose: First of all, this subroutine analyses the cable between the copper nodes and the RAU, in order to remove double-counted cables. This is accomplished using the subroutine *RemoveUrbanDoubleBackMainCable*, as described below.

Secondly, it identifies whether a copper node should in fact be an LPGS and, if so, whether it should have fibre, wireless or even satellite backhaul. It runs through each pillar in turn:

- The RAU cannot be an LPGS, so it is labelled as a RAU.
- For all other pillars, the maximum loop length across all the locations in the pillar cluster (from FDP to RAU) is calculated.
- If this distance is less than a maximum threshold for using an LPGS(gdMaxCableDistanceBeforeUsingLPGS), then a pillar is still used and the jointing required between the pillar and the RAU is derived and stored in gdPillarRAUJointing().
- If this distance is higher than the threshold, then an LPGS is required. The type of backhaul link is then determined as follows:
  - if the network is either including all pillars in a fibre ring, or is linking all pillars with fibre-fed locations into fibre rings and this pillar has fibre-fed locations, then it will already have a backhaul ink via the fibre ring. So, we remove the pillar–RAU link, provided that it does not form part of the fibre ring
  - for the urban deployment, LPGS are otherwise assumed to be linked to the RAU by a fibre.

Having completed the backhaul determination, the nature of each location in the ESA can then be finalised using the subroutine *DetermineLocationType*.

► *RemoveUrbanDoubleBackMainCable*

Location:	Found in the *PillarClusterToPillarCluster*module
Purpose:	For each pillar cluster in turn, this subroutine considers the route that the cable takes from the pillar back to the RAU. Specifically,:

- counts the number of times that this cable passes through each link in the trench network
- reduces this count where it is more than 1 to remove instances where the cable doubles back on itself
- adjusts the recorded length of the cable for the pillar in gdSheathLengthToConnectPillarClusters() accordingly

► *DetermineLocationType*

Location:	Found in the *OutputResults* module
Purpose:	For the urban deployment, this subroutine first identifies all DP locations, labelling all other locations as FDPs. It then overwrites the main node locations, namely the RAU, the pillars and LPGS.

2.3.7 Result storage phase

The remaining outputs of the network asset volumes are printed to the output worksheet for the ESA by the subroutine *OutputTheResults*.

OutputTheResults

Location:	Found in the *OutputResults* module
Purpose:	This prints the remaining network volumes to the output worksheet for the ESA in the relevant Access DATA workbook.

Specifically, the subroutine:

- calculates the average loop length for each pillar cluster
- prints the network volumes for the cluster containing the RAU
- prints the aggregated volumes for the ESA, including:
  - trench between pillars and the RAU
  - fibre sheath for the fibre rings
  - fibre length for the fibre rings
  - number of fibre rings
  - number of relay stations
- prints the network volumes for every other pillar cluster

- prints the identity of the next pillar on the fibre ring for each pillar, if applicable
- prints the fibre links from the RAU if there is more than one fibre ring (the line for the RAU cluster can only indicate one of these links)
- prints the pillar indices for each DP cluster, into column U
- prints the edges in the spanning trees at the DP–pillar and pillar–RAU level of the network, using *WriteConnectClustersResults*
- prints the incremental trench for the fibre rings, using *WriteFibreRingResults*
- prints the duct requirements for each link in the trench network, using *WriteDuctOutputs*
- identifies each location as being served by either copper or fibre
- estimates the cable lengths by cable size, using *CalculateTotalSheathLengthByCableSize*, as explained below.

► *WriteFibreRingResults*

Location: Found in the *BuildFibreRing* module

Purpose: Identifies the incremental trench links required for the fibre rings and calculates the additional manholes required (if any) on these links.

► *WriteDuctOutputs*

Location: Found in the *CalculateDuct* module

Purpose: For each link in the trench network, the number of ducts *required* by type are calculated, determined by how many cables of each type there are passing through the link and the capacity of each type of duct.

The number of ducts that are *provisioned* (based on the allowed multiples) is also determined for each link. The length of each link, using either crow-flies or p-function, is also derived.

Finally, the type of pit required at each DP location is also determined.

► *CalculateTotalSheathLengthByCableSize*

Location: Found in the *OutputResults* module

Purpose: For the urban deployment, this estimates the cable lengths by cable size within the DP clusters:

- for each level of demand, the number of sheaths of each cable size used to serve that demand are stored in the array glCableRequirementsByDemand()

- across all locations, the sheath requirements for each cable size are then aggregated in the array gdSheathLengthByCableSize()
- the total length required of each cable size is then printed in the output worksheet for the ESA
- the sheath length by cable size within the distribution network is also printed here, having been pre-calculated in *ApplyDijkstra*.

2.3.8 Assumption storage phase

The assumptions used within the calculation are printed onto the output worksheet for the ESA, using the subroutines *RecordAssumptions.*

RecordAssumptions

Location:	Found in the *CommonCode* module
Purpose:	This prints all of the assumptions used in the calculation of access network asset volumes.

For an urban deployment, it prints:

- capacity and distance constraints for the nodes used in the network
- technical constraints for the fibre rings and copper jointing
- cables used in the non-tapered distribution network (if applicable)
- coefficients for the p–function used
- coefficients for the proxy cost function
- cost assumptions used (in the urban case, for the fibre and wireless backhaul cost comparisons for LPGS backhaul).

Finally, in order to reduce congestion in the computer's memory, the subroutine *EraseArrays* (found in the *MainMacros* modules) uses the Erase statement to destroy global arrays populated separately for each ESA, releasing the allocated memory.

2.4 Rural deployment path

There are thirteen phases when using a rural deployment for an ESA:

- initial copper clustering phase, which is described in section 2.4.1
- copper or wireless determination phase, which is described in section 2.4.2
- copper clustering phase, which is described in section 2.4.3

- copper pillar cluster spanning tree phase, which is described in section 2.4.4
- copper cluster connection phase, which is described in section 2.4.5
- backhaul determination phase, which is described in section 2.4.6
- fibre determination phase, which is described in section 2.4.7
- copper result storage phase, which is described in section 2.4.8
- wireless clustering phase, which is described in section 2.4.9
- satellite determination phase, which is described in section 2.4.10
- wireless backhaul determination phase, which is described in section 2.4.11
- copper and fibre result storage phase, which is described in section 2.4.12
- assumption storage phase, which is described in section 2.4.13.

2.4.1 Initial copper clustering phase

This is run through the subroutine *AllClusteringMethods*, which is found in the *MainMacros* module. Three subroutines are driven by *AllClusteringMethods* in the rural case:

- *IdentifyRAU*
- *DivisiveClustering*
- *PassDirectToPillarClusterLevel*.

These are explained in more detail in the following sub-sections.

IdentifyRAU

Location:	Found in the *Clustering* module
Purpose:	Determines a location from a list of input locations to be a suitable location for the RAU in an ESA. Currently, each ESA in the Access DATA workbook uses the first location in the list as the location of the RAU, since that is where the actual location given by ExchangeInfo is stored. In this case, the value of the variable glStatedRAUvertex must be positive and the RAU is assumed to be at this location. Otherwise, the location closest to the demand-weighted centre of the locations in the ESA is used. There are three sets of objects that can be used for this calculation by *IdentifyRAU*: locations and pillars. In relation to the rural clustering phase, the individual locations are used.

DivisiveClustering

In this case, FDPs are grouped directly into clusters served by pillars, using the subroutine *DivisiveClustering.* This clusters a set of locations based on a capacity and a distance constraint. For the rural deployment, the initial clustering stages are the same as the urban deployment.

However, when the refinement stages start, only the following subroutines are used:

- *SimpleReassignment*
- *Swap*
- *FullOptimisation*
- *HighDemandSimpleReassignment*.

These are reused frequently throughout the code for different clustering requirements and are explained below. All of the subroutines can move any locations in the rural deployment, since the *FullAccessNetworkBuild* subroutine will not calculate ESAs with both a rural deployment and a fibre ring deployment.

PassDirectToPillarClusterLevel

Location:	Found in the *Clustering* module
Purpose:	This subroutines edits the arrays used so that the second stage of the clustering that occurs in the urban deployment is circumvented for the rural deployment.

Pillar clusters are consolidated using *ConsolidatePillars*. This runs exactly as specified in the urban deployment in section 2.3.1, except that merges are stated to be possible only if the two pillar cluster taken together would satisfy the (absolute) pillar capacity constraint and distance constraint with their new pillar location.

The pillar locations are identified using the subroutine *IdentifyPillar*.

The subroutine *TransferSmallCopperAreasToWireless* is not used at this stage in the code: see section 2.4.3.

2.4.2 Copper or wireless determination phase

Up until this point, the path traced by the rural deployment has been almost identical to the urban deployment. The path then diverges into a subroutine called *CalculateWirelessAndSatelliteServedDemand* in the *WirelessAndSatellite* module, which governs the rest of the rural deployment.

The first step is to determine, on a cost basis, whether a location should be served by copper or wireless technology (the 'copper-wireless decision'). This is completed by the subroutine *ChooseRadioOrCopperCluster*.

ChooseRadioOrCopperCluster

Location: Found in the *WirelessAndSatellite* module

Purpose: The pillar locations that have already been calculated are stored. The G–NAF co-ordinates corresponding to each of the locations are recalled and then the locations are clustered according to the wireless assumptions.

The decision then begins by assuming that all of the pillar locations are served by copper, with everything else served by wireless.

As described in the main documentation, two deployment scenarios are investigated (enumerated by the variable lComparison). Each of these deployment scenario checks through each of the pillar clusters in turn twice (enumerated by the variable lIteration).

The value of lComparison and lIteration determine the copper capacity/distance constraints used in the heart of the subroutine, as summarised in Table 2.2 and Table 2.3 below:

	lComparison =1	*lComparison =2*
lIteration=1	Minimum (or 'critical') capacity (~20 units of demand)	
lIteration=2	Standard pillar capacity used in copper clustering	

Table 2.2: Copper capacity constraints employed [Source: Analysys]

	lComparison =1	*lComparison =2*
lIteration=1	Based on distance from pillar	LPGS assumed in all cases,

Table 2.3: Copper distance

```
lIteration=2
```

The heart of the subroutine proceeds as follows and runs for each deployment scenario and pass of the pillar locations:

- The first pillar cluster to be checked is always the RAU location.
- For the current pillar location, the wireless-fed point closest to that pillar is identified and then the point in the copper cluster that is closest to this wireless-fed point is identified.
- The wireless-fed location is assigned to this copper cluster and the average cost per unit demand are calculated for:
  - the copper cluster
  - the original wireless cluster.
- If the average copper cost is lower than the average wireless cost, then the pillar location is re-calculated assuming that this location is now fed by copper.
- If the updated copper cluster is found to satisfy the distance constraint, then the wireless-fed location becomes a copper-fed location permanently. The total costs of each cluster are also updated.

Several clean-up processes then refine the outputs for each of the deployment scenarios. The first checks whether a copper cluster that has survived the process is in fact surrounded by another copper cluster. If so, then the two clusters are merged. Secondly, any clusters that are smaller than the minimum capacity are converted to wireless. The copper and wireless cluster costs are updated accordingly.

The deployment scenario that gives the lowest total cost is taken as the final output of the algorithm. Finally, each location is provisionally stated in the output worksheet for the ESA as being served by either copper or wireless based on this decision.

Having completed the copper-wireless decision, *CalculateWirelessAndSatelliteServedDemand* checks whether the RAU is served by wireless or copper. If the former is true, but there exist other locations served by copper, then the RAU is reset to be served by copper, to be consistent with the scorched-node assumption.

2.4.3 Copper clustering phase

Having identified the subset of points that are said to be served by copper, these points are fed into the *AllClusteringMethods* subroutine. This executes *IdentifyRAU* and *DivisiveClustering* as described in section 2.4.1. However, when executing the *PassDirectToPillarClusterLevel* subroutine, there is an extra stage that is completed at the end using the subroutine *TransferSmallCopperAreasToWireless*.

