RW Net 4.23

A Network Analysis System SDK

Table of Contents

Part I

User Manual

1 User Manual

1.1 Introduction

RW Net 4.23

RW Net is a general purpose routing library. It is flexible enough to be used together with almost any GIS system available and it will also work together with most programming tools on the market.

RW Net uses it's own format for storing street networks and included are functions for importing 25 street databases from most common GIS formats. This topological format is targeted towards routing purposes and is described here 11. RW Net always loads the topological network into memory before doing any calculations.

The basic structure in the topological network is a one-to-one relationship, where the first link in the network matches the first record in your GIS file, second link matches second record etc. This makes the network files very compact and fast to use.

All attribute information (road class, one-way information etc.) is held in a separate data structure which can easily be updated without having to re-create the topological network.

1.2 Feature matrix

Features

	Standard	Pro
Network size	500,000 links	100,000,000 links
Import 25	Single file only	Multiple files
Import formats	MIF, SHP, TAB	MIF, SHP, TAB, events
Import from SQL 33 database		Yes (VCL, DLL)
Node-2-node routing	Yes	Yes
Matrices (*)	Yes	Yes
Spatial searches 64	Yes	Yes
Max list length (*)	300 items	No limit
Shortest / fastest / cheapest route	Yes	Yes
32 road classes	Yes	Yes
Geographic & projected coordinates 10	Yes	Yes
Alpha parameter 87 for improved speed	Yes	Yes
Output to MIF, SHP, KML, GML, CSV, DBF, array, GeoJSON	Yes	Yes
Output to TAB on win32/win64	Yes	Yes
Turn restrictions	Yes	Yes
Limits 9 (max weight, width etc.)	Yes	Yes
Dynamic segmentation	Yes	Yes
Driving directions 92 (*)	Yes	Yes
Travelling salesman optimization 122 (TSP) (*)	Symmetrical	Asymmetrical too
Nearest N facilities 72 (*)	Yes	Yes
Isochrone functions - link based (*)	Yes	Yes
Isochrone functions - voronoi 98 based (*)	Yes	Yes
Export of network 44	Yes	Yes
Topological checks (subnets 87, missing snap 65, parallel links 53, cul-de-sac 43, overpasses 65 etc.)	Yes, up to 10,000 links	Yes

Encryption of network files 27		Yes
Smoothing of isochrones 102		Yes
Hierarchical routing 7		Yes
Approach based routing 90		Yes
Multi-threaded calculation 82		Yes
TSP with curb approach 125		Yes
Mixed Rural Postman Problem (arc routing)		Yes
Join links 62		Yes
Clustering 115		Yes
Minimum Spanning Tree 85		Yes
Weighted Center of graph 84		Yes
Traffic Assignment 91		Yes

Functions marked with (*) only accepts 300 items in Standard version.

1.3 System requirements

Available for:

- .NET (fully managed)
- .NET Silverlight

- Delphi XE2 - XE3 - XE4 - XE5 - XE6 - XE7 - XE8 - 10 (32/64-bit)

- DLL 12 for 32-bit MapInfo / MapBasic 7.5 - 15.0
- DLL 12 for 64-bit MapInfo / MapBasic 12.5 -

Some versions are only available in Pro version - see license terms 20.

All versions are fully self-contained and 100% "native" on each their own platform (no "wrappers").

RW Net 4 is 100% Unicode enabled.

The older RW Net 2 also includes versions for older Delphi compilers.

1.4 Quick overview

A normal setup includes these steps:

1. Import geographic coordinate data into RW Net's own format with class TImport 25
2. Import attribute information such as street names, one-way information etc. with class TImportAttributes 30
3. Open the generated files with class TNetwork 35
4. Perform spatial searches with class TSpatialSearch 64 or
5. Perform isochrone calculations (one-to-many, matrices etc.) with class TCalc 68 or
6. Perform one-to-one route calculation with class TRouteCalc 87

Routes can be exported to a lot of standard formats using one of the TGISwrite 104 classes. Several of the functions write directly to one of the GIS formats.

Drivetime isochrones

A typical use of the software is the calculation of drivetime isochrones, showing how far you can get in 5, 10, 15 minutes etc.
We have explained the various options in more detail here 16.

Optimization

Matrices can be used in one of the TSP classes (TTSP 122 and TTSPcurb 125) to perform an optimization of the sequence.

TOptimizer 115 is used for creating territories according to various criteria: Load, size etc.

1.5 Network terminology

Terminology used to describe the various elements of a street network:

A *link* consists of several connected vertices (2 or more, blue squares on the map below). The first vertex of a link is called the *from-node* and the last vertex is called the *to-node*. See function Link2FromNode 49 and Link2ToNode 49.

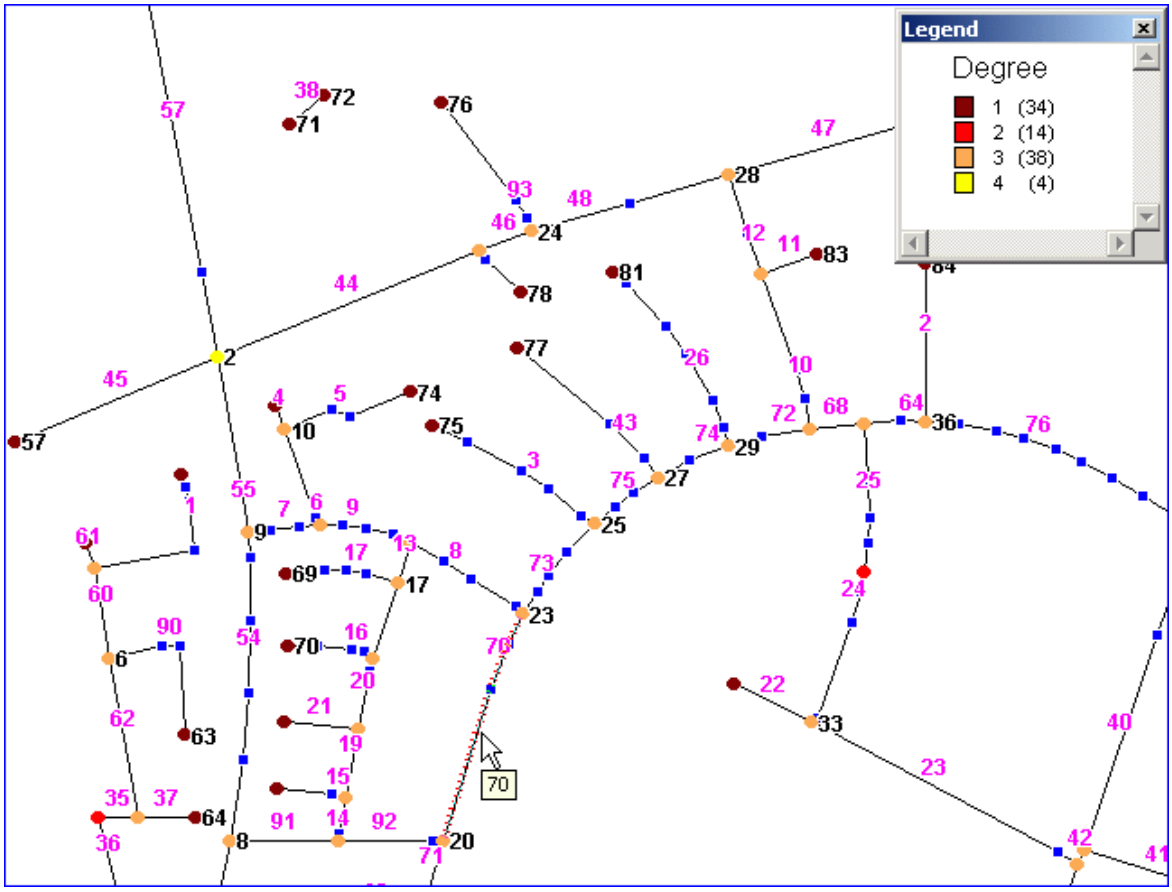
Most of the nodes share coordinates with nodes of other links. The number of links sharing a node is referred to as the degree of the node. You should normally never reach more than 10. See function Degree 43.

A node is also called an *intersection* - even if the degree is 1 or 2. A link where one of the nodes has degree 1 is called a *dangling* link. A node with degree 2 is called a *pseudo-node* (see function Join 62). The red node on the map is such a pseudo-node.

A link is identified by it's internal ID (Magenta text on the map: 1, 2, 3 ....), which corresponds to the record ID's of the input dataset used by function TImport.execute 27.

A node is identified by it's ID (Black text on the map: 1, 2, 3 ....). Node ID's are primarily ordered by degree in descending order and secondarily by x-coordinate in ascending order. Node ID's are assigned during network import and can not be controlled by the user.

A *location* is a position on a link: e.g. 50% along link 70 - counted in the same direction as that the link has been digitized in. This appr. matches the cursor on the map below. Locations are used when doing dynamic routing. The percentage needs to be between 0 and 1 (both included).



1.6 Link information

1.6.1 Attributes

The attribute for each link in the network play a key role in defining how the link is used in the routing calculations. This is defined through a bit-pattern:

1. Road class, 0-31 (5 bits)
These have no predefined meaning, but their value can be translated into a drive time using function CalculateTime 40

2. Hierarchy (*), 1-5, (3 bits, 32-64-128, bit 5-7)
A topological hierarchy can be used for speeding up TRouteCalc 87 calculations.
0 is also allowed, if you don't use the hierarchy at all.
See further explanation here: Hierarchy 7.

3. No-drive through (*), true/false (1 bit, 256, bit 8)
This can be used to define areas, where you are not allowed to drive through to get to the target.
Applies to TRouteCalc 87 calculations.
See TNetwork.NoDriveThroughCheck 51, TNetwork.NoDriveThroughInit 52 and TRouteCalc.NoDriveThrough 79

4. One-way, To-From direction not allowed, (1 bit, 512, bit 9)

5. One-way, From-To direction not allowed, (1 bit, 1024, bit 10)

If both bit 9 and 10 are set, the link is closed for driving.

6. Roundabout, true/false (1 bit, 2048, bit 11)
Can be used in creating driving directions.

7. Non-driving link, such as a ferry or car-train, true/false (1 bit, 4096, bit 12)
Can be used in creating driving directions.

8. True if not allowed to make U-turns at the From-end of the link. (1 bit, 8192, bit 13)

9. True if not allowed to make U-turns at the To-end of the link. (1 bit, 16384, bit 14)

10. SkipInSearch (*), true/false (1 bit, 32768, bit 15)
For use with function NearestLocation 66
See AttributeSetSkipInSearchBit 39

(*): Changed from RW Net 2.

An example:

A road of class 4, which can only be travelled in the direction of digitization: 4 + 512 = 516.

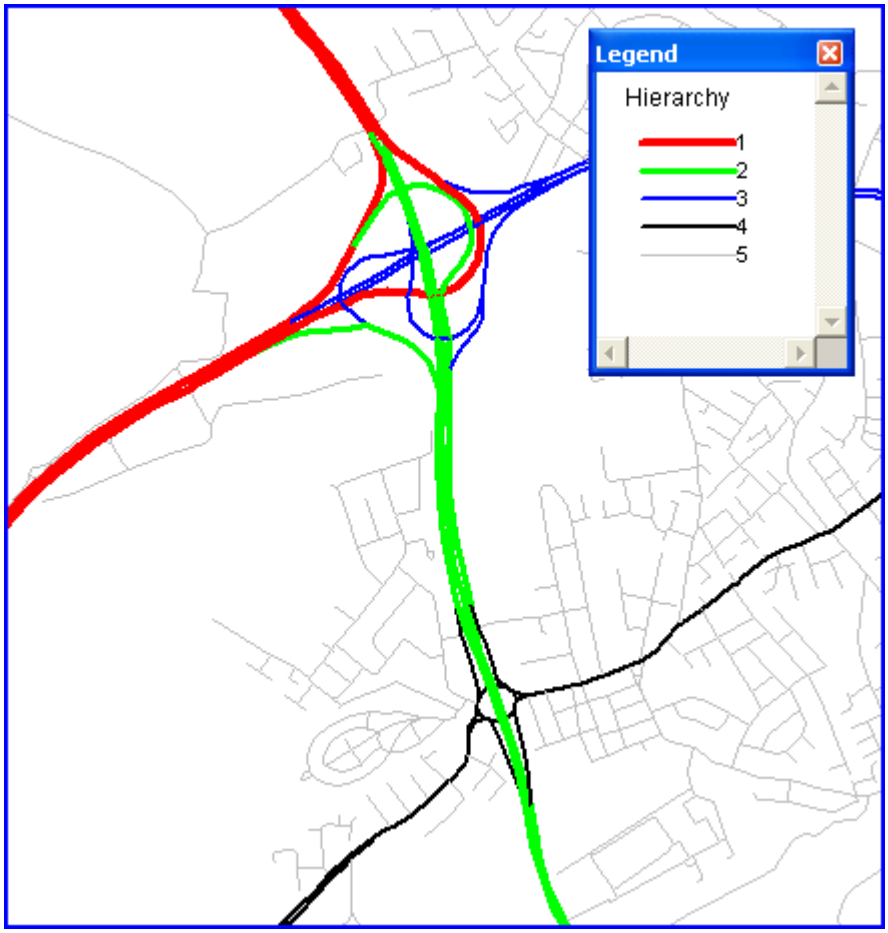
1.6.1.1 Hierarchy

Some street databases has special attributes for the most important streets, the ones being used as part of long routes.
This will typically be motorways, but can also be ferries, bridges and some minor streets which are required to have a connected network.

The advantages of restricting routes to these more important streets are:

- Much faster point-2-point route calculations for long routes (TRouteCalc 87).
- Simpler routes, which doesn't make short-cuts via minor roads to make a long route a little shorter / faster.

The map below shows an example from TomTom Multinet data with 5 layers of importance (hierarchies):



RW Net 4 uses a method where the calculation of the route is restricted to level 1..X as soon as level X has been reached on the route unless you are
within a certain distance Y of the final target. Then additional levels are included in the search again.

For the algorithm to work properly the parameter Y has to be supplied for levels 2 to 5. Level 1 (the top level) is of course always included in the search.
The best values for these parameters depend on the geometric properties of the network and how the hierarchy attribute has been setup.

If you have less than 5 levels in your data source - 3 for instance - use levels 1, 2 and 3.

If you choose small parameters values, a smaller part of the network is considered when you get close to the target and this improves calculation speed.
The downside is you risk not finding the target at all (!), because there are no major streets within the limits you have defined.
The solution to this problem is to re-calculate without the hierarchy setting or just use a more relaxed setting (larger parameter values).
Such re-calculations are costly and when choosing parameters it is important to find a balance between normal, fast calculations and the slow re-calculations.

Functions for working with hierarchies:

TNetwork.Hierarchy 49 For getting / setting hierarchy for a single link

TRouteCalc.Hierarchy 89	For enabling / disabling hierarchy for a calculation
TRouteCalc.SetHierarchyLevel 90	For defining parameters for the hierarchy - suggestions for TomTom and Navteq databases

1.6.2 External ID

Each link can have an associated external ID as opposed to the internal ID (1, 2, 3...). The external ID can be either integer (0..2147483647) or string based.

The advantage is an external ID can be constant over time and globally unique - even when working with a subset of a larger database.

RW Net includes functions for translating between internal and external ID, but otherwise all functions uses internal ID for input / output.

1.6.3 Limit

Besides the routing options available as part of the attribute 6 bit pattern, it is also possible to define 2 other kind of route restrictions for links in the network:

1. A scalar quantity such as a maximum weight, height, width etc. If the limit for a certain link in the network is 100 and you calculate a route for a vehicle with a value >100, that link will be avoided in the route. It is mandatory to scale your limits into the 1-255 interval.

2. A ***bit pattern*** for defining special links such as ferries, toll roads etc. which you may want to avoid in your routing. If the limit for a link is 3 = 00000011 it may mean it is both a ferry and a toll "road" (most ferries are not free, so that seems logical). If your value has either bit 1 or 2 set, that link will be avoided in the route. It is possible to define 8 such bits within each limit.

For both types, a link value of 0 means no limitations at all.

A maximum of 9 such limitations can be created.

See TImportAttributes 30, TNetwork 35 and TCalc.SetLimit 56.

1.6.4 Road name

It is possible to have a road name for all links. Multiple sets can be created so a link can have the name "Main Street" in one setup, but "Main Street, Smalltown" in another setup. Or use different languages.

Road names are stored using Unicode and always converted before output, depending upon the chosen file format and codepage.

Road names can be used in driving directions 92 and in functions ExportLinks 44 and Join 62.

1.7 Turn restrictions

There are 2 types of restricted turns:

- Banned turns
- Delayed turns

Normally you will be using banned turns only, since for most normal routing purposes, setting different road speeds for road classes are sufficient for giving a realistic route choice.

Generally you can change choice of route A LOT by using wrong values for delays, so take care.

All methods about turn restrictions has an index parameter, which points to one of the turn indices, created by CreateArrayTurn 42.

If you add a turn restriction, where one of the links making up the restriction is already marked as one-way, it is skipped.

1.8 Coordinate units

Two kind of coordinate units are supported:

- Spheric / Latitude-longitude
- Cartesian / Projected

When working with spheric coordinates, all distance calculations are performed using great circle distances and the Earth is considered a perfect sphere with radius 6378.13 km.

When working with Cartesian coordinates, all distance calculations are performed using straight Pythagoras formula. Several different Cartesian units are supported.

It is worth noting, that RW Net *never* performs any transformation between coordinate systems. It always works with the native coordinates of the base dataset used when creating the network. It will return strange results, if you set the coordinates as spheric, while they are really meters or vice-versa. It is YOUR responsibility to make sure this is correct.

See also TCoordinateUnit 142

1.9 Coordinate system

When importing 25 from a GIS street database, information about the coordinate system is stored in the INI file.

This is used for generating output files with class TGISwrite 104 - either internally or by the user.

Depending on the output formats you plan to use, this information is needed:

TAB / MIF	Coordsys clause
SHP	PRJ file
GML / GeoJSON	EPSG code
KML	Always uses lat/long, WGS 84

These should be set before importing.

1.10 Units

RW Net 4 uses metrical units almost everywhere in the setup:

- Distances: Km
- Speeds: Km/h
- Time: Minutes

The exception is miles & mph can be used in a few output-to-file functions, where the output is directly aimed at end-users:

TCalc.MatrixOut 77
TCalc.MatrixDynOut 77
TDrivingDirections 92
TRouteCalc.NearestNDyn 88

See TCalc.DistanceUnit 70 and TDrivingDirections.DistanceUnit 94 if you prefer miles.

1.11 File structure

When a network is imported, several binary files are created on disk, which together define the topological network. This gives a short description of the content of the various files:

Filename	Mandatory	Encrypted	Explanation
Attribute.bin		X	Attributes of links
Coord.bin	X	X	Coordinates of all intersections (start / end node)
Coord3.bin	X		Coordinates of the rest of the vertices
Coord3i.bin	X	X	Index into Coord3.bin
Index1.bin & index2.bin			Index for conversion between link id (1, 2, 3..) and external id.
Length.bin	X	X	Length of all links in the network
Limit?.bin		X	Information about limits on links such as max heights etc.
Link.bin & node.bin	X	X	Information about link-node relationship ("topology")
Roadname??.bin			List of possible road names, Unicode
Roadnumber??.bin			Index into roadname??.bin
Rwnet_config.ini	X	X	INI file, text format
Spatialindex.bin			Spatial index of both links and nodes

Turn restrictions can be stored in files with flexible naming.

If you set the Encryption 27 property, files marked as such in the table will be encrypted during creation and decrypted during load.

1.12 Password protection

You need to enter a password when using a non time-limited version of RW Net.

Call method InitPassword for any of these classes after instantiation:
TImport 25, TNetwork 35, TTSP 122, TTSPcurb 125

It is sufficient to supply the password *once* in an application.

1.13 Progress events

Progress events are available for these methods:

TImport.execute 27
TImportAttributes.execute* 30
TImportSQL.executeMSSQL 34

TNetwork.AttributeSave 39
TNetwork.ExportLinks 44
TNetwork.ExportLocationList 45
TNetwork.ExportNodes 45

TNetwork.Join 62
TNetwork.ObjectCheck 52
TNetwork.Open 53
TNetwork.ParallelLinks 53

TSpatialSearch.FindNonConnected 65
TSpatialSearch.FindOverPasses 65
TSpatialSearch.SplitAndSnap 68

TCalc.MatrixOut 77
TCalc.MatrixDynOut 77
TCalc.SubNet 87

TRouteCalc.MatrixDynCurbIsoChrone 89
TRouteCalc.MatrixDynCurbRoute 90
TRouteCalc.SubNetEx 91

TTSP.execute 123
TTSPcurb.executecurb 125

TVoronoi.execute 101

Assign the OnProgress event to follow progress and eventually cancel the calculations. The events steps from 0 to 100 and as a minimum for every 2 seconds.

1.14 MapBasic DLL

The rwnet4.dll is aimed for use with MapInfo / MapBasic.

Since MapInfo is a single user application, we have made several changes to make development easier.

Rather than doing Create/Free methods, we use pre-allocated objects.
Objects are referenced either indirectly (single instance) or by their index (multi instance).

Classes referenced indirectly:

- TCalc 68 / TRouteCalc 87
- TDrivingDirections 92
- TGISwrite 104
- TImport 25
- TImportAttributes 30
- TNetwork 35
- TOptimizer 115
- TRandom 136
- TRoadClassSpeed 137
- TRoadClassTurnCost 137
- TRoute 148
- TSpatialSearch 64
- TStepList 134
- TTrafficList 134
- TTSP 122
- TTSPcurb 125
- TVoronoi 98

Classes / types referenced by index (handle):

	Number of instances	Null-element
TApproachArray 141	2	
TBitArray 134	3	Yes
TCostArray 143	2	Yes
TCurbMatrix 143	1	
TFloatPointArrayEx 144	2	
TIntegerArray 145	2	
TIntegerList 133	2	Yes
TLocationList 134	2	Yes
TMatrix 146	2	
TStringList 137	2	Yes
TWordArray 150	2	

If null-element is true, you can pass 0 as index / handle, when you want to pass nil as parameter.

Function naming convention
TImport.Execute becomes TImport_Execute.
TCalc.IsoCost becomes TCalc_IsoCost.
etc.

Some method names have been shortened due to max length = 31 characters.
TCalc 68 and TRouteCalc 87 are both referenced as TCalc.

All definitions can be seen in the rwnet4.def file along with a sample application, covering key areas.

Password initialization
Call method "InitPassword 11".

Codepage
Since RW Net 4 is Unicode enabled and Mapbasic isn't, it is required to do an internal conversion in all function calls involving strings.
This is handled automatically through a global variable, which sets the codepage you are using in MapBasic.
Default is the system codepage.
Methods: GetCodepage / SetCodepage.

GIS output format
There is a global variable for output format, which is gfMITAB 144 by default.
This means it is skipped from all function calls having a gisformat parameter.
Methods: GetGISformat / SetGISformat.

Error handling
If an error happens when calling a method, you can use one of these 2 functions to test it:
GetLastExceptionClass
GetLastExceptionMessage
The messages are cleared after each successful method call.

Progress Events
These can all be turned on/off by calling SetProgress with 0/1 as parameter.
The progress is then shown with a built-in dialog.

Missing functionality compared with VCL / .NET version
- GISarray 71 output as format
- Direct access to TPolyGeneration 136

1.15 Data Sources

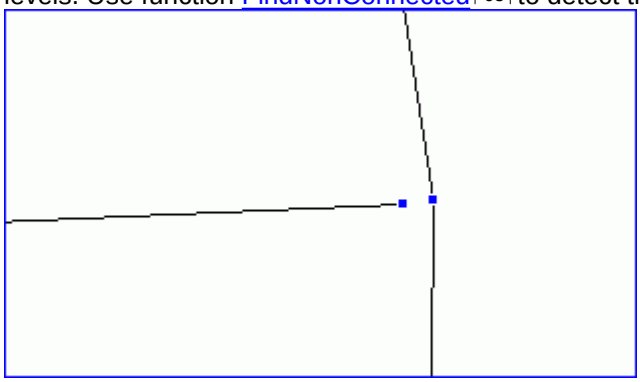
At RouteWare website you will find a list of street data providers for various parts of the world. Data from these providers usually have a topological correct structure, which means they are almost ready to be used in RW Net.

But how should your own street data look like, in order to be used in RW Net?

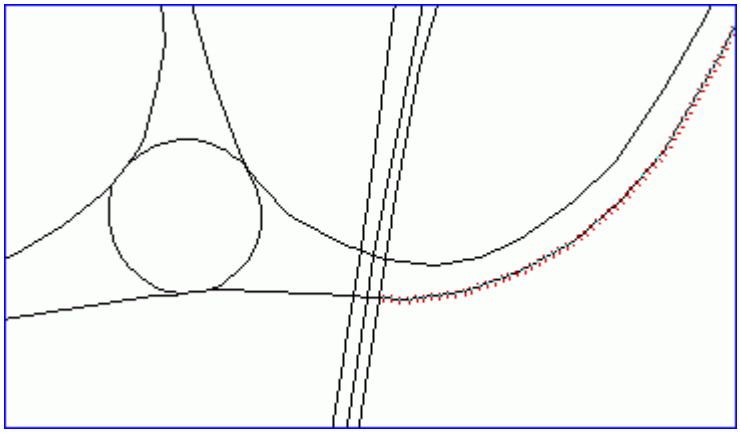
- ***They need to snap***
- ***They need to split at intersections***
- ***The network should be plane unless there is an overpass***
- ***You should avoid subnets (islands)***
- ***You should avoid very long links, which have a negative impact on speed of certain calculations***

Below is shown some examples on networks, which are NOT correct, but all look correct unless you check out the details:

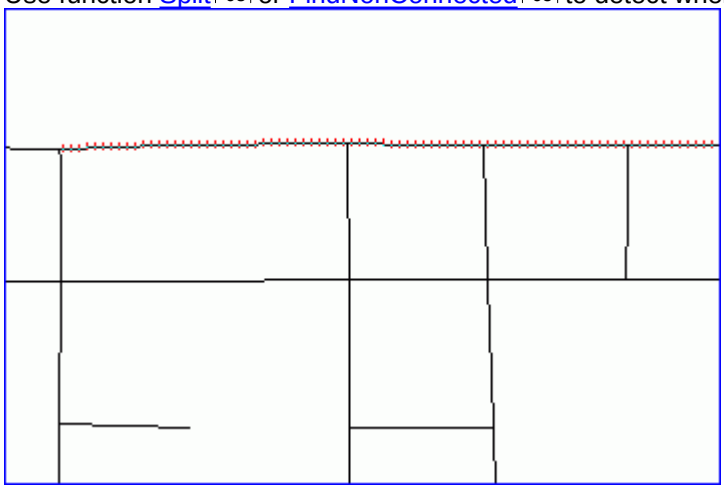
Example 1: Missing snap at an intersection
This means the network doesn't connect and the movement to / from the disconnected section, isn't possible. In the example below, the gap is just 1 meter and can't be seen at normal zoom levels. Use function FindNonConnected 65 to detect these situations.



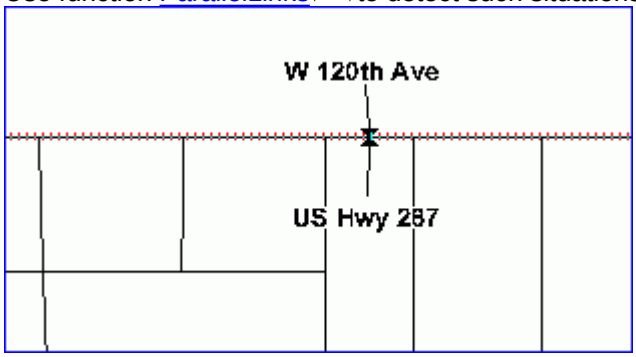
Example 2: Split at overpass / underpass
This means a lot of impossible turn movements are suddenly made possible. This is a typical problem with TIGER data.
There is no single logical check to detect these situations, it is a simple shortcoming of the data source, if there are no Z-levels 16.



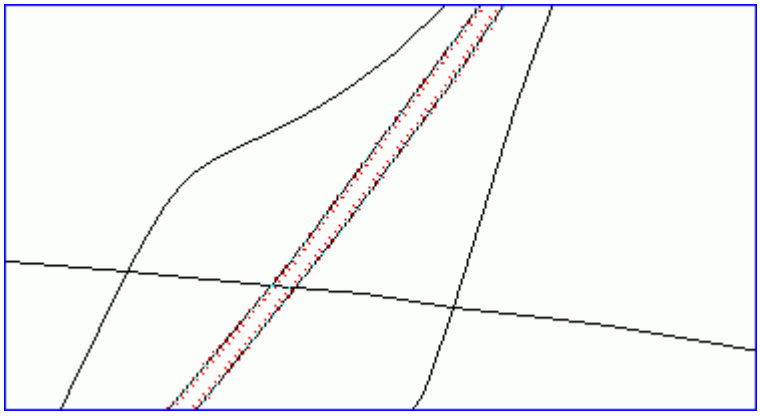
Example 3: Doesn't split/break at intersections
This means turns are not possible at most intersections.
Use function Split 68 or FindNonConnected 65 to detect where this is most likely an issue.