TransferSmallCopperAreasToWireless

Location:	Found in the *WirelessAndSatellite* module
Purpose:	This subroutine is used after the copper-wireless decision and looks at the subset of points designated as served by copper following their re-clustering using the copper assumptions. If clusters are identified which have less than the minimum capacity, then the points within these clusters are stated to be served by wireless and the necessary arrays are updated.

2.4.4 Copper pillar cluster spanning tree phase

As with the urban deployment, *ConstructTreeFollowingClusterMainPointIdentification* is then used to derive minimum spanning trees within each of the pillar clusters. As stated previously, the rural deployment does not use DP clusters. In the rural deployment, it is often attempting to create a spanning tree of over 300 points, so this process can take much longer per cluster than in the urban deployment. This begins with the subroutines *GetMaxPointsInCluster* and *SetupArraysForSpanningTree*.

GetMaxPointsInCluster

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Identifies the maximum number of points in a pillar cluster.

SetupArraysForSpanningTree

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose: Dimensions the key arrays for the minimum spanning tree process:

- glUnattachedPoint() – the list of locations in the cluster that are unattached at any point in the algorithm

- glAttachedPoints() – the list of locations in the cluster that are attached at any point in the algorithm
- glVertexRoute() – for any location, the location that it passes through in order to get back to the node location
- gdDistanceMatrix() – stores the distance between any two points in the cluster
- gobjEdges() – stores the vertices and lengths associated with each edge in the spanning tree.

Following these two subroutines, each cluster is then treated in turn. The following are used in order to create and store the minimum spanning trees:

- *SetupPointsInCluster* – identifies the central point in the cluster
- *SetupGdDistanceMatrix* – calculates the required distances
- *ConstructTree* – constructs the minimum spanning tree for the cluster
- *StoreRoutes* – for each point P, identifies the point P passes through to get back to the node
- *GetTotalDistance* – calculates the total trench within the tree
- *GetRuralTaperedSheathLength*/*GetRuralNonTaperedSheathLength* – calculates the total copper sheath within the tree depending on the nature of the cabling network deployed
- *GetRuralTaperedCopperLength*/*GetRuralNonTaperedCopperLength* – calculates the total copper pair length within the tree depending on the nature of the cabling network deployed
- *WriteCopperNetworkResults* – stores the list of edges in the spanning tree on the output worksheet for the ESA.

SetupPointsInCluster

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• Calculates the number of points in the cluster. • Identifies the pillar location for the cluster and states this location as the central point cp.

SetupGdDistanceMatrix

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For the rural deployment, calculates the p-function distance between any two pairs of points in the pillar cluster and stores these distances in the array gdDistanceMatrix().

ConstructTree

In order to derive the spanning tree for the cluster, the algorithm begins with the central point (cp) identified in *SetupPointsInCluster*. All other locations in the cluster are assumed to be unattached. Locations are then added to the tree incrementally and the lists of attached and unattached locations are updated.

For each incremental addition, the algorithm cycles through each pair of attached and unattached points and calculates the average cost per line of the whole tree were the two points linked together by a trench and a cable. This average cost is determined by *AverageCostPerLine*.

► *AverageCostPerLine*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For the rural deployment, it determines:

- the extra capacity and the copper pair requirements (*c*) needed to serve the unattached point, depending on the nature of the cabling network within the pillar cluster:
  - for tapered, glCablingForRuralDemandIncUtilisation() is used, which was populated using inputs from the 'Inputs' worksheet and accounts for the cable utilisation assumed in the network
  - for non-tapered, *GetNonTaperedCableSize* and *GetNonTaperedSheath* are used
- the length of extra trench distance (*d*) to link the attached and unattached point.

The cost of the new link is then calculated using the expression

$$k_1*d + k_2*c + k_3*d*c + k_4*\sqrt{c}$$

and calculates the new cost per unit of demand for the entire tree.

For each unattached point, the edge to an attached point that gives the lowest new average cost per line is stored in the array objEdgeList() using *AddToEdgeList*.

► *AddToEdgeList*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Having identified the best edge by which to join a particular point to the existing tree, this subroutine stores the vertices of this edge and its average cost per unit of demand.

This is repeated for every unattached point. The edge in objEdgeList() that gives the lowest new average cost per line is then linked to the tree using *AddCheapestEdgeInListToObjEdges*.

► *AddCheapestEdgeInListToObjEdges*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• Adds the best link in objEdgeList(), in terms of average cost per line, to the list of edges for the minimum spanning tree. • Updates the array glVertexRoute() for this new edge in the tree, defined by: glVertexRoute(unattached location on new edge) = attached locations on new edge • Updates the total copper length required to link the location all the way back to the node.

The number of unattached points is reduced by 1 and the lists of attached and unattached points updated using *IdentifyAttachedAndUnattachedPoints*, which can be found in the *ModifiedPrimSpanningTree* module.

The loop in *ConstructTree* continues until all locations in the cluster have been attached to the cluster node.

StoreRoutes

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For each point P in the cluster, stores the point Q that it passes back through in order to reach the cluster node, with the array defined as: glRouteToCentre(P)=Q

GetTotalDistance

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total trench in the spanning tree for a pillar, by cycling through all points P in the cluster and calculating the distance between P and the point that it passes through on it way back to the

pillar.

GetRuralTaperedSheathLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total cable sheath in the spanning tree assuming that the cabling is tapered. For each point P it calculates the distance between P and its predecessor back to the pillar. The number of sheaths needed for the cabling are determined on the basis of the downstream capacity at that point, using the array glCablingForRuralDemandIncUtilisation(). This accounts for the assumed utilisation of cable. The number of sheaths is then multiplied by the length of the link between P and its predecessor and this is aggregated onto the total.

GetRuralNonTaperedSheathLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total cable sheath in the spanning tree assuming that the cabling is non-tapered. For each point P it calculates the distance between P and its predecessor back to the pillar. The number of sheaths needed for the cabling are determined on the basis of the downstream capacity at that point, using the function *GetNonTaperedSheath*, which is explained below. This accounts for the assumed utilisation of cable. The number of sheaths is then multiplied by the length of the link between P and its predecessor and this is aggregated onto the total.

GetRuralTaperedCopperLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total copper pair length in the spanning tree assuming that the cabling is tapered. For each point P, it calculates the distance between P and its predecessor back to the pillar. The number of copper pairs needed for the cabling are determined on

the basis of the downstream capacity at that point, using the array glCablingForRuralDemandIncUtilisation(). This accounts for the assumed utilisation of cable.

The number of copper pairs is then multiplied by the length of the link between P and its predecessor and this is aggregated onto the total.

In addition, the size of the sheath deployed is derived and the total length required of that size is aggregated into the array gdDistnNetworkSheathBySize(), which stores total length of sheath by cable size.

GetRuralNonTaperedCopperLength

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Calculates the total cable pair length in the spanning tree assuming that the cabling is non-tapered. For each point P, it calculates the distance between P and its predecessor back to the pillar.

It determines the number of pairs needed for the cabling on the basis of the downstream capacity at that point, using the functions *GetNonTaperedSheath* and *GetNonTaperedCableSize*, which are explained below. These account for the assumed utilisation of cable.

The number of copper pairs is then multiplied by the length of the link between P and its predecessor and this is aggregated onto the total.

In addition, the size of the sheath deployed is derived and the total length required of that size is aggregated into the array gdDistnNetworkSheathBySize(), which stores total length of sheath by cable size.

► *GetNonTaperedSheath*

Location:	Found in the *ClusterToCluster* module
Purpose:	Given a capacity in terms of units of demand, it determines the cable size required in a non-tapered network. This accounts for the assumed level of cable utilisation in the distribution network.

Non-tapered cabling is assumed to come in only two types: a main cable size and a minor cable size. If the capacity required, having accounted for utilisation, is smaller than the minor cable capacity then the minor cable is used. Otherwise, the necessary multiples of the main cable is used.

If the main cable is used, then the number of sheaths is calculated by rounding up the ratio of the capacity required and the main cable size.

If the minor cable size is assumed to be zero, then the main cable size is always used. This is a default assumption.

► *GetNonTaperedCableSize*

Location:	Found in the *ClusterToCluster* module
Purpose:	Given a capacity in terms of units of demand, it determines the cable size required in a non-tapered network. This accounts for the assumed level of cable utilisation in the distribution network.

Non-tapered cabling is assumed to come in only two types: a main cable size and a minor cable size. If the capacity required, having accounted for utilisation, is smaller than the minor cable capacity then the minor cable is used. Otherwise, the main cable is used.

WriteCopperNetworkResults

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose:

- Writes the points and their co-ordinates in the output worksheet for the ESA (in rows BF–BM) that define every edge in the spanning trees for the pillar clusters.
- Determines how many extra pits are required along these edges.
- Writes the copper locations, their co-ordinates and their parent pillar in the output worksheet for the ESA (in rows BX–CD).

2.4.5 Copper cluster connection phase

This phase is run through the *ConnectClusters* subroutine, which is found in the *ClusterToCluster* module.

After this revision, we then join up all the pillar locations in the ESA through the subroutine *RunAtRuralPillarClusterLevel*, which also lies in *ClusterToCluster* module. This subroutine is only used if there is more than one pillar cluster.

RunAtRuralPillarClusterLevel executes several subroutines:

- *SetUpClusterPairIndex*
- *IndexPointsWithinPillarCluster*
- *SortPairsOfPoints*
- *RoutePointsForRuralPillarCluster*
- *ApplyDijkstra* (contained within *RoutePointsForCluster*).

SetUpClusterPairIndex

Location:	Found in the *ClusterToCluster* module
Purpose:	Indexes pairs of pillars in an ESA so that each unordered pair occurs exactly once: use triangular numbers: e.g. for four pillars, (P1,P2) → 1, (P1,P3) → 2, (P1,P4) → 3, (P2,P3) → 4, (P2,P4) → 5, (P3,P4) → 6

IndexPointsWithinPillarCluster

Location:	Found in the *ClusterToCluster* module
Purpose:	Creates a new indexing PointIndex() and its inverse InversePointIndex() that are effectively identity mappings in this case: this is included for the process to have a consistent structure with the urban equivalents.

Following *IndexClusterWithinPillarCluster*, the pillar that is the RAU location is identified.

SortPairsOfPoints

Location: Found in the *ClusterToCluster* module

Purpose:

- For each pair of pillars in the ESA:
  - calculate how many unique pairs of points there are which have one point from each pillar cluster
  - explicitly identify these pairs of points and, for each pair, calculate the distance between the two points using the p-function
  - sort the pairs of points in order of this distance, with the closest pair of points listed first.

RoutePointsForPillarCluster

Location:	Found in the *ClusterToCluster* module

Purpose:

- Finds the least proxy cost route for connections between pillar clusters and then connections to the RAU.
- This assumes that, if the route for a pillar location back to the RAU goes through other pillar clusters, then it passes through their pillar locations.
- It also assumed that each pillar–RAU link is a discrete cable.
- The proxy cost function when costing up linkages with new trench assume additional trench cost [uses a cost multiplier of $M_1 = k_1 * + (k_3 *$ pillar–RAU cabling capacity $C_D) + (k_4 * \sqrt{C_D})$, to multiply by the new trench length L_N].
- The proxy cost function when costing up linkages through existing trenches assume no additional trench cost [uses a cost multiplier of $M_2 = (k_3 *$ pillar–RAU cabling capacity $C_D) + (k_4 * \sqrt{C_D})$, to multiply by the existing trench length L_E].
- When costing the links between any two pillar locations, for each unique pair of pillar clusters:
  - identify the pair of points (with 1 from each pillar cluster) which are the closest (a distance *d* apart, calculated with the p–function)
  - calculate the sum of their distances back to their respective pillars, D_T.
- The proxy cost of linking the two pillars clusters together through these points is then assumed to be $(M_1 * d) + (M_2 * D_T)$.
- For comparing the costs of linking two pillars, assuming a constant pillar–RAU cabling capacity means jointing proxy costs will be the same for all pillar cluster pairs, so it is not included in our sum (so k_2 is not included above).
- For each pair of pillar clusters, identify the pair of points which give the lowest linking proxy cost.
- Add on the jointing cost for each of these best linkages.
- This gives us a fully meshed set of linkages between all pillars.

- These linkages are stored in the array C2CEdgePillar().

ApplyDijkstraForRuralPillarClusters (within RoutePointsForRuralPillarCluster).

Location:	Found in the *ClusterToCluster* module

Purpose:

- Apply the Dijkstra algorithm to derive a least proxy cost route between any pillar and the RAU, using these linkages.
- Assume, provisionally, that all pillars are connected directly to the RAU.
- Start with the RAU.
- For every other pillar, identify from storage the requirements for linking it to the RAU:
  - extra trench
  - cost of linking the two pillars
  - cabling cost of linking the two pillars (i.e. excluding trench cost)
  - (effectively) total sheath length between them.
- Then, execute the following loop whilst there are still unconnected pillars.
- For a given (connected) pillar i (with i starting off as the RAU), look through the remaining unconnected pillars and decide which pillar (*j*) is the most cost effective to link directly to pillar i.
- Then, for each unconnected pillar *k*, test to see if there is a cheaper proxy cost in linking it back to the RAU by going through i, or via the current provisional path, by:
  - calculating all the extra jointing costs of going through pillar i (possibly via other pillars) back to the RAU, rather than its existing path
  - if ([total cabling cost of linking *k* to RAU via i] +[extra jointing costs of linking *k* to RAU via i] + [the cost of linking *k* to i]) < cost of linking *k* to the RAU directly, then set the link for *k* to be via pillar i.
- Set i to be the pillar that was just connected (*j*) and return to the start of the loop.
- When all pillars have been connected, calculate:

- total extra trench required to join all pillars to the RAU
- incremental copper sheath required to link the pillar location back to the RAU, for each pillar cluster
- for each pillar, the previous pillar on its way back to the RAU, stored in glPreviousPillarCentre().

As the last step in *ConnectCluster*, the subroutine *CalculateDuctByType* is used to calculate the duct requirements within each pillar cluster.

CalculateDuctByType

Location:	Found in the *CalculateDuct* module
Purpose:	Identifies the number of sheaths by cabling type within each link in the pillar clusters, as preparation for the derivation of the number of ducts within each link in the subroutine *WriteDuctOutputs*.

If the cabling within the pillar cluster is assumed to be tapered, then 1 sheath is assumed to be required within each link.

If the cabling within the pillar cluster is assumed to be non-tapered, then the sheath requirements are derived with *GetNonTaperedSheath*.

For each link in the pillar cluster networks, the number of sheaths required is calculated and stored as intra-pillar (copper) duct.

Back in the subroutine *CalculateWirelessAndSatelliteServedDemand*, the number of branching kits required in the network is calculated and then the function *GetRuralJointingCosts* is used to calculate the jointing required in the network.

GetRuralJointingCosts

Location:	Found in the *WirelessAndSatellite* module
Purpose:	Calculates the jointing requirements within the pillar clusters. For distance-related jointing, we assume a maximum distance that cable can be pulled within the distribution network without a full joint.

Firstly, for each location, it determines the number of copper pairs

	that will be heading back towards the pillar at that location, having accounted for the utilisation of the cable. For the tapered case, this uses glCablingForRuralDemandIncUtilisation. For the non-tapered case, this uses *GetNonTaperedCableSize* and *GetNonTaperedSheath*.
	The demand-related jointing is calculated first on an edge-by-edge basis. For the non-tapered case, this is taken to be the demand at that point. For the tapered case, it is taken to be the demand at that point if it is an extreme point. Otherwise, it is taken to be the total demand downstream of that point. If the edge within the link exceeds the maximum pulling distance in length, then we include additional jointing as required within the edge.
	The distance-related jointing is only calculated explicitly for the non-tapered case. For each location, its predecessor back to the pillar is calculated. The cable distance of both locations back to the pillar are known. If a multiple of the maximum pulling distance is passed within the link, then a full joint is assumed to occur on the cable at the location nearest to the pillar.

Back in the subroutine *CalculateWirelessAndSatelliteServedDemand*, the demand fed by fibre in each pillar cluster is calculated using the subroutine *CalculateFibreUnitsOfDemand*.

CalculateFibreUnitsOfDemand

Location:	Found in the *Clustering* module
Purpose:	Aggregates fibre demand on a pillar cluster basis for the purpose of printing the outputs for this ESA. For each fibre-fed location, one unit of demand is assumed to be still fed by copper: the remainder by copper.

2.4.6 Backhaul determination phase

Having calculated the copper and fibre networks for a rural deployment for the ESA, the backhaul requirements for each access node are then derived.

For example, pillars may be too far from the RAU to be linked by copper, in which case a LPGS is installed and a backhaul link replaces the copper link. This is accomplished by the subroutine *DetermineBackhaulForCopperServedAreas*.

DetermineBackhaulForCopperServedAreas

Location: Found in the *WirelessAndSatellite* module

Purpose: First of all, this subroutine analyses the cable between the copper nodes and the RAU, in order to remove double-counted cables. This is accomplished using the subroutine *RemoveRuralDoubleBackMainCable*, as described below.

Secondly, it identifies whether a copper node should in fact be an LPGS and, if so, whether it should have fibre, wireless or even satellite backhaul. It runs through each pillar in turn:

- The RAU cannot be an LPGS, so it is labelled as a RAU.
- For all other pillars, the maximum loop length across all the locations in the pillar cluster (from FDP to RAU) is calculated.
- If this distance is less than a maximum threshold for using an LPGS (gdMaxCableDistanceBeforeUsingLPGS), then a pillar is still used and the jointing required between the pillar and the RAU is derived and stored in gdPillarRAUJointing().
- If this distance is higher than the threshold, then an LPGS is required. The type of backhaul link is then determined as follows:
- If the network is either including all pillars in a fibre ring, or is linking all pillars with fibre-fed locations into fibre rings and this pillar has fibre-fed locations, then it will already have a backhaul link via the fibre ring. So, we remove the pillar–RAU link, provided that it does not form part of the fibre ring. The rural deployment should never use fibre rings, so this is irrelevant for the rural case.
- Otherwise, the cost of linking the LPGS by a fibre and a wireless link is calculated. The wireless cost includes the cost of relay stations, which are derived by using *DeriveWirelessLinkQuantities*, which is explained below.
- If the fibre cost is the cheaper option, then the pillar–RAU copper link is replaced with a fibre cable.
- If wireless is the cheaper option, but needs more than a certain number of relay stations (glMaxNumRelaysInWiBackhaul), then the copper link is replaced with a satellite link.

- Otherwise, the incremental trench and copper link is removed and the LPGS is assumed to be served by a wireless link. The number of relay stations required is also stored.
- Having completed this determination, the nature of each location is then determined using *DetermineLocationType*, as described below.

► *RemoveRuralDoubleBackMainCable*

Location: Found in the *PillarClusterToPillarCluster*module

Purpose: For each pillar cluster in turn, this subroutine considers the route that the cable takes from the pillar back to the RAU. Specifically,:

- counts the number of times that this cable passes through each link in the trench network
- reduces this count where it is more than 1 to remove instances where the cable doubles back on itself
- adjusts the recorded length of the cable for the pillar in gdSheathLengthToConnectPillarClusters() accordingly

► *DeriveWirelessLinkQuantities*

Location: Found in the *ModifiedPrimSpanningTree* module

Purpose: Given the length of a wireless link, this derives how many relay stations are required on the link, by dividing the link distance by the assumed maximum distance of a wireless link without a relay station (~30km, as defined in the 'Inputs' worksheet of the Access CODE workbook) and then rounding up.

► *DetermineLocationType*

Location: Found in the *OutputResults* module

Purpose: For the rural deployment, this subroutine first identifies all copper points as DPs and all wireless locations as FDPs. It then overwrites the main node locations, namely the RAU, the pillars and LPGS.

The duct required to link up the pillars and LPGS to the RAU is calculated using the subroutine *CalculateDuctBetweenRuralPillars*.

CalculateDuctBetweenRuralPillars

Location:	Found in the *CalculateDuct* module
Purpose:	For the rural deployment, for each pillar/LPGS, this subroutine identifies the full route taken back to the RAU and inserts a duct in the relevant trenches. Ducts for pillar–RAU and LPGS–RAU links are recorded separately.

2.4.7 Fibre determination phase

Having completed the trench network for the copper network, the algorithm then seeks to overlay a fibre network to serve the locations of high demand. For a rural deployment, this can only be done using point-to-point links. This is handled by the subroutine *LinkFibrePointsDirectlyToPillar*, which is found in the *BuildFibreRing* module.

LinkFibrePointsDirectlyToPillar

Location: Found in the *BuildFibreRing* module

Purpose: This calculates the fibre sheath and length requirements for joining each fibre-fed location back to its parent RAU via its parent DP and pillar. This uses the path determined by the nominal unit of demand assigned to each fibre-served location in the copper network.

This function does not need incremental trench, since the existing trench network is assumed to be used as the path back to the RAU.

For the rural deployment:

- Fibre-fed locations are flagged in the array bolPointFedByFibre().
- The fibre sheath needed to join each fibre-fed location back to its parent RAU is calculated, by using the same path followed by the copper network.
- The fibre length needed to join each fibre-fed location back to its parent RAU is calculated for each pillar cluster, by multiplying the total FDP–DP, DP–pillar fibre sheath lengths by their respective assumed fibre cable sizes.
- The pillar–RAU fibre sheath length is assumed to be the same as that for the analogous connection in the copper network.
- The pillar–RAU fibre length is assumed to be the fibre sheath

length multiplied by the assumed fibre cable size in the DP–pillar part of the network.

- A discrete cable is assumed for each fibre-fed location all the way back to the RAU (point-to-point architecture).
- The amount of fibre-specific duct required is calculated using *CalculateFibreDuctByType* for links back to the pillar and *CalculateDuctsForFibreBackFromPillar* for links from the pillar back to the RAU.

► *CalculateFibreDuctByType*

Location:	Found in the *CalculateDuct* module
Purpose:	This calculates the duct required for the entire route for each fibre cable from the fibre-fed FDPs back to the pillar.

► *CalculateDuctsForFibreBackFromPillar*

Location:	Found in the *CalculateDuct* module
Purpose:	Fibre rings can never be used in the rural deployment, so this calculates the duct required for a point-to-point fibre link from each fibre-fed FDP, starting from the parent pillar and going back to the RAU.

2.4.8 Copper result storage phase

The remaining outputs of the copper network asset volumes are printed to the output worksheet for the ESA by the subroutine *OutputTheCopperResults*.

OutputTheCopperResults

Location: Found in the *OutputResults* module

Purpose: This prints the remaining network volumes to the output worksheet for the ESA in the relevant Access DATA workbook.

Specifically, the subroutine:

- calculates the average loop length for each pillar cluster
- prints the network volumes for the cluster containing the RAU
- prints the aggregated volumes for the ESA, including the trench between pillars and the RAU and the number of relay stations

- prints the network volumes for every other pillar cluster
- prints the pillar cluster indices for each location, via *WriteCopperClusterResults*, as explained below
- prints the edges in the spanning trees at the pillar–RAU level of the network, using the subroutine *WriteConnectRuralClustersResults*, as explained below
- prints the duct requirements for each link in the trench network, using *WriteDuctOutputs*, as explained below
- estimates the cable lengths by cable size, using the subroutine *CalculateTotalSheathLengthByCableSize*, as explained below.