Example 4: Double digitization with two street names, here name + route number
Not a really big problem, but the result of a route calculation may include one of the two streets in a more or less random fashion.
Use function ParallelLinks 53 to detect such situations.



Example 5: Multi sectioned polylines
Polylines with more than 1 section are ignored. They will not be part of any route, since there is no logical start/end of the link.
These will be reported during network import in either ImportErrorList 27 or in the network_report.txt 27.



1.16 Z-Levels

Z-level is an integer from -9 to 9, which specifies the horizontal level of streets: One number for the start of the link (Z-from) and one for the end of the link (Z-to).

The information is used during the import 25 process to adjust coordinates slightly (10 cm) to prevent nodes at different Z-levels to have the same coordinates. The modification is only applied, if Z-level<>0.

It is commonly found in commercial street databases (Navteq, TomTom, ITN etc.).

If your dataset contains fromnode and tonode for the links instead of Z-level information, use ExternalNodeId2ZLevels 46 for a transformation.

1.17 Isochrones - overview

Generating nice-looking isochrones has always been a key functionality of routing software and RW Net 4 offers several methods, which are shown below.

As you will see, generating the same N km isochrone with different methods do not give the exact same output.
That is also why we generally do not recommend using the isochrones for point-in-polygon analysis as a way of finding out which customers are less then N km away.

Rather use the various matrix functions for finding distance between multiple points. This also allows you to include the off-road part in the calculations.

This table gives an overview of the key differences between the methods:

	DriveTimeSimple 70	Voronoi 98	IsoLinkDriveTime 73	Alpha shapes 83
Input	1 node	nodes & locations	nodes or 1 location	nodes & locations
TPolyGeneration input		Yes		Yes
Speed of calculations	16 ms	78 ms	217 ms	452 ms
Holes		Yes		
Islands		Yes		Yes
Doughnut mode	Yes	Yes		
Smoothing	Yes	Yes		

Shown on map as:	Blue line	Yellow polygon	Black network	Brown line

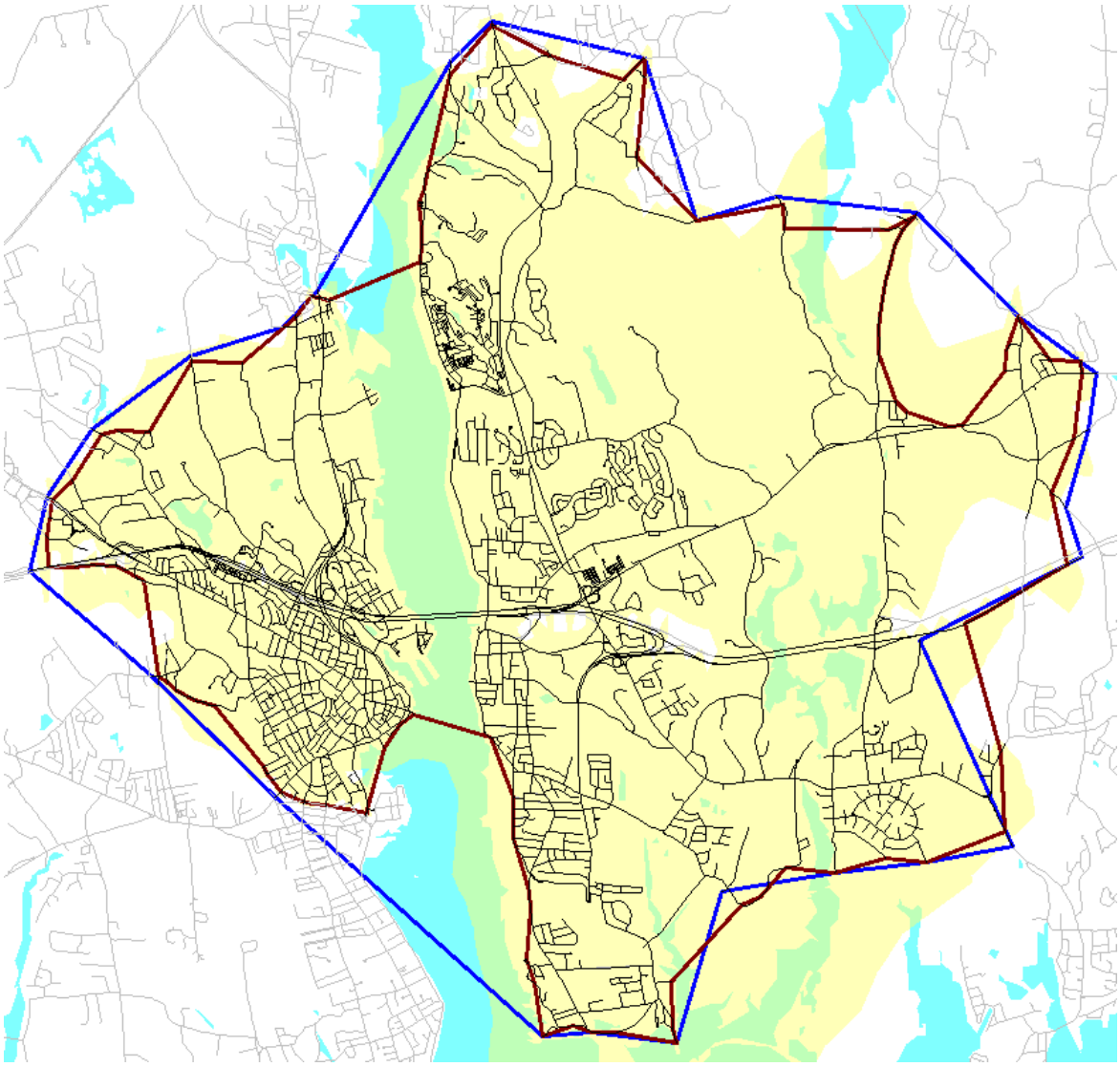
Alpha shapes and DriveTimeSimple both tries to follow the perimeter of the network, which can be reached from the starting point(s).
They do not take into consideration any unreachable parts of the network (the grey lines), so they may get included in the output polygon anyway.

Voronoi on the other hand follows the line between what can be reached (black network) and what can not be reached (grey network).

Alpha shapes can not be calculated in doughnut mode, since 2 polygons may actually be intersecting, due to the way they are calculated.

IsoLinkDriveTime is the most accurate, so for comparison it is included. But it is a much different kind of output.

Timings above are for the 7 km isochrones shown below with addnodes = 0.3 km 76:



1.18 Check List

Sometimes a calculation returns a different result (route) compared to what you expected or you get no route at all. Both situations are due to issues somewhere in the network and these can be hard to locate.

This is a list of things to check:

- Look inside network_report.txt (generated when calling TImport.execute 27):

1. Are you using the latest version of RW Net?
2. Is the coordinate unit detected correctly?
3. Should Z-level information have been applied?
4. Do some of the objects have errors?
5. Is the average object length realistic?

- Check for network errors. See also data sources 14.

- Reduce the setup as much as possible:

1. Call Open 53(false,false,0) (removes one-way restrictions)
2. Set Turnmode = false, when creating TCalc instance
3. Set Hierarchy 89 = false
4. Skip Limits 81
5. Use SetShortest 81
6. Set Alpha 87 = 0
7. Set NoDriveThrough 79 = false

If this solved the problem, enable these again one by one, until it fails. Then you know where to look.

- Create maps in your GIS identifying the problem:

1. Look at basic map for no physical connection (missing bridge / ferry etc)
2. Create a thematic map of one-way directions and closed links
3. Create a thematic map of attribute field
4. Create a thematic map of hierarchy attributes etc.
5. Call TurnExportGIS 59 to view turn restrictions
6. Call FindNonConnected 65 from RW Net Pro
7. Call SubNet 87 from RW Net Pro

If you have multiple points (Matrix or TSP function), it can be tricky to locate which point makes the trouble. One method is to calculate with 2 points first. If that works OK, try with 3 points, then 4 points etc, until the problem pops up. Now the problem is usually somewhere near the last point being added.

1.19 Changes from RW Net 2

New functionality

Full Unicode support
Improved .NET support (Compact Framework and Mono for instance)
Nodes can be "closed" in point-to-point routing
Links can be "closed" in point-to-point routing (No-Drive-Through bit)
Complex turn restrictions
Automatic identification of left/right turns

Travelling salesman optimization with support for curb approach
Travelling salesman optimization with time windows
Output to array format instead of files on disk (known from RW NetServer 3)
Minimum spanning trees

Improved functionality

Better developer experience (more OOP)
Improved flexibility
Much faster spatial searches
Overall calculation speed

Changes in behaviour

Loop links are not allowed by default.
Networks with loop links do not work in route calculations (TCalc, TRouteCalc, TDrivingDirections).
Starting 1-1-2012.
Percentages in locations can now also be exact 0 or 1. No need to use 0.0001 or 0.9999
Side in locations are defined differently
The attribute field is differently defined for a few of the bits ("mode" is gone)
Alpha parameter is by default 1 (enabled)

Changes in setup

File format is different with an INI file introduced (can be replaced with an event if desired). Even though some file names may be the same, the content may be changed
Road names are now stored in Unicode format, making import/export easier
Coordinate system is normally detected automatically during import
Coordinate system don't have to be specified when opening network, since it is stored in the INI file
Routes modes are now shortest, fastest or cheapest - it used to be shortest or fastest/cheapest
Distances, speeds etc. are in km (SI-unit). Miles can only be used in a few output-to-file functions
Password for initialization is now also needed in the Delphi versions.

Functions removed

Isogrid
NWcreateCGF
ResultFile & ResultSave
RouteSave
All functions listed as obsolete in RW Net 2 documentation
COM version

Functions renamed

AirDistNode: DistanceBetweenNodes 43
AirDistPos: DistanceBetweenPoints 43
AttributeCreate(2): ExecuteAttribute 31
AttributeLoad: Open 53
Assignment: TrafficAssignment 91
CloseLink: OneWaySet 52
Coordinate2Location: NearestLocation 66
Coordinate2LocationSimple: NearestLocationSimple 66
Coordinate2Node: NearestNode 67
CulDeSac: CulDeSac 43 & SubNetEx 91 & Bridges 89
District: District 119
ExternidImport: ExecuteExternalidInt 31 / ExecuteExternalidString 32
ExternIDfindID: Link2ExternalID 50

ExternIDfindIndex: ExternalID2Link 46
FindCloseNodes: FindNonConnected 65
GetLinkDist: LinkLength 49
GetOpenStatus: OneWayGet 52
IsoLink2: IsoLinkDriveTime 73
IsoLink2Dyn: IsoLinkDriveTimeDyn 74
IsoLink4: IsoLinkServiceArea 74
IsoPoly2: TVoronoi 98 with mode = vmIsoChrone
IsoPoly3: TVoronoi 98 with mode = vmSimple
IsoPoly4: TVoronoi 98 with mode = vmServiceArea
LimitCreate: ExecuteLimit 32
LimitLoad: OpenLimit 53
LimitLoad_bitpattern: OpenLimit 53
Linkmax: Linkcount 49
NetworkCenter: CenterNode 84 / Cluster2 117 / Cluster3 118
NetworkLength: Length 49
NodeCreate: ExportNodes 45
NodeLinkCheck: FindNonConnected 65
Nodemax: Nodecount 51
NodeX, NodeY: Node2Coordinate 51
NWcreate: TImport.execute 27, ExportNodes 45, ObjectCheck 52
NWload: Open 53
NW3Dnodes: ExternalNodeId2ZLevels 46
Overpasses: FindOverPasses 65
RoundAbout: AttributeGetBit 39(link,11)
RouteList: TDrivingDirections 92
UnusedLinks: UnusedLinks 87
Valency: Degree 43

Functionality not added yet

CPP / MRPP & EulerRoute
FindRoundAbout
HierarchyCheck1
HierarchyCheck2

1.20 License Terms

Platforms

A license gives access to available versions as listed in the schema below. Within the first year licensor has access to updates and new versions.

	Standard	Pro
Delphi XE		Yes
Delphi XE2 - XE8, 32 bit	Yes	Yes
Delphi XE2 - XE8, 64 bit		Yes
.NET	Yes	Yes
.NET Silverlight		Yes
DLL for MapInfo / MapBasic		Yes

Support

A license gives access to support as listed below for the first year:

	Standard	Pro / Standard site

E-mail support	ASAP after 24 hours	ASAP
Telephone / Skype support		Yes

E-mail support includes answering questions, which do not involve writing source code of >10 lines. Only persons with a license can receive support.

After first year, support and maintenance can be extended for one more year at a time.

Deployment / distribution

This table lists how deployment of applications is allowed for different versions of RW Net:

	Standard	Pro
Deployment of desktop applications	Only within own organization	Allowed
Deployment of server application	See website for price	See website for price
Deployment of TImport 25 functionality	See website for price	See website for price

It is a server application if:
- The application has an API, which makes the routing functionality accessible from other applications or
- The routing functionality is available from other computers through a network (web service, REST, cgi etc.)

General terms

Licensor is allowed to use RW Net for as long as he/she doesn't violate this license.

Licensor is not allowed to:

- Distribute applications outside it's own organization, which competes directly with RouteWare's own applications: RouteFinder, RW NetServer and FleetEngine.
- Wrap up RW Net in component-like structures and distribute it.

If licensor holds a personal license, he/she can either:
1) Have only 1 named person using RW Net on as many computers as he/she like or
2) Install it on 1 physical computer (doesn't include terminal services, citrix and similar setups) and let several persons use it from there (support is still only given to 1 person)

If licensor holds a site license, it allows an unlimited number of persons at licensors site to use RW Net at the same time. Ask RouteWare for enterprise-wide licenses.

Licensors of RW Net are issued a personal password 11 to activate the software. This password must not be readable to end-users of deployed applications. It is the responsibility of licensor to ensure that this is taken care of.

The usual legal stuff

All copyrights belong to RouteWare (Uffe Kousgaard).
Disassemble or reverse engineering of RW Net binaries are not allowed.
Licensor is not allowed to install RW Net on a network drive or shared drive except for backup purposes.
Licensor is not allowed to sell or in any other way hand over the right to use the software to any other party.
RouteWare is not responsible for any problems, direct or indirect, which RW Net may cause - no matter what the reason may be.
Any problem / error will be corrected as fast as possible within normal business hours. If

RouteWare is not able to correct problems, which to a severe degree affect the functionality of the software, a refund is made, which matches the degree to which the software doesn't function properly. This refund is based on what the licensor has paid within the last 12 months and cannot exceed this amount.

Updated Jan 2014

Part II

Main Classes

2 Main Classes

2.1 TImport

These classes (TImport & TCustomImport) are used for importing the main geography part of a network.

Using this class in applications that you deploy to other users, requires an additional runtime license. See license terms 20.

TImport
Add 25
AddFiles 26 (only available in RW Net Pro)
Execute 27
ZfromField & ZtoField 29 *

TCustomImport (only available in RW Net Pro)
Execute 27
OnImportLink 28 *

Shared properties / methods:
AllowLoops 26 *
CoordinateUnit 26 *
CoordSys 26 *
Directory 27 *
EPSG 27 *
Encryption 27 * (only available in RW Net Pro)
MaxNodesPerCell 28 *
PRJ 29 *
SkipSpatialIndex 29 *

CreateReport 27
ImportErrorList 27
LinkCount 28
MaxDegree 28
MBR 28
NodeCount 28
StartTime 29
StopTime 29
TotalLength 29

* = Properties that may be set before calling execute 27. Default values are sufficient in most cases.

2.1.1 Add

Call this method to add a single file to the list of files for processing by Execute 27 method.

Adding a TAB file requires that the corresponding MAP and ID files are also present.
Adding a SHP file requires that the corresponding SHX file is also present.
Adding a MIF file requires no further files.

You can not mix different file types.

Extended TAB files from MapInfo 15.2 (64-bit) and onwards ("NATIVEX") are supported.

Syntax: Add(filename: string)

In RW Net Standard you can only add 1 file in total. If you add another one, the previous one gets removed from the list.

2.1.2 AddFiles

This method can be used to add 25 multiple files at a time.

Syntax: AddFiles(files: TStringList 137)

Only available in RW Net Pro.

2.1.3 AllowLoops

This property can be used to define if loop links are allowed in datasets. A loop link is one, where the first and last vertex is the same.

If AllowLoops is false (default), loop links will be reported 27 during the import process.

If AllowLoops is true, the existence of any loop links will prevent the use of TRoute 148, TRouteCalc 87 and TDrivingDirections 92.

We recommend that loop links are split into 2 links in advance.

Databases without loop links: TomTom, Navteq, OSM, ITN and Meridian 2.
Databases with loop links: NVDB and DAV.

Type: Boolean

2.1.4 CoordinateUnit

Coordinate unit will automatically be detected for MIF and TAB files.
For SHP files it will happen automatically if a PRJ file exists.

For other situations, it should be set by the user or an error will be raised during import.

Default: cuUnknown

Type: TCoordinateUnit 142

2.1.5 CoordSys

This string property is a MapInfo coordinate clause.

If you import from a MIF or TAB file, it property will automatically be set during execution.

You can set it manually, if you plan to export to MIF or TAB using TGISwrite 104 or one of the other functions writing to GIS files.
Otherwise it will be set to a default value (Non-Earth coordsys) that matches CoordinateUnit 26 and the coordinates in the file.

Type: String

2.1.6 CreateReport

Call this method after calling execute to generate a report on the import process. This is similar to the report from RW Net 2.

Filename is always network_report.txt and it is stored in the same folder as the other bin files.

Syntax: CreateReport

2.1.7 Directory

This property points to where the output files are stored.

Default: Current directory.

Type: String

2.1.8 EncryptionKey

You can set this property if the imported files should be encrypted to prevent other users from using the files. Encrypting makes it harder, but can't fully prevent the very determined and skilled user from getting to your data.

Default value is 0 (no encryption). You can only set it in RW Net Pro.

Type: Int64

2.1.9 EPSG

The EPSG code should be set before importing, if you plan to export to GML files later on.

Default value is 4326 (Lat/Long, WGS84).

Type: Integer

2.1.10 Execute

Call this method when you have defined all input parameters. This is what does the main job.

Syntax: Execute

2.1.11 FailOnDifferentCoordSys

This property is used for controlling import of multiple TAB or MIF files.

If true (default), it will stop when different CoordSys' are encountered.

Type: Boolean

2.1.12 ImportErrorList

This read-only property keeps a list of problematic links in the input data source, found during import process. Content of the list is also written to the report 27.

Type: TImportErrorList 133

2.1.13 LinkCount

This read-only property returns the total number of links after calling execute 27.

Type: Integer

2.1.14 MaxDegree

This read-only property returns the maximum degree of the network after calling execute 27.

Type: Integer

2.1.15 MaxNodesPerCell

This property can be used to define how detailed the spatial index should be. Set the value before importing.

A higher value decreases the size of of the file and memory foot print, but reduces speed of spatial searches.

Default value is 50. Minimum value is 25.

Type: Integer

2.1.16 MBR

This read-only property reports the minimum bounding rectangle (MBR) after import

Type: TFloatRect 144

2.1.17 NodeCount

This read-only property returns the total number of nodes generated after calling execute 27.

Type: Integer

2.1.18 OnImportLink

In class TCustomImport assign this event to read custom data:

Parameters:
1) Link is automatically increased by one every time.
2) Vertices: This is a list of vertices (coordinates).
3) VertexCount: Indicates the number of vertices on the list. The list may be longer than actual number of elements.
4) Zfrom, Zto: Z-levels 16 for the link.
5) LastLink: Set this to true, when you have reached the last link to be read.

All links will be traversed twice. Depending upon your source of data, it may be faster to extract to a MIF file first.

Syntax: OnImportLink(link: integer; var vertices: TFloatPointArray 144; var vertexcount: TVertexCount 149; var Zfrom, Zto: integer; var LastLink: boolean)

2.1.19 PRJ

This string gets updated during the import process, if a PRJ file exists along with a SHP file and no coordinate unit has been specified.

Type: String

2.1.20 SkipSpatialIndex

This property allows you to skip creation of the spatial index during import to save time and disk space.

Spatial index is needed by TSpatialSearch 64.

The spatial index can not be created later on, unless you re-import the data.

Default: false

Type: Boolean

2.1.21 Starttime

This read-only property reports when importing started.

Type: TDatetime

2.1.22 Stoptime

This read-only property reports when importing stopped.

Type: TDatetime

2.1.23 TotalLength

This read-only property returns the total length for all links after calling execute 27.

Type: Double

2.1.24 ZfromField & ZtoField

These 2 properties are used for describing Z-level 16 in input data.

The 2 properties refer to the field ID in the same way as it is being done in class TImportAttributes 30: First field has index 0.

For TAB files, the execute command will automatically look for .DAT files.
For MIF files, the execute command will automatically look for .MID files.
For SHP files, the execute command will automatically look for .DBF files.

Set both values >=0 to apply.

Default: -1

Type: Integer

2.2 TImportAttributes

This class is for importing attribute information for the links in the network.

	File-based	Event-based
Attributes	ExecuteAttribute 31	ExecuteAttributeEvent 31
External ID's (integers)	ExecuteExternalidInt 31	ExecuteExternalidIntEvent 32
External ID's (strings)	ExecuteExternalidString 32	ExecuteExternalidStringEvent 32
Limits	ExecuteLimit 32	ExecuteLimitEvent 32
Road names	ExecuteRoadname 33	ExecuteRoadnameEvent 33

Before calling any of the file-based methods above, call function Add 30 to add a list of files to import from. Supported file formats include DBF, DAT, MIF and CSV.

If you rather want to import using events, use one of these procedures mentioned in the column with event-based methods.

2.2.1 Add

Call this method to add a single file to the list of files for processing by one of the execute methods.

You can add DBF, DAT, MID and CSV files, but not mix different file types.

If you use CSV files, remember to set CodePageCSV 30. Codepage for other file types gets auto-detected: DAT files from their TAB counterpart and MID files from their MIF counterpart. DBF gets detected from the internal header.

Syntax: Add(filename: string)

In RW Net Standard you can only add 1 file in total. If you add another one, the previous one gets removed from the list.

2.2.2 AddFiles

This method can be used to add 30 multiple files at a time.

Syntax: AddFiles(files: TStringList 137)

Only available in RW Net Pro.

2.2.3 CodepageCSV

When reading from CSV files, set this property to the codepage used.

Default: Codepage of the local system.

Type: TCodePage 141

2.2.4 CodepageDBF

When reading from DBF/DAT files, set this property to the codepage used. Normally leaving it to 0 is sufficient, but DBF files from OpenStreetMap uses UTF-8, which isn't supported natively by DBF format. In that case set it to 65001.

If set to 0, it uses the codepage byte inside the file header. If that byte is 0 too, it uses the codepage of the local system.

Default: 0.

Type: TCodePage 141

2.2.5 Directory

This property points to where the output files are stored.

Default: Current directory.

Type: String

2.2.6 EncryptionKey

You can set this property if the imported files should be encrypted to prevent other users from using the files. Encrypting makes it harder, but can't fully prevent the very determined and skilled user from getting to your data.

Default value is 0 (no encryption).

Type: Int64

Only available in RW Net Pro.

2.2.7 ExecuteAttribute

Call this procedure to import attribute information from one or more files.

Specify fieldindex (0-based) or fieldname for fk = fkDBF. If fieldname is specified, it takes precedence.

Syntax: ExecuteAttribute(fieldindex: integer; fieldname: string; fk: TFileKind 143)

2.2.8 ExecuteAttributeEvent

Assign event OnReadAttribute for importing attributes. When the last record has been reached, set lastrecord = true.

TAttributeReadEvent = procedure(Sender: TObject; link: integer; var attribute: word; var lastrecord: boolean);

Syntax: ExecuteAttributeEvent

2.2.9 ExecuteExternalIDInt

Call this procedure to import external ID information from one or more files, when the field is an integer field.

If the value read doesn't fit into an integer 141, specify useint64 = true and an int64 141 will be used instead.

Int64 formatted files can only be opened with RW Net 4.18 or newer.

Specify fieldindex (0-based) or fieldname for fk = fkDBF. If fieldname is specified, it takes precedence.

Syntax: ExecuteExternalidInt(fieldindex: integer; fieldname: string; fk: TFileKind 143; useint64:

boolean)

2.2.10 ExecuteExternalIDIntEvent

Assign event OnReadExternalIDInt for importing external ID's that are integer based. When the last record has been reached, set lastrecord = true.

Specify useint64 = true and int64 141 will be used for storage instead of integers 141, allowing much bigger numbers.

TExternalIDReadIntEvent = procedure(Sender: TObject; link: integer; var externalID: int64; var lastrecord: boolean)

Syntax: ExecuteExternalIDIntEvent(useint64: boolean)

2.2.11 ExecuteExternalIDString

Call this procedure to import external ID information from one or more files, when the field is a string field.

Specify fieldindex (0-based) or fieldname for fk = fkDBF. If fieldname is specified, it takes precedence.

Syntax: ExecuteExternalidString(fieldindex: integer; fieldname: string; fk: TFileKind 143)

2.2.12 ExecuteExternalIDStringEvent

Assign event OnReadExternalIDString for importing external ID's that doesn't fit in an integer. When the last record has been reached, set lastrecord = true.

TExternalIDReadStringEvent = procedure(Sender: TObject; link: integer; var exteralnID: string; var lastrecord: boolean)

Specify the maximum width of the strings to be read.

Syntax: ExecuteExternalidStringEvent(width: integer)

2.2.13 ExecuteLimit

Call this procedure to import limit information from one or more files.

Specify fieldindex (0-based) or fieldname for fk = fkDBF. If fieldname is specified, it takes precedence.

Syntax: ExecuteLimit(fieldindex: integer; fieldname: string; fk: TFileKind 143)

See also LimitFileIndex 33

2.2.14 ExecuteLimitEvent

Assign event OnReadLimit for importing limits. When the last record has been reached, set lastrecord = true.

TLimitReadEvent = procedure(Sender: TObject; link: integer; var Limit: byte; var lastrecord: boolean)

Syntax: ExecuteLimitEvent

2.2.15 ExecuteRoadname

Call this procedure to import road names from one or more files.

Specify fieldindex (0-based) or fieldname. If fieldname is specified, it takes precedence.
If reading from MID file, it will automatically lookup the codepage from the MIF file.

Syntax: ExecuteRoadname(fieldindex: integer; fieldname: string; fk: TFileKind 143)

2.2.16 ExecuteRoadnameEvent

Assign event OnReadRoadname for importing road names. When the last record has been reached, set lastrecord = true.

TRoadNameReadEvent = procedure(Sender: TObject; link: integer; var name: string; var lastrecord: boolean)

All links will be traversed twice. Depending upon your source of data, it may be faster to extract to a CSV file first.

Syntax: ExecuteRoadnameEvent

2.2.17 LimitFileIndex

When calling ExecuteLimit 32 or ExecuteLimitEvent 32 this property is used in the naming of the output file.

Allowed interval is 1 to 9.

Default: 1

Type: integer

2.2.18 RoadNameFileIndex

When calling ExecuteRoadname 33 or ExecuteRoadnameEvent 33 this property is used in the naming of the output file.

Allowed interval is 1 to 99.

Default: 1

Type: integer

2.3 TImportSQL

This class can be used to import directly from a GIS enabled database.
MS SQL Server, IBM DB2, Oracle and PostGIS all offers storage of gis data, directly inside the database.

We have implemented it for MSSQL and only for 32/64 bit DLL and Delphi XE7 - XE8 - 10 platforms.

You can reuse the majority of the shared properties from TImport 25 with this class (CreateReport 27, NodeCount 28 etc.).

This class is available with Pro only.

To make it work in XE7, you have to make an edit of the XE7 source code:

```
1. Open file \source\data\firedac\FireDAC.Phys.ODBCBase.pas
2. Locate TFDPhysODBCCommand.SQL2FDColInfo method
3. Find there case with SQL_BLOB (app. line 1467)
4. Add SQL_SS_UDT to this case
```

2.3.1 ExecuteMSSQL

This is the main method and is a single call to do all the processing.

The first six parameters always need to be set.
For the rest at least one need to be set. This way you can create just the attribute.bin file or similar if you have the rest in advance.

If geography is true, the field with the geography is automatically detected. For the remaining the fieldname need to be set.

Example set of parameters for a mapinfo table "roads" that has been uploaded to a local MS SQL Server Express database.