► *WriteCopperClusterResults*

Location:	Found in the *Clustering* module
Purpose:	Prints the pillar cluster alongside each copper-fed location.

► *WriteConnectRuralClustersResults*

Location:	Found in the *PillarClusterToPillarCluster* module
Purpose:	Prints the incremental trench links required to join the pillars and LPGS with fibre backhaul to the RAU. Also calculates the number of extra manholes needed along these links.

► *WriteDuctOutputs*

Location:	Found in the *CalculateDuct* module
Purpose:	For each link in the trench network, the number of ducts *required* by type are calculated, determined by how many cables of each type there are passing through the link and the capacity of each type of duct. The number of ducts that are *provisioned* (based on the allowed multiples) is also determined for each link. The length of each link, using either crow-flies or p-function, is also derived. Finally, the type of pit required at each DP location is also determined.

► *CalculateTotalSheathLengthByCableSize*

Location:	Found in the *OutputResults* module
Purpose:	For the rural deployment, the sheath length by cable size is estimated slightly differently depending on whether the network is tapered or

non-tapered.

In the tapered case, the sheath requirements for each point and its predecessor back to the pillar are calculated individually using glCablingForRuralDemandIncUtilisation and then aggregated by cable size.

In the non-tapered case, the sheath requirements for each point and its predecessor back to the pillar are calculated individually using *GetNonTaperedCableSize* and *GetNonTaperedSheath* and then aggregated by cable size.

The sheath by cable size is then printed onto the output worksheet for the ESA.

2.4.9 Wireless clustering phase

Having completed the copper and fibre deployments, a wireless phase is undertaken by the subroutine *CalculateWirelessandSatelliteServedDemand* if either

- There are points that have been designated as being served by wireless.
- There are LPGS that are served by wireless backhaul, which must be linked back to the RAU as part of a wireless backhaul network.

For each wireless-fed location, the co-ordinates of its corresponding location from the G–NAF are restored and these co-ordinates are used to cluster these locations using the wireless assumptions and the subroutine *DivisiveClustering* in the *Clustering* module.

2.4.10 Satellite determination phase

When the wireless locations have been clustered, *CalculateWirelessandSatelliteServedDemand* calculates the cost of serving each wireless cluster by wireless or by satellite.

The wireless cost of a cluster is given by:

Cost of the wireless BTS + (Number of locations in cluster × Cost of a wireless CPE)

The satellite cost of a cluster is given by:

(Number of locations in cluster × Cost of connecting a location with satellite)

If a cluster is found to have a higher wireless cost, then it is assumed to be served by satellite, so the BTS is not needed for the consideration of the wireless backhaul network.

2.4.11 Wireless backhaul determination phase

As stated above, a wireless backhaul network is required if there are BTS and/or LPGS with wireless backhaul that need to be connected back to the RAU. For each cluster served by a wireless BTS, the demand-weighted centre is calculated. The data for the wireless clusters is then stored and a set of points is created consisting of (in order):

- the RAU
- all LPGS with wireless backhaul
- wireless BTS.

The subroutine *ConstructTreeForWirelessBTS* is then used to create a minimum spanning tree of these points. This can be found in the *ModifiedPrimSpanningTree* module and begins with the subroutines *GetMaxPointsInCluster* and *SetupArraysForSpanningTree*.

GetMaxPointsInCluster

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Identifies the number of nodes in the wireless backhaul network.

SetupArraysForSpanningTree

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	Dimensions the key arrays for the minimum spanning tree process:

- glUnattachedPoint() – the list of locations in the cluster that are unattached at any point in the algorithm
- glAttachedPoints() – the list of locations in the cluster that are attached at any point in the algorithm
- glVertexRoute() – for any location, the location that it passes through in order to get back to the node location
- gdDistanceMatrix() – stores the distance between any two points in the cluster: here, it is calculated to be the crow-flies distance
- gobjEdges() – stores the vertices and lengths associated with each edge in the spanning tree.

Following these two subroutines, each cluster is then treated in turn. The following subroutines are used in order to create and store the minimum spanning trees:

- *SetupPointsInCluster* – identifies the central point in the cluster
- *ConstructTree* – constructs the minimum spanning tree for the cluster
- *StoreRoutes* – for each point P, identifies the point P passes through to get back to the node

- *WriteWirelessNetworkResults* – stores the list of edges in the spanning tree on the output worksheet for the ESA.

SetupPointsInCluster

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• In this case, we are creating a single backhaul network, so all the points we are considering are in the cluster. • The main node has been set up to be the RAU, which is identified and stated to be the central point cp.

ConstructTree

In order to derive the spanning tree for the cluster, the algorithm begins with the central point identified in *SetupPointsInCluster*. All other locations in the cluster are assumed to be unattached. Locations are then added to the tree incrementally and the lists of attached and unattached locations are updated.

For each incremental addition, the algorithm cycles through each pair of attached and unattached points and calculates the average cost per line of the whole tree were the two points linked together by a trench and a cable. This average cost is determined by *AverageCostPerWirelessLink*.

► *AverageCostPerWirelessLink*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For the wireless backhaul network, it determines: • The extra backhaul capacity (*c*) required to serve the unattached point, using the subroutine *GetBackHaulMultiplierNeeded*, in the *WirelessAndSatellite* module. • The crow-flies distance (*d*) between the attached and unattached point. • The number of relay stations (*n*) required, derived using *DeriveWirelessLinkQuantities*. The cost of the new link is then calculated using the expression: $k_1*d + k_2*c + k_3*n$

and calculates the new cost per unit of demand for the entire tree.

For each unattached point, the edge to an attached point that gives the lowest new average cost per line is stored in the array objEdgeList() using *AddToEdgeList*, which can be found in the *ModifiedPrimSpanningTree* module.

This is repeated for every unattached point. The edge in objEdgeList() that gives the lowest new average cost per line is then linked to the tree using *AddCheapestWirelessLinkInLinkListToObjEdges*.

► *AddCheapestWirelessLinkInLinkListToObjEdges*

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• Adds the best link in objEdgeList(), in terms of average cost per unit of demand, to the list of edges for the minimum spanning tree. • Updates the array glVertexRoute() for this new edge in the tree, defined by: glVertexRoute(unattached location on new edge) = attached locations on new edge • Updates the total capacity and costs required to link the location all the way back to the RAU.

The number of unattached points is reduced by 1 and the lists of attached and unattached points updated using *IdentifyAttachedAndUnattachedPoints*, which can be found in the *ModifiedPrimSpanningTree* module.

The loop in *ConstructTree* continues until all locations in the cluster have been attached to the cluster node.

StoreRoutes

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	For each point P in the cluster, stores the point Q that it passes back through in order to reach the cluster node, with the array defined as glRouteToCentre(P)=Q

WriteNetworkResults

Location:	Found in the *ModifiedPrimSpanningTree* module
Purpose:	• Writes the location co-ordinates in the output worksheet for the ESA (in rows BF–BM) that define every edge in the spanning trees for the wireless backhaul network. • Classifies each link as either wireless LPGS–BTS, BTS–BTS, BTS–RAU, BTS–wireless LPGS, wireless LPGS–wireless LPGS.

For each edge in the wireless backhaul network, the number of relay stations is calculated using *DeriveWirelessLinkQuantities* and aggregated. Finally, all the wireless locations are then restored.

2.4.12 Wireless and satellite result storage phase

Having derived the wireless clusters and backhaul network, the network asset volumes for these clusters are printed to the output worksheet for the ESA by the subroutine *OutputTheWirelessAndSatelliteResults*.

OutputTheWirelessAndSatelliteResults

Location:	Found in the *OutputResults* module
Purpose:	This prints the remaining network volumes to the output worksheet for the ESA in the relevant Access DATA workbook. Specifically, the subroutine: • prints the number of wireless relay stations required • prints out network volumes for the RAU, which can depend on whether the ESA has copper deployed or not • prints out network volumes for wireless clusters using *OutputAWirelessRow* • prints out network volumes for satellite clusters using *OutputASatelliteRow* • prints the cluster indices for each location in a wireless or satellite

cluster using *WriteWirelessClustersResults*.

► *WriteWirelessClustersResults*

Location:	Found in the *OutputResults* module
Purpose:	For each location in a wireless cluster, a cluster index is printed. These are enumerated so that an ESA containing both copper, wireless and satellite clusters will have a unique index for each cluster.

Back in the *CalculateWirelessAndSatelliteServedDemand* subroutine, each location is then identified as being served by copper , wireless or satellite. The original set of all points are then restored to the main array.

2.4.13 Assumption storage phase

The assumptions used within the calculation are printed onto the output worksheet for the ESA, using the subroutines *RecordAssumptions*.

RecordAssumptions

Location: Found in the *CommonCode* module

Purpose: This prints all of the assumptions used in the calculation of access network asset volumes.

For a rural deployment, it prints:

- capacity and distance constraints for the nodes used in the network
- technical constraints for the copper jointing
- cables used in the non-tapered distribution network (if applicable)
- coefficients for the p–function used
- coefficients for the proxy cost functions
- cost assumptions used (in the rural case for the fibre and wireless backhaul cost comparisons for LPGS backhaul, the copper and wireless cost-based decision and the satellite decision).

Finally, in order to reduce congestion in the computer’s memory, the subroutine *EraseArrays* (found in the *MainMacros* modules) uses the Erase statement to destroy global arrays populated separately for each ESA, releasing the allocated memory.

3 Script for the core network algorithms

3.1 Introduction

Visual Basic code has been developed, using the compiler in Excel, for the purpose of determining the efficient backhaul routes, using a spur and ring topology, from the local exchanges (LEs) to the local access switch (LAS) (or their NGN equivalents).

This chapter outlines the structure of the underlying code, which can be located in the Excel workbook *LE_LAS_ring.xls*.

The code is divided into two main programs:

- **Find PoCs –** for each LAS area, this part of the code identifies which LEs should be designated as points of confluence (PoCs); and for all the other LEs, the parent PoC to which they should join.
- **RunTSP –** this code forms the PoC rings within each LAS area.

These programs are described in detail in sections 3.2 and 3.3 respectively.

3.2 Find PoCs

Backhaul from the LE nodes (AT1 nodes in NGN) are aggregated at PoCs prior to being backhauled to the parent LAS (regional node in NGN).

The following conditions govern whether an LE/AT1 may be designated as a PoC:

- If the demand at the LE/AT1 node is greater than a defined threshold limit, then that LE/AT1 node is designated as a PoC.
- For the other nodes, if the demand of a clustered group of LE/AT1 nodes is greater than a defined threshold limit, then the LE/AT1 node at the line-weighted centre of the cluster is designated as the PoC.

Having determined the appropriate PoC locations, the algorithm calculates the trench and fibre distance to join the LE/AT1 nodes to the appropriate PoC – the clustering process identifies the parent PoC to each LE/AT1 node. For each LE, this code requires the following data inputs (sorted by ‘Parent LAS’ then ‘Distance to parent LAS’):

Input required	*Description*
LE ID	An identifier for the local exchange
Parent LAS	An identifier for the LAS area
Distance to parent LAS	The straight-line distance from the LE to its corresponding LAS
SIOs at LE	The number of services in operation at the LE
Latitude	Latitude coordinate of the LE/AT1 node
Longitude	Longitude coordinate of the LE/AT1 node

Table 3.1: Data inputs required by the 'Find PoCs' algorithm [Source: Analysys]

In addition, the algorithm requires the following parameters to be defined:

Parameter	*Description*
LEs.Per.POC	The maximum number of LEs that may be attached to a PoC
SIOs.For.POC	The SIO threshold of a PoC – above which an LE is automatically assigned to be a PoC by itself
Trench.Cost	The cost per metre of digging a trench
Fibre.Cost	The cost per metre of deploying the fibre

Table 3.2: Parameters required by the 'Find PoCs' algorithm [Source: Analysys]

Using these inputs, the 'Find POCs' algorithm runs through the following subroutines:

- *ClusterToPoCs*
- *Setup_Output_Sheet*
- *ReadInputParameters*
- *ReadInALAS*
- *Write_GPOC_dis_to_PoC*
- *Calc_Distance_Matrix*
- *RunThisLAS*
- *Identify_PoCs*
- *Cal_weighted_centre*
- *Prepare_and_run_spanning_tree*
- *WriteOutresults*.