Parameter	Value
Server	"127.0.0.1\SQLEXPRESS"
Database	"GIS1"
Username	"GIS_user1"
Password	"secret_code"
Schema	"mapinfo"
Tablename	"roads"
Geography	true
Attribute	"attribute4"
Roadname	"streetname"
Externalid	"ID"
Limit	""
ZFrom	"ZFromLevel"
ZTo	"ZToLevel"

Syntax: ExecuteMSSQL(Server, Database, Username, Password, Schema, TableName: string; geography: boolean; attribute, roadname, externalid, limit, ZFrom, ZTo: string);

2.3.2 LimitFileIndex

When calling ExecuteMSSQL 34 this property is used in the naming of the output file.

Allowed interval is 1 to 9.

Default: 1

Type: integer

2.3.3 RoadNameFileIndex

When calling ExecuteMSSQL 34 this property is used in the naming of the output file.

Allowed interval is 1 to 99.

Default: 1

Type: integer

2.4 TNetwork

This is the main class that holds all the information about the street network, while the other classes (TSpatialSearch 64, TRouteCalc 87, TCalc 68) link to this, when doing calculations. Whatever you define here, is shared by all the other classes linking to it.

Besides holding the core network (geometry, topology, spatial index), it also allows you to work with other types of information: Attributes, time / speed, cost, road names, turn restrictions, limits etc.

This is a list of available methods / grouped by area: (*) = Pro only.

Basic opening & closing of the network

Directory 43
EncryptionKey 44
LinkLimit 49
Open 53
Close 41

Geometry & topology

These are generally fairly simple lookup functions returning information requiring little processing.

CoordinateUnit 42
CoordinateWindow 42
CulDeSac 43
Degree 43
ExtractSection 47
GetGISSection 48
Length 49
Link2FromNode 49
Link2ToNode 49
LinkCount 49
LinkLength 49
Location2Coordinate 50
Location2CoordinateList 50
LoopLink 50
LoopLinks 50
MaxDegree 51
MBR 51
Node2Coordinate 51
Node2Link 63 (*)
NodeCount 51
SwapList 57

Attributes

See this introductory chapter on attributes 6

AttributeGet 39
AttributeGetBit 39
AttributeSave 39
AttributeSet 39
AttributeSetBit 39
AttributeSetBits 39
AttributeSetSkipInSearchBit 39
Hierarchy 49
OneWayGet 52
OneWaySet 52
NoDriveThroughInit 52
NoDriveThroughSet 52
RoadClass 55
SwapOneWay 57

Time

Time is defined as minutes and is the criteria for routing in fastest 81 mode

CreateArrayTime 42
CalculateTime 40
ReadTime 55
GetTime 48
SetTime 56

Speed

Internally speed is always stored as time for each link, so if you change one, you also change the other.

ReadSpeed 54
GetSpeed 48
SetSpeed 56

Cost

Use cost, when you want a more flexible routing criteria than just time or distance.

CreateArrayCost 42
CalculateCost 40
ReadCost 54
GetCost 48
SetCost 56

Turn restrictions

See this introductory chapter on turn restrictions 9.

CreateArrayTurn 42

TurnAutoProcess 57
TurnImportBin 59
TurnImportTxt 60

TurnRestriction 61

Road Names

These are mostly used when creating driving directions.

External ID

See this introductory chapter on external ID 9.

Limits

See this introductory chapter on limits 9.

TRoute methods

Methods operating on a TRoute 148 instance (output from route calculation).

Check functions

These are functions for verifying input. They are used internally by most of the other methods, so you generally don't need to call them on your own.

CheckTurnIndex 41

Export

Methods for exporting data to a GIS file, so you can view the actual data.

ExportLinks 44
ExportLinksFullSplit 45
ExportLocationList 45
ExportNodeList 45
ExportNodes 45
ExportPolyGeneration 45
ExportTrafficList 62 (*)
TurnExportGIS 59

Advanced methods

These are more complex methods doing various sorts of calculations / analysis.

Direction 43
DistanceBetweenNodes 43
DistanceBetweenPoints 43
DistanceToLink 43
DistanceToLinkSimple 44
DistanceToNode 44
DownStream 62 (*)
ExternalNodeId2ZLevels 46
Join 62 (*)
Matrix 51
MatrixDyn 51
ObjectCheck 52
ParallelLinks 53
Select 55
Split 56
Trace 63 (*)
UpdateAlphas 61
UpStream 64 (*)

GIS output

These 5 properties all define various settings, used when generating GIS output. See TGISwrite 104 class.
They are automatically populated when the network is opened.

Codepage 42
CompactMIF 42
CoordSys 42
EPSG 44
PRJ 54

When calling GISoutputInit 48 it also inherits these values.

GISarray 48 is for storing output, when you have chosen array as output format.

Random places

When you just need some random input data for testing.

RandomPoint 54
RandomNode 54
RandomLocation 54

2.4.1 AttributeGet

This function returns the attribute 6 for the link.

Syntax: AttributeGet(link: integer): word

2.4.2 AttributeGetBit

This function returns the attribute 6 value for a single bit of the link.

Syntax: AttributeGetBit(link: integer; bit: byte): boolean

Example: AttributeGetBit(link,11) returns true if link 11 has been marked as a round-about link.

2.4.3 AttributeSave

If you have made changes to the attributes, you can save the whole content as a new attribute.bin file. It will use the folder as specified by Directory 43 and overwrite an existing file.

Syntax: AttributeSave

2.4.4 AttributeSet

This method will change the whole attribute 6 of a link.

Syntax: AttributeSet(link: integer; value: word)

2.4.5 AttributeSetBit

This method will change an attribute 6 bit of a single link. Bit is a value from 0 to 15.

Syntax: AttributeSetBit(link: integer; bit: byte; value: boolean)

2.4.6 AttributeSetBits

This method will change an attribute 6 bit for all links, according to BA. Bit is a value from 0 to 15.

BA should have one more elements than LinkCount 49, since Links are 1-based and TBitArray 134 is 0-based.

Syntax: AttributeSetBits(bit: byte; BA: TBitArray 134)

2.4.7 AttributeSetSkipInSearchBit

This method will set the SkipInSearch bit for all closed links (both oneway bits set). For open links, the bit is cleared.

Syntax: AttributeSetSkipInSearchBit

2.4.8 CalculateCost

This will update a Cost array index as a linear combination of length and time:

Cost(,costindex) = weightlength * Length(*) + weighttime * Time(*, timeindex)*

Syntax: CalculateCost(costindex: integer; weightlength, weighttime: TCost; timeindex: integer)

Call CreateArrayCost 42 in advance to allocate the index.

Timeindex points to one of the arrays defined through CreateArrayTime 42. If weighttime = 0, value of timeindex is ignored.

If both weights are <> 0, we recommend setting weighttime = 1. This ensures the values can be interpreted as time easily and be used together with turn delays if needed.

2.4.9 CalculateTime

This will calculate time on all links, by looking up the speed in the RCS array, based upon the road class attribute 6:

Time(,index) = Length(*) / RCS(attribute(*)) * 60*

Call CreateArrayTime 42 in advance to allocate the index.

See also TCalc.MaxSpeed 78.

Syntax: CalculateTime(index: integer; RCS: TRoadClassSpeed 137)

2.4.10 CheckCoordinate

This will check if a coordinate is valid.

If using degrees / radians / grads there are natural limits for valid values (-180 to 180, -90 to 90 etc).

For all coordinate units the coordinatewindow 42 is used for checking that P is within a certain bounding box of the street network.
By setting CoordinateWindow < 0, this part of check is disabled.

Syntax: CheckCoordinate(P: TFloatPoint 143)

2.4.11 CheckExternalOpen

This method checks if the external ID 9 has been opened through Open 53.

Syntax: CheckExternalOpen

2.4.12 CheckLink

This will check if a link number is valid, i.e between 1 and LinkCount 49. At the same time LinkLength 49 has be <> 0.

Syntax: CheckLink(link: Integer)

2.4.13 CheckLocation

This will check if a location number is valid, i.e link is between 1 and LinkCount 49 and percent is between 0 and 1.

Syntax: CheckLocation(loc: TLocation 146)

2.4.14 CheckLocationList

Checks 41 all elements in LL:

Syntax: CheckLocationList(LL: TLocationList 134)

2.4.15 CheckNode

This will check if a node number is valid, i.e between 1 and NodeCount 51.

Syntax: CheckNode(node: Integer)

2.4.16 CheckNodeList

Checks 41 all elements in NL:

Syntax: CheckNodeList(NL: TIntegerList 133)

2.4.17 CheckOpen

This method checks if the network has been opened through Open 53.

Syntax: CheckOpen

2.4.18 CheckTurnIndex

This checks if index is valid for referencing sets of Turn restrictions. See CreateArrayTurn 42.

Syntax: CheckTurnIndex(index: integer)

2.4.19 Close

This method closes the network and releases all memory related to it. This includes arrays setup using CreateArray* functions, all roadname files etc.

Syntax: Close

2.4.20 CloseRoadNameFile

This method closes a single roadname file and releases the memory related to it.

Syntax: CloseRoadNameFile(FileNumber: integer)

2.4.21 Codepage

Property Codepage: TCodePage 141

2.4.22 CompactMIF

Property CompactMIF 107: boolean

2.4.23 CoordinateUnit

Read-only property. Is set when calling Open 53.

Type: TCoordinateUnit 142

2.4.24 CoordinateWindow

This property controls checking of coordinates 40 when entered into functions that accept coordinates.

It is used in checking if coordinates are within the Minimum Bounding Rectangle 51 + X % of the street network. This will prevent situations where you by mistake swap x and y coordinate or use lat/long coordinates when the street network was in a projected coordinate system or vice versa.

Use a negative number to skip checking.

An example: If the coordinate should be between 0 and 50 and CoordinateWindow = 20 (default), then only coordinates between -10 and 60 will be accepted.

Type: double

2.4.25 CoordSys

This property is set when calling Open 53.

Property CoordSys: string

2.4.26 CreateArrayCost

Call this method to allocate room for n cost arrays. Cost arrays can be used, when you want to a route that isn't shortest or fastest, but rather some other expression.

Syntax: CreateArrayCost(n: integer)

2.4.27 CreateArrayTime

Call this method to allocate room for n time arrays. Time arrays are primarily used for fastest path routing. Multiple arrays can be setup for different uses (vehicle types, time of day etc).

Syntax: CreateArrayTime(n: integer)

2.4.28 CreateArrayTurn

Call this method to allocate room for n turn arrays. Each array is a list of turn restrictions / turn delays.

When calling Open 53, this is automatically initialized for 1 array, so normally it isn't needed to call at all.

Syntax: CreateArrayTurn(n: integer)

2.4.29 CulDeSac

This read-only property returns true if a link is part of a Cul-De-Sac / dead end link. See also NonCulDeSacNodes 52.

Syntax: CulDeSac[Index: Integer]: boolean

2.4.30 Degree

This method returns the degree of a node. See network terminology 5 for details.

To iterate through the links connected to a node, use function Node2Link 63.

Syntax: Degree(node: integer): integer

2.4.31 Direction

Returns the turning angle (0-359) at node2 when moving from link1 to link2 via node2. This is based on the exact coordinates of the polylines and the node.

Link1 and link2 must both be connected to node2. Specifying node2 may seem superfluous, but is required since link1 and link2 could be parallel links.

Straight on is 0, to the left is 90, backward is 180 and to the right is 270.

Syntax: Direction(link1,node,link2: integer): integer

2.4.32 Directory

This property defines the location of all binary files used by RW Net. Default directory is the current path.

Type: string

2.4.33 DistanceBetweenNodes

Calculates the as-the-crow-flies distance between two nodes.

Syntax: DistanceBetweenNodes(node1,node2: integer): double

2.4.34 DistanceBetweenPoints

Calculates the as-the-crow-flies distance between P1 and P2.

Syntax: DistanceBetweenPoints(P1,P2: TFloatPoint 143): double

2.4.35 DistanceToLink

This method calculates the distance from P to link.

It returns this information:

- Percentage along the link (0 .. 1)

- Side of the link (-1: Left or 1: Right)
- Distance
- Coordinates of location on link

Syntax: DistanceToLink(P: TFloatPoint 143; link: integer; out percent: double; out side: integer; out distance: double; out Pnew: TFloatPoint 143)

See also DistanceToLinkSimple 44 and NearestLocation 66.

2.4.36 DistanceToLinkSimple

This method calculates the distance from P to link.

Syntax: DistanceToLinkSimple(P: TFloatPoint 143; link: integer): double

See also DistanceToLink 43 and NearestLocationSimple 66.

2.4.37 DistanceToNode

This method calculates the distance from P to a node.

Syntax: DistanceToNode(P: TFloatPoint 143; node: integer): double

2.4.38 EncryptionKey

Set this property before calling Open 53, if your data are encrypted.

Type: int64

2.4.39 EPSG

This property is set when calling Open 53.

property EPSG: integer

2.4.40 ExportLinks

This method will export the currently open network, including external ID, limit and roadname information where available.

LL can be used if you want to split some of the links. Typically setup LL using FindOverPasses 65 or SplitAndSnap 68.
If you prepare LL on your own, remember to call these 2 methods after filling in the list: RemoveDuplicates and RemoveStartEndPos.

BA can be used to specify a selection - a subset of the links.

LL and BA can both be nil.

Syntax: ExportLinks(filename: string; GF: TGISformat 144; LL: TLocationList 134; BA: TBitArray 134)

2.4.41 ExportLinksFullSplit

This method will export the currently open network, including external ID, limit and roadname information where available.

All links are split into short sections, between vertices. See map here: network terminology 5 (blue dots). The method is good for preparing OpenStreetMap data for use in RW Net. Use method Join 62 with topology=2 on the exported dataset and possibly also FindOverPasses 65 function.

BA can be used to specify a selection - a subset of the links. BA can be nil.

Syntax: ExportLinksFullSplit(filename: string; GF: TGISformat 144; BA: TBitArray 134)

2.4.42 ExportLocationList

This method will export LL, so it can be viewed externally. As a minimum the coordinate part of the items need to be filled in.
Use Location2CoordinateList 50 for this, if only the location is filled in.

Syntax: ExportLocationList(filename: string; GF: TGISformat 144; LL: TLocationList 134)

2.4.43 ExportNodeList

This method will export NL, so it can be viewed externally.

Syntax: ExportNodeList(filename: string; GF: TGISformat 144; NL: TIntegerList 133)

2.4.44 ExportNodes

This method will export the nodes of the currently open network.

Syntax: ExportNodes(filename: string; GF: TGISformat 144)

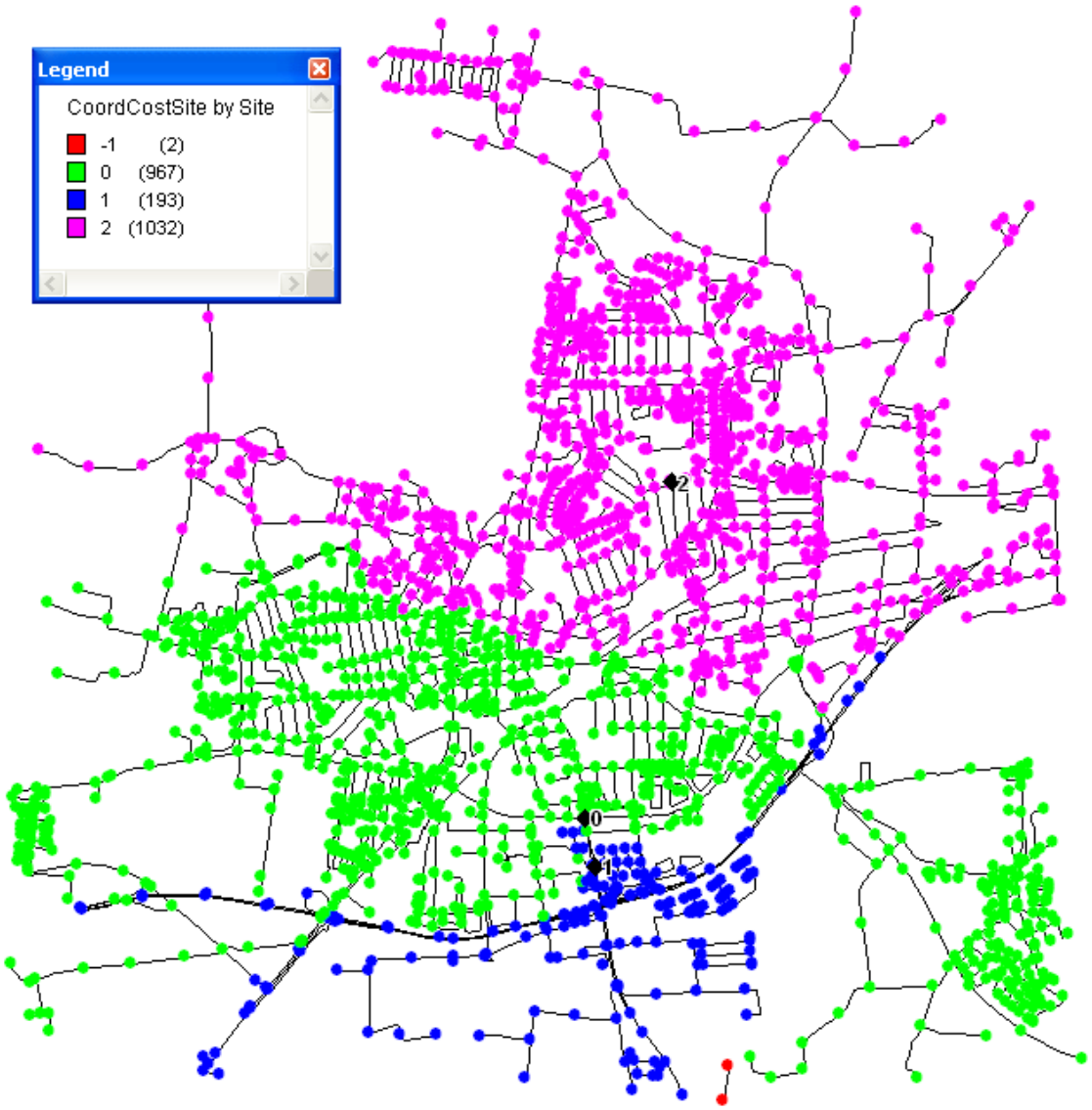
2.4.45 ExportPolyGeneration

This function is mostly for debugging purposes. It allows you to export the content of PG to 2 GIS files, so both facilities (startpoints) and the main data (CoordCostSiteList) are shown. Files called StartPoints and CoordCostSite are generated as specified in the same folder as the main network files.

Syntax: ExportPolyGeneration(filename: string; GF: TGISformat 144; PG: TPolyGeneration 136)

Example: ExportPolyGeneration("test", gfSHP, PG) will generate files test_startpoints.shp and test_coordcostsite.shp.

Example with 3 start points and thematic map of nearest facility:



2.4.46 ExternalID2Link

This method translates an external ID into the internal ID.

Syntax: ExternalID2Link(id: string): integer

2.4.47 ExternalNodeId2ZLevels

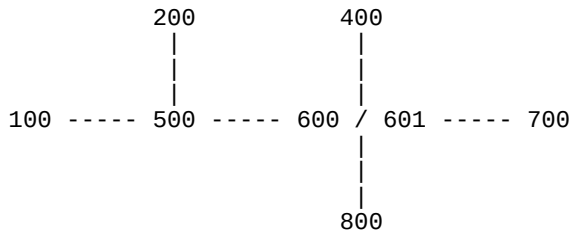
In most datasets Z-levels are used to describe when streets intersect at different levels such as with bridges.

However, some use external node numbers instead to indicate that a shared coordinate belongs at different levels.

In this example we have 6 links with these fromnode and tonode values:
1: 100 - 500
2: 200 - 500
3: 400 - 601
4: 500 - 600

5: 600 - 700
6: 601 - 800

As can be seen on the simplified map below, node 600 and 601 is really the same coordinate, but since the external node is different, they have different Z-levels.



This method helps you translating from external node numbers to Z-levels for the links.

After you have imported and opened a network, create a text file like this:
100,500
200,500
400,601
500,600
600,700
601,800

After calling the function you will get an output file with pairwise Z-levels like this:
0,0
0,0
0,1
0,2
2,0
1,0

Once applied to your dataset, you can import it again, this time declaring Z 29 during import.

Syntax: ExternalNodeId2ZLevels(InputFile, OutputFile: string; GF: TGISFormat 144);

2.4.48 ExtractSection

This method can be used to extract a part of a whole link. Start calling GetGISSection 48

Start and stop should be from 0 to 1 and if stop<start the order of the coordinates is swapped.

Example: ExtractSection(list,1,0) will return the whole list in reverse order.

Syntax: ExtractSection(list: TFloatPointArrayEx 144; start,stop: TPercent 148): TFloatPointArrayEx 144

2.4.49 GeoJSON

When using gfGeoJSON 144 as output format in functions like ExportNodes 45 etc, this read-only property holds the output.

Property GeoJSON: string

2.4.50 GetCost

This method returns cost for a single link and array index.

Syntax: GetCost(index,link: integer): TCost 143

See also CreateArrayCost 42, CalculateCost 40 and SetCost 56

2.4.51 GetGISSection

This method returns a list of coordinates for a link in the network.

Syntax: GetGISSection(link: integer): TFloatPointArrayEx 144

2.4.52 GetGISSectionRoute

This method returns a list of coordinates for a whole route.

Syntax: GetGISSectionRoute(route: TRoute 148): TFloatPointArrayEx 144

2.4.53 GetLimit

Returns limit (0-255) for the specified limit and link. You need to have called OpenLimit 53 in advance to setup the memory.

Syntax: GetLimit(LimitID,link: integer): byte

2.4.54 GetSpeed

Returns speed for the specified array index and link.

Syntax: GetSpeed(index,link: integer): TCost 143

2.4.55 GetTime

Returns time (minutes) for the specified array index and link.

Syntax: GetTime(index,link: integer): TCost 143

2.4.56 GISarray

When using gfArray 144 as output format in functions like ExportNodes 45 etc, this read-only property holds the output.

Property GISarray: TGISarray 110

2.4.57 GISoutputInit

This function can be used to create a TGISwrite object.

Please see Feature Matrix 3 for supported formats in your version.

Syntax: GISoutputInit(filename: string; GF: TGISformat 144): TGISwrite 104

2.4.58 Hierarchy

This array property can be used to get or set the hierarchy of a link. A hierarchy is a value from 1 to 5. See attributes 6.

If the hierarchy hasn't been set, it returns 0.

Property Hierarchy[Index: Integer]: integer

2.4.59 Length

This returns the length of the complete network.

Syntax: Length: TCost 143

2.4.60 Link2FromNode

Returns the number of the node at the start of the link. This is where digitizing has started.

Syntax: Link2FromNode(link: integer): integer

2.4.61 Link2ToNode

Returns the number of the node at the end of the link. This is where digitizing has ended.

Syntax: Link2ToNode(link: integer): integer

2.4.62 LinkCount

Return the highest link-number in the currently loaded network, which should equal to the number of links in the corresponding GIS-network.

Syntax: LinkCount: integer

2.4.63 LinkLength

This function returns the length of the link.

If the value is 0, it means the link isn't valid for routing. Please go back and check the result of the import process.

Syntax: LinkLength(link: integer): TCost 143

2.4.64 LinkLimit

Returns the maximum number of links according to your license. See feature matrix 3 (network size).

Syntax: LinkLimit: integer

2.4.65 Link2ExternalID

This method returns the external ID 9 from the internal ID.

Syntax: Link2ExternalID(link: integer): string

2.4.66 Link2RoadName

This method returns the roadname for a specific link and RoadFileID. The roadfile 53 should have been opened in advance.

Syntax: Link2RoadName(RoadFileID,link: integer): string

2.4.67 Link2RoadNameID

This method returns the roadnameID for a specific link and RoadFileID. The roadfile 53 should have been opened in advance.

A roadnameID is an integer, that corresponds to a roadname. It is more compact than a roadname and it is faster to do comparisons using roadnameID's.

RoadnameID's can be translated to road names this way 55.

Syntax: Link2RoadNameID(RoadFileID,link: integer): integer

2.4.68 Location2Coordinate

This method translates a location into a set of coordinates, with the ability to offset it to one of the sides of the link. When offset is positive, it will be on the right side of link, negative means left side. This is the same setup as DistanceToLinkExtended 43 uses for side.

Syntax: Location2Coordinate(loc: TLocation 146; offset: double): TFloatPoint 143

2.4.69 Location2CoordinateList

Same as Location2Coordinate, just for a whole list. LL is updated with the coordinates.

Syntax: Location2CoordinateList(LL: TLocationList 134; offset: double)

2.4.70 LoopLink

This function returns true if a link is a loop link.

Syntax: LoopLink(Index: Integer): boolean

2.4.71 LoopLinks

This function returns true if any link in the network is a loop.

Starting from 1-1-2012, networks with loop links will not work in TCalc 68, TRouteCalc 87 and TDrivingDirections 92.

Syntax: LoopLinks: boolean

2.4.72 Matrix

This method calculates a matrix of distances between all combinations of nodes in NL. Distance uses as-the-crow fly distances. If extra is true, the matrix will have an additional row and column, allowing for special optimization in class TTSP 122.

Syntax: Matrix(NL: TIntegerList 133; extra: boolean): TMatrix 146

2.4.73 MatrixDyn

This method calculates a matrix of distances between all combinations of locations in LL. Distance uses as-the-crow fly distances. If extra is true, the matrix will have an additional row and column, allowing for special optimization in class TTSP 122.

You should have called Location2CoordinateList 50 in advance.

Syntax: MatrixDyn(LL: TLocationList 134; extra: boolean): TMatrix 146

2.4.74 MaxDegree

Returns the maximum degree in the network. This is typically 5-6.

Syntax: MaxDegree: integer

2.4.75 MBR

Returns the minimum bounding rectangle of the currently loaded network. Is set when calling Open 53.

Syntax: MBR: TFloatRect 144

2.4.76 Node2Coordinate

Returns the coordinates of a node.

Syntax: Node2Coordinate(node: Integer): TFloatPoint 143

2.4.77 NodeCount

Return the highest node-number in the currently loaded network. The import 25 process assigns node numbers automatically, this can not be controlled by the user.

Syntax: NodeCount: integer

2.4.78 NoDriveThroughCheck

This function checks if NoDriveThrough 79 bit is set for any link on the route and the logical area is different from that of the first and last link on the route.

Returns true if the route isn't valid.

Syntax: NoDriveThroughCheck(route: TRoute 148): boolean

2.4.79 NoDriveThroughInit

If you change the NoDriveThrough 6 bit for some of the links after loading the network, you should call this function again to have various internal datastructures reset.

Syntax: NoDriveThroughInit

2.4.80 NoDriveThroughSet

This function checks if NoDriveThrough 79 bit is set for any link in the network.

Syntax: NoDriveThroughSet: boolean

2.4.81 NonCulDeSacNodes

This function returns a list of all nodes, that are not completely surrounded by CulDeSac 43 links.

It can be used together with function Nearest 79 to move from a node in a CulDeSac area.

Syntax: NonCulDeSacNodes(NL: TIntegerList 133)

2.4.82 ObjectCheck

This method checks individual objects for issues. I.e. not definite errors, but just issues.

It looks for:
- Duplicate nodes
- Self-intersecting objects
- Objects with sharp turns (set turn_angle parameter to define threshold, 90 is a good value)

Returns the number of objects with issues.

Syntax: ObjectCheck(filename: string; GF: TGISformat 144; turn_angle: integer): integer

In standard version you are limited to networks with <10000 links.

2.4.83 OneWayGet

This method returns information about one-way status for a link.

0: No restrictions
512: Travel only allowed in the direction of digitization
1024: Travel only allowed in the opposite direction of digitization
1536: Closed

Syntax: OneWayGet(link: integer): word

2.4.84 OneWaySet

This method sets information about one-way status for a link.

0: No restrictions
512: Travel only allowed in the direction of digitization
1024: Travel only allowed in the opposite direction of digitization

1536: Closed

Syntax: OneWaySet(link: integer; value: word)

2.4.85 Open

This method opens a street network and loads all information into memory. Files are loaded from Directory 43 property.

If attributes is true, attribute.bin is also opened.

Set coord3cache to true, unless you have limited RAM.

Set spatialindex to true, if you want to load the spatial index and use TSpatialSearch 64 methods.

ExternalID parameter:
0: Do not open
1: Open, but no caching
2: Open, cache index
3: Open, cache index + keys

Syntax: Open(attributes, Coord3Cache, spatialindex: boolean; externalid: integer)

2.4.86 OpenLimit

Use this method to open limit files. Filenumber should refer to the naming of the file, while LimitID is from 1 to 9. It is important to open them in sequence or the routing restrictions will not work. For instance open limitID 1 and 2, but not 4 or higher.

Syntax: OpenLimit(FileNumber, LimitID: integer; bitpattern: boolean)

See also Limits 9, LimitFileIndex 33, GetLimit 48, SetLimit 56 and TCalc.SetLimit 81.

2.4.87 OpenRoadName

This method opens a roadname 9 file, previously setup through import.

Specify the number of the file (1..99) and if it should be cached. It is only relevant to cache if you plan to generate MANY driving directions.

Syntax: OpenRoadName(RoadFileID: integer; cache: boolean)

2.4.88 ParallelLinks

Identifies group of links, which start and end at the same two nodes. These might give problems in some networking algorithms ("emme/2" for instance).

RW Net has no problem with parallel links, unless you want to apply a turn restrictions from one parallel link to another and only want it at one of the 2 nodes they have in common.

The function returns the number of parallel links found.

The generated GIS file contains fields for:
- Link: Original link ID
- Group: 1, 2, 3 ....
- SameLength: Logical value, which is true if all links in the group has the same length. This usually means the same link has been digitized twice.

Syntax: ParallelLinks(filename: string; GF: TGISformat 144): integer

In Standard version you are limited to networks with <10000 links.

2.4.89 PRJ

This property is set when calling Open 53.

property PRJ: string

2.4.90 RandomLocation

Returns a random location.

Syntax: RandomLocation(r: TRandom 136): TLocation 146;

2.4.91 RandomNode

Returns a random node.

Syntax: RandomNode(r: TRandom 136): integer;

2.4.92 RandomPoint

Returns a random point on the network.

It is a coordinate within 10 meters of a location.

Syntax: RandomPoint(r: TRandom 136): TFloatPoint 143;

2.4.93 ReadCost

This method allows you to read speed for all links from a single DAT or DBF file.

Specify the full filename, including path.
Fieldindex is 0-based.
If fieldname is defined, fieldindex is ignored.

If cost is 0 or negative for any link, it is ignored.

Syntax: ReadCost(index: integer; filename: string; fieldindex: integer; fieldname: string);

2.4.94 ReadSpeed

This method allows you to read speed for all links from a single DAT or DBF file.

Specify the full filename, including path.
Fieldindex is 0-based.
If fieldname is defined, fieldindex is ignored.

If speed is 0 or negative for any link, it is ignored.

Syntax: ReadSpeed(index: integer; filename: string; fieldindex: integer; fieldname: string);

2.4.95 ReadTime

This method allows you to read speed for all links from a single DAT or DBF file.

Specify the full filename, including path.
Fieldindex is 0-based.
If fieldname is defined, fieldindex is ignored.

If time is 0 or negative for any link, it is ignored.

Syntax: ReadSpeed(index: integer; filename: string; fieldindex: integer; fieldname: string);

2.4.96 RoadClass

This array property can be used to get or set the road class of a link. A road class is a value from 0 to 31. See attributes 6.

Property RoadClass[Index: Integer]: integer

2.4.97 RoadName2RoadNameID

This method translates a roadname into the corresponding roadname ID.

Syntax: RoadName2RoadNameID(RoadFileID: integer; roadname: string; ignorecase: boolean): integer

2.4.98 RoadNameID2RoadName

This method returns roadname for a roadname ID.

Syntax: RoadNameID2RoadName(RoadFileID,RoadNameID: integer): string

2.4.99 RoadNameMaxWidth

This method returns the maximum width for an open roadfile and for a specific codepage. This can be used when writing to a TGISwrite 104 output with fixed field width, such as DBF, SHP, MIF and TAB.

Syntax: RoadNameMaxWidth(RoadFileID: integer; Codepage: TCodePage): integer

2.4.100 RouteLength

This method returns the length of a route.

See also RouteCost 80 and RouteTime 80.

Syntax: RouteLength(Route: TRoute 148): TCost 143

2.4.101 Select

This method can be used for selecting from the street network.

Output is stored in BA. New selections are set and added to any previous selections in BA.

Roadclass_min and roadclass_max specifies the interval for selections. Use 0 and 31 to ignore.

Hierarchy_min and hierarchy_max specifies the interval for selections. Use 0 and 5 to ignore.

You can specify a RoadFileID and RoadNameID to select a specific roadname. Use 0 to ignore.

For each of the bits 8 - 15 in the attribute 6 you can specify the value 0 or 1. Use 2 to ignore.

Syntax: Select(BA: TBitArray;
roadclass_min,roadclass_max,hierarchy_min,hierarchy_max: integer;
RoadFileID,RoadNameID: integer;
bit8,bit9,bit10,bit11,bit12,bit13,bit14,bit15: byte)

2.4.102 SetCost

This method sets cost for a single link and array index.

Syntax: SetCost(index,link: integer; cost: TCost)

See also CreateArrayCost 42, CalculateCost 40 and GetCost 48

2.4.103 SetLimit

This method sets limit for a single limitID and link. You need to have called OpenLimit 53 in advance to setup the memory.

This doesn't change any file on disk.

Syntax: SetLimit(limitID,link: integer; value: byte)

2.4.104 SetSpeed

Sets speed for the specified array index and link. Internally it is the corresponding time, that is stored.

See also MaxSpeed 78.

Syntax: SetSpeed(index,link: integer; speed: TCost 143)

2.4.105 SetTime

Sets time (minutes) for the specified array index and link.

Syntax: SetTime(index,link: integer; time: TCost 143)

2.4.106 Split

This method will create entries in LL for all links in the network and for every x km.

Example: If the value of distance parameter is 1 km and a link is 3.6 km long, entries will be created like this:
Evenout = false: 1, 2 and 3 km (3 entries)
Evenout = true: 0.9, 1.8 and 2.7 km (3 entries)
No entries are created for links shorter than 1 km.

Call ExportLinks 44 afterwards to have the network saved, but with shorter links.

Syntax: Split(distance: TCost 143; evenout: boolean; LL: TLocationList 134)

2.4.107 SwapList

This method swaps the order of coordinates in the variable.

Syntax: SwapList(var list: TFloatPointArrayEx)

2.4.108 SwapOneWay

This method swaps all oneway restrictions, so they point in the opposite direction. If both oneway bits 6 (9 and 10) are set, nothing happens.

It can be used to calculate isochrones from many-to-one, by first swapping the restrictions, doing it one-to-many and then swapping back again.

This is for instance relevant, when doing a drivetime isochrone and it is more important how fast you can get TO the center (example: hospitals), rather than getting FROM the center (example: fire stations).

You should not combine this with turn restrictions, since these can not be "swapped" easily.

Syntax: SwapOneWay

2.4.109 TurnAutoProcess

This method allows to automatically detect turns and add turn delays through out your network.

You can either use the built-in rules for adding delays for T-junctions and normal junctions, or override these with events.

In any case, it should be specified if traffic is right- or left-hand. Left-hand is known from UK, Ireland, Australia, New Zealand, Japan, India, South Africa etc.

It should also be specified if any nodes should be skipped completely. This could be nodes / junctions which are part of ramps or use traffic lights, so you want to set up different rules.

We suggest calling TurnExportGIS 59 once you have called TurnAutoProcess to see what it actually gives in minutes.

This method is quite slow for large networks, so use it with care.

Events

When using the events you will just get a list of links back, making up the intersection. This includes intersections or nodes with degree > 2.
The list is ordered in the same way as is shown on the small maps below.

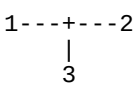
T-junctions:
TTurnTEvent = procedure(Sender: TObject; node, link1, link2, link3: integer)

Normal junctions:
TTurnEvent = procedure(Sender: TObject; node: integer; links: TIntegerArray 145)

Built-in rules

Delays for each road class in the network is supplied as a TRoadClassTurnCost 137 object. For all links in each intersection the delay is then looked up, based upon their road class. If a turn involves crossing multiple traffic flows in the intersection, these are added together as can be seen here:

For a T-junction, where 1-2 is the main road:



Delays for right-hand traffic:

From 3 to 1: 1.5 * (delay1+delay2)
From 3 to 2: delay1
From 2 to 3: delay1
Other turns: No delay

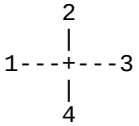
Delays for left-hand traffic:

From 3 to 1: delay2
From 3 to 2: 1.5 * (delay2 + delay1)
From 1 to 3: delay2
Other turns: No delay

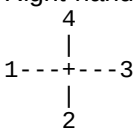
Main road is determined from geometry: The link combination closest to a straight line is the main road.

For a normal intersection, where 1-3 is the main road:

Left-hand traffic



Right-hand traffic



From 1 to 4: delay3
From 2 to 1: 1.5 * (delay1 + delay3 + delay4)
From 2 to 3: delay1
From 2 to 4: 1.5 * (delay1 + delay3)
From 3 to 2: delay1
From 4 to 1: delay3
From 4 to 2: 1.5*(delay1 + delay3)
From 4 to 3: 1.5*(delay1 + delay2 + delay3)
Other turns: No delay

Main road gets detected from the delays. Largest delay means main road.

If opposing roads, 1-3, can't be identified as the main road, the intersection is skipped. This happens if for instance 1-2 has the largest delay.

If the delay for all 4 roads is the same:

Delays for right-hand traffic:

Right turns: delay1
Straight ahead: 2 * delay1
Left turns: 4.5 * delay1

Delays for left-hand traffic:

Left turns: delay1
Straight ahead: 2 * delay1
Right turns: 4.5 * delay1

For intersections with >4 links:

No processing occurs. You can use the events instead.

Syntax: TurnAutoProcess(index: integer; LeftHandTraffic: boolean; RCTC: TRoadClassTurnCost 137; SkipNodes: TBitArray 134)

2.4.110 TurnExportBin

This method saves all turn restrictions to a file on disk, which can later be loaded with function TurnImportBin 59.

Specify a full filename, preferably with along this pattern "turn*.bin" (makes it easier to recognize the file).

Syntax: TurnExportBin(index: integer; filename: string)

2.4.111 TurnExportGIS

This method writes all turn restrictions to a TGISwrite, so it is easier to graphically view the turn restrictions.

Specify a full filename.

Syntax: TurnExportGIS(index: integer; filename: string; GF: TGISformat 144)

2.4.112 TurnExportTxt

This method saves all turn restrictions to a text file on disk, which can later be loaded with function TurnImportTXT 60. Internal link ID's are used in the output.

Specify a full filename.

Syntax: TurnExportTxt(index: integer; filename: string)

2.4.113 TurnImportBin

This method loads turn restrictions from a file on disk, created by TurnExportBin 59.

Specify a full filename.

Syntax: TurnImportBin(index: integer; filename: string)

2.4.114 TurnImportTxt

This method loads turn restrictions from a text file on disk. Supply a full filename, including folder.

The format is one or more lines, where each line stores one restriction with parameters stored in space separated format. Different types of restrictions are possible:

0: Simple Turn restriction, 2 external link ID's + 1 cost value
1: Simple Turn restriction 61, 2 link ID's + 1 cost value
2: TurnStandard 61, coordinates for node
3: Mandatory turn, 2 external link ID's
4: Mandatory turn 60, 2 link ID's
5: Complex Turn restriction, >2 external link ID's + 1 cost value
6: Complex Turn restriction 61, >2 link ID's + 1 cost value

File example:

```
// Comment
0 A4003234 A4003127 -1
1 456 230 -1
2 -77.024098 38.902711
3 A4003234 A4003127
4 456 230
5 A4003279 A4003234 A4003127 -1
6 89 456 230 -1
```

Lines starting with // are ignored as comments.

The ITN converter will create turn restriction files in this format.

If you use turn restrictions with external ID's (type 0, 3 and 5), make sure you have called Open 53 with externalID>0 or you will get an error code returned.

Type 0, 2, 3 and 4 gets translated into one or more type 1 during import and type 5 gets translated into type 6 during import.

Syntax: TurnImportTxt(index: integer; filename: string)

2.4.115 TurnMandatory

This method defines that turns from link1 is only allowed if the next turn is link2.

Internally this is translated into a number of turn restrictions. These are only applied at the end of link1, where it is actually possible to connect to link2.

If link1 and link2 are parallel links, you will get an error.

Syntax: TurnMandatory(index: integer; link1,link2: integer)

2.4.116 TurnReset

Clears the list of turn restrictions.

Syntax: TurnReset(index: integer)

2.4.117 TurnRestriction

This method defines a restriction on turns from link1 to link2.

cost < 0: Turn prohibited
cost = 0: Remove turn restriction
cost > 0: Additional cost related to the turn (= delay).

If link1=link2 the restriction (a U-turn) is skipped. See here how to apply U-turn restrictions 83. It is not possible to have delays for U-turns, they can only be either allowed or banned.

If link1 and link2 are parallel links, a turn restriction is added at both nodes. Prevent this by breaking up one of the links.

Syntax: TurnRestriction(index,link1,link2: integer; cost: TCost 143)

2.4.118 TurnRestrictionComplex

This is the same method as TurnRestriction 61 except up to 6 links can be defined as making up the restriction. If you need more than 6 links, use TurnImportTXT 60 instead.

Syntax: TurnRestrictionComplex(index,link1,link2,link3,link4,link5,link6: integer; cost: TCost 143)

2.4.119 TurnStandard

Adds turn restriction on standard 4-degree intersection, which means no turns are allowed - only driving straight through. The method also works for nodes with higher, but still even degree.

Syntax: TurnStandard(index,node: integer)

2.4.120 UpdateAlphas

Alpha is a parameter used internally by TRouteCalc 87 to direct routes faster towards the target.

After changing speed, time or cost and before calculating routes with TRouteCalc 87, you should call this method.

For shortest path routing with TRouteCalc, the method is not required.

Syntax: UpdateAlphas

2.4.121 Write

These methods are for writing results from various calculations directly to a DBF or DAT file. DAT is part of a MapInfo TAB file and similar to a DBF file.

All 4 variations below allow you to write to gfDecimal, gfFloat, gfInteger, gfSmallInt and gfLogical fields.
In any case values are written so they best possibly are stored in the underlying field.

Values which are too big, makes it raise an error, such as storing 100000 in a SmallInt field (valid range -32767 to 32767).
If you try to store a number in a Logical field, all values >0 are treated as True.

Length of TIntegerArray, TCostArray and TBitArray need to match the number of records in the file.

TIntegerList is treated as a list of records marked as True. If you have reset=true at the same time, all other records are marked as false.

Fieldindex is 0-based. If you specify fieldname, it is used instead of fieldindex.

Syntax:
Write1(filename: string; fieldindex: integer; fieldname: string; value: TIntegerArray 145);
Write2(filename: string; fieldindex: integer; fieldname: string; value: TCostArray 143);
Write3(filename: string; fieldindex: integer; fieldname: string; value: TBitArray 134);
Write4(filename: string; fieldindex: integer; fieldname: string; value: TIntegerList 133; reset: boolean);

2.4.122 Pro Methods

2.4.122.1 DownStream

This method can be used for tracing in an oriented network. It will start from a link and trace in the forward (downstream) direction as long as there is only one directed link from the next node (unique direction for flow). Links without direction are ignored. You can use direction 512 / 1024 as in a normal street network.

This method is only useful for utility networks, such as sewers, water pipes etc. It has little relevance for street networks.

Output linklist is in the order of flow, starting with the input link.

Syntax: DownStream(link: integer; linklist: TIntegerList 133)

See also Trace 63 and UpStream 64

2.4.122.2 ExportTrafficList

This method will export TL, so it can be viewed externally.

If lines = false:
Output is shown as point objects with origin / destination records shown as "O" and "D".

If lines = true:
Output is shown as lines connecting origin and destination.

Syntax: ExportTrafficList(filename: string; GF: TGISformat 144; TL: TTrafficList 134; lines: boolean)

2.4.122.3 Join

This will identify neighbouring links and join them in groups. The grouping can be defined by setting 3 parameters, where at least one of them need to be <> "ignore":

Topology
0: Ignore it
1: Connected
2: Intersection to intersection (intersection: Node with degree 43 >= 3)
3: Intersection to intersection, but ignoring cul-de-sac links

RoadFileID
0: Ignore roadname
N: Split when roadname changes

Attributes:
False: Ignore attributes
True: Split according to attributes

Result is stored in IA array: Indices with the same value belong to the same group.
Result can also be written to a TGISwrite output, if filename is specified. If GF = gfArray, just set filename to something.

Using parameter combination (0,0,false) is not allowed, since it would join ALL links into one large object.

When using topology=2, joins that would result in loops, are avoided.

Normally it used with these parameters, when the output is to be used for routing:
Topology = 2
RoadfileID >0, if the network is to be used with driving directions.
Attributes = true.

If turn restrictions are defined, they are exported to a file with ".turn" as extension with the updated link ID's as reference.

Syntax: Join(filename: string; GF: TGISformat 144; topology,RoadFileID: integer; attributes: boolean; var IA: TIntegerArray 145)

2.4.122.4 Node2Link

This method returns the ID of the links connected to a node.

Iterate through the links this way:

```
for index = 1 to Degree 43(node)
  print node2link(node,index)
next
```

Syntax: Node2Link(node,linkindex: integer): integer

2.4.122.5 Trace

This method can be used for tracing in a network. It will start from a link and trace in all directions until a node is reached that is marked with True in the Valves input parameter.

Oneway restrictions are ignored.

This method is only useful for utility networks, such as sewers, water pipes etc. It has little relevance for street networks.

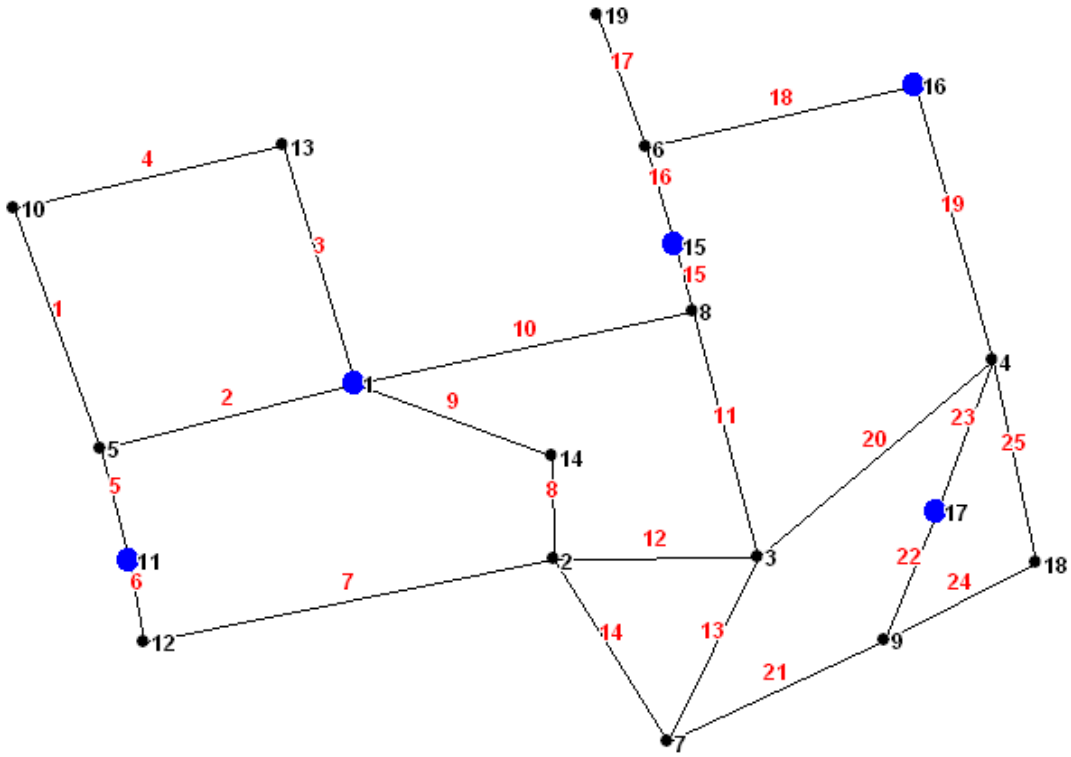
Output parameter ValvesReached shows which of the valves was reached.
Output parameter LinksReached shows which of the links was reached.

Syntax: Trace(link: integer; Valves, ValvesReached, LinksReached: TBitArray 134)

See also DownStream 62 and UpStream 64

Example:
Input:
Start link: 1 (red labels)
Valves: 1, 11, 15, 16, 17 (blue dots)

Output:
ValvesReached: 1, 11
LinksReached: 1, 2, 3, 4, 5.



2.4.122.6 UpStream

This method can be used for tracing in an oriented network. It will start from a link and trace in the reverse (upstream) direction, branching if required. Links without direction are ignored. You can use direction 512 / 1024 as in a normal street network.

This method is only useful for utility networks, such as sewers, water pipes etc. It has little relevance for street networks.

Upstream links are marked as true in the output, including the input link.

Syntax: UpStream(link: integer; links: TBitArray 134)

See also DownStream 62 and Trace 63

2.5 TSpatialSearch

This class is for making spatial searches in the network. Generally as either searching for nodes or links (locations).

On top of this, various topological checks can also be performed: FindNonConnected 65, FindOverPasses 65 and Split 68.

When opening the network 53, make sure parameter "spatialindex" is true.

2.5.1 Create

When creating an instance of TSpatialSearch, it is required to specify a network.

Syntax: Create(NW: TNetwork 35)

2.5.2 FindOverPasses

This method finds where 2 links intersect and add these locations to LL (LL is not cleared first). Two entries are added every time, one for each link.

There should only be overpasses, where there is also a bridge / tunnel in real life.

You can call ExportLinks 44 afterwards if you want to split the links where an overpass was found.

The FindOverPasses algorithm makes sure that exactly the same coordinates are used for both pairs of links, so snap is guaranteed if you call TImport on the resulting output from ExportLinks. See also SplitAndSnap 68.

Syntax: FindOverPasses(LL: TLocationList 134)

In standard version you are limited to networks with <10000 links.

2.5.3 FindNonConnected

This method performs a topological checks:

It checks if there within a radius of all nodes in the network are other nodes / links, which are not connected to the first node.
Usually set radius = 0.005 km or an even smaller value.

Output is a TGISwrite with fields for node and link numbers and the distance from the first node to the node / link. Visually a line is drawn.

Syntax: FindNonConnected(filename: string; GF: TGISformat 141; Radius: double): integer

In standard version you are limited to networks with <10000 links.

2.5.4 GeoJSON

When using gfGeoJSON 144 as output format in functions like FindNonConnected 65 etc, this read-only property holds the output.

Property GeoJSON: string

2.5.5 GISarray

When using gfArray 144 as output format in function FindNonConnected 65, this read-only property holds the output.

Type: TGISarray 110

2.5.6 NearestLink

This function will locate which link you are driving on, when both GPS coordinate and bearing (0-359) are known.
0 = North, 90 = East, 180 = South, 270 = West.

This is especially useful, when there are many intersecting roads or two parallel roads.

Specify a search radius, such as 0.025 km, since it uses SelectLinks 67 internally.

The result is returned as a list of possible matches, with the best guess at the top of the list.

Syntax: function NearestLink(P: TFloatPoint 143; Bearing, SearchRadius: double): TGPSMatchList 133;

See also NearestLocation 66

2.5.7 NearestLocation

This locates the nearest location from P.

It returns this information:
- Location
- Side of the link (-1: Left or +1: Right)
- Distance
- Coordinates of location on link

A typical use is converting large amounts of GPS coordinates into network locations. An example of performance is:
A street network with 200,000 links, latitude/longitude coordinates: 2000 calcs per sec (using an AMD A6-5400K)

Larger street networks makes it slightly slower, while using projected coordinates makes it faster. Setting MaxVerticesPerCell 28 at a lower value than default can also make it slightly faster.

Syntax: NearestLocation(P: TFloatPoint 143; var Loc: TLocation 146; var side: integer; var distance: double; var Pnew: TFloatPoint 143)

See also DistanceToLink 43, NearestLocationSimple 66 and SkipInSearch 68.

2.5.8 NearestLocationSimple

This finds the nearest location from P.

Syntax: NearestLocationSimple(P: TFloatPoint 143): TLocation 146

See also DistanceToLinkSimple 44, NearestLocation 66 and SkipInSearch 68.

2.5.9 NearestLocationSimpleList

Same as NearestLocationSimple 66, except it processes LL.

Syntax: NearestLocationSimpleList(LL: TLocationList 134)

2.5.10 NearestNode

This locates the nearest node from P. Returns distance to the node too.

Syntax: NearestNode(P: TFloatPoint 143; var node: integer; var distance: double)

2.5.11 NearestVertex

This locates the nearest vertex from P.

It returns this information:
- Link
- Index of vertex (0-based)
- Distance

Syntax: NearestVertex(P: TFloatPoint 143; var link, index: integer; var distance: double)

See also GetGISSection 48

2.5.12 SelectLinks

This method selects all links within a radius from P.

Result is returned as a list of links in List and as a bit pattern in BA.

Syntax: SelectLinks(P: TFloatPoint 143; Radius: double; List: TIntegerList 133; BA: TBitArray 134)

See also SkipInSearch 68.

2.5.13 SelectLinksArray

Same as SelectLinks 67, but result is only returned in the array.

Syntax: SelectLinksArray(P: TFloatPoint 143; Radius: double; BA: TBitArray 134)

2.5.14 SelectLinksList

Same as SelectLinks 67, but result is only returned in the list.

Syntax: SelectLinksList(P: TFloatPoint 143; Radius: double; List: TIntegerList 133)

2.5.15 SelectNodes

This method selects all nodes within a radius from P.

Result is returned as a list of nodes in List and as a bit pattern in BA.

Syntax: SelectNodes(P: TFloatPoint 143; Radius: double; List: TIntegerList 133; BA: TBitArray 134)

2.5.16 SelectNodesArray

Same as SelectNodes 67, but result is only returned in the array.

Syntax: SelectNodesArray(P: TFloatPoint 143; Radius: double; BA: TBitArray 134)

2.5.17 SelectNodesList

Same as SelectNodes 67, but result is only returned in the list.

Syntax: SelectNodesList(P: TFloatPoint 143; Radius: double; List: TIntegerList 133)

2.5.18 SkipInSearch

This property defines if the attribute bit 6 for skipping links is used in method NearestLocation 66 and SelectLinks 67.

Default: False

Type: boolean

See also AttributeSetSkipInSearchBit 39

2.5.19 SplitAndSnap

This methods performs a search around nodes with degree <3. If any location on a link is found in the search, and the link is not connected to the original node and not a node at the same time (i.e. start or end of the link), the location is added to LL (LL is not cleared first). The same coordinate is also added to LL, with the original node as reference together with special codes, that can be handled by ExportLinks 44, so links can be split and updated correctly for exact snap in future TImport 25 runs.

You can also choose to call ExportLocationList 45 to visually check and edit, where issues has been found.

The method returns the number of positive searches. LL holds ~2-3 times as many items.

Syntax: SplitAndSnap(Radius: double; LL: TLocationList 134): integer

In standard version you are limited to networks with <10000 links.

2.6 TCalc

This class is used for one-to-many route calculations, the Dijkstra algorithm is used. Use TRouteCalc 87 for one-to-one route calculations.

Typical sequence when using TCalc is like this:

Call SetTime 56 and/or SetCost 56 if you want to calculate more than just length of routes.

Define which criteria you want and call the corresponding method: SetShortest 81, SetFastest 81 or SetCheapest 81.

The SkipLinkList 81 can be used to ignore certain links in the route calculations.

Eventually set MaxCost 78, if you want to create a smaller isochrone than otherwise.

MaxSpeed 78 can be used to override the speed for the network, in case of slow vehicles.

NoDriveThrough 79 can be set to avoid areas, where you are not allowed to drive through ("no access").

Finally call one of the actual isochrone methods, possibly followed by additional query methods:

If you just want cost:
- IsoCost 72 or IsoCostList 72 or IsoCostListN 72 > NodeCost 79 or LinkCost 76
- IsoCostDyn 72 or IsoCostListDyn 72 or IsoCostListNDyn 73 > NodeCost 79 or LinkCostDyn 76
- IsoCostDynApproach 84 > LinkCostDynApproach 85

If you want cost and the route:
- IsoCost 72 or IsoCostList 72 > RouteFind 80 > RouteCost 80, RouteLength 55 and / or RouteTime 80
- IsoCostDyn 72 or IsoCostListDyn 72 or IsoCostListNDyn 73 > RouteFindDyn 80 > RouteCost 80, RouteLength 55 and / or RouteTime 80
- IsoCostDynApproach 84 > RouteFindDynApproach 86 > RouteCost 80, RouteLength 55 and / or RouteTime 80

Matrix methods: (to TMatrix 146)
- Matrix 51
- Matrix2 77
- MatrixDyn 51
- MatrixDyn2 77

Matrix methods: (output to GIS files)
- MatrixOut 77
- MatrixDynOut 77
- MatrixPOut 78

Other methods:
- Nearest 79
- NearestDyn 79
- NearestOpen 79
- NearestOpenDyn 79

Methods for isochrones (see also here 16):
1. DriveTimeSimple 70
2. IsoPoly 75
3. AlphaShape 83 (Pro only)
4. IsoLinkDriveTime 73

It is worth noting that Cost in TNetwork is different from Cost in TCalc:

- In TNetwork it is a generalized cost for a single link (or turn delay), much similar to the length or time of a link.