These subroutines are described in detail in the following subsections.

3.2.1 ClusterToPOCs

Location: Found in the *ClusterToPOCs* module

Purpose: This is the entry point and skeletal part of the 'Find POCs' code. It calls the following routines:

- *Setup_Output_Sheet* – prepares the output sheet
- *ReadInputParameters* – reads in the input parameters (*LEs.Per.POC* and *SIOs.For.POC*)

Then it loops through each LAS calling the following routines:

- *ReadINALAS* – reads in the input data for the LAS
- *write_GPOC__dist_to_PoC* – for LEs that are PoCs by themselves (henceforth referred to as GPoCs), we can write out the distance to the PoC as 0
- *Calc_Distance_Matrix* – calculates the distances between every pair of LEs within the LAS area
- *RunThisLAS* – the main clustering algorithm within the code. Clusters the LEs into PoC areas
- *Identify_POCs* – for each PoC area, identifies the LE nearest the centre of the cluster to use as the PoC
- *prepare_and_run_spanning_tree* – produces the minimum spanning tree for each PoC cluster. That is, identify what trenches need to be dug in order to attach each LE to its PoC
- *WriteOutResults* – writes out the results of the clustering and spanning tree

3.2.2 Setup_Output_Sheet

Location:	Found in the *ClusterToPOCs* module
Purpose:	This subroutine clears all the cells in the output range

3.2.3 ReadInputParameters

Location:	Found in the *ClusterToPOCs* module
Purpose:	This subroutine reads in the input parameters *LEs.Per.POC* and *SIOs.For.POC*

3.2.4 ReadInALAS

Location:	Found in the *ClusterToPOCs* module
Purpose:	This subroutine runs through the data ranges and continues while the Parent LAS string is the same as the previous line. The first line it comes across is the LAS (assuming the data is sorted by distance to LAS). Then, if it is an LAS or if the *SIOs.For.POC* condition is met, it makes the LE into a GPoC (a PoC by itself with no other LEs in the PoC area). If this is the case, then we assign all the input data to an *objGPOCData* object. If it is not the case, then we assign the input data to an *objInputData* object.

3.2.5 write_GPOC__dist_to_PoC

Location:	Found in the *Spanning Tree* module
Purpose:	For each of the GPoCs identified in the *ReadInALAS* subroutine, this subroutine writes out the distance from the LE to the PoC to be 0 (since the LE is the PoC in these cases).

3.2.6 Calc_Distance_Matrix

Location:	Found in the *ClusterToPOCs* module
Purpose:	Calculates the straight-line distances between every pair of LEs within the LAS area. The calculation takes account of the Earth's curvature.

3.2.7 RunThisLAS

Location:	Found in the *ClusterToPOCs* module
Purpose:	This subroutine sets up the global variable *glNumPoints,* which represents the number of points that need clustering. It then calls *Divisive_clustering*, which performs the clustering of LEs into PoCs. For a description of the algorithm, see *DivisiveClustering* and all of its associated routines in section 2.3.1.

3.2.8 Identify_POCs

Location: Found in the *Clustering* module

Purpose: For each cluster that has been identified (for reference, there are *gNumChildClusters* identified), this subroutine calls the function *cal_weighted_centre*, which calculates the centre of the cluster. The subroutine then loops through each LE in the cluster and identifies the LE that is nearest the centre. That identified LE is designated as the PoC. The algorithm then assigns the global variable *glPOCPoints(lCluster)* to point to that LE.

3.2.9 cal_weighted_centre

Location: Found in the *Clustering* module

Purpose: This subroutine calculates the weighted centre of a cluster of points. In the algorithm, all of the points are weighted equally (the variable *dCap*).

3.2.10 prepare_and_run_spanning_tree

Location: Found in the *SpanningTree* module

Purpose: For each PoC cluster of LEs, this subroutine sets up the data into the format required to run the minimum spanning tree algorithm. It then calls the algorithm *construct_tree*, which constructs the minimum spanning tree for that PoC cluster. That is, it identifies the trenches that need to be dug to join each LE to its PoC. For a description of the algorithm see the *ConstructTree* algorithm in section 2.3.2. In summary, the spanning tree algorithm uses the trench costs and fibre costs to identify the cheapest way of joining the LEs to the PoC.

3.2.11 WriteOutResults

Location: Found in the *ClusterToPOCs* module

Purpose: This has two data types as arguments: (i) *objGPOCData*, which is the data for the LEs that are PoCs by themselves (i.e. are either an LAS or have so many SIOs to justify being a PoC – these have no other LEs in the PoC area) and (ii) *objInputData*, which is the data for the other LEs within the LAS.

3.2.12 Outputs of the Find PoCs algorithm

For each LE, the *WriteOutResults* subroutine writes out the following named ranges results of the algorithm to the 'Input Table' worksheet:

Named range	*Column title (on the 'Input Table' worksheet)*	*Description*
Is.LAS	Is a LAS?	Whether the LE is also the LAS (a "Y" means yes, otherwise the column is left blank)
PoC.ID	PoC ID	a number indicating an identifier for the PoC that the LE is associated with
PoC.Name	PoC Name	the name of the LE that is also the corresponding PoC
Dist	Distance to PoC	the crow flies distance from the LE to the PoC
Next.LE	Next LE in the spanning tree	the next LE which this LE will pass through to reach its PoC
LE.Dist	Trench Distance to Next LE	the crow flies distance to the next LE or PoC if connected directly
Fibre.Dist	Fibre Distance to POC	the distance to the PoC considering intermediate LEs

Table 3.3: Outputs of the 'Find PoC' algorithm written to the 'Input Table' worksheet [Source: Analysys]

For each PoC, the *WriteOutResults* subroutine writes out the following named ranges results to the 'Input POCs':

Named range	*Column title (on the 'Input PoCs' worksheet)*	*Description*
Cluster.Centre.Lat	ClusterCentre Latitude	The latitude of the centre of the LE cluster in the PoC area
Cluster.Centre.Long	ClusterCentre Longitude	The longitude of the centre of the LE cluster in the PoC area
Number.LEs.In.POC	Number of LE's in the POC	The number of LEs that are in the PoC cluster, determined by the subroutine *num_points_in_cluster* in the Clustering module

Table 3.4: Outputs of the 'Find PoC' algorithm written to the 'Input POCs' worksheet [Source: Analysys]

For each PoC, the subroutine also writes out the following named ranges results (found on the 'Input POCs' worksheet), which form the basis of inputs into the second part of the code, the *RunTSP* algorithm:

Named range	*Column title (on the 'Input PoCs' worksheet)*	*Description*
Input.TSP.POC.ID	POC Id	An identifier for the PoC to which the LE is associated
Input.TSP.POC.Name	LE Id	The name of the PoC (which is an LE name)
Input.TSP.Lat	POC Latitude	The latitude coordinate for the PoC
Input.TSP.Long	POC Longitude	The longitude coordinate for the PoC
Input.TSP.LAS	LAS	The parent LAS ID to the PoC
Number.Of.POCs	Number of POCs in LAS	The number of PoCs in the LAS area
Input.TSP.Is.a.LAS	Is a LAS?	Flag as to whether the PoC is also the LAS (a "Y" means yes, otherwise the column is left blank)
Input.TSP.Num.SIOs	SIOs	The aggregate number of SIOs in the PoC area (summed over all LEs in the PoC area)

Table 3.5: Outputs of the Find PoCs algorithm that feed into the RunTSP algorithm [Source: Analysys]

3.3 RunTSP

The 'travelling salesman problem' (TSP) is a well-recognised problem in optimisation.

In its traditional form, the TSP considers the situation of a salesman who needs to find the least-cost round-trip route between a number of cities – the route must visit each city exactly once, and must end at the same city at which the route started.

It is employed in the Analysys cost model to determine the most efficient ring backhaul topology between a group of PoCs.

However, the TSP algorithm has been extended beyond its traditional solution in order to enable backhaul solutions to consider more than one ring. Either several separate rings can form, each including the LAS as a node, or 'parent' rings (connecting to the LAS) can link 'child' rings back to the LAS. Consequently, the additional problem of identifying which PoCs should belong to which ring is incurred.

The *RunTSP* algorithm uses the following sequence:

- For each LAS area, the algorithm identifies to which ring cluster each PoC belongs.

- For those LAS areas where the number of PoCs is sufficiently small (up to *Num.POCs.GA.Threshold* – see below), the algorithm performs an exhaustive search considering every possible way in which the PoCs could be partitioned into ring clusters.
- Where the number of PoCs is greater than this threshold number, the algorithm uses a genetic algorithm approach instead – an exhaustive search would be prohibitive in terms of processing time.
  - This genetic algorithm sets up an initial random "population", each member of which represents a different partitioning of the PoCs into ring clusters. That population is then evolved – at each generation in the evolution and for each new member in that generation, two members of the old population are selected with a preferential bias towards those with the cheapest ring costs. These are thencrossed (so the new member has some of the properties of one parent and the rest from the other) and mutated to obtain some new properties. More details can be found in the relevant sections below.
- Whichever method is used to select the partitioning, the same 'travelling salesman' algorithm is then used within each ring cluster to identify the order in which those PoCs should be joined.
  - An exhaustive Branch and Bound implementation that builds up the test ring along a search path, adding one PoC at a time, is used. If that ring exceeds the current best upper cost bound, then that search path is rejected. The algorithm proceeds along a new search path until either all search paths have been exhaustively searched, or until a full-sized ring that is cheaper than the current best cost bound is found. In the latter case, the best cost bound is set to be the newly found cost; and the algorithm continues to proceed along the next search path. Using this methodology, the algorithm exhaustively searches all of the possible paths until it has found the optimal solution. More details can be found in the relevant sections below.

The *RunTSP* code takes the outputs, as defined in Table 3.6, from the *Find POCs* algorithm as input. In addition, it takes the following input parameters:

Parameter	*Description*
Max.POCs.Per.Ring	The maximum number of PoCs allowed on any ring. Thus, if the number of PoCs in any LAS area is greater than this threshold, we would need more than one ring
Bridging.Nodes	The number of nodes which bridge a child ring to a parent ring. Valid inputs are 1 or 2.
Num.POCs.GA.Threshold	The maximum number of PoCs in an LAS area in which the algorithm would calculate the optimal ring using an exhaustive search mechanism. If the number of PoCs is above this threshold then a genetic algorithm is employed to find an efficient ring[1]. Note that if the number of PoCs is less than or equal to *Max.POCs.Per.Ring* then there is only one ring required – all the PoCs can fit on it. Thus only the travelling salesman part of the algorithm to identify the order in which these PoCs are joined needs to be implemented
GA.Num.Generations	The number of generations to run when using a genetic algorithm to identify efficient rings. The more generations, the more likely one is to find a more efficient ring. However, the more generations, the longer the algorithm will take to process

Table 3.6: Parameters required by the RunTSP algorithm [Source: Analysys]

The *RunTSP* algorithm runs through the following routines starting at *RunTSP*:

3.3.1 RunTSP

Location: Found in the *RunTSP* module

Purpose: This is the entry point and skeletal part of the *RunTSP* algorithm. It calls the following routines:

- *ClearOutputSheet* – clears the output sheet ready to receive the new results
- *SetupInputParams*–reads in the input parameters
- *InitOutput* – sets up the first row to start the output from

Then it loops through for each LAS area calling the following routines:

- *ClearTSPInputData* – gets called for the *objTSPInputData* object, clearing out the values. The object will contain the input data

[1] Whilst this method cannot guarantee an optimal solution, as it is not an exhaustive approach, it does employ optimisation algorithms to check that a near-optimal (which may in fact be optimal) solution is generated.