- In TCalc it is the result of a route / isochrone calculation from a starting point to somewhere else. The cost can be either distance (SetShortest 81), time (SetFastest 81) or "cost" (SetCheapest 81), depending upon which criteria has been set up.

2.6.1 Create

When creating an instance of TCalc, it is required to specify a network and if turnmode should be true or false.

Syntax: Create(NW: TNetwork 35; Turnmode: boolean)

2.6.2 DistanceUnit

When generating output, you can use this property to use miles in the output.

This affects MatrixOut 77, MatrixPOut 78 and MatrixDynOut 77. But no other methods !!

Default: duKm

Type: TDistanceUnit 143

2.6.3 DriveTimeSimple

This is a simpler version of the voronoi 98 based method for drivetime isochrones. It uses a single node as center.

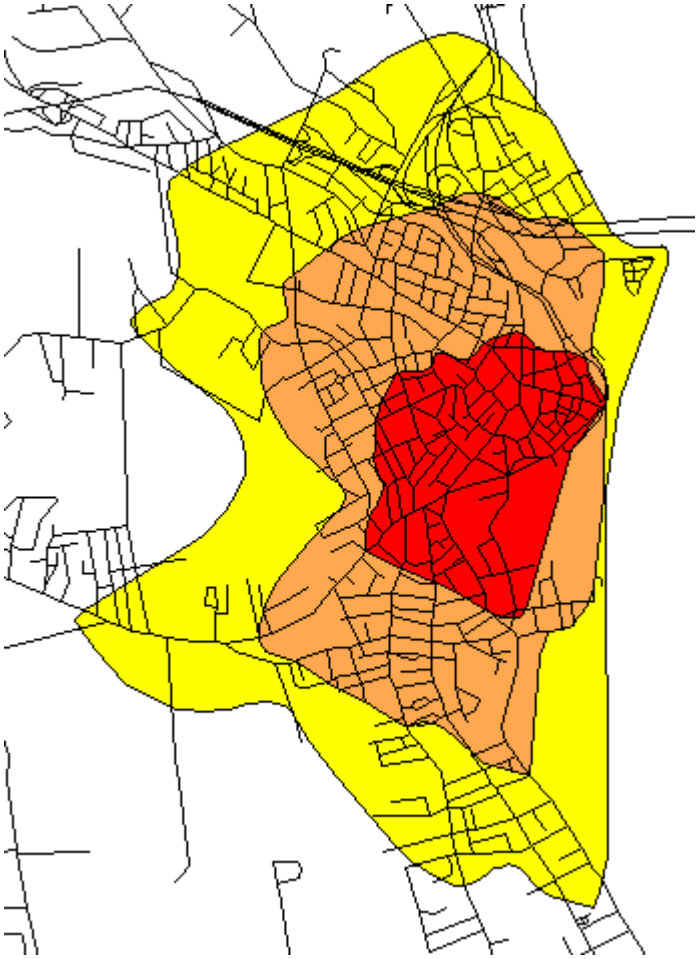
Angle should be in the range 0 to 45, with 0 giving the convex hull. Small values make the isochrone follow the network more closely and larger values makes it closer to the convex hull.

Smoothing 86 can be enabled, but may give degenerate results when combined with multiple steps.

Syntax: DriveTimeSimple(filename: string; GF: TGISformat 144; node: Integer; Steps: TStepList 134; angle: double; doughnut: boolean);

See also Isochrones - overview 16

Example with 1-2-3 km, angle = 3 degree, doughnut = true and smoothing = (5,3,5):



2.6.4 GeoJSON

When using gfGeoJSON 144 as output format in functions like MatrixOut 77 etc, this read-only property holds the output.

Property GeoJSON: string

2.6.5 GISarray

When using gfArray 144 as output format in functions like MatrixOut 77 etc, this read-only property holds the output.

Type: TGISarray 110

2.6.6 IgnoreOneway

Set this property to true, if you want to ignore one-way restrictions in the route calculations.

Default: false

Type: boolean

2.6.7 IsoCost

This method calculates an isochrone from the node. The size of the isochrone can be restricted by setting MaxCost 78.

Syntax: IsoCost(node: integer)

2.6.8 IsoCostDyn

This method calculates an isochrone from the location. The size of the isochrone can be restricted by setting MaxCost 78.

An error will be raised if location is on a loop link. Check with LoopLink 50 function in advance.

Syntax: IsoCostDyn(Loc: TLocation 146)

2.6.9 IsoCostList

This method calculates an isochrone from node, which extends until all nodes in NL has been reached. If MaxCost 78 has been set, it may stop sooner.

Syntax: IsoCostList(node: integer; NL: TIntegerList 133)

2.6.10 IsoCostListDyn

This method calculates an isochrone from the location, which extends until all locations in LL has been reached.

An error will be raised if location is on a loop link. Check with LoopLink 50 function in advance.

Syntax: IsoCostListDyn(Loc: TLocation 146; LL: TLocationList 134)

2.6.11 IsoCostListN

This method calculates an isochrone from node, which extends until the first N nodes in NL has been reached.

If MaxCost 78 has been set, it may stop sooner.

Result is returned in IL as a sorted index into NL. Length of IL may be < N, if not all nodes in NL is reached.

Example:
NL = {100, 200, 300, 400, 500, 600}
N = 3
cost(100) = 32
cost(200) = 45
cost(300) = 103
cost(400) = 77
cost(500) = 80
cost(600) = 10

Output: IL = {5, 0, 1}
Cost of index 5, 0 and 1 is 10, 32 and 45.

Syntax: IsoCostListN(node: integer; NL, IL: TIntegerList 133; N: integer)

2.6.12 IsoCostListNDyn

Same method as IsoCostListN 72, just using locations instead:

An error will be raised if location is on a loop link. Check with LoopLink 50 function in advance.

Syntax: IsoCostListNDyn(Loc: TLocation 146; LL: TLocationList 134; IL: TIntegerList 133; N: integer)

2.6.13 IsoCostMulti

This method calculates an isochrone from a list of facilities (NL, nodes), identifying which facility is nearest. No more than 65535 nodes are allowed in the list.

For each facility you can define if there is an offset, i.e. a cost>0 value that is added to the cost. This can for instance be used to create drivetime regions around a number of fire-stations, which has different start times. Set parameter to nil, if offset=0 for all facilities.

You can set MaxCost if you only want smaller isochrones in your calculations.

BestCost returns the cost to the nearest facility.
BestFacility returns an index into NL. If value is 65535, it means the node wasn't in reach of any facility.

Both have as many elements as there are nodes.

You can not combine this method with subsequent calls to RouteFind 80, LinkCostDyn 76, NodeCost 79 or any other methods. You should only use the two output parameters as result.

Syntax: IsoCostMulti(NL: TIntegerList 133; Offset: TCostArray 143; var BestCost: TCostArray 143; var BestFacility: TWordArray 150)

2.6.14 IsoLinkDriveTime

This method shows the distance from one more centers (nodes) to each location on a street network.

Internally it uses IsoCostMulti 73 and shares the NL and Offset parameters with this method.

StepList is a number of cost values, indicating which values are used as steps in generating the output. For instance steps 1, 2 and 3 will generate steps 0-1, 1-2 and 2-3.

Output is a polyline theme and the polylines are dynamically segmented to show the exact position where it changes, which center is the nearest. Polylines are oriented so they point away from the center.

Syntax: IsoLinkDriveTime(filename: string; GF: TGISformat 144; NL: TIntegerList 133; Offset: TCostArray 143; StepList: TStepList 134)

See also Isochrones - overview 16

Example:

2.6.15 IsoLinkDriveTimeDyn

The same as IsoLinkDriveTime 73, except it uses a single location as center.

Syntax: IsoLinkDriveTimeDyn(filename: string; GF: TGISformat 144; loc: TLocation 146; OffSet: TCost 143; StepList: TStepList 134)

2.6.16 IsoLinkServiceArea

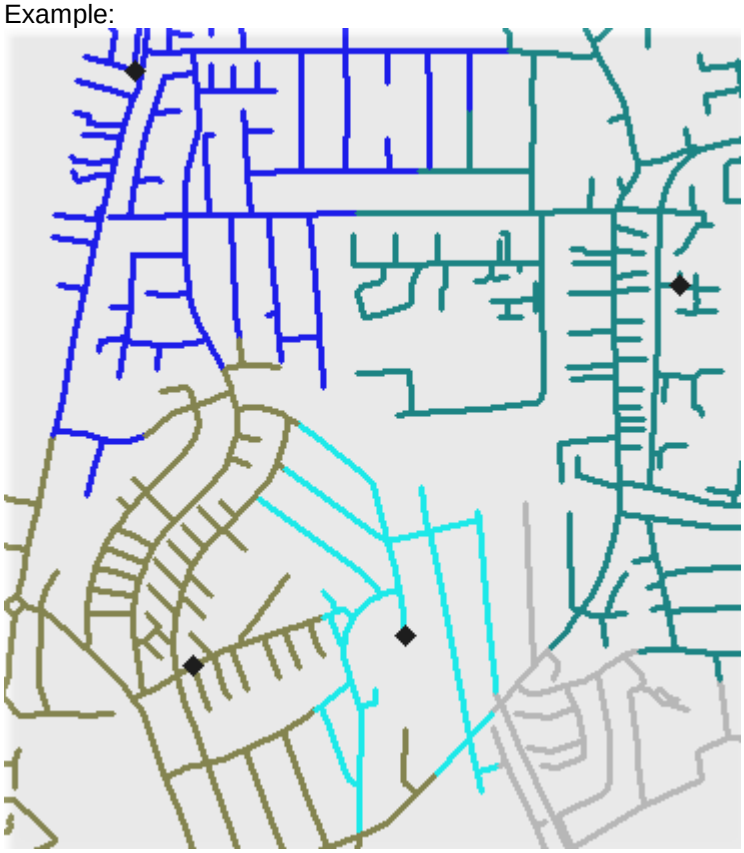
This method shows which center (node) is the nearest on a street network.

Internally it uses IsoCostMulti 73 and shares the NL and Offset parameters with this method.

Output is a polyline theme and the polylines are dynamically segmented to show the exact position where it changes, which center is the nearest. Polylines are oriented so they point away from the center.

Syntax: IsoLinkServiceArea(filename: string; GF: TGISformat 144; NL: TIntegerList 133; Offset: TCostArray 143)

Example:



2.6.17 IsoPoly

This method is for calculating input to the voronoi 98-based methods for drivetime isochrone, service areas etc.

Main input is two lists with nodes and locations. If you only have nodes or locations, set the other list parameter to nil. No more than 65535 items are allowed in the lists in total.

The lists contain your facilities or just a single facility. Isochrones are calculated for each of them and the output keeps track of which node / location was the nearest and cost. This is done for all nodes in the network.

For each facility you can define if there is an offset, i.e. a cost>0 value that is added to the cost. This can for instance be used to create drivetime regions around a number of fire-stations, which has different start times. Set parameter to nil, if offset=0 for all facilities.

Addnodes can be used to define if the calculation should be done for additional locations along long links. If the value of addnodes is, say 1 (km), and a link is 3.6 km long, additional nodes will be inserted at 0.9, 1.8 and 2.7 km in the output. No additional nodes are added if the link is shorter than 1 km.

MaxCost 78 & MBR can both be used to define if the isochrone should be restricted in size. If MaxCost 78=0 and MBR=cFRNull, the whole network is covered.

See also IsoPolyFast 76, IsoPolyRandomnization 76 and Isochrones - overview 16.

Syntax: IsoPoly(NL: TIntegerList 133; LL: TLocationList 134; Offset: TCostArray 143; addnodes: TCost 143; MBR: TFloatRect 144): TPolyGeneration 136

2.6.18 IsoPolyFast

This function is the same as IsoPoly 75, except the MBR parameter is replaced with a buffer parameter.

Internally the function automatically calculates MBR from the items in NL and LL, MaxCost 78 and buffer. This makes it much faster and the recommended solution for preparing input for drive time polygons.

Buffer should be specified in km. Suggested values are 2 km for urban areas and 10 - 20 km in rural areas (for voronoi 98 calculations).
If you use the output for Alpha shapes 83, then buffer = 0 is sufficient.

Syntax: IsoPolyFast(NL: TIntegerList 133; LL: TLocationList 134; Offset: TCostArray 143; addnodes, buffer: TCost 143): TPolyGeneration 136

2.6.19 IsoPolyRandomnization

Set this property to true, if you want to add a very small randomnization to the coordinates output from the IsoPoly 75 methods.

This is sometimes needed if you are using low values for the addnodes parameter due to numerical instabilities in the

Default: false

Type: boolean

2.6.20 LinkCost

TurnMode 83 = false:
Returns the maximum cost of the two end nodes of the link.

TurnMode 83 = true:
Link>0: Returns the cost of going to the ToNode of the link.
Link<0: Returns the cost of going to the FromNode of the link.

Syntax: LinkCost(link: integer): TCost 143

2.6.21 LinkCostDyn

Returns the cost of getting to a specific location of a link.
This method can be used after a call to either IsoCost 72 or IsoCostDyn 72 function.

Syntax: LinkCostDyn(loc: TLocation 146): TCost 143

2.6.22 Matrix

This method calculates a matrix, based upon the nodes in NL.
Eventually set extra = true for use with TTSP 122 - see explanation in TTSPmode 149.
Set symmetric = true, if you can do with a symmetric matrix. This makes calculations faster.

If you want to do a N x M matrix rather than N x N, use Matrix2 77.

Syntax: Matrix(NL: TIntegerList 133; extra,symmetric: boolean): TMatrix 146

2.6.23 Matrix2

This method calculates a matrix, based upon the nodes in NL1 and NL2. Calculations are fastest if NL1 is the smallest list.

See also Matrix 76.

Syntax: Matrix2(NL1, NL2: TIntegerList 133): TMatrix 146

2.6.24 MatrixBuffer

When calculating matrices for a small area in a big street network, it is possible to speed up calculations by restricting calculations to the relevant part + a buffer.
Experience show a factor 2 can be obtained in the best case.

If the buffer gets specified too small, you risk not finding the correct route or even not finding a route at all.

We recommend using 5 km for urban areas, in more rural areas use a larger value.

If you use 0 or a negative value, the whole network is considered (default).

Applies to: Matrix 51, Matrix2 77, MatrixDyn 51, MatrixDyn2 77, MatrixOut 77, MatrixDynOut 77 and MatrixPOut 78.

Default: 0 km

Property MatrixBuffer: double;

2.6.25 MatrixDyn

Same method as Matrix 76, just with locations instead of nodes:

Syntax: MatrixDyn(LL: TLocationList 134; extra, symmetric: boolean): TMatrix 146

2.6.26 MatrixDyn2

Same method as Matrix2 77, just with locations instead of nodes:

Syntax: MatrixDyn2(LL1, LL2: TLocationList 134): TMatrix 146

2.6.27 MatrixDynOut

Same method as MatrixOut 77, just with locations instead of nodes.

It also adds an additional parameter nearestopen, which updates locations in LL1 and LL2 where needed by making calls to NearestOpenDyn 79.

Syntax: MatrixDynOut(filename: string; GF: TGISformat 144; LL1, LL2: TLocationList 134; SL1, SL2: TStringList 137; dist, time, cost, symmetric, routeobject, nearestopen: boolean)

2.6.28 MatrixOut

This method calculates a matrix, based upon the nodes in NL1 and NL2.

SL1 and SL2 contains strings, identifying the records. This can be as simple as the record ID or another text.

SL1 and NL1 need to hold the same amount of items.
SL2 and NL2 need to hold the same amount of items.

Optionally SL1 and SL2 can be nil, then list index is used in the output.
If SL1 or SL2 contains all integers, the field type in the output is generated accordingly.

Dist, time and cost can be set to false/true to determine which fields should be included in the output.
Set routeobject = true, if you want the route to be part of the output.

If symmetric is true, NL2 should be the same as NL1 and only combinations in one direction between members in NL1 is part of the output.

If threads 82 > 1, symmetric is ignored / is always false.

Output files can get very big if you have many items in the lists, especially if routeobject is also true.
See the notes about TGISwrite 104.

Syntax: MatrixOut(filename: string; GF: TGISformat 144; NL1,NL2: TIntegerList 133; SL1,SL2: TStringList 137; dist, time, cost, symmetric, routeobject: boolean)

2.6.29 MatrixPOut

Same method as MatrixDynOut 77, just with positions instead of locations.
This means you should use the coordinate of TLocationList 146 items, rather than the locations.

It also adds an additional parameter offroadspeed (km/h), which allow you to include the offroad part in the output. If speed=0, then it is skipped.

If nearestopen is "active" for a specific element (i.e. another element is used as the starting point, rather than the nearest), then the offroad part is skipped.

Syntax: MatrixDynOut(filename: string; GF: TGISformat 144; LL1, LL2: TLocationList 134; SL1, SL2: TStringList 137; dist, time, cost, symmetric, routeobject, nearestopen: boolean; offroadspeed: double)

2.6.30 MaxCost

This property can be used to restrict the size of isochrones, matrices etc.
Unit is whatever is used as cost criteria: Cost, time or distance.

Default: 0

Type: TCost 143

2.6.31 MaxSpeed

This property can be used to limit the speed of all links in the network, if you are calculating for a vehicle that can not go as fast as is otherwise possible.

It only affects the route choice when working in Fastest 81 mode.

Default: 0

Type: TCost 143

2.6.32 Nearest

This method will locate the nearest item in NL, calculated from a node. It returns the index of the item.

If returning -1, nothing was found.

Syntax: Nearest(node: integer; NL: TIntegerList 133): integer

2.6.33 NearestDyn

This method will locate the nearest item in LL, calculated from the location. It returns the index of the item.

If returning -1, nothing was found.

Syntax: NearestDyn(Loc: TLocation 146; LL: TLocationList 134): integer

2.6.34 NearestOpen

This method will find the nearest open node, from a starting node. An open node is one where at least one of the connected links is open for driving.

Syntax: NearestOpen(node: integer; var NearestNode,NearestLink: integer; var cost: TCost 143)

2.6.35 NearestOpenDyn

This method will find the nearest open link and node, from a starting location. An open node is one where at least one of the connected links is open for driving.

Obstacles it ignores while doing so:
1) Oneway restrictions
2) Limits
3) Links being skipped
4) NoDriveThrough setting

Syntax: NearestOpenDyn(Loc: TLocation 146; var NearestNode, NearestLink: integer; var cost: TCost 143)

2.6.36 NodeCost

Returns the cost for a single node. This requires that an isochrone has been calculated previously.

Syntax: NodeCost(node: integer): TCost 143

2.6.37 NoDriveThrough

This property controls if the attribute 6 bit for NoDriveThrough areas should be respected in calculations.

Default: false

Type: boolean

2.6.38 RouteCost

This method returns the cost of a route, according to how Cost has been setup 81.

See also RouteTime 80 and RouteLength 55.

Syntax: RouteCost(route: TRoute 148): TCost 143

2.6.39 RouteFind

This method will return a TRoute list to a node, if a route / isochrone has already been calculated from another node.

2 examples with the same functionality:

TCalc.IsoCost(node1)
cost = TCalc.NodeCost(node2)
route = TCalc.RouteFind(node2)

cost = TRouteCalc.Route(node1,node2)
route = TCalc.RouteFind(node2)

IsoCost 72 method is faster if you have many calculations to do for the same node1. But class TRouteCalc 87 offers more fine-tuning options.

Syntax: RouteFind(node: integer): TRoute 148

2.6.40 RouteFindDyn

This method will return a TRoute list to a location, if an isochrone has already been calculated from another location.

2 examples with the same functionality:

TCalc.IsoCostDyn(location1)
cost = TCalc.RouteFindDyn(location2,route)

cost = TRouteCalc.RouteDynEx(location1,location2,route)

IsoCostDyn 72 method is faster if you have many calculations to do for the same location1. But class TRouteCalc 87 offers more fine-tuning options.

Syntax: RouteFindDyn(Loc: TLocation 146; var route: TRoute 148): TCost 143

2.6.41 RouteTime

This method returns the time (minutes) of a route, according to how Time has been setup 81.

See also RouteCost 80 and RouteLength 55.

Syntax: RouteTime(route: TRoute 148): TCost 143

2.6.42 SetCheapest

This sets the calculation target to be cost. At the same time it can be specified if turn restrictions should be included.

Syntax: SetCheapest(turncosts: boolean)

2.6.43 SetCost

This defines which cost array should be used in cheapest 81 route calculations.

Default: 0 as index

Syntax: SetCost(index: integer)

2.6.44 SetFastest

This sets the calculation target to be time. At the same time it can be specified if turn restrictions should be included.

Syntax: SetFastest(turncosts: boolean)

2.6.45 SetLimit

This allows you to restrict routing to links where a limit exists. See Limit 9 for further details.

LimitID is from 1 to 9 and value is from 0 to 255.

Default is value = 0, i.e. no restriction.

Syntax: SetLimit(LimitID: integer; value: byte)

2.6.46 SetShortest

This sets the calculation target to be distance. At the same time it can be specified if turn restrictions should be included.

If they are included, only restrictions (<0) are applied. Delays (>0) are ignored.

Default: SetShortest(false)

Syntax: SetShortest(turncosts: boolean)

2.6.47 SetSkipLinkList

You can set up a list of links that should be excluded in routing.

Default: no list

See also SetSkipNodeList 91.

Syntax: SetSkipLinkList(list: TBitArray 134)

2.6.48 SetTime

This defines which time array should be used in fastest 81 route calculations.

Default: 0 as index

Syntax: SetTime(index: integer)

2.6.49 SetTurn

This defines which turn restriction array should be used, when turncosts = true in SetCheapest 81, SetFastest 81 or SetShortest 81.

Default: 0 as index

Syntax: SetTurn(index: integer)

2.6.50 SkipCulDeSacOptimization

This property controls if cul-de-sac optimization should be skipped during route calculations, increasing calculation time by 25-30%.

It can be used to make sure RouteDynApproach 90 calculations gives the same result as IsoCostDyn 72 followed by RouteFindDyn 80 - even in rare situations.

Default: false

Type: boolean

2.6.51 SmartInit

When doing short routes / small isochrones in very large networks (>2 million links), we have added this feature, which allows the routing engine to only initialize the network as it works it way through it, by using the spatial index for determing when new areas are visited.

It can improve performance by a factor 2-5 in such cases. The advantage disappears when length of routes reaches app. 50 km. For very long routes, it is even a disadvantage.

Default: false

Type: boolean

2.6.52 Starttime

This property is used for defining when a route calculation starts. Not implemented yet.

Type: TCost 143

2.6.53 Threads

Set this property to decide how many threads are used when calling these methods:

IsoCostMulti 73
IsoLinkDriveTime 73 (*)
IsoLinkServiceArea 74 (*)
IsoPoly 75
IsoPolyFast 76

Matrix 76
Matrix2 77
MatrixDyn 77
MatrixDyn2 77
MatrixDynCurbIsochrone 89

MatrixOut 77 (*)
MatrixDynOut 77 (*)

Methods that involve writing a lot to disk (*), do not always benefit much from running multi-threaded.
Some fileformats are even slower in multi-threaded mode.

The setting only applies in RW Net Pro. Valid values are 1 to 16. Default is 1.
We do not recommend using higher than the "number of cores - 1".

When running with multiple threads, additional TCalc objects are created internally.
This means a much higher amount of memory is allocated. This is especially something to be aware of with large street networks.

Progress events for these functions are disabled when threads > 1.

Type: integer

2.6.54 Turnmode

This read-only property returns if the object was created 69 with turnmode enabled.

Type: boolean

2.6.55 UTurnAllowed

Defines if U-turns are allowed when Turnmode = true.

False: All U-turns are banned.
True: All U-turns are allowed, unless banned through attribute 6 settings.

U-turns are always allowed on cul-de-sac 43 links.

Default: false

Type: boolean

2.6.56 Pro methods

2.6.56.1 AlphaShape

Alpha shapes is one of many views to create isochrones around a set of points.
It requires the presence of alphashape.dll or alphashape64.dll.

See also Isochrones - overview 16

Syntax: AlphaShape(PG: TPolyGeneration 136; SL: TStepList 134; filename: string; GF: TGISFormat 144);

2.6.56.2 CenterNode

This method finds the center node of a network, the one which minimizes this expression:

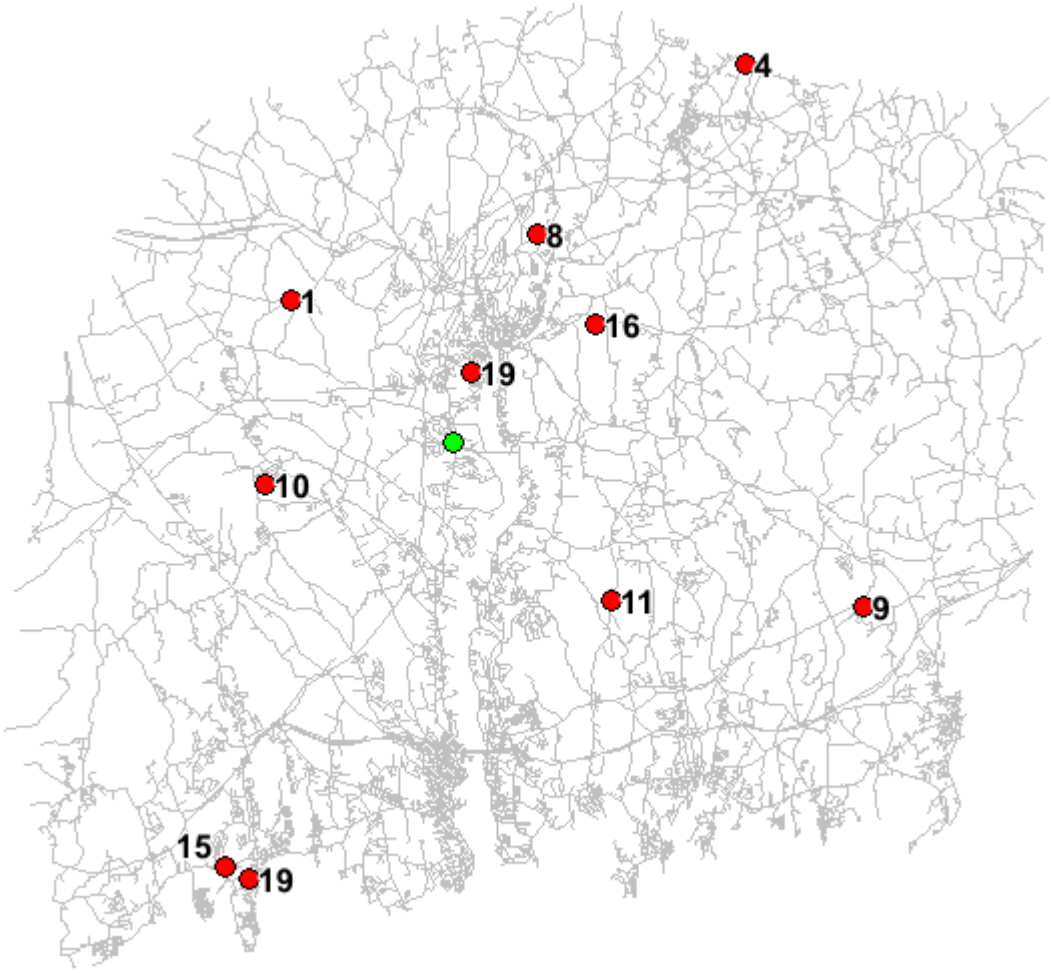
∑ Weight(node) * distance(node, CenterNode)

Parameter nodeweights need to have as many elements as NodeCount 51 +1 and contain 0's or positive weights.

The method is aimed at having not too many elements > 0 or it gets slow.

Syntax: CenterNode(var nodeweights: TCostArray 143): integer;

Example with 10 nodes with weights and the green centernode:



2.6.56.3 IsoCostDynApproach

This method calculates an isochrone from the location, but with a specific approach. The size of the isochrone can be restricted by setting MaxCost 78.

An error will be raised if location is on a loop link. Check with LoopLink 50 function in advance.

Requires turnmode 83 = true !

Syntax: IsoCostDynApproach(Loc: TLocation 146; Approach: TApproach 141)

2.6.56.4 IsoCostListDynApproach

This method calculates an isochrone from the location, but with a specific approach. It stops when all items in LL has been reached.

An error will be raised if location is on a loop link. Check with LoopLink 50 function in advance.

Requires turnmode 83 = true !

Syntax: IsoCostListDynApproach(Loc: TLocation 146; LL: TLocationList 134; Approach: TApproach 141)

2.6.56.5 LinkCostDynApproach

Returns the cost of getting to a specific location of a link and with a specific approach.
This method can be used after a call to IsoCostDynApproach 84 method.

2 examples with the same functionality:

TCalc.IsoCostDynApproach 84(location1,approach1)
cost = TCalc.LinkCostDynApproach(location2,approach2)

cost = TRouteCalc.RouteDynApproach 90(location1,location2,approach1,approach2)

If location1.link = location2.link you will have to use the TRouteCalc method.

Syntax: LinkCostDynApproach(loc: TLocation 146; approach: TApproach 141): TCost 143

2.6.56.6 MST

This method calculates a minimum spanning tree for the network, using Prims algorithm. Result is stored in links as 1's if the link is part of the tree.

By default length is used as cost, but by calling SetFastest 81 or SetCheapest 81, you can change to another criteria.

Oneway restrictions are not taken into account, neither is limits or the SkipLinkList 81.

Performance examples:
13,500 links: 0.5 sec
200,000 links: 165 sec

Syntax: MST(links: TBitArray 134);

Example output, subset of larger network:

2.6.56.7 RouteFindDynApproach

This method will return a TRoute list to a location, if an isochrone has already been calculated from another location.

2 examples with the same functionality:

TCalc.IsoCostDynApproach 84(location1,approach1)
cost = TCalc.RouteFindDynApproach(location2,approach2,route)

cost = TRouteCalc.RouteDynApproachEx 90(location1,location2,approach1,approach2,route)

If location1.link = location2.link you will have to use the TRouteCalc method.

Syntax: RouteFindDynApproach(loc: TLocation 146; approach: TApproach 141; var route: TRoute 148): TCost 143

2.6.56.8 SetSmoothing

Same functionality as here 102, but for method DriveTimeSimple 70.

Syntax: SetSmoothing(passes, rounded, deviation: integer);

2.6.56.9 SubNet

This method calculates which part of a network is a subnet. A subnet is defined as a part of the network, which isn't connected to the rest of the network. It can typically be an island without a ferry or a similar situation.

This is done with IgnoreOneWay 71 set to true temporarily, so one-way restrictions may in fact make even more links in-accessible. See function SubNetEx 91 on how to detect such situations.

IA returns the subnet ID for each link, while the method returns the number of subnets.

0 = main net
1, 2, 3... = sub nets

The main net is defined as the part of the network with node 1.

See also SubNetEx 91

Syntax: SubNet(var IA: TIntegerArray 145): integer

2.6.56.10 Tree

This method allows you to calculate a tree from a single starting point, typically used for verification of the street network.

This can be seen as a simpler version of TRouteCalc.TrafficAssignment 91. It only records which links are in use and all traffic is from a single node.

Syntax: Tree(filename: string; GF: TGISformat 144; startnode: integer; var NL: TIntegerList 133)

2.6.56.11 UnusedLinks

This method can be used for locating links which are not part of a route between any 2 nodes. This is done using the current cost criteria.

Invalid objects are not marked in the output.

Syntax: UnusedLinks(links: TBitArray 134)

2.7 TRouteCalc

This class inherits all properties and methods from TCalc 68 and adds methods and properties related to one-to-one route calculations, the A* algorithm is used.

In particular it adds these properties:

Alpha 87	Makes it possible to increase speed of calculations. No further data requirements.
Hierarchy 89	Makes it possible to increase speed of calculations, if hierarchical information is available in the attributes 6.
SkipNodeList 91	Allows you to avoid passing through certain nodes in the network.

2.7.1 Alpha

This property allows you to speed up calculations. By using 1.0 as value you will still get the actual best route, while increasing the value also increases the risk of getting a route that is closer to a straight line between start and end, but not necessarily the best route.

We recommend not increasing to more than 1.3. That may improve calculation speed with a factor

10 or so: Largest improvement is seen for long routes.

Default: 1.0

See also UpdateAlphas 61.

Type: TCost 143

2.7.2 NearestNDyn

This method finds the N nearest elements in LL2 for every element in LL1.

Typically LL1 has many elements, such as single addresses.
LL2 has much less elements, typically some centers (schools etc).

Routes are calculated from center to address for optimized speed of calculations. If you want the other direction, call SwapOneWay 57 before and after calling the function.

If maxcost 78 is set, it is used as cutoff and less than N elements may be found.

SL1 and SL2 can be nil or contain text identifiers for the output.

Dist, time and cost can be set to false/true to determine which fields should be included in the output.
Set routeobject = true, if you want the route to be part of the output (slows down calculations).

Output is a GIS file with one or more of these fields:
1. ID1
2. ID2
3. N
4. Distance
5. Time
6. Cost

Syntax: NearestNDyn(filename: string; GF: TGISformat 144; LL1, LL2: TLocationList 134; N: integer; SL1, SL2: TStringList 137; dist, time, cost, routeobject: boolean)

2.7.3 Route

Returns the cost of a route from node1 to node2.

You can call RouteFind 80 afterwards, if you want the actual route and not just the cost.

Syntax: Route(node1,node2: integer): TCost 143

2.7.4 RouteDyn

Returns the cost of a route from location1 to location2.

See also RouteDynEx 89.

Syntax: RouteDyn(loc1,loc2: TLocation 146): TCost 143

2.7.5 RouteDynEx

Returns the cost of a route from location1 to location2, including the actual route.

Syntax: RouteDynEx(loc1,loc2: TLocation 146; var Route: TRoute 148): TCost 143

2.7.6 Pro methods

2.7.6.1 Bridges

Detects if removing a link from the network, breaks up the network in separate subnets. The problematic links are marked with 1 in the IA array.

CulDeSac links are not marked, since testing for CulDeSac can easily be done with calling CulDeSac 43 for all links.
Links with oneway restrictions are not marked, use SubNetEx instead.

The function returns the number of elements set in the array.

See also SubNetEx 91

Syntax: Bridges(var IA: TIntegerArray 145): integer

2.7.6.2 CulDeSacCurb

Detects which links can not be used in curb approach mode (TTSPcurb 125), when U-turns are not allowed. The problematic links are marked with 1 in the IA array.

UTurnAllowed 83 and IgnoreOneWay 71 should be left to false, to find all problems.

Simple CulDeSac links are not marked, since testing for CulDeSac can easily be done with calling CulDeSac 43 for all links.
The function returns the number of elements set in the array.

The difference between the two functions, is this one also locates links where oneway restrictions are the cause of the problems.

Syntax: CulDeSacCurb(var IA: TIntegerArray 145): integer

2.7.6.3 Hierarchy

Set this property to true, if you want to enable hierarchical routing.

Default: false

Type: boolean

2.7.6.4 MatrixDynCurbIsochrone

Same as MatrixDyn 77, but including curb approach, for use with TTSPcurb 125.

This version is fastest if you have more than ~10 items in LL.
See also MatrixDynCurbRoute 90.

Requires turnmode 83 = true !

Syntax: MatrixDynCurbIsochrone(LL: TLocationList 134; extra: boolean): TCurbMatrix 143

2.7.6.5 MatrixDynCurbRoute

Same as MatrixDyn 77, but including curb approach, for use with TTSPcurb 125.

This version is fastest if you have less than ~10 items in LL.
See also MatrixDynCurbIsochrone 89.

Syntax: MatrixDynCurbRoute(LL: TLocationList 134; extra: boolean): TCurbMatrix 143

2.7.6.6 RoadNameTest

This function tests the roadname as part of the driving directions. The theory is, if a road name occur more than once on a route, it may be an error and the links between the 2 occurences might have the wrong name in the database. An example:

Link 1, 2, 3: Main Street
Link 4, 5: Old Road
Link 6, 7: Main Street

Here link 4 and 5 should probably have had the name Main Street as well. The function will report such instances and then leave it to the user to decide, if any edits should be performed.

RoadFileID defines the database with road names to use with the function and NumOfRoutes defines how many random routes to calculate as part of the test.

The fields in the generated GIS file are linkID, count of links that should be changed in the same way (2 in the example above), present roadname and suggested roadname. Generally, a low number of links to be changed, indicates a higher likelihood, that it is a required change. Using a filter of count<20 is a good idea, before viewing the output. Roundabouts are always skipped in the output.

Syntax: RoadNameTest(filename: string; GF: TGISformat 144; RoadFileID, NumOfRoutes: integer);

2.7.6.7 RouteDynApproach

Returns the cost of a route from location1 to location2, but with specific approach at both locations.

See also RouteDynApproachEx 90.

Syntax: RouteDynApproach(loc1,loc2: TLocation 146; Approach1,Approach2: TApproach 141): TCost 143

2.7.6.8 RouteDynApproachEx

Returns the cost of a route from location1 to location2, but with specific approach at both locations. Actual route is also included in the output.

Syntax: RouteDynApproachEx(loc1,loc2: TLocation 146; Approach1,Approach2: TApproach 141; var Route: TRoute 148): TCost 143

2.7.6.9 SetHierarchyLevel

Sets the 4 hierarchy parameters for use in hierarchical routing. Values should be expressed in km.

Input requirement: h2 >= h3 >= h4 >= h5 >= 0.
By default all parameters are set to infinite, meaning no hierarchy is applied.

We have executed tests with TomTom (netbclass field) and Navteq (func_class field) databases and recommend these values:

	Km
TomTom	130, 120, 100, 22
Navteq	145, 90, 40, 7

Tests were executed on UK data with a large number of random routes. Compared to not using a hierarchy, calculations were 6 times faster with TomTom data and 11 times faster with Navteq data. Navteq has better hierarchy attributes and a little less details in the network, hence the differences.

For short routes (<50 km) there is only little difference between using a hierarchy or not, while calculation of longer routes (>400 km) in the UK may be as much as 20-40 times faster (Navteq) and 6-30 times faster (TomTom).

Syntax: SetHierarchyLevel(h2, h3, h4, h5: double)

2.7.6.10 SetSkipNodeList

You can set up a list of nodes that should be excluded in routing.

Default: no list

See also SetSkipLinkList 81.

Syntax: SetSkipNodeList(list: TBitArray 134)

2.7.6.11 SubNetEx

Detects if a route between any 2 nodes can only be found when going in one of the directions. The links with the problematic one-way restrictions are identified and marked with 1 in the IA array. If any links are marked, it means the whole network isn't strongly connected.

The function returns the number of elements set in the array.

See also SubNet 87 and Bridges 89

Syntax: SubNetEx(var IA: TIntegerArray 145): integer

2.7.6.12 TrafficAssignment

This method is for assigning traffic to a street network.

Key input is TL, which holds traffic as volume between two pairs of coordinates. All traffic is allocated to the street network, using the all-or-nothing principle. For each link it keeps track of the total volume in both directions.

The map shows traffic from the red dot to all the blue dots. Width of line corresponds to volume.

Errors is used for keeping track of records within TL for which no route could be calculated. If Errors is unassigned, no records are marked.

Output contains these fields:
1. Link ID
2. Volume in forward direction
3. Volume in reverse direction

See also TrafficAssignmentDyn 92

Syntax: TrafficAssignment(filename: string; GF: TGISformat 144; TL: TTrafficList 134; var Errors: TIntegerList 133)

2.7.6.13 TrafficAssignmentDyn

This is the same method as TrafficAssignment 91, but it uses dynamic segmentation which means volumes are assigned to partial links and locations are used internally, instead of nodes.

Output contains these fields:
1. Link ID
2. Start percent
3. End percent
4. Volume in forward direction
5. Volume in reverse direction

Syntax: TrafficAssignmentDyn(filename: string; GF: TGISformat 144; TL: TTrafficList 134; var Errors: TIntegerList 133)

2.8 TDrivingDirections

This class can be used for creating driving directions (turn left/right etc), but also simpler setups aimed at just mapping. Output goes to a TGISwrite 104 instance.

The 4 main methods:
Route 95, RouteList 96, RouteDyn 95 and RouteListDyn 96.

Route calculation properties

SortedIndex 96 allows you to visit the location in a different order than natural. Typically as a result from TTSP 122 / TTSPcurb 125 calculations.

RoundTrip 95 should be set, so it matches TTSP.mode 149 if used in combination with TTSP 122 / TTSPcurb 125.

OffRoadSpeed 94 can be used with RouteDyn and RouteListDyn, when coordinates are present in the LocationList.

SideInArray 96 and SideOutArray 96 can be used to define approach, when used in combination with TTSPcurb 125.
They can also be populated by calling CalcSideInOutArray 93, if sequence is known.

Output properties

The key property controlling the kind of output is ConcatenationMode 94.

These 7 properties control if each field should exist in the output or not:
Cost 94, Dist 94, Time 97 and Speed 97
TotalCost 97, TotalDist 98 and TotalTime 98

Dist and TotalDist are always possible, while Cost, Speed and Time require that the parent TRouteCalc 87 is set up correctly (see SetTime 82 and SetCost 81).

Speed, cost and total cost are disabled by default. The rest are enabled.

These 3 properties control a possible time stamp field in the output:
StartTime 97, StopTime 97 and TimeStampFormat 97.

RoadFileID 95 is used for defining which road names should be used.

ViaList 98 is for including a textual description of the locations / nodes.

Driving directions properties

These properties are only relevant for mode cmDrivingDirections 142:

POI 95
RoundAboutCounting 95
SharpTurn 96
TurnText 98

2.8.1 Create

When creating an instance of TDrivingDirections, it is required to specify a TRouteCalc instance.

Syntax: Create(Calc: TRouteCalc 87);

2.8.2 CalcSideInOutArray

This method is for preparing SideInArray 96 and SideOutArray 96 with optimum values (avoiding U-turns as much as possible), when the elements in LL is already in the correct sequence.

It will test all possible combinations, so calculation time increases if LL has many elements or the elements are far apart.
As an example 100 locations takes 3 second, while 500 locations take 13 secs on the sample

street network.

It should be used before calling RouteListDyn 96.

Syntax: CalcSideInOutArray(LL: TLocationList 134)

Only available in the Pro version.

2.8.3 ConcatenationMode

This key property controls the kind of output performed when calling one of the methods.

Default: cmDrivingDirections

Type: TConcatenationMode 142

2.8.4 Cost

This property controls if cost should be part of the output.

Default: false

Type: boolean

2.8.5 Dist

This property controls if dist should be part of the output. Format of field is determined by DistanceUnit 94.

Default: true

Type: boolean

2.8.6 DistanceUnit

When generating output, you can use this property to use miles & mph instead of km & km/h.

Default: duKm

Type: TDistanceUnit 143

2.8.7 OffRoadSpeed

This property can be used to define the speed while moving from the exact coordinates (which are off road) to the nearest link.

It can only be used in combination with method RouteListDyn 96. The LL parameter need to have both coordinates and locations defined internally. This is done by adding coordinates first and then use TSpatialSearch.NearestLocationSimpleList 66.

If it is 0 and ConcatenationMode = cmCompactOffRoad, it is the same as using cmCompact.

A typical value would be 5 km/h, for walking speed.

Default: 0

Type: TCost 143

2.8.8 POI

It is possible to define a list of POI (Points-Of-Interest), that you want included in the output. For each link part of the result, it is checked if it contains any POI.

POI may be roadside signs, petrol stations etc. They are not possible during roundabouts.

Default: nil

Type: TPOIList 134

2.8.9 RoadFileID

This property is used to describe which roadname 53 file is used for the driving directions.

If ConcatenationMode = cmDrivingDirections, it needs to be set.
If ConcatenationMode = cmSeparate, it can be set and shall then be included in the output.
For other modes, the roadname is not part of the output.

Default: 0

Type: integer

2.8.10 RoundAboutCounting

This property controls how exit links are counted as part of driving directions in roundabouts.

False: Only exit links are counted
True: All links are counted

Default: false

Type: boolean

2.8.11 RoundTrip

This property controls if the output should be generated as a round trip (A-B-C-A) or not (A-B-C).

If you call method Route 95 or RouteDyn 95 (2 points only), you may like to set it to false first.

Default: true

Type: boolean

2.8.12 Route

Same method as RouteList 96, just with 2 nodes and no need to setup a list of nodes.

Syntax: Route(output: TGISwrite 104; node1,node2: integer)

2.8.13 RouteDyn

Same method as RouteListDyn 96, just with 2 locations and no need to setup a list of locations.

Syntax: RouteDyn(output: TGISwrite 104; loc1,loc2: TLocation 146)

2.8.14 RouteList

This method calculates a route between all the nodes in NL and writes the result to output.

Syntax: RouteList(output: TGISwrite 104; NL: TIntegerList 133)

2.8.15 RouteListDyn

This method calculates a route between all the locations in LL and writes the result to output.

See also OffRoadSpeed 94.

Syntax: RouteListDyn(output: TGISwrite 104; LL: TLocationList 134)

2.8.16 SharpTurn

If this property is >0, it is possible to trigger a turn description in the output even when the street name doesn't change, but the road makes a clear turn at an intersection. Just define how sharp the turn should be. Suggested value is 60-75 degrees. This only applies to sharp turns at intersections - not halfway down a link.

Default: 0

Type: Integer

2.8.17 SideInArray

Set this property in combination with TTSPcurb 125 optimization, to control how locations are approached (in-bound).

Default: nil

Type: TApproachArray 141

Only available in the Pro version.

2.8.18 SideOutArray

Set this property in combination with TTSPcurb 125 optimization, to control how locations are approached (out-bound).

Default: nil

Type: TApproachArray 141

Only available in the Pro version.

2.8.19 SortedIndex

This property controls the order of the nodes / locations in the output. This is typically the output from TTSP.SortedIndex 124.

Alternatively you can setup your own TIntegerArray. It should be zero-indexed and contain all values from 0 to Count-1 only once, starting with 0.

Default: nil

Type: TIntegerArray 145

2.8.20 Speed

This property controls if speed should be part of the output. Format of field is determined by DistanceUnit 94.

Default: false

Type: boolean

2.8.21 StartTime

This property defines when time stamps start in the output and is defined as a fraction of a day.

Default: 0

Type: double 141

2.8.22 StopTime

This property defines when time stamps stops in the output and is defined as a fraction of a day. If StartTime 97 <> 0, it is ignored.

Default: 0

Type: double 141

2.8.23 Time

This property controls if time should be part of the output.

Default: true

Type: boolean

2.8.24 TimeStampFormat

This property controls the format for time stamp in the output. Works in connection with StartTime 29 and StopTime 97.

Default: tfSkip

Type: TTimeStampFormat 149

2.8.25 TotalCost

This property controls if total cost should be part of the output.

Default: false

Type: boolean

2.8.26 TotalDist

This property controls if total dist should be part of the output. Format of field is determined by DistanceUnit 94.

Default: true

Type: boolean

2.8.27 TotalTime

This property controls if total time should be part of the output.

Default: true

Type: boolean

2.8.28 TurnText

If this property is defined an additional field is added to the output with textual description of the turns instead of just the values from 0 to 379.

Default: nil

Type: TTurnTexts 137

2.8.29 ViaList

This is for including textual descriptions and / or a fixed service time for each of the locations.

Default: nil

Type: TViaArray 150

2.9 TVoronoi

This class is used for generating Voronoi polygons and Delaunay triangulations. A detailed description of these can be seen in Wikipedia: Voronoi & Triangulation

The primary target is calculation of service areas and drivetime isochrones.

The sample application shows how to do it for isochrones and service areas. The other modes are done in a similar fashion.

You can also use the class independently from the routing functions, if you create and populate the PolyGeneration parameter on your own.

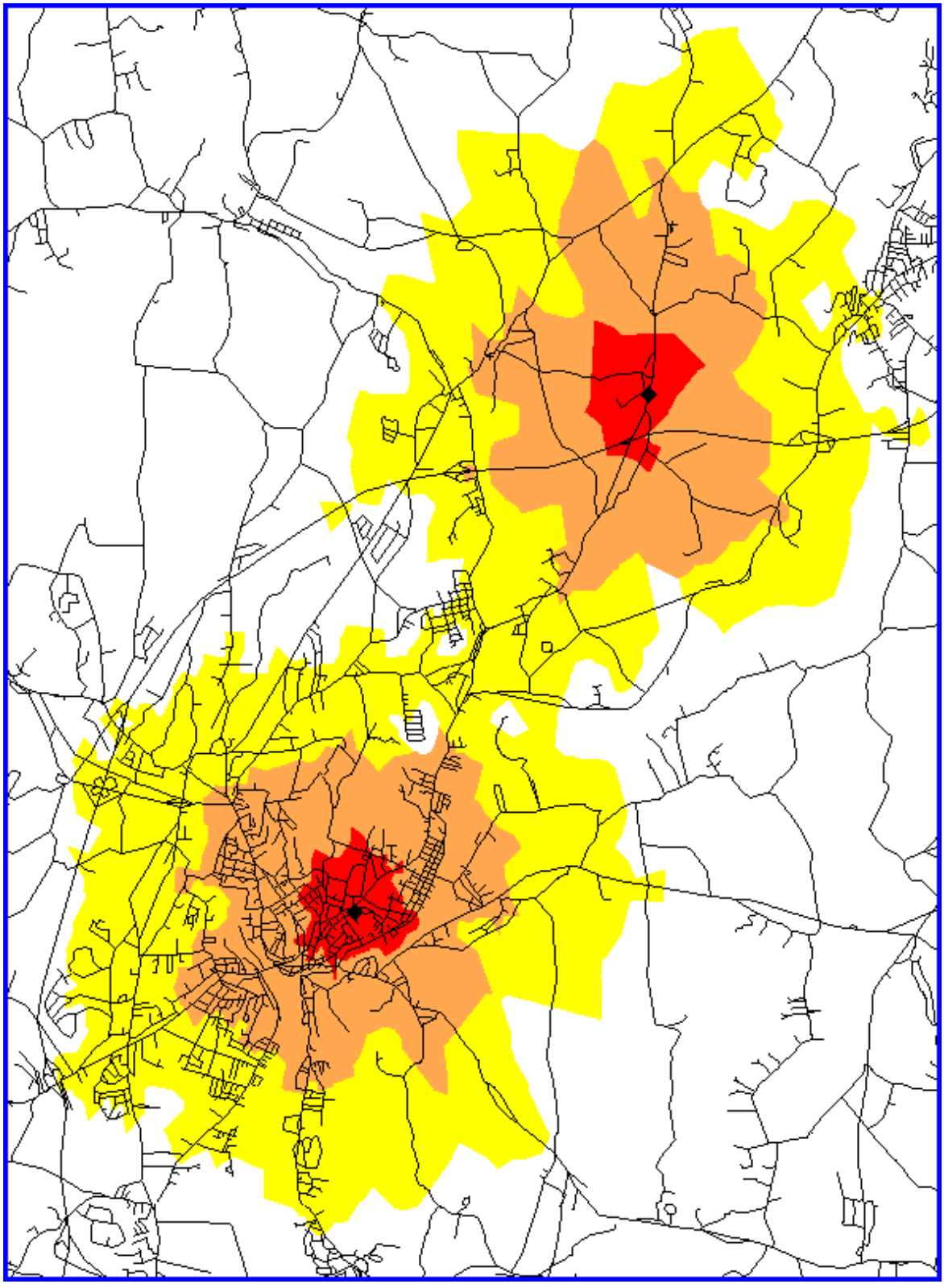
Properties relevant for each mode:

Mode	GISwrite 101	PolyGeneration 102	Slope 103	Zfieldname 104	Doughnut 101	IncludeHoles 102	StepList 103	Smoothing 102
vmTriangulationLine	X	X	X	X				
vmTriangulationSimple	X	X	X	X				
vmSimpleLine	X	X						

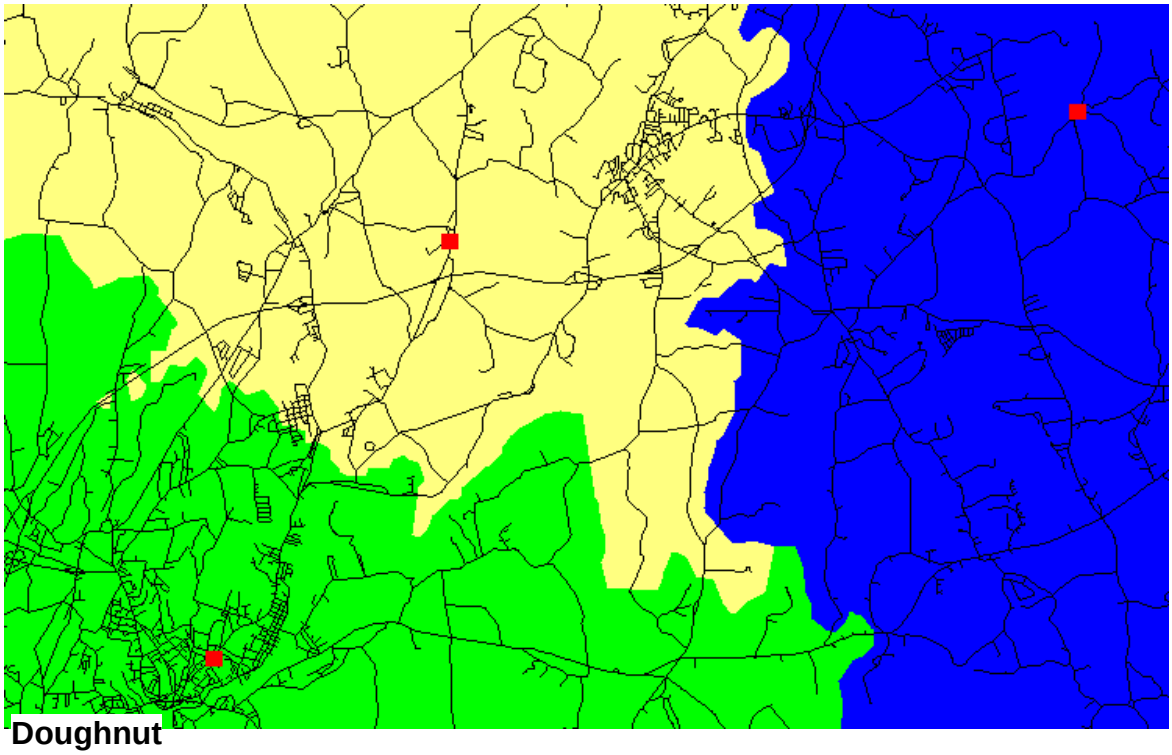
vmSimple	X	X						
vmIsochrone	X	X			X	X	X	X
vmServiceArea	X	X						

GISwrite, PolyGeneration and StepList need to be set. Slope, Zfieldname, Doughnut and IncludeHoles have default values and can be left unchanged.

Example of drivetime isochrone:



Example of service area calculation:



2.9.1 Doughnut

This property controls if output is generated as doughnut when mode = vmIsoChrone.

Example: If StepList 103 holds values 1, 2 and 3, you will get these 3 records in the output, depending upon the value:

false	true
0 - 1	0 - 1
0 - 2	1 - 2
0 - 3	2 - 3

Non-doughnut polygons (false) are overlapping.
Doughnut polygons (true) are not overlapping.

Default: true

Type: boolean

2.9.2 Execute

This is the main method for starting calculations.

Syntax: execute: integer

2.9.3 GISwrite

This is a reference to a TGISwrite instance, for holding output from the calculations.

Default: nil

Type: TGISwrite 104

2.9.4 IncludeHoles

This property controls if holes (and islands) are allowed in the output, when mode 102 = vmIsoChrone and doughnut 101 = false.

These two maps, show the same 1 km isochrone with IncludeHoles = true and false:

Default: true

Type: boolean

2.9.5 Mode

This key property controls the kind of calculation and output performed when calling execute 101.

Default: vmIsoChrone

Type: TVoronoiMode 150

2.9.6 PolyGeneration

This holds the main data used for the calculations.

Default: nil

Type: TPolyGeneration 136

2.9.7 SetSmoothing

This method allows you to smooth the output when mode 102 = vmIsochrone. Call it before calling Execute 101.

It is worth noting that the generated polygons do not get any more accurate, but may look more "visually" attractive on a map.

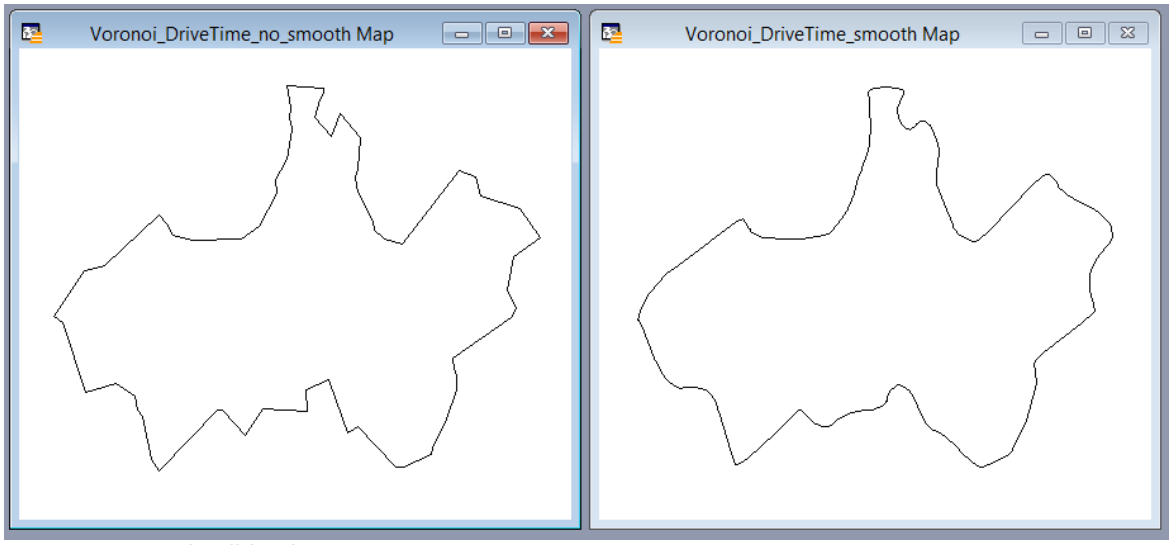
The number of nodes in the generated polygons will increase significantly, so use the function with

care.