- *ReadLASInputs* – reads in the input data for the LAS, setting up the *objTSPInputData* object, which contains the input data
- *RunThroughCombinations* – runs through either (i) an exhaustive search of all possible combinations of rings that could be used in order to find the optimal ring; or (ii) sets of the genetic algorithm to find the best ring that can find. It also writes out the results.

3.3.2 ClearOutputSheet

Location:	Found in the *RunTSP* module
Purpose:	This subroutine clears the contents of all the named output ranges.

3.3.3 SetupInputParams

Location:	Found in the *RunTSP* module
Purpose:	This subroutine reads in the four input parameters *Max.POCs.Per.Ring, Bridging.Nodes, Num.POCs.GA.Threshold, GA.Num.Generations.*

3.3.4 InitOutput

Location:	Found in the *OutputTSPResults* module
Purpose:	This subroutine sets the first row of the output ranges to write to.

3.3.5 ClearTSPInputData

Location:	Found in the *RunTSP* module
Purpose:	This subroutine initialises the passed-in object, generally setting the object’s values to zero. The passed-in object will be a clsTSPInputData type, which will eventually contain the input data required to run the TSP algorithm.

3.3.6 ReadLASInputs

Location:	Found in the *RunTSP* module

Purpose:	This subroutine reads in all the data for an LAS area. It loops through each and adds the PoC data to the *objTSPInputData* object. Once it finishes looping, the object is then ready to be passed into the processing stage of the algorithm.

3.3.7 RunThroughCombinations

Location: Found in the *RunTSP* module

Purpose: This is called with the *objTSPInputData* object passed in as an argument. First it finds the maximum number of ring clusters that we will consider using by calling *getMaxRings*

It tests the number of PoCs in the LAS area against *glMaxPOCsForExhaustiveSearch*, which is the global variable representing *Num.POCs.GA.Threshold*.

If it is less then it:

- First assigns all PoCs to Ring 1, then calls *UseThisCombination* and then calls *RecurseCombinations* to find the optimal solution

Otherwise it:

- Calls *RunGeneticAlgorithm* to find a near-optimal solution

In either cases, when it returns from these calls it calls *RunForBestRings*, which runs through the travelling salesman part of the algorithm again but this time writing out the results.

3.3.8 GetMaxRings

Location: Found in the *RunTSP* module

Purpose: This subroutine determines the number of rings required for a given number of PoCs and given value of *glMaxPOCsPerRing* (which takes the value of the named range *Max.POCs.Per.Ring*)

3.3.9 UseThisCombination

Location: Found in the *RunTSP* module

Purpose: This subroutine takes the array *lRings ()* as an argument, which identifies which ring each PoC is assigned to in the current test ring.

It calls *TestRingValidityBeforeSetup* to determine the number of rings required and to test whether the rings are valid (i.e. there aren't more than the maximum allowed number of PoCs on any ring or less than the minimum number). If this returns False then a solution for this combination is not followed. If this returns True, then the algorithm creates an *objTSPData* object to hold the TSP data and calls *ClearTSPData* to initialise its values. It then uses *lRings ()* along with the input data in *objTSPInputData* to set up *objTSPData* by calling *bolSetUpTSPData*.

Having created the *objTSPdata* object, the subroutine then calls *TestRingValidityAfterSetup* to test whether the rings are valid and, if this returns True, then it then calls *RunTheAlgorithm* with the first argument as False so that no results are written out. *RunTheAlgorithm* identifies the best order to join up the PoCs given the ring association for each PoC. It stores these results in the object array *objBestRings()* and that includes the distance cost associated with the setup. If the total distance cost is the best found so far for this LAS area then the results are copied into the global object array *glBestRings()*. Finally, the algorithm calls *ClearBestRings* to clear out the memory used in creating *objBestRings()*

3.3.10 TestRingValidityBeforeSetup

Location: Found in the *RunTSP* module

Purpose: Firstly, this subroutine determines the number of rings (i.e. clusters) needed for the array argument *lRings()* passed through, which says which ring each PoC is assigned to in the current test ring. It then determines which of these clusters contains the LAS node.

The subroutine then performs the tests. The ring validity test will fail if one of the following is true:

- The number of PoCs in any cluster is larger than *glMaxPOCsPerRing* (which represents *Max.POCs.Per.Ring*)
- If any cluster that does not contain the LAS and has the number of PoCs in the cluster equal to *glMaxPOCsPerRing*. This is because the cluster will have to be attached to the ring that contains the LAS and thus the number of PoCs would then exceed *glMaxPOCsPerRing*
- If the total number of PoCs across all clusters is larger than

MinPOCsPerRing, but the number of PoCs in any one cluster is less than *MinPOCsPerRing* (or less than that number minus 1 if the cluster does not contain the LAS). Note *MinPOCsPerRing* is defined as 3. This condition means that we will never have a cluster smaller than this size if we could join them into bigger clusters

3.3.11 TestRingValidityAfterSetup

Location: Found in the *RunTSP* module

Purpose: This subroutine tests the validity of each ring (i.e. cluster) in the solution. A candidate ring fails if any of the following are true:

- The number of PoCs in any cluster is larger than *glMaxPOCsPerRing* (which represents *Max.POCs.Per.Ring*)
- If the total number of PoCs across all clusters is larger than *MinPOCsPerRing*, but the number of PoCs in any one cluster is less than *MinPOCsPerRing* (or less than that number minus 1 if the cluster does not contain the LAS).

3.3.12 ClearTSPData

Location: Found in the *RunTSP* module

Purpose: This clears out and initialises the *objTSPData* object ready to be populated

3.3.13 SetUpTSPData

Location: Found in the *RunTSP* module

Purpose: This takes *lRings ()* as an argument, which says which ring each PoC is assigned to in the current test ring. It calculates the number of clusters (i.e. rings) required by the *lRings()* array. Then copies the longitude and latitude from *objTSPInputData*. It determines which cluster has the LAS and also the number of PoCs in each cluster (i.e. ring). Finally, it calls *bolJoinClusterRings*, which joins all the other rings up to the ring that contains the LAS

3.3.14 bolJoinClusterRings

Location: Found in the *RunTSP* module

Purpose: This loops through each cluster (i.e. ring), *lCluster*, and determines whether it contains the LAS. If it does, then there is nothing more to do with this cluster here. Otherwise, it looks through all the PoCs in all the clusters that do contain the LAS and determines the PoC that is nearest to any of the PoCs in *lCluster*. It then adds this nearest PoC to the list of PoCs in *lCluster* and sets this PoC to be the bridging PoC (the one that joins the two rings). If it turns out that this bridging PoC is the LAS, then *lCluster* now contains the LAS. Thus it may have been better for a previous cluster to have joined to this *lCluster* rather than what it has joined to. Hence, we clear out the bridging PoCs of the previous clusters and start *bolJoinClusterRings* again knowing that *lCluster* is now a cluster containing the LAS and has been joined

3.3.15 RunTheAlgorithm

Location: Found in the *RunTSP* module

Purpose: Creates a new worksheet and chart (by calling *CreateSheet* and *CreateChart* respectively) ready to output the results for that particular LAS. It then loops through each cluster (i.e. ring) performing the following tasks:

- Calls *SetUpInputs* to set up the distances between pairs of PoCs and the order in which to consider the PoCs
- Calls *FindInitialBound*, which uses a simple dynamic programming technique to get an initial upper bound for the best ring solution – meaning the best order in which to join the ring's nodes (PoCs)
- Calls *InitBranchPoints* which sets up the identifiers for test rings that form branch points in the Branch and Bound algorithm that is employed to solve the travelling salesman problem. See the section *InitBranchPoints* below for more details
- Sets up memory for the 3 initial test rings of size 4. Note that a size-3 ring (one with 3 nodes) is unique in how it can be ordered since we assume that a link from A to B is the same as a link

from B to A. A ring of size 4 has 3 possible ways to join the nodes up. In general, a ring of size *n* has (*n*-1)!/2 possible combinations

- Calls *FindShortestRing*, which runs through the Branch and Bound method for solving the travelling salesman problem
- Copies the results of the best ring order into the *objBestRings(lCluster)* object, which represents the best ring order solution for that cluster (collection of PoCs)
- If the *bolOutputRings* flag is set to True, calls *OutputARing* which outputs the results for the ring cluster. Note that this argument is passed into the function and is only passed in as True once we know the best way to partition the PoCs into ring clusters. That is, only once we know which ring cluster each PoC should belong to. For LAS areas with 8 or fewer PoCs then all PoCs belong to the same cluster, so we can be sure we have the best partition

Finally, if *bolOutputRings* is set to True then we call *OutputFinalResults* which provides the summary over all the ring clusters in the LAS area

3.3.16 CreateSheet

Location: Found in the *OutputTSPResults* module

Purpose: Creates a new worksheet with the given name (the LAS area). Or, if that worksheet already exists, then *CreateSheet* clears the sheet

3.3.17 CreateChart

Location: Found in the *OutputTSPResults* module

Purpose: Creates a new chart in the new worksheet created above in *CreateSheet*. Or, if that chart already exists, then *CreateChart* clears the chart

3.3.18 SetUpInputs

Location: Found in the *InitialiseTSP* module

Purpose: This calls *GetInitialDistances* to calculate the distance costs between every pair of PoCs in the ring cluster. It then calls *SetUpInitialRingAndReorderSites* which re-orders the PoCs in memory in an attempt to improve the efficiency of the Branch and Bound algorithm. It also sets up the initial ring of 3 PoCs formed from the first 3 PoCs in the re-ordered list

3.3.19 GetInitialDistances

Location: Found in the *InitialiseTSP* module

Purpose: Calculates the distance costs between every pair of PoCs in the ring cluster

3.3.20 SetUpInitialRingAndReorderSites

Location: Found in the *InitialiseTSP* module

Purpose: This loops through to find the 3 PoCs that form the longest triangle. It sets these 3 PoCs as the initial ring in the algorithm as this makes for an efficient algorithm. It then calls *ReorderSites* to re-order the rest of the PoCs into an efficient order; followed by *SetUpReorderedDistances* to update the distance cost matrix to reflect the re-ordering of the PoCs

3.3.21 ReorderSites

Location: Found in the *InitialiseTSP* module

Purpose: Re-orders the PoCs so that the first 3 form the largest triangle. The PoCs are then ordered in descending order in terms of distance to a PoC on that triangle

3.3.22 SetUpReorderedDistances

Location: Found in the *InitialiseTSP* module

Purpose: Having re-ordered the PoCs, we need the distance matrix to reflect this re-ordering so that its elements refer to the correct PoCs

3.3.23 FindInitialBound

Location:	Found in the *ShortestRing* module

Purpose:

This uses a quick dynamic programming technique to find a good first solution to the travelling salesman problem. This solution is used to form an initial upper bound to the overall cost of the ring – we know any better solution must have a lower cost. The dynamic programming technique employed here takes the initial ring of 3 PoCs as the size-3 "current ring". It then takes the next PoC from the re-ordered list and considers where it should be added to the current ring. There are 3 possibilities – (i) between the 1st and 2nd PoC, (ii) between the 2nd and 3rd PoC, or (iii) between the 3rd and 1st PoC. It places the new (4th PoC) in the position that minimises the total ring cost of the size-4 ring to form the new size-4 current ring. It then takes each PoC in turn and considers every possible position with the current ring. Finally, once all PoCs have been added, we have the dynamic programming solution. Note that, in general, this will not be optimal since an early decision on where to place PoC 5, for instance, may mean higher additional costs when we add a later PoC (say PoC 7) than if PoC 5 had been placed elsewhere. Nevertheless, it will form a good solution and provides a useful upper bound to the costs

It is used within the *FindInitialBound*routine in the following manner:

- Forms the initial size-3 ring from the first 3 PoCs
- Calls *FindDPBest* to find the best dynamic programming solution for the re-ordered PoCs

We then trya few other dynamic programming solutions (calling *FindDPBest* each time) with different orderings of PoCs to see if we can improve on the initial bound

3.3.24 FindDPBest

Location:	Found in the *ShortestRing* module

Purpose:

Finds the best result from the dynamic programming method of adding 1 PoC at a time using the PoC ordering passed in by the array *bPermuteOrder()*. Note that the result depends on the order of the PoCs and so is unlikely to be optimal. Having formed the initial size-3 ring (passed into this routine), loops through the rest of the PoCs in the *bPermuteOrder* order and at each stage:

- Loops through all possible positions where the new PoC could be added, initially between PoCs 1 and 2. At each loop:
    - Calculates the distance cost "saved" – i.e. initially the distance between PoC1 and PoC2 since there is no longer a direct join between these 2 PoCs if we are adding the new PoC in between
    - Adds on the two parts to the extra distance costs – so initially these parts are (i) the extra distance cost between PoC1 and the new PoC and (ii) the extra distance cost between PoC2 and the new PoC
- Inserts the new PoC in the position where the extra distance cost added less the distance cost saved is the smallest (i.e. the smallest net addition to the distance cost) and proceeds to the next PoC in the *bPermuteOrder* order

Once we have processed all the PoCs, we have the solution using the dynamic programming method and we set the best ring so far to be this ring order and set the initial upper bound to be the associated distance cost

3.3.25 InitBranchPoints

Location: Found in the *ShortestRing* module

Purpose: This sets out the branch points in the search tree. The way the Branch and Bound algorithm works is to build up the full-size ring from an initial size-3 ring adding a PoC at each stage. Potentially, it will consider every possible combination of ways to order the full-size ring. However, there is an inequality we can use that means we do not need to search through every possible combination. The inequality is that any size-(n+1) ring will always have at least as much total distance cost as a size-n ring provided we keep the n PoCs in the same order in the ring. That is, adding a new PoC anywhere between two existing PoCs in the ring cannot reduce the distance cost. This is the same as saying the net addition to the distance cost in the *FindDPBest* routine cannot be negative

Using this inequality, once we have identified a size-n ring that exceeds the bound (the cost of the best full solution found so far), there is no point in proceeding along that branch because we can only add extra costs to it by adding on the rest of the PoCs. We cannot reduce the costs. Thus we should give up on that branch and proceed to the next branch point. See the *FindShortestRing* sub-section for more details on the algorithm

The *InitBranchPoints* sets out those branch points. Specifically, we have an enumeration of the rings being considered and *InitBranchPoints* stores the enumeration of the branch points in the global array *giBranchPoints()*

The search tree can be seen in the following diagram:

```
----1-----4----8----13.....etc
                ----9
                ---10
                ---11
                ---12
          -----5----8
                ----9
                ---10
                ---11
               ---12
          -----6...etc
          -----7
----2-----4
          -----5
         -----6
          -----7
----3-----4
          -----5
          -----6
          -----7
```

Table 3.7: Example search tree for Branch and Bound method [Source: Analysys]

The numbers in the diagram represent the enumeration of the rings. The rings in the first column represent the unique combinations for 4 POCs, the second column for 5 POCs, the third column for 6 POCs and so on. The top row describes rings being developed starting with one of the three possible combinations of 4 POCs, Ring 1. Reading across the top row:

- Ring 4 is formed by adding a single PoC to Ring 1 between Ring 1's first and second nodes.
- Ring 8 is formed by adding a single PoC to Ring 4 between its first and second nodes
- Ring 13 is formed by adding a single POC to Ring 8 between its first and second node etc.

If Ring 8 exceeds the cost bound, then all subsequent rings developed from it will also exceed the cost bound. Therefore, there is no need to consider any more of the rings developed along that specific search path. The top row are said to be the *branch points* since they are the first rings considered at each size.

Columns in the diagram describe all combinations of a fixed number of POCs into unique rings. For example, reading down the third column from the top of the diagram:

- ring 9 represents adding a single PoC to Ring 4 between its second and third nodes
- ring 10 represents adding a PoC to Ring 4 between the third and fourth nodes
- ring 11 represents adding a PoC to Ring 4 between the fourth and fifth nodes
- ring 12 represents adding a PoC to Ring 4 between the fifth and first nodes.

Having generated Rings 8–12 from Ring 4 and considered all the necessary full-size rings that can be formed from them, we can re-use the memory assigned to these rings. Thus, five new rings numbered 8–12 are constructed from Ring 5, which will be different to the previous rings labelled 8 through 12. This principle of reuse is used throughout the algorithm and requires significantly less memory. For example, if only rings containing up to six POCs are required, then memory for only 12 rings is required at any one time.

3.3.26 FindShortestRing

Location: Found in the *ShortestRing* module

Purpose: This runs through the Branch and Bound algorithm for solving the travelling salesman problem:

- We start off with a given PoC order and add the PoCs one at a time in this order. So the initial ring of size 3 is made up of PoCs 1, 2 and 3. Rings of size 4 are made up of PoCs 1, 2, 3 and 4 but may be in any of three orders – 4, 1, 2, 3 (Ring 1); or 1, 4, 2, 3 (Ring 2); or 1, 2, 4, 3 (Ring 3). Note the labelling of the rings Ring1, Ring2 and Ring3. This number forms the array argument inside the *objTestRings()* object.

- Note cyclic and mirror symmetry means that these 3 rings cover all possible rings of size 4 made up of the first 4 PoCs.

- We form a tree of Rings 1, 2 and 3 investigating all three possible rings of size 4. We then go back a BranchPoint, so back to Ring 1 in order to consider a search path from Ring 1.

- From Ring 1, we form a new subtree of Rings 4, 5, 6, 7 from that BranchPoint (Ring 1 is a branch point) investigating all four possible rings of size 5 (from the given 4-ring – Ring 1;

and given PoC order), then go back a BranchPoint (so back to Ring 4).

- We then form a new subtree of Rings 8, 9, 10, 11, 12 from that BranchPoint for all five possible rings of size 6 (from the given 5-ring – Ring 4; and given PoC order) then go back a BranchPoint (so back to Ring 8), etc.

- Suppose we only need to go up to rings of size 7 (assuming there are only 7 PoCs in this cluster), then we need only go up to Ring 18 in this list and thus only store 18 rings at any one time.

- Once we have processed the full-sized size-7 rings, Rings 13, 14, 15, 16, 17, 18 (formed from Ring 8), we then go back to Ring 9 and form a new set of Rings 13, 14, 15, 16, 17, 18 starting from Ring 9 as opposed to Ring 8.

- We then do the same but starting from Ring 10, and again starting from Ring 11 and once more starting from Ring 12.

- Then we go back to Ring 5 (as opposed to Ring 4) and form a new set of Rings 8, 9, 10, 11, 12 from which to go from to form the new sets of Rings 13, 14, 15, 16, 17, 18.

- Then go from Ring 6 and Ring 7.

- Then go back to Ring 2.

- Then finally go from Ring 3.

Note that if any ring exceeds the current best cost bound then there is no point considering any further rings along that branch as they will also exceed the best cost bound. Contrariwise, any full-size rings that have lower distance cost than the current best cost bound are stored and the best cost bound set to that ring’s distance cost.

The algorithm is employed in the code in the following manner:

- Copies the initial size-3 ring into the *objTestRings*() array for Rings 1, 2 and 3. Then for each of these in turn, call *AddBranch* to add on the 4th PoC in the appropriate position for that ring.

- If there are only 4 PoCs then finds which of those 3 rings above is the best.

- Otherwise, calls the routine *RecurseThroughTree* to recurse through the search tree in accordance with the algorithm above.

3.3.27 AddBranch

Location: Found in the *ShortestRing* module

Purpose: Takes an existing ring, and adds on the next PoC to it in the stated position. Updates the distance cost and kills off the branch (by setting the *objRing*'s *bolActive* flag to False) if the distance cost exceeds the current best cost bound. On the other hand, if the ring is now full-size and its cost is lower then the current best cost bound, it stores the ring in *objBestRing* and updates the current best cost bound to be the cost of the ring

3.3.28 RecurseThroughTree

Location: Found in the *ShortestRing* module

Purpose: This is the core of the Branch and Bound algorithm. It employs a recursion technique to search through all possible ring combinations that have the potential to be better than the current best ring solution. *RecurseThroughTree* looks to see whether the branch from the current ring is active and if so, it copies the current ring into each of a new set of rings enumerated along the branch (see *InitBranchPoints* for more details). Then to each of these new rings, it calls *AddBranch*, which adds a PoC to the ring in the appropriate position. If this new ring survives (i.e. its cost is still less than the current best cost bound), it calls *RecurseThroughTree* to recurse from this new ring

3.3.29 OutputARing

Location: Found in the *OutputTSPResults* module

Purpose: This calculates the capacity requirements of the ring. The capacity requirement is the sum of the capacity at each PoC on the ring except for the LAS PoC, plus the capacity at any PoC on any other rings that are joined to this ring to get to the LAS.

It then updates the sheet, created in *RunTheAlgorithm* that is named after the current LAS, with the results of the best ring found for that cluster. Furthermore, it updates the chart (by adding the data and calling *FormatChartSeries*) on that sheet to display these results. Finally, it add the results to the output table in the 'Output PoCs' worksheet

3.3.30 OutputFinalResults

Location:	Found in the *OutputTSPResults* module
Purpose:	This updates the sheet, created in *RunTheAlgorithm* that is named after the current LAS, with an overall summary for all the ring clusters in that LAS area. For example, outputting the total distance cost of all the best rings

3.3.31 FormatChartSeries

Location:	Found in the *OutputTSPResults* module
Purpose:	Sets the line size and format of the output chart series. If we're adding the first series, then also calls *FormatTheChart* to format the whole chart

3.3.32 FormatTheChart

Location:	Found in the *OutputTSPResults* module
Purpose:	Formats the output chart. For example, removing gridlines and the legend

3.3.33 ClearBestRings

Location:	Found in the *RunTSP* module
Purpose:	Clears out the memory of the *objBestRings()* object ready to be used for another combination of partitioning the PoCs into ring clusters

3.3.34 RecurseCombinations

Location:	Found in the *RunTSP* module
Purpose:	This creates the partitioning of the PoCs into rings, forming the

array *lRings()* which stores this partitioning. It proceeds in the following manner:

- Loops *lPOCToChange* through from the last PoC back to the passed-in argument *lLastPOCToChange*
- At each stage in that loop, loops *lRingForPOCToChange* from 2 up to the maximum number of rings (obtained earlier in *RunThroughCombinations*)
- Within this inner loop:
    - Sets the *lLastPOCToChange* PoC to be *lRingForPOCToChange* – the cluster ring that it has been incremented to be in.
    - Sets all later PoCs to be in ring cluster 1
    - Calls *UseThisCombination* to test out this partitioning to see if it improves upon the best rings found so far
    - Calls *RecurseCombinations* with *lPOCToChange + 1* passed in as *lLastPOCToChange*

For example, let us suppose there are 3 rings to assign 6 PoCs to. (Note this is unrealistic since 6 PoCs would fit on one ring, but it is useful for illustration.) We start off assigning them all to Ring 1 (which we can represent by the series 111111) and call *UseThisCombination* on that. Then we increment the last PoC to 2. So the PoCs are assigned to rings according to the series 111112 and again call *UseThisCombination*. We increment it again to get the series 111113 each time calling *UseThisCombination*. Then the last PoC is at the maximum number of rings, so we go to the next PoC back and set all later PoCs to 1. Thus we get to 111121. The series continues as follows:

- 111122
- 111123
- 111131
- 111132
- 111133
- 111211
- 111212
- 111213
- 111221

- 111222
- 111223
- 111231
- 111232
- 111233
- 111311

etc.