Recommendations:

- Call it with only 1 step 103. With >= 2 steps there is a risk of unwanted overlaps.
- Leave doughnut 101 = false or you risk gaps between rings.

Example without and with settings (5,3,5):



Parameters and valid values:

Passes (1..5) defines how smoothed the output gets. A typical value is 3-4.

Rounded (3..6) defines how close the output fits the original input. 3 means any sharp angles almost disappear, while 5-6 for instance maintains the original look closer.

Deviation (0..15) allows you to remove some of the added nodes again to keep the total number of nodes lower without changing the look of the generated polygon too much.
Deviation is expressed in degrees. 1 degree will remove very few nodes, while 4-5 degrees will be good for most applications.

(0,0,0) is default value and means no smoothing at all. Use it for resetting.

Syntax: SetSmoothing(passes, rounded, deviation: integer; coord: TCoordinateUnit 142);

2.9.8 Slope

This is the slope of the triangulations. X, Y and Z (Cost) need to be in the same unit for it to work.

Default: false (which means not calculated in output)

Type: boolean

2.9.9 StepList

This property is used to define the steps used in mode vmIsoChrone. See Doughnut 101 too.

Default: nil

Type: TStepList 134

2.9.10 Zfieldname

Change this property if you want a different fieldname for the Z (cost) value. This is only relevant for the triangulation modes.

Default: "Cost"

Type: string

2.10 TGISwrite

This class is used for generating output from calculations. Typically as a GIS file with coordinates, but CSV and DBF files are also possible.

It is mostly used internally, but made available to users too. There is less error checking in this class, so you are to a higher degree responsible for what you are doing, if you use it directly. The sample uses it several times.

This table lists the 8 classes, which all has the same interface:

	Contains geographic data	File components
TGISwriteArray	X	
TGISwriteCSV		CSV
TGISwriteDBF		DBF
TGISwriteGeoJSON	X	GEOJSON
TGISwriteGML2	X	XML, XSD
TGISwriteKML2	X	KML
TGISwriteMIF	X	MIF, MID
TGISwriteMITAB	X	TAB, MAP, ID, DAT
TGISwriteSHP	X	SHP, SHX, DBF, PRJ, CPG

Despite most of the file formats can hold mixed object types 147 (SHP being the exception), we only support using a single object type.

Array
This is not file based opposed to the other formats. Not suited for very large datasets or you may hit an out-of-memory error.

CSV
This always uses , as field delimiter, no matter regional settings.
First line in the file contains the field names.

DBF
Stores codepage information in byte 29 in the header. This is standard, but not all software reads the information.
Limited to 2 GB.

GeoJSON
This is a string. Not suited for very large datasets or you may hit an out-of-memory error.
If you don't specify a filename for output, no file output is created.

GML2
2.1.2 format.

KML2

2.2 format.
You should only use KML if your coordinate system is already lat/long, WGS84.

MITAB
Requires MITAB.DLL or MITAB64.DLL on the path.
Limited to 2 GB.

SHP
CPG file is a simple text file with the codepage number. ArcGIS can read this information.
Limited to 2 GB.

2.10.1 AddField

Call this method to add fields after creating the header 109.

Syntax: AddField(Fieldname: string; Field: TGISField 144; Width, decimals: byte)

SHP / DBF do not support field names with more than 10 characters.

Width should be specified for fChar and fDecimal.
Width is a maximum of 254 for fChar in DBF, MIF, SHP and MITAB.
For KML and GML there is no limit and width is ignored.

Decimals should be specified for fDecimal.

2.10.2 Adding objects

There are 5 ways to add objects:

- AddPoint 106
- AddPoint2 106
- AddLine 106
- AddLine2 106
- AddObject 106 followed by AddSection 106 / AddSection2 107

In all methods the attributes for the object is added as a comma-delimited string.

Always use , as delimiter.
Always use . as decimal point.

When using gfChar fields, use Unicode and " around the text.
When using gfDate fields, use this format: YYYYMMDD
When using gfLogical fields, use this format for true: "T", "t", "Y", "y" or 1.
When using gfTime fields, use this format: HHMMSSsss (where sss = millisec, required !)
When using gfDateTime fields, use this format: YYYYMMDDHHMMSSsss (where sss = millisec, required !)

For SHP/DBF files, gfTime and gfDateTime are stored as text.
For TAB files, use of gfTime or fgDateTime means a version 9.00 file is generated.

Example:
A dataset consists of 9 fields, one of each type:
gfChar, gfInteger, gfSmallInt, gfDecimal, gfFloat, gfDate, gfLogical, gfTime, gfDateTime

Attribute string:
"test_text",1234567,123,123.45,123.45,19991231,1,123456000,19991231123456000

19991231123456000 = 31st of Dec 1999, 12:34:56.000

2.10.2.1 AddPoint

This adds a single point to the dataset.

Syntax: AddPoint(X, Y: double; Attrib: string)

2.10.2.2 AddPoint2

This adds a single point to the dataset.

Syntax: AddPoint2(P: TFloatPoint 143; Attrib: string)

2.10.2.3 AddLine

This adds a simple line to the dataset.

Syntax: AddLine(X1, Y1, X2, Y2: double; Attrib: string)

2.10.2.4 AddLine2

This adds a simple line to the dataset.

Syntax: AddLine2(P1, P2: TFloatPoint 143; Attrib: string)

2.10.2.5 AddObject

This adds the first part of a polyline / region object to the dataset.

Syntax: AddObject(NumParts: integer; MultiPolygon: boolean; Attrib: string)

After calling this method you should call AddSection 106 or AddSection2 107 as many times as stated in NumParts parameter.

If NumParts is 0, you will get an ungeocoded object in the dataset (works with point objects too). This is valid for all the formats, but we have seen some software not being able to deal correctly with SHP files with ungeocoded objects.

If you write to region output, have multiple outer rings and use GeoJSON, set MultiPolygon to true. For other situations, value do not matter.

NumParts can not be higher than 32000 for MIF and TAB formats.

2.10.2.6 AddSection

Call this method to add the actual coordinates in SegList:

Syntax: AddSection(Index: integer; var SegList: TFloatPointArrayEx 144)

For polyline datasets, the index parameter has no effect and you can just set it to 0.

For region / polygon objects it is important to store information about outer / inner rings (holes) correctly and different file formats has different requirements:

GML, KML, MITAB

Direction of coordinates: No requirements
Should be stored as first 1 outer and then N inner polygons.
This can be followed by further outer/inner sequences.
Index should be 0, 1, 2, 3 .... Change sign, if it is an outer polygon.

SHP
Direction of coordinates for outer polygons: Clockwise.
Direction of coordinates for inner polygons: Anti-clockwise.
Order of polygons and index parameter doesn't matter.

MIF and Array
No requirements

Common set of rules for all file formats
Direction of coordinates for outer polygons: Clockwise.
Direction of coordinates for inner polygons: Anti-clockwise.
Should be stored as first 1 outer and then N inner polygons.
This can be followed by further outer/inner sequences.
Index should be 0, 1, 2, 3 .... Change sign, if it is an outer polygon.
First and last coordinate should be the same for polygons or an error is raised.

2.10.2.7 AddSection2

Call this method to add a simple line object:

Syntax: AddSection2(Index: integer; X1, Y1, X2, Y2: double)

2.10.3 Brush

This property applies to regions in TAB / MIF output.

Default: BrushDefault 146

Type: TMIBrush 146

2.10.4 Close

Call this method to close the file, when you are done writing.

2.10.5 Codepage

This property describes the codepage used, when MIF, TAB, SHP, DBF and CSV files are generated.

KML and GML always uses UTF-8.
Array format uses native Unicode.

Default: System default codepage.

Type: TCodePage 141

2.10.6 CompactMIF

This property describes if MIF files should be written in a compact form, without any object drawing styles (Brush 107, Pen 109 or Symbol 109).

Default: False (meaning style is included by default).

Type: boolean

2.10.7 Coordsys

This property is used when writing MIF and TAB files.

Default: CoordSys Earth Projection 1, 104 (Lat/Long, WGS84).

Type: String

2.10.8 Drop

This method will close and delete any generated files.

2.10.9 EPSG

The EPSG property should be set if you write to GML or GeoJSON.

Default: 4326 (Lat/Long, WGS84).

Type: Integer 141

2.10.10 Filename

Fill in this property for all file types, except Array format.

Type: String

2.10.11 GeoJSON

When writing to GeoJSON format, this string contains the output.

Type: string

2.10.12 GISarray

When writing to array format, this object contains the output.

You should create the (empty) object first and then assign it to the TGA property.

Type: TGISarray 110

2.10.13 GreatCircleDist

This property should be set if your output is lat/long coordinates and you want to add additional nodes for every X km, so that the output is shown in your GIS application as great circles between start and end. A typical value could be 500 km, so this is only for very large objects.

It is the users responsibility only to use it with lat/long data or nonsense output may be generated.

Default value is 0.

Type: double 141

2.10.14 MITAB_supported

This function returns true, if writing to TAB is supported. This means if the library can find the relevant mitab.dll or mitab64.dll, depending upon the platform.

Type: boolean

2.10.15 OptimizePLinesSections

This property describes if consecutive matching polyline segment should be joined before output.

Default: False.

Type: boolean

2.10.16 Pen

This property applies to polylines and regions in TAB / MIF output.

Default: PenDefault 146

Type: TMIPen 146

2.10.17 PRJ

This property is used when writing the PRJ file in a SHP file collection.

Default:
GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137,298.257223563]],PRIMEM["Greenwich",0],UNIT["Degree",0.0174532925199433295]]

(Lat/Long, WGS84).

Type: String

2.10.18 StartHeader

Call this method when you are ready to create a new file.

As a minimum these properties should have been set in advance:
Array: TGA 108
CSV, DBF, KML2: Filename 108
GML2: Filename 108, EPSG 108
MIF, MITAB: Filename 108, Coordsys 108
SHP: Filename 108, PRJ 109

Syntax: StartHeader(NumFields: integer; ObjectTypes: TObjectTypes 147)

After calling this method you should call AddField 105 as many times as stated in NumFields parameter.

2.10.19 Symbol

This property applies to style of points in TAB / MIF output.

Default: SymbolDefault 147

Type: TMISymbol 147

2.10.20 WrittenRecords

This read-only property keeps track of how many records has been written, since calling StartHeader 109.

If no records has been written after a process, you can safely call method Drop 108 to delete the empty files.

Type: integer 141

2.11 TGISarray

This class holds output information from TGISwriteArray.

The sample application shows how to iterate through the whole data structure.

2.11.1 OT

Information about the object type in the array.

Type: TObjectTypes 147

2.11.2 MBR

Minimum bounding rectangle for the whole array.

Type: TFloatRect 144

2.11.3 Field

This is a list of fields in the array.

Field: array of TFieldInfo 110

2.11.3.1 TFieldInfo

```
TFieldInfo = record
  FieldType: TGISField 144
  Width, decimals: byte
  Name: string
end
```

2.11.4 Rec

This is the actual data in the TGISarray

Rec: array of TRec 110

2.11.4.1 TRec

This is each record in the TGISarray:

```
TRec = record
  Attr: array of Variant (array of Object in .NET version)
  Coord: array of TFloatPointArray 144
end
```

Attr is the attribute information for the object. Length of array is the same as that of Field 110.

Coord is the lists of coordinates making up the object. Multiple lists are required for regions with holes for instance. See AddSection 106 for details.

2.11.5 RecCount

The number of records in the array. Rec 110 may have room for more records, since it is extended in size in steps.

Type: Integer 141

2.11.6 Clear

Call this method to clear all memory allocated.

Part III

Optimization classes

3 Optimization classes

Optimization classes are not part of all levels:

RW Net Standard & Pro:
TTSP 122

RW Net Pro
TOptimizer 115
TTSPcurb 125

3.1 TOptimizer

This class holds various optimization methods:

- Cluster1 116
This is when customers should be grouped into clusters of a uniform load. Minimizing geometric size of clusters.

- Cluster2 117
This is when customers should be grouped into a number of clusters. Minimizing total distance between cluster center and customers.

- Cluster3 118
This is when customers should be grouped into a number of clusters. Minimizing the maximum distance between cluster center and customers (minimax strategy).

- District 119
This is when customers should be assigned to existing centers with a capacity. Minimizing distance between centers and customers.

See also TCalc.CenterNode 84 to locate center of a single cluster.

3.1.1 Assignment

This property is read-only and holds the result of a calculation.

Property Assignment: TIntegerArray 145;

3.1.2 Capacity

This describes capacity of each center.

Property Capacity: TCostArray 143;

3.1.3 Center

This property is read-only and holds the result of a calculation.

Property Center: TIntegerArray 145;

3.1.4 Cluster1

This function solves the problem of clustering customers (with demands 119), so load 121 within the cluster is lower than sCapacity and geometric size of cluster is minimized.

Cost is defined through a matrix 122, which can be calculated by TCalc.Matrix 76, TCalc.MatrixDyn 77, TNetwork.Matrix 51, TNetwork.MatrixDyn 51 or on your own.

Demand should be a much smaller number than sCapacity. Otherwise the algorithm isn't very good at finding a solution.

If Demand parameter is nil (not set), the algorithm assumes 1 for all customers. See also Swap 122.

The function returns number of clusters. Property Center 115 holds information about which customer is the center of the cluster.

Property	Dimension
Demand	No of customers
Matrix	No of customers x customers
Assignment (output)	No of customers
Center (output)	No of clusters
Load (output)	No of clusters

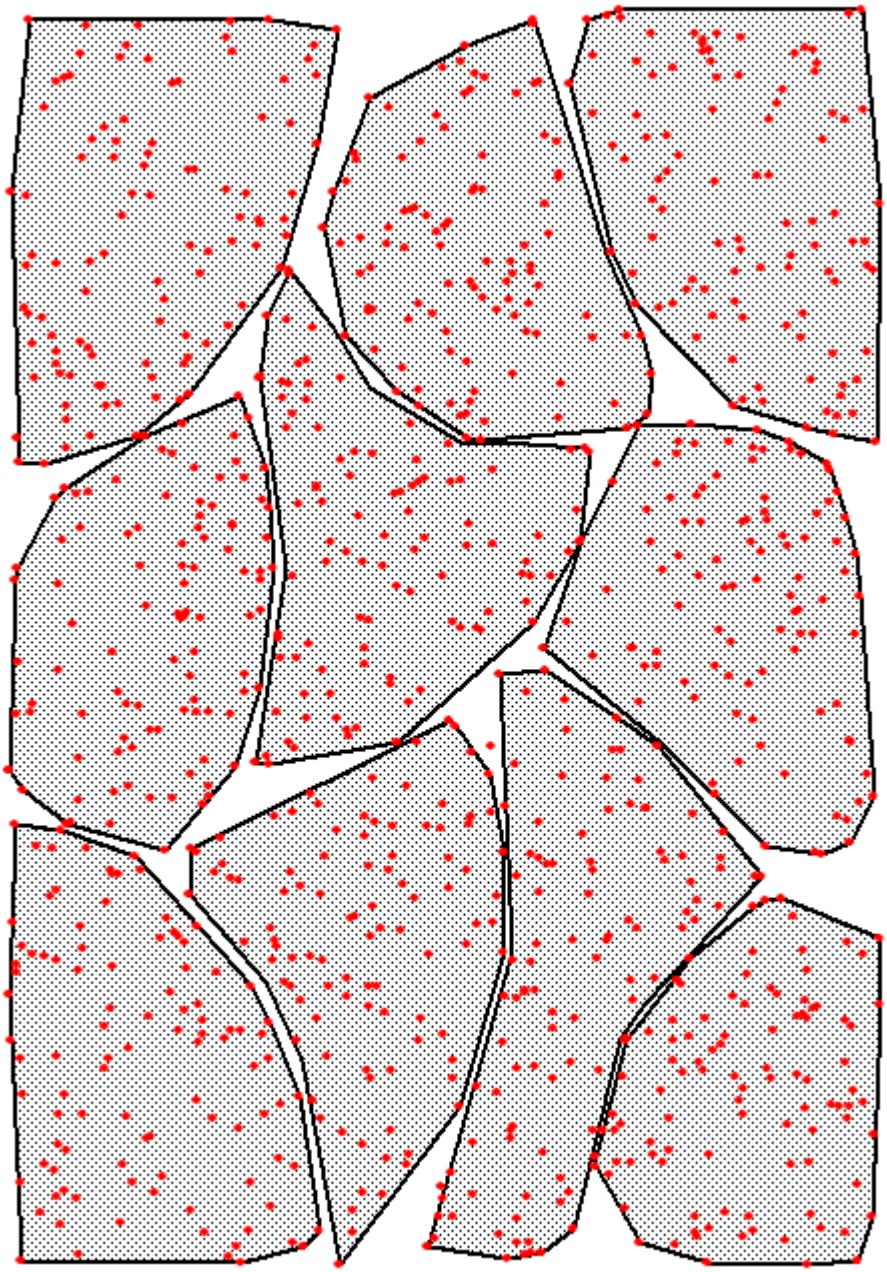
Sample calculation time (demand = 1 for all customers):

Customers	Clustersize	No of clusters	Calculation time (msec)
100	10	10	~0
1000	100	10	31
1000	10	100	250
10000	1000	10	2500
10000	100	100	22219 (22 sec)
10000	10	1000	219656 (~ 4 minutes)

With 50000 customers the matrix has reached a size of 10 GB - to give an indication of the largest instances that can be handled.
On win32 the limit is appr. 25000 customers.

Syntax: Cluster1(sCapacity: TCost): integer;

This is an example with 1000 customers and 10 clusters. Clusters are here highlighted as polygons:



3.1.5 Cluster2

This function solves the problem of clustering customers (with weights defined through demands 119 property), so total distance between cluster center and customers is minimized.

Cost is defined through a matrix 122, which can be calculated by TCalc.Matrix 76, TCalc.MatrixDyn 77, TNetwork.Matrix 51, TNetwork.MatrixDyn 51 or on your own.

If Demand parameter is nil (not set), the algorithm assumes 1 for all customers.

You can also call the function with NCluster = 1, if you just want to find the weighted center.

The function do not return any values, but populates these properties:

- Property Assignment 115 holds a number in the range 0 .. NCluster-1 about the cluster ID.
- Property Center 115 holds information about which customer is the center of the cluster.

Property	Dimension
Demand (used as weight)	No of customers
Matrix	No of customers x customers
Assignment (output)	No of customers
Center (output)	No of clusters

Sample calculation time:

Customers	No of clusters	Calculation time (msec)
100	10	32
1000	1	16
1000	10	47
1000	100	31
10000	10	3219
10000	100	3532
10000	1000	2422
20000	100	11891
20000	1000	8391

Syntax: Cluster2(NCluster: integer);

3.1.6 Cluster3

This function solves the problem of clustering customers, so maximum distance within each cluster between center and customers is minimized.

Cost is defined through a matrix 122, which can be calculated by TCalc.Matrix 76, TCalc.MatrixDyn 77, TNetwork.Matrix 51, TNetwork.MatrixDyn 51 or on your own.

You can also call the function with NCluster = 1.

The function do not return any values, but populates these properties:

- Property Assignment 115 holds a number in the range 0 .. NCluster-1 about the cluster ID.
- Property Center 115 holds information about which customer is the center of the cluster.

Property	Dimension
Matrix	No of customers x customers
Assignment (output)	No of customers
Center (output)	No of clusters

Sample calculation time:

Customers	No of clusters	Calculation time (msec)

100	10	~0
1000	1	16
1000	10	47
1000	100	31
10000	10	7031
10000	100	3968
10000	1000	3281
20000	100	19469
20000	1000	13750

Syntax: Cluster3(NCluster: integer);

3.1.7 Demand

This describes demand of each customer.

Property Demand: TCostArray 143;

3.1.8 District

This function solves the problem of assigning customers (with demands 119) to centers (with capacities 115), so
total load 121 is within the capacity and cost of travel is minimized for all customers.

Cost is defined through a matrix 122, which can be calculated by TCalc.Matrix2 77, TCalc.MatrixDyn2 77 or on your own.

Normally you will have many more customers than centers.

Demand should be a much smaller number than capacity. Otherwise the algorithm isn't very good at finding a solution.

Optimum results can only be achieved if demand is the same value for all customers. Such as 1.

If Demand parameter is nil (not set), the algorithm assumes 1 for all customers.

The function returns the cost of the solution.

The heuristics parameter can take two values, 1 and 2. With method 1 you will typically get the lowest
overall cost values, while method 2 gives "nicer" looking solutions, but requires more time to get the solution.
We recommend trying both and pick the result you prefer.

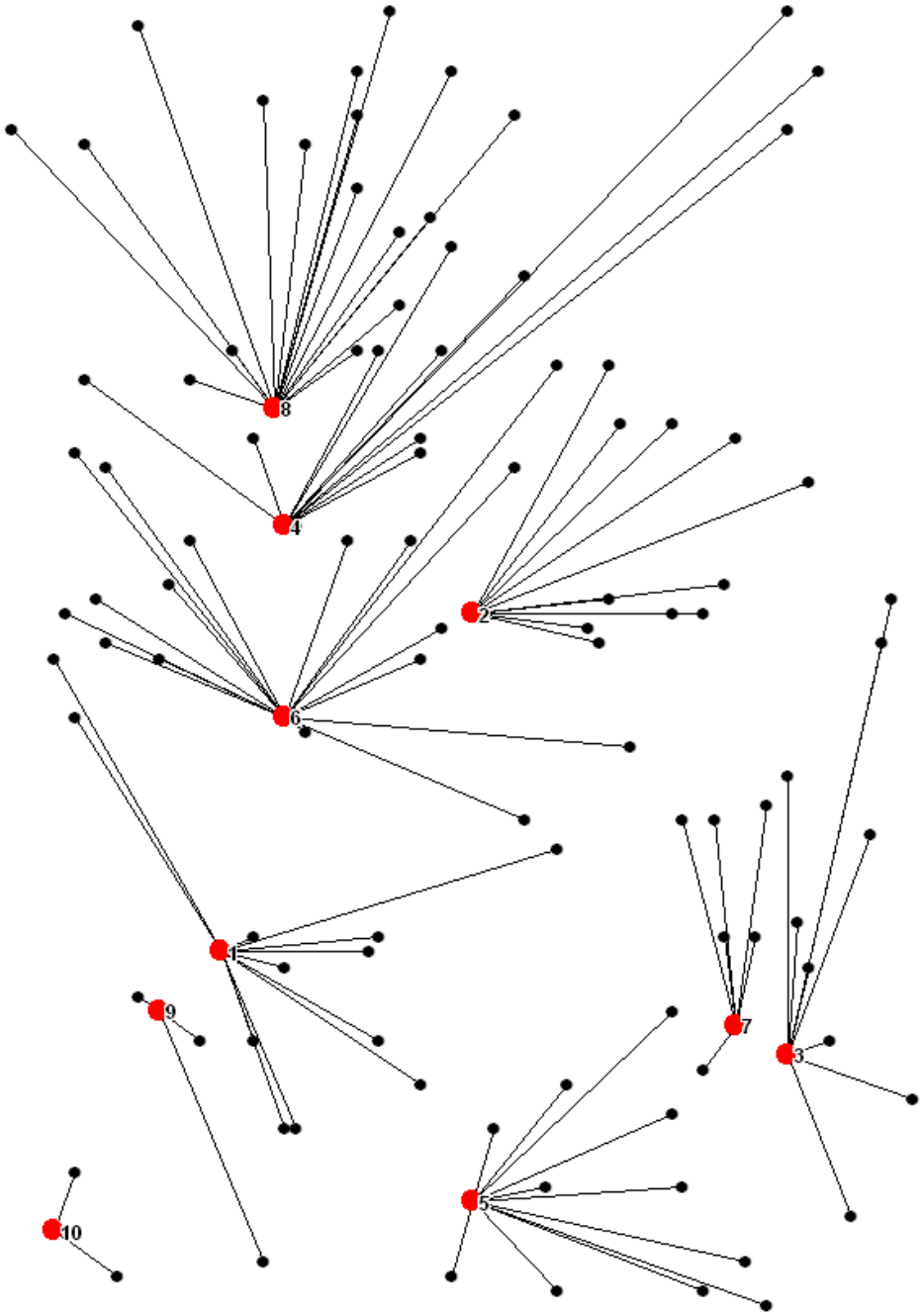
Property	Dimension
Capacity	No of Centers
Demand	No of Customers
Matrix	No of Centers x Customers
Assignment (output)	No of Customers
Load (output)	No of Centers
Unassigned	No of unassigned customers

Sample calculation time (demand = 1 for all customers, random center capacity, but sufficient in total):

Customers	No of centers	Calculation time - heuristic 1	Heuristic 2
100	10	16 ms	16 ms
1000	10	719 ms	734 ms
1000	100	1109 ms	860 ms
5000	10	220 sec (~ 4 minutes)	19 sec
5000	100	169 sec (~ 3 minutes)	240 sec (4 min)
5000	1000	15 sec	3754 sec (62 min)

Syntax: District(heuristic: integer): TCost 143;

This is an example with 100 customers, assigned to 10 centers with varying capacity. Customers and centers are connected with lines to make the result easier to view:



3.1.9 Load

This property is read-only and holds the result of a calculation, how much demand were assigned to each center.

Property Load: TCostArray 143;

3.1.10 Matrix

This is the input to the optimization, describing cost of matching centers and customers.

Property Matrix: TMatrix 146;

3.1.11 Swap

Set this to make the Cluster1 116 function perform additional steps of swapping and improving quality of solution a lot.
This makes it slower and should only be used with uniform demand 119.

Property Swap: boolean;

3.1.12 Unassigned

This property is read-only and reports how many customers wasn't assigned to a center in the District 119 method.

Property Unassigned: integer;

3.2 TTSP

This class is for travelling salesman optimization.

There is support for asymmetric matrices in the Pro version. This is especially important if you have many places in dense urban areas with many one-way restrictions.

In the Standard version, the input matrix is made symmetrical before optimization.

The algorithm uses random permutations, but you can control the randomness using the RandSeed 124 property.

Typically you will get the true optimum solution for instances with up to ~100 places to visit. With >100 places the quality of the solution degrades slowly.

Set properties MaxIterations 123, PercentWithoutImproveStop 124 and TimeLimit 125 to control for how long time the optimization should continue.

By default PercentWithoutImproveStop is the active criteria, while other properties (MaxIterations and TimeLimit) has default values, which means they are not active unless you set them.

Set Mode 124 before running the optimization 123.

It is also possible to monitor progress event 11, and eventually ask the algorithm to stop earlier.

After the optimization has finished, you have access to cost 122 and optimized sequence 124 (key result).

3.2.1 Cost

This read-only property holds the cost of the calculated sequence. It gets updated during execution too, if you monitor progress events 11.

Property Cost: TCost 143

3.2.2 Execute

This procedure starts the actual optimization.

Prepare the matrix using
- TNetwork.Matrix 51 or TNetwork.MatrixDyn 51 (for as-the-crow fly distances)
- TRouteCalc.Matrix 76 or TRouteCalc.MatrixDyn 77 (for real routes)

Eventually call MatrixPreProcess 123 to refine processing.

See also ExecuteFull 123.

Syntax: Execute(mat: TMatrix 146)

3.2.3 ExecuteFull

This works the same way as Execute 123, except it testes all possible combinations and ignores MaxIterations 123.

Calculation time increases fast as the dimension of the matrix increases. Just 10 elements means more than 3 million combinations (10!) and a calculation time of appr. 1 sec. With 12 elements you are reaching a couple of minutes.

Use it for testing if the solution found by Execute is the best possible. It almost always is with just 10 elements.

Syntax: ExecuteFull(mat: TMatrix 146)

3.2.4 MatrixPreProcess

Very often an optimization will result in some links being traversed more than once (either in the same or opposite directions). If there are multiple locations along that same link, it won't matter if the locations are visited first or second time the link is traversed. At least not from an optimization point of view. But for humans it feels most natural if all stops along the same link is just after each other.

This method will update the matrix, so short and long distances between locations are preferred to two medium distances. It essentially takes the square-root of the normalized cost: Matrix(*) = sqrt(Matrix(*) / max(Matrix(*))).

Example: Costs 0.25 + 0.75 has the same total as 0.5 + 0.5. After processing we will now see that sqrt(0.25) + sqrt(0.75) < sqrt(0.5) + sqrt(0.5), so 0.25 and 0.75 are preferred to 2x 0.5.

Syntax: MatrixPreProcess(var mat: TMatrix 146)

3.2.5 MaxIterations

The algorithm performs a number of iterations before it stops. We recommend these settings for maxIterations in TTSP class:

dimension < 500: 150000 iterations
500 < dimension < 2000: 150000 - dimension * 60 iterations
dimension > 2000: 30000 iterations

If threads 124 <> 1, you should multiply the number of iterations by the number of threads.

Expected run times:

100 elements in matrix: 10 sec
1000 elements in matrix: 4 minutes
2000 elements in matrix: 10 minutes

(Currently no recommendations for TTSPcurb class, but significantly higher values are required so far)

Default value: "large number"

Property MaxIterations: Integer

3.2.6 Mode

This describes how the optimization is performed.

Default value: tspRoundTrip

property Mode: TTSPmode 149

3.2.7 PercentWithoutImproveStop

This property controls when the optimization should stop.

If for instance the value is 50% and last improvement was after 20 sec, then it will stop after 30 sec, if no further improvements happened inbetween.

Set it to 0, to make it inactive.

Default: 100.

Property PercentWithoutImproveStop: integer

3.2.8 RandSeed

This property controls which seed is used for the optimizations, so the same calculation can be run again, if needed. Or different one.

Default: 1

Property RandSeed: integer

3.2.9 SortedIndex

This read-only property holds the optimized sequence after calculation has ended.

Property SortedIndex: TIntegerArray 145

3.2.10 Threads

This property defines how many threads are being used in the optimization phase.

The main advantage is reaching a slightly better solution in shorter time, since you can usually get an equally good solution running with threads=1 for longer time. I.e. no wonders from multiple threads !

It has little or no effect on problems with dimension < 100.

Default: 1 (can only be changed in Pro version)

Property Threads: integer

3.2.11 TimeLimit

This property controls for how many msec, the optimization phase runs.

Default: 0 (no limit)

Property TimeLimit: integer

3.3 TTSPcurb

This class is similar to TTSP, except it also takes curb (kerb) approach and U-turns into consideration.

It shares these methods / properties with TTSP: Cost 122, MaxIterations 123, Mode 124, PercentWithoutImproveStop, RandSeed, SortedIndex 124, Threads and TimeLimit.

After the optimization has finished, you have access to cost 122, optimized sequence 124, SideIn 126 and SideOut 126.

Known issue:
Using this method with only 3 elements may not always give the optimium result. This shall be fixed.

See also CulDeSacCurb 89

3.3.1 ExecuteCurb

This procedure starts the actual optimization.

Prepare the matrix using MatrixDynCurbIsochrone 89 or MatrixDynCurbRoute 90

Eventually call MatrixPreProcess 126 to refine processing.

UTurnCosts should either be:
= 0: Allowed
> 0: Allowed, but at an additional cost
< 0: Turn not allowed.

To avoid U-turns, use a high cost or negative value.

DesiredSide:
See the sample on how to setup the array. It can basically be apIgnore or apReverse / apForward. The 2 last ones depend upon left / right driving.

Syntax: ExecuteCurb(mat: TCurbMatrix 143; UTurnCosts: TCostArray 143; DesiredSide: TApproachArray 141)

3.3.2 ExecuteCurbFull

This works the same way as ExecuteCurb 125, except it testes all possible combinations and ignores MaxIterations 123.

Calculation time increases fast as the dimension of the matrix increases. Just 10 elements means more than 3 million combinations (10!) and a calculation time of appr. 1 sec. With 12 elements you

are reaching a couple of minutes.

Use it for testing if the solution found by Execute is the best possible. It almost always is with just 10 elements.

Syntax: ExecuteCurbFull(mat: TCurbMatrix 143; UTurnCosts: TCostArray 143; DesiredSide: TApproachArray 141)

3.3.3 MatrixPreProcess

Same method as TTSP.MatrixPreProcess 123, just with a different parameter.

Syntax: MatrixPreProcess(var mat: TCurbMatrix 143)

3.3.4 SideIn

This function returns from which side a location should be approached (in-bound).

Syntax: SideIn(index: integer): TApproach 141

3.3.5 SideInArray

This read-only property returns from which side all locations should be approached (in-bound). Can be used together with TDrivingDirections.SideInArray 96

Type: TApproachArray 141

3.3.6 SideOut

This function returns from which side a location should be approached (out-bound).

Syntax: SideOut(index: integer): TApproach 141

3.3.7 SideOutArray

This read-only property returns from which side all locations should be approached (out-bound). Can be used together with TDrivingDirections.SideOutArray 96

Type: TApproachArray 141

3.4 TTSPwindow

NOTE: You should use the special FleetEngine license instead of this class, since it is about to be removed starting from version 4.16

This class is similar to TTSP 122, but with some extra features:

- You can define one time window per job
- You can define a break
- Supports a matrix larger than the number of jobs, but using an index
- Can optimize for time (driving & waiting) or distance or a combination
- Standby jobs

There is only support for TTSPmode 149 = tspRoundTrip.

If no time windows are defined, the algorithm uses the normal TTSP 122 class internally.

Breaks and standby jobs are different in the sense, a standby job is within a time window and there can be many of them. The break on the other side is more flexible and is generally assigned where it fits the best with the desired time for the break. Both breaks and standby jobs can take place anywhere.

Unit for time is minutes and for distance it is km, but strictly speaking it can be anything as long as you stick to the same everywhere. Since unit for time is an integer, you may need to use seconds if much accuracy is needed.

3.4.1 Create

When creating a new instance, state how many jobs you have.

Syntax: Create(N: integer 141)

3.4.2 Execute

This function starts the actual optimization and returns a code:

```
  0: No error
 -1: Total of all work times do not fit in the whole day
 -2: Break is not within the planning period
 -3: Job N1 127 can not be reached from starting point within time window
 -4: Job N1 127 can not reach end point within time window
 -5: Job N1 127 & N2 127 overlap including drivetime between them
 -6: Length of the best plan exceeds the length of the planning period
 -7: No route found when testing all possibilities
 -8: Jobs with time windows do not fit
 -9: Jobs without time windows do not fit
-10: Resource ID has illegal value
-11: Matrix ID has illegal value
```

Output is best understood by reading the sample code and looking at the generated file.

Syntax: Execute: integer 141

3.4.2.1 N1

This property is set if the execute 127 method returns error code -3, -4 or -5.

Property N1: integer 141

3.4.2.2 N2

This property is set if the execute 127 method returns error code -5.

Property N2: integer 141

3.4.3 JobCount

This read-only property is set when you call create 127.

Property JobCount: integer

3.4.4 Input

When defining input, you should as a minimum set these properties:

DistMatrix 128, Jobs 128, TimeMatrix 129 and WorkStop 129.

The rest has default values, which are acceptable for many setups.

3.4.4.1 BreakStart

This is when the break starts, in minutes after midnight.

Default: 0

Property BreakStart: TTime 148

3.4.4.2 BreakTime

This is the length of the break, in minutes.

Default: 0 (meaning no break)

Property BreakTime: TTime 148

3.4.4.3 DepotMatrixID

This is the index for the depot into the distmatrix 128 and timematrix 129.

Default: 0

Property DepotMatrixID: integer 141

3.4.4.4 DistMatrix

This is a matrix of distances.

Property DistMatrix: TMatrix 146

3.4.4.5 Jobs

This is the main property with information about jobs. The array is automatically setup when you create the instance, but you still have to setup each individual job.

The array gets updated during optimization, so it is also part of the output.

Property Jobs: TJobsArray 145

3.4.4.6 MaxDepth

If JobCount 127 <= MaxDepth, an exhaustive search is performed, so the optimum solution is found. Maximum value is 20.

Default: 8

Property MaxDepth: integer 141

3.4.4.7 MaxIterations

This controls how many iterations are executed for non-exhaustive searches.

Default: 100 x JobCount 127.

Property MaxIterations: integer 141

3.4.4.8 TimeMatrix

This is a matrix of time.

Property TimeMatrix: TTimeMatrix 149

3.4.4.9 WeightDriveDistance

This is the weight on drive distance in the optimization.

Default: 0

Property WeightDriveDistance: Double 141

3.4.4.10 WeightDriveTime

This is the weight on drive time in the optimization.

Default: 1

Property WeightDriveTime: Double 141

3.4.4.11 WeightWaitTime

This is the weight on wait time in the optimization.

Default: 1

Property WeightWaitTime: Double 141

3.4.4.12 WorkStart

This property defines when the planning period starts.

Default: 0

Property WorkStart: TTime 148

3.4.4.13 WorkStop

This property defines when the planning period ends. Should be larger than WorkStart 129.

Default: 0

Property WorkStop: TTime 148

3.4.5 Output

Output is best understood by reading the sample code and looking at the generated file.

The Jobs 128 property is also part of the output.

All the output properties are read-only.

3.4.5.1 BestCost

This is the total cost, after optimization has finished.

Property BestCost: TCost 143

3.4.5.2 BestDist

This is the total distance, after optimization has finished.

Property BestDist: TCost 143

3.4.5.3 BestDriveTime

This is the total drive time, after optimization has finished.

Property BestDriveTime: TTime 148

3.4.5.4 BestStart

This is the starting time for the planning period, after optimization has finished.

Property BestStart: TTime 148

3.4.5.5 BestStop

This is the stopping time for the planning period, after optimization has finished.

Property BestStop: TTime 148

3.4.5.6 BestWait

This is the total waiting time, after optimization has finished.

Property BestWait: TTime 148

3.4.5.7 FirstDriveDistToNext

This is the distance from the depot to the first job.

Property FirstDriveDistToNext: TCost 143

3.4.5.8 FirstDriveTimeToNext

This is the time from the depot to the first job.

Property FirstDriveTimeToNext: TTime 148

3.4.5.9 SortedIndex

This is the key result of the optimization, that is the sequence of jobs.

Property SortedIndex: TIntegerArray 145

Part IV

Helper Classes

4 Helper Classes

These are classes that primarily are for input / output from the main classes.

4.1 TBaseList

Various generic lists are used throughout RW Net, see the sub-chapters for implementations:

TBaseList

This is a basic, unsorted list. T is the list item.

method Add(Item: T): Integer
method Clear
method Delete(Index: Integer)
method Extract(Index: Integer): T
method Insert(Index: Integer; Item: T)
property Capacity: Integer
property Count: Integer (read-only)
property Items[Index: Integer]: T

TBaseListSort adds these methods to TBaseList:

method IndexOf(Item: T): Integer
method RemoveDuplicates (calls Sort internally)
method Sort
property ReverseIndex: Boolean
property ReverseItems[Index: Integer]: integer (read-only)
property Sorted: Boolean (read-only)

4.1.1 TCoordCostSiteList

This is an implementation of TBaseListSort 133

List item: TCoordCostSite 142

4.1.2 TGPSMatchList

This is an implementation of TBaseListSort 133

List item: TGPSMatch 144

4.1.3 TImportErrorList

This class is an implementation of TBaseList 133 with errors that gets recorded during data import 25.

List item: TImportError 145

4.1.4 TIntegerList

This is an implementation of TBaseListSort 133

List item: integer

Adds two methods:

1) RemoveBlanks, which removes items that are 0.

2) SetFromBitArray, which creates a list of "true" elements in the TBitArray 134.

4.1.5 TLocationList

This is an implementation of TBaseListSort 133.

This is a list of not just TLocations 146, but also a corresponding TFloatPoint 143. Depending upon how the list is being used, the requirements regarding the location and coordinate part may be different.

See also Location2CoordinateList 50 and NearestLocationSimpleList 66.

List item: TLocationListItem 146

Additional methods:
Add1(Item: TLocation 146): integer
Add2(link: integer; percent: TPercent 148): integer
Add3(P: TFloatPoint 143): integer
Add4(Item: TLocation 146; P: TFloatPoint 143): Integer
RemoveStartEndPos. Removes all items, where percent = 0 or percent = 1.

4.1.6 TPOIList

This is an implementation of TBaseListSort 133.

List item: TPOI 148

4.1.7 TStepList

This is an implementation of TBaseListSort 133

List item: TCost 143

Adds function Max, which returns the largest item.

4.1.8 TTrafficList

This class is an implementation of TBaseList 133 for use in traffic assignment 91.

List item: TTraffic 149

Only available in Pro

4.2 TBitArray

This class is simply an array of boolean values, but with additional functions built-in.

It is more or less similar to BitArray in .NET and TBits in VCL.

4.2.1 Bits

This property allows you to get or set individual bits in the array.

Property: Bits[Index: Integer]: Boolean

4.2.2 CountFalse

This method returns the number of false in the array.

Syntax: CountFalse: integer

4.2.3 CountTrue

This method returns the number of true's in the array.

Syntax: CountTrue: integer

4.2.4 P_And

This calculates logical *and* with the B parameter.

Syntax: P_And(B: TBitArray)

4.2.5 P_Not

This calculates logical *not* of the whole array. I.e. switches all values between false and true.

Syntax: P_Not

4.2.6 P_Or

This calculates logical *or* with the B parameter.

Syntax: P_Or(B: TBitArray)

4.2.7 SetAll

Sets all elements to the specified value

Syntax: SetAll(Value: boolean)

4.2.8 SetAllFalse

Sets all elements to false.

Syntax: SetAllFalse

4.2.9 SetAllTrue

Sets all elements to true.

Syntax: SetAllTrue

4.2.10 SetFromIntegerArray

This sets the size automatically and assigns to true, when the elements of IA is different from 0.

Syntax: SetFromIntegerArray(IA: TIntegerArray)

4.2.11 Size

This properties specifies the size of the array. If the array is extended, new elements are initialized to false.

Property Size: Integer

4.3 TPolyGeneration

This class is normally only used as a place holder for output from IsoPoly 75 / IsoPolyFast 76 method, which is used as input to TVoronoi 98.

IsoPoly 75 creates the instance, but you should free it on your own.

It can be exported using ExportPolyGeneration 45 for viewing of the content.

It contains 2 public fields, which can be accessed directly:

```
CoordCostSiteList: TCoordCostSiteList 133
StartPoints: TFloatPointArray 144
```

4.4 TRandom

This is for generating pseudo random numbers, but implemented as a class so you have full control and can use it in threads too.
It is also independant of the compiler used.

It uses the same formula as used in Delphi:
http://en.wikipedia.org/wiki/Linear_congruential_generator

4.4.1 NextDouble

Returns a number, 0 <= x < 1.

Syntax: NextDouble: double;

4.4.2 NextInt

Returns an integer, 0 <= x < value.

Syntax: NextInt(value: integer): integer;

4.4.3 Randomize

Initializes the random number generator from the compiler built-in random seed generator.

Syntax: Randomize;

4.4.4 SetSeed

Define your own seed, so you repeat a certain sequence of random numbers.

Syntax: SetSeed(value: Int64);

4.5 TRoadClassSpeed

This class is used for storing a set of speeds related to each road class 6.

It is a fixed array of doubles with index 0 to 31. Default value is 60 km/h. Only values >0 are allowed.

It can be accessed directly using its index.

It has a single method for loading from an INI file in the same format as used by FleetEngine and RouteWare Studio.
Default speed is 60 km/h for undefined classes.

Syntax: LoadFromINI(filename, section: string);

Example: LoadFromINI('c:\fleetengine.ini' , 'Net1');

[net1]
Speed1 = 110
Speed2 = 90
etc.

4.6 TRoadClassTurnCost

Same as TRoadClassSpeed 137, except valid range is >=0 and default value is 0. Unit can be anything, but we recommend minutes.

Used in TurnAutoProcess 57.

4.7 TStringList

This parameter is slightly different, depending on the platform:

- .NET: List<string>
- Delphi: TStringList

4.8 TTurnTexts

This is a pre-populated array of strings, which you can use when generating driving directions 92.

All elements are accessible for reading / writing through property Items[], so you can modify them for your own liking.

Default values are in English.

Degrees	Text
0 - 22	Straight on
23 - 67	Slight turn to the left
68 - 112	Turn to the left

113 - 157	Sharp turn to the left
158 - 202	U-turn like
203 - 247	Sharp turn to the right
248 - 292	Turn to the right
293 - 337	Slight turn to the right
338 - 360	Straight on
361	Take exit 1 from roundabout
362	Take exit 2 from roundabout
363	Take exit 3 from roundabout
..	..
379	Take exit 19 from roundabout

Part V

Simple types

5 Simple types

These types are the simple ones, without a constructor / destructor.

5.1 Single

A single is a 4-byte floating point number. It is generally used for costs, distances etc. in RW Net.

5.2 Double

A double is a 8-byte floating point number. It is generally used for coordinates.

5.3 Word

Word means a 2-byte unsigned integer (uint16).
Range: 0 to 65,535.

5.4 Integer

Integer means a 4-byte signed integer (int32).
Range –2,147,483,648 to 2,147,483,647.

5.5 Int64

Int64 means a 8-byte signed integer.
Range: –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

This is typically used when working with unique identifiers in databases like TomTom, Navteq and OpenStreetMap.

5.6 TApproach

Enumeration: (apIgnore, apForward, apReverse)

5.7 TApproachArray

Array of TApproach 141

5.8 TCodePage

On .NET: Same as System.Text.Encoding

On other platforms: Word 141

See also Wikipedia

5.9 TColor

Same as integer 141 on .NET

5.10 TConcatenationMode

This enumeration describes the various modes for TDrivingDirections 92 output:

cmVeryCompact	The whole result as one record
cmCompact	All segments between two locations as one record
cmCompactOffRoad	As above, but with off road segments separately
cmDrivingDirections	As driving directions
cmSeparate	With all segments as separate records - very detailed and includes link ID

5.11 TCoordCostSite

This type is used as input to voronoi generation.

```
TCoordCostSite = record
                   Cost: TCost 143
                   Site: Integer 141
                   P: TFloatPoint 143
                 end;
```

If site = 65535 it means no nearest facility was in reach.

5.12 TCoordinateUnit

This enumeration informs about the coordinate units in use. It can only be set before importing 25 a dataset.

Geographic coordinates:
cuRad (radians, -pi - +pi, -pi/2 - +pi/2)
cuDeg (degrees, -180 - +180, -90 - +90)
cuGrad (grads, -200 - +200, -100 - +100)

Projected coordinates (SI units):
cuMm
cuCm
cuDm
cuM
cuKm

Projected coordinates (non-SI units):
cuPoint
cuInch
cuLink
cuFt
cuSurveyft
cuYard
cuChain
cuRod
cuMiles
cuNmi
cuUnknown

Data using geographic coordinates are checked during import for valid range.

By far the most usual ones are cuDeg and cuM.

5.13 TCost

This is used for cost, time, turn delays and speed of routes, links etc.

Alias for Single 141

5.14 TCostArray

Array of TCost 143

5.15 TCurbMatrix

A 3D array of TCost 143 elements.

See TTSPCurb 125 and MatrixDynCurbRoute 90 / MatrixDynCurbIsochrone 89.

5.16 TDistanceUnit

Enumeration: (duKm,duMiles)

5.17 TErrorCode

An enumeration:

ecDeleted	Object is deleted
ecNotGeoCoded	Object is not geocoded
ecNotPolyLine	Object is not a (poly)line, but type <value> (value only for SHP/TAB file)
ecMultiSection	Object has <value> sections
ecZeroOrOneVertices	Object has only <value> vertices
ecLoopLink	Object is a loop link
ecTooManyVertices	Object has >65535 vertices

Not all error codes are possible for both TAB, MIF and SHP files.

5.18 TFileKind

Enumeration: fkCSV, fkDBF

Use fkCSV when working with comma-separated files, such as CSV and MIF
Use fkDBF when working with DBF and DAT files.

5.19 TFloatPoint

This is a record describing a point:

```
TFloatPoint = record
  x,y: double 141
end
```

If you are working with spherical / geographic coordinates, use x for longitude and y for latitude.

5.20 TFloatPointArray

Array of TFloatPoint 143

5.21 TFloatPointArrayEx

```
TFloatPointArrayEx = record
  Pnt: TFloatPointArray 144
  Count: Integer
end
```

Count keeps track of how many positions in Pnt is in use.

5.22 TFloatRect

This is a record describing a rectangle:

```
TFloatRect = record
  xmin,ymin,xmax,ymax: double 141
end
```

5.23 TGISField

Field type identifier:

Enumeration: (gfChar, gfInteger, gfSmallInt, gfDecimal, gfFloat, gfDate, gfLogical, gfTime, gfDateTime)

5.24 TGISFormat

GIS format identifier:

Enumeration: (gfMIF, gfDBF, gfSHP, gfCSV, gfArray, gfMITAB, gfGML2, gfKML2, gfGeoJSON, gfMFAL)

gfMFAL is not available for .NET.

Output from gfGeoJSON is also stored as a string on the calling object:
TNetwork.GeoJSON 47
TSpatialSearch.GeoJSON 65
TCalc.GeoJSON 71
TGISwrite.GeoJSON 108

5.25 TGPSMatch

This record type is used for storing results for possible matches from function TSpatialSearch.NearestLink 66

```
TGPSMatch = record
  Loc: TLocation 146
  Distance: double
  DifBearing: double
  Reverse: boolean
  OneWayMisMatch: boolean
end;
```

Distance is in km. The smaller, the better
DifBearing is in degrees. The smaller, the better.
Reverse: True, if the record is for driving in the opposite direction of digitization.
OneWayMisMatch: True, if the road is oneway and it doesn't match the bearing.

5.26 TImportError

```
TImportError = record
  fileindex: integer, 0-based, refers to items in list of files 26
  linklocal: integer, 1-based, refers to a link inside a file
  link: integer, 1-based, refers to total sequence of links (internal ID)
  errorcode: TErrorCode 143
  value: integer
end
```

5.27 TIntegerArray

Array of integer 141

5.28 TJob

This record is used when defining a job, as part of TTSPwindow 126 class.

It consists of 2 parts, input and output.

Servicetime must fit within the time window, i.e. ServiceTime <= WindowStopTime - WindowStartTime

If Standby is true, the job has no "location" (MatrixID is ignored) and the length of the job includes drivetime before and after standby period.
Also the timewindow need to be specified and is used as total period (worktime is also ignored).

Output is best understood by reading the sample code and looking at the generated file.

```
TJob = record
  // input
  ServiceTime:     TTime 148     // minutes, default 0
  WindowStartTime: TTime 148     // timestamp, default 0
  WindowStopTime:  TTime 148     // timestamp, default 0
  MatrixID:        integer 141   // default index+1
  StandBy:         Boolean   // default false

  // output
  Wait:            TTime 148     // minutes
  BreakBefore:     TTime 148     // minutes
  BreakAfter:      TTime 148     // minutes
  StartTime:       TTime 148     // timestamp
  DriveTimeToNext: TTime 148     // minutes
  DriveDistToNext: TCost 143     // distance
end
```

5.29 TJobsArray

Array of TJob 145

5.30 TLocation

See network terminology 5 for details.

```
TLocation = record
  link: integer
  percent: TPercent 148
end
```

5.31 TLocationListItem

```
TLocationListItem = record
  loc: TLocation 146
  P: TFloatPoint 143
end
```

5.32 TMatrix

A 2D array of TCost 143 elements.

When used in optimizations it need to be square.

5.33 TMIBrush

This is used for defining region style in TAB / MIF files.

```
TMIBrush = record
  pattern: TMIBrushPattern 146
  fg_color : TColor 141 (foreground)
  bg_color : TColor 141 (background)
end
```

Predefined TMIBrush constants:

```
BrushDefault: Black outline, white fill
OutlineOnly: Black outline, no fill
```

5.34 TMIBrushPattern

This describes a MapInfo pattern from 1 to 71. How they look can be seen in MapInfo's documentation.

Type: Integer 141

5.35 TMILinePattern

This describes a MapInfo line pattern from 1 to 118. How they look can be seen in MapInfo's documentation.

Type: Integer 141

5.36 TMIPen

This is used for defining polyline and region linestyle in TAB / MIF files.

```
TMIPen = record
  Width: TMIPenWidth 147
```

```
  pattern: TMILinePattern 146
  Color: TColor 141
end
```

Predefined TMIPen constants:

PenDefault: Solid narrow line
PenInvisible: Invisible line (used by regions)

5.37 TMIPenWidth

This describes a MapInfo line width from 1 to 2047. See MapInfo's documentation for further documentation.

Type: Integer 141

5.38 TMISymbol

This is used for defining polyline and region linestyle in TAB / MIF files.

```
TMISymbol = record
  Shape: TMISymbolNo 147
  Color: TColor 141
  Size: TMISymbolSize 147
end
```

Predefined TMISymbol constant:

SymbolDefault: Small black dot

5.39 TMISymbolNo

This describes a MapInfo symbol style from 31 to 67. How they look can be seen in MapInfo's documentation.

Type: Integer 141

5.40 TMISymbolSize

This describes a MapInfo symbol size from 1 to 48. See MapInfo's documentation for further documentation.

Type: Integer 141

5.41 TObjectTypes

Enumeration:

otNone	No object
otPoint	Point
otPline	Polyline
otRegion	Region / Polygon

5.42 TPercent

See network terminology 5 for details.

Alias for double 141

5.43 TPOI

TPOI is used for "Points-Of-Interest" that can be included in driving directions 92. Name and location fields should be self-explanatory, while approach parameter can be used, if a POI can only be seen when driving in one direction.

If you know the coordinates of a POI, use TSpatialSearch.NearestLocation 66 to get both location and side of road.

This table shows how to translate from side to approach, if we assume a POI is only visible in the same side of the road as the vehicle is moving:

Side	Right-hand driving	Left-hand driving
-1 (left)	apReverse	apForward
+1 (right)	apForward	apReverse

If a POI is visible when driving in both directions, just set Approach to apIgnore.

```
TPOI = record
  Name: string
  Location: TLocation 146
  Approach: TApproach 141
end
```

5.44 TRoute

A TRoute describes a sequence of links and nodes, together making up a route between 2 nodes or 2 locations:

If there are one less links than nodes: Between 2 nodes (and percent1=percent2=0)
If there are one less nodes than links: Between 2 locations

```
TRoute = record
  nodes: TIntegerArray 145
  links: TIntegerArray 145
  percent1,percent2: TPercent 148
end
```

If a link number is negative it is travelled in the reverse direction.

5.45 TTime

This is used in class TTSPwindow 126 for setting timestamp.

Alias for integer 141

5.46 TTimeMatrix

A square 2D array of TTime 148 elements. Used in TTSPwindow 126 class.

5.47 TTimeStampFormat

This enumeration describes the format for time stamps in TDrivingDirections 92:

tfSkip	Skip in output
tf24hour	24 hour format: "23:59"
tf12hour	am/pm format: "11:59 PM"
tfFloat	Floating point number for your own formatting (fraction of a day). It may be >1, but not negative. Example: 0.25 = "6:00 AM" = "6:00"

5.48 TTraffic

This type is used for a volume of traffic between coordinates P1 and P2 in traffic assignment 91.

```
TTraffic = record
  P1,P2: TFloatPoint 143
  Volume: TVolume 150
end
```

Only available in Pro

5.49 TTSPmode

This enumeration describes the various modes for TSP optimization:

tspRoundTrip	This is the classic round trip mode
tspStartEnd	This starts at the first item in the list and ends at the last item
tspOpenEnd	This starts at the first item, but can end at any item
tspOpenStart	This can start anywhere, but ends at the list item
tspOpen	This can start and end anywhere

When optimizing for all other modes but tspRoundTrip, set extra = true in methods Matrix 51 and MatrixDyn 51.

5.50 TVertexCount

This is used for the number of vertices on a link.
Minimum is 2 (first and last).
Maximum is 65535.
It is the same as a 2-byte unsigned integer (word 141).

5.51 TVia

This type and corresponding TViaArray 150 can be used when creating driving directions 92.

Field *name* is a textual description and *time* is the time in minutes it takes to make the stop.

```
TVia = record
  Name: string
```

```
  Time: TCost
end
```

5.52 TViaArray

array of TVia 149

5.53 TVolume

This is used in traffic volumes in TTraffic for traffic assignments 91.

Alias for single 141

Only available in Pro

5.54 TVoronoiMode

This enumeration describes the various modes for voronoi 98 output:

vmTriangulationLine	Basic triangulation, as line output
vmTriangulationSimple	Basic triangulation, as polygon output
vmSimpleLine	Basic voronoi, as line output
vmSimple	Basic voronoi, as polygon output
vmIsochrone	Drivetime isochrone
vmServiceArea	Service area

5.55 TWordArray

Array of Word 141