Eventually, all possible partitions are considered. Note that many of these partitions will not be valid – for example they may have too many PoCs in any one ring cluster. These will be rejected in the call to *TestRingValidity* inside *UseThisCombination*

3.3.35 RunGeneticAlgorithm

Location:	Found in the *GeneticAlgorithmForRings* module
Purpose:	This is called from *RunThroughCombinations* when the number of PoCs in the LAS is more than *glMaxPOCsForExhaustiveSearch*. In these cases, an exhaustive search would be prohibitive upon processing time and a genetic algorithm is employed to find a near-optimal solution

This routine proceeds as follows:

- Calls *SeedPopulation* to set up an initial "population" of test partitions of PoCs into ring clusters. Each member of the population represents a partitioning of the PoCs into ring clusters.

- Calls *TestPopulation* to run through each of these partitions to find the best possible rings for each partition and records their results. If any of these results improve upon the best result found so far across all partitions tested in terms of least distance cost, then it records it

Loops through "generations" from 1 up to *glNumGenerations* – the number for *GA.Num.Generations*. In each generation:

- Calls *SortThePopulation* to order the population, putting the best partitions first and the worst ones last. The best partitions are those for whose best rings incur the least distance cost

- Calls *SeedPopulation* to set up a few new test partitions –

although this time the seeded population is only part of the full population

- Calls *MergePopulation* to produce "offspring" preferentially from the better partitions found in the previous generation
- Calls *MutatePopulation* to make some small random adjustments to the population
- Calls *TestPopulation* to run through each of the partitions in the new population, again finding the best possible rings for each partition, recording the results and updating the best solution where appropriate

In each generation, there will be POPULATIONSIZE (=100) members in total, NEWRANDOMS (=10) of these will be newly seeded members and the rest will be from merging two parents

Once we have run through *GA.Num.Generations*, we use the best partitioning found so far to output as the results. Note that this may or may not be the optimal solution but should always be a very good solution

3.3.36 SeedPopulation

Location: Found in the *GeneticAlgorithmForRings* module

Purpose: Sets the population of PoC partitions for the genetic algorithm. Each member of the population represents a partitioning of the PoCs into ring clusters. These are seeded at random. Note that arguments passed in to this function determine which members of the population are seeded. The first time it is called, all members of the population are seeded. On later calls, only some members are seeded – others are formed by merging and mutating existing members of the population

3.3.37 TestPopulation

Location: Found in the *GeneticAlgorithmForRings* module

Purpose: Runs through each member of the population and calls *UseThisCombination* on that combination. *UseThisCombination* tests whether the partition is valid (e.g. the number of PoCs in any ring cluster does not exceeded the maximum permitted), then identifies the best order of PoCs in each ring represented by the partitioning and determines the associated distance cost

3.3.38 SortThePopulation

Location:	Found in the *GeneticAlgorithmForRings* module
Purpose:	Orders the population according to which partition can have the cheapest set of rings formed for it. It calls the routine *SortTheScores* to perform this task on the temporary array *lPopulationOrder()*. It then updates the ring partitioning arrays *lMemberRings()* to reflect this re-ordering

3.3.39 SortTheScores

Location:	Found in the *Sort* module
Purpose:	Uses the standard Quicksort method to order the scores (cheapest way of forming the rings given the partition). It calls the *ScoresQuickSort* routine to do this

3.3.40 ScoresQuickSort

Location:	Found in the *Sort* module
Purpose:	Recursively sorts scores by swapping points from the lower half with points from the upper half of the population when a point in the lower half scores higher than the medium value and a point in the upper half scores lower than the medium value. It then recurses within each of these halves until all the points are sorted. It uses the routine *SwapMembers*, which just swaps two points

3.3.41 SwapMembers

Location:	Found in the *Sort* module
Purpose:	Swaps two points within an array

3.3.42 MergePopulation

Location: Found in the *GeneticAlgorithmForRings* module

Purpose: This runs down through the new generation population members from POPULATIONSIZE - NEWRANDOMS to REPRODUCERS + 1. At each stage in the loop, two random parent members from the top scoring REPRODUCERS number of the previous generation population are used to generate the partition for the new generation population member

It then runs down the new generation population members from REPRODUCERS To 1 counted by *lMember*, only selecting parents from the top scoring *lMember* number of the previous generation population

In this way, higher-scoring members are preferentially selected for being used to form members in the next generation

It calls *MergeTwoParents* in order to produce the next generation member from the two current generation parents

3.3.43 MergeTwoParents

Location: Found in the *GeneticAlgorithmForRings* module

Purpose: This takes two members from an existing population and produces one new member for the next generation. In the “parent” members, each PoC has been assigned to a ring cluster. In the new member, for each PoC, there is a 50-50 chance that it will take the first parent’s ring cluster assignment and otherwise will take that of the second parent

3.3.44 MutatePopulation

Location: Found in the *GeneticAlgorithmForRings* module

Purpose: This adds some more random element into the process by adjusting a random number of PoC cluster ring assignments. For each member of the population, it will adjust a random number up to MAXPOPMUTATE of the PoCs. When it adjusts them, it sets the new cluster ring that the PoC is assigned to be to a random number up to the maximum number of cluster rings

3.3.45 RunForBestRings

Location:	Found in the *RunTSP* module
Purpose:	Whichever method is used from *RunThroughCombinations*, we end up calling *RunForBestRings*. At this stage, we have either found the optimal rings for the sets of PoC by using an exhaustive search mechanism through *RecurseCombinations* or we have found a very good and still possibly optimal solution through *RunGeneticAlgorithm*. We then call *RunForBestRings* to run through this best solution, but this time to output the results as we go. Hence this routine calls *SetUpTSPData* as before but using the best rings found already, then calls *RunTheAlgorithm* again but this time with the first argument set to True so it knows to output the results. Finally, it calls *ClearBestRings* to clear out the memory used by *objBestRings()*

3.3.46 Outputs of the RunTSP algorithm

The *RunTSP* algorithm produces the following outputs in the 'Output PoCs' worksheet:

Named range	*Column title (on the 'Output PoCs'] worksheet)*	*Description*
TSP.POC.ID	POC Id	An identifier for the PoC
TSP.POC.Name	POC Name	The name of the PoC (which is an LE name)
TSP.Lat	POC Latitude	The latitude coordinate for the PoC
TSP.Long	POC Longitude	The longitude coordinate for the PoC
TSP.LAS	LAS	The parent LAS ID to the PoC
Ring	Ring	An identifier for the ring on which the PoC should be a member
TSP.Num.POCs	Number of POCs in LAS	The number of PoCs in the LAS area
TSP.Is.a.LAS	Is a LAS?	Flag as to whether the PoC is also the LAS (a "Y" means yes, otherwise the column is left blank)
Bridge	Bridging Node	Flag as to whether the PoC is also on the Parent ring (connects to a LAS) (a "Y" means yes, otherwise the column is left blank). Note that such bridging nodes appear more than once in the outputs since they appear for each ring they are on. The "Y" will appear in the column for the entry representing the child ring. In addition, the node representing the LAS will appear as a bridging node on one (but only one) of the rings connecting to the LAS.
SIOs	SIOs	The aggregate number of SIOs in the PoC area (summed over all LEs in the PoC area)
TSP.Next.Node	Next Node	The next node (PoC) along the ring that this PoC gets joined to
TSP.Dist.To.Next.Node	Dist To Next Node	The distance to the next PoC that the PoC is joined
Ring.Joined	Ring Joined To	Where there are Child rings, the ring joined indicate the Parent ring of the Child ring
In.Las.Ring	Is In LAS Ring	Flag as to whether the PoC is a Parent ring (contains the LAS)
SDH.Transmission.Capacity	SDH Transmission Capacity	The capacity that must be sustained at the PoC. If the ring it is on contains the LAS, then the "SDH.Transmission.Capacity" is calculated as the sum of the SIOs in the LAS area at all the PoCs, excluding the LAS. Otherwise it is just the sum of the SIOs in the PoC's ring, excluding the bridging node
SDH.Transmission.Capacity.Formula	SDH Transmission Capacity Formula	The "SDH.Transmission.Capacity" output as a formula so that it is updated if the number of SIOs changes

Table 3.8: Table outputs of the RunTSP algorithm [Source: Analysys]

In addition, for each LAS area, a new worksheet is generated. In each worksheet, there is:

Title	*Description*
Total number of POCs	The number of PoCs in the LAS area
Total number of Nodes	The number of PoCs in the LAS area but double counting those that form bridging nodes (join two rings together)
Total Cost of All Rings	The distance cost associated with all the rings in the LAS area
Time Taken	The time taken to perform the calculation for this LAS area

Table 3.9: Worksheet summary outputs of the RunTSP algorithm [Source: Analysys]

Then for each ring in each LAS area, the following outputs are provided:

Title	*Description*
Ring	The identifier for the ring
Number of Sites	The number of PoCs in the ring
Cost of Ring	The distance cost associated with the ring
Optimal Node Order	The order in which the PoCs should be joined in the ring
Node	An identifier for the Node
Latitude	The latitude of the PoC
Longitude	The longitude of the PoC
Site Capacity	This is set to 1
Stored Node	The re-ordered numbering of the Nodes used in the algorithm and can be ignored. It is useful for debugging the algorithm
POC ID	The identifier for the PoC that the Node corresponds to
POC Name	The name of the PoC
LAS	Marked next to the nodes that represent the LAS

Table 3.10: Worksheet ring outputs of the RunTSP algorithm [Source: Analysys]

3.4 Other routines

The longitude / latitude distance measure used throughout the algorithms for the core network are executed by the functions calc_dist_between_two_points and calc_dist_between_two_DblPoints. Both functions take the coordinates (LongA, LatA) and (LongB, LatB) of two points as inputs and return the spherical distance D between them by the formula:

$$D = R * A\cos\left[Cos\left(\pi * \frac{(90 - LatA)}{180}\right) * Cos\left(\pi * \frac{(90 - LatB)}{180}\right) + Sin\left(\pi * \frac{(90 - LatA)}{180}\right) * Sin\left(\pi * \frac{(90 - LatB)}{180}\right) * Cos\left(\pi * \frac{LongA - LongB}{180}\right)\right]$$

where R is the radius of the Earth.

Both of these functions can be found in the *Clustering* module for the core routine.

The following routines used in this code have almost identical equivalents described in section 2, as they also form part of the geoanalysis and access network module. The mapping of the two sets of subroutines is shown below in Table 3.11.

Location	*Core routine*	*Equivalent access routine*
Clustering module	*Divisive_clustering*	*DivisiveClustering*
	AllocatePointToCluster	*AllocatePointToCluster*
	simple_reassignment	*SimpleReassignment*
	Swap	*Swap*
	full_optimisation	*FullOptimisation*
	initialise_cluster_allocations	*InitialiseClusterAllocations*
	initialise_parent_capacity	*InitialiseParentCapacity*
	cal_maxd_in_P	*CalSquareMaxDInP*
	cal_unweighted_centre	*CalUnweightedCentre*
	choose_first_child_member	*ChooseFirstChildMember*
	cal_total_distance	*CalcTotalDist*
	write_cluster_results	*WriteClusterResults*
SpanningTree module	*construct_tree*	*ConstructTree*
	identify_attached_and_unattached_points	*IdentifyAttachedAndUnattachedPoints*
	test_add_unattached_point_to_attached_point	*AverageCostPerLine*
	add_to_edge_list	*AddToEdgeList*
	add_cheapest_in_edge_list_to_objEdges	*AddCheapestEdgeInListToObjEdges*
	StoreRoutes	*StoreRoutes*
	GetTotalDistance	*GetTotalDistance*
	setup_DistanceMatrix	*SetupGdDistanceMatrix*
	setup_points_in_cluster	*SetupPointsInCluster*

Table 3.11: Other routines documented in the "Script for the access network algorithms" [Source: Analysys]