



digital
vision
group

TRIP as a Graph Database

Tutorial

End User License Agreement

All rights to this software, its documentation and logotypes of the TRIP product family and software (altogether “Software”) supplied by DVG Operations GmbH (DVG) are exclusively owned by DVG.

The transfer of this Software, solutions or parts thereof requires the prior written agreement of DVG. Furthermore, the customer has the right to use licensed Software and / or process solutions supplied by DVG to the extent specified in his contract with DVG.

The free-to-use non-commercial version doesn't require a prior written agreement with DVG but such customers, organizations and/or third parties agree by using the software and / or solution of DVG to be strongly obliged to keep all rights to this software, documentation and logotypes of the TRIP product family absolutely un infringed and protected.

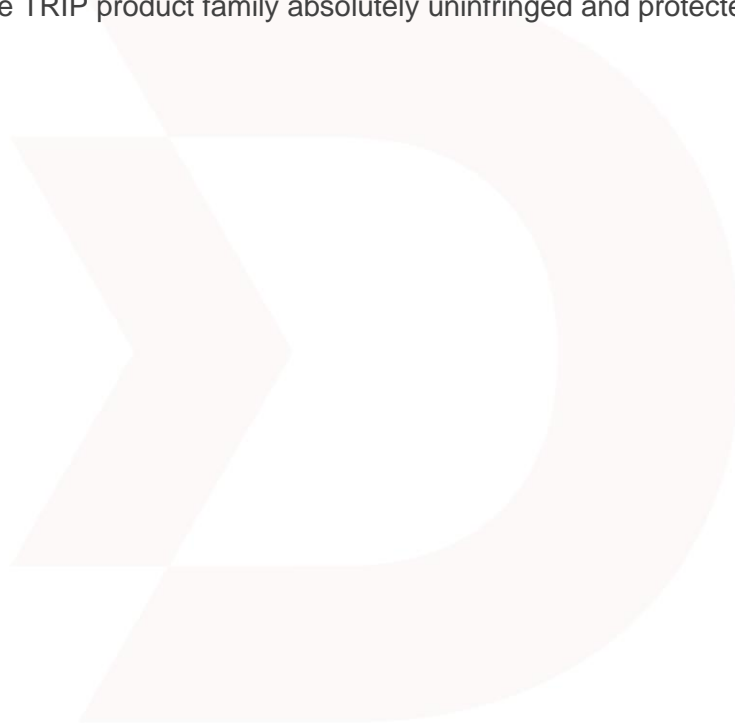


Table of Contents

About this Document	4
Graphs in General	5
Overview.....	5
Graphs.....	5
Graph databases.....	5
Graph Terminology.....	5
Vertex (node).....	5
Edge (relation)	5
Arc	5
Directed Graph	6
Graphs in TRIP.....	6
Motivation	6
The TRIP Graph Database.....	6
Design	6
Vertices.....	7
Edges	7
Create.....	7
Customize.....	8
Graph Construction and Maintenance with TRIPnpx and TRIPjxp	8
Add Vertices	8
Add Edges	8
Vertex Deletion.....	9
Dealing with Broken References	9
Using Graphs from TRIPnpx and TRIPjxp	9
Traversing the Graph.....	9
Find Vertices and Edges via the Graph.....	10
Read Vertices and Edges.....	12
Graph Functionality and the TRIPtoolkit API.....	12
Create a Graph Database	12
Adding Edges and Vertices	12
Graph Search	13
Graph Path Analysis.....	14

About this Document

This document provides a technical overview over the TRIPsystem graph database features. The graph features make it possible use of graph-oriented structures and operations in TRIP.

NOTE: Graph support is EXPERIMENTAL. Its use is encouraged and DVG would like your feedback on it. This feedback will help determine the future direction of development of graph database support in TRIP.



Graphs in General

Overview

Graphs

A graph is a structure that uses vertices, edges and properties to represent data. A graph database is a database that use graph structures.

Graphs in general can be directed or undirected. An edge in an undirected graph states that the associated vertices are related in a bidirectional manner. If A and B are related, it goes both ways. An edge in a directed graph is unidirectional - it only goes one way. In order to state bi-directionality in such a graph, two edges are required.

Graph databases

A graph database can be thought of as an object database. However, a major distinction is that the relations between objects are also objects. Promoting relations to actual entities have some interesting consequences.

- Increased performance when querying the relationships between objects.
- Easier identification of relations between objects, even when the objects are not directly related.
- The difference between the data model and the way it is stored is lower. This smaller "impedance mismatch" means that queries can be executed with greater performance and that applications can be written more quickly.

Graph Terminology

Vertex (node)

A vertex (or node) is the unit out of which graphs are formed. As far as graph theory is concerned, a node is an opaque object. In specific types of graphs, however, nodes may have additional structure.

A semantic network is a type of graph in which the vertices represent concepts or classes of objects.

Edge (relation)

An edge (or relation) is a binary association between two vertices (nodes). Edges may be directed (asymmetric) or undirected (symmetric).

Although the information that two nodes are associated using a particular edge is available, the semantics of graph traversal gets clearer if only asymmetric edges are used. For example; many people may know a celebrity, but the celebrity is not likely to know all his/her fans.

Arc

An arc is a triplet formed by two nodes and an edge. For asymmetric edges, an arc $A=(X,Y)$ is considered to be directed from node X to node Y; Y is called the head and X is called the tail of the arc.

If there are two arcs $A1=(Y,X)$ and $A2=(X,Y)$, each arc is considered to be inverted with respect to the other. $A1$ is the inverted arc of $A2$ and vice versa.

Directed Graph

A graph is a set of arcs. More formally, a directed graph is an ordered pair $D=(V,A)$ where V is a set of vertices (nodes) and A is a set of arcs.

A directed graph is considered to be a symmetric graph, if for every arc A in D , the corresponding inverted arc also belongs to D .

Graphs in TRIP

Graph database features were introduced in TRIPsystem 7.1. A graph database is a class of NoSQL database in which data is structured as a graph.

Motivation

The benefits of a graph database include:

- A **lower impedance mismatch** between the physical data model and the conceptual model / application class hierarchy. Especially if the data is graph-oriented to begin with.
- Following and exploring relationships in data (a.k.a. graph **traversal**, or navigation) is easier that it would be with many other types of databases.

A graph database can be used in a wide number of scenarios, including:

- **Data structures:** any data structure that makes use of pointers, references or relations to link data together is making use of a graph of some kind.
- **Path problems:** trying to find shortest or longest paths from some location to a destination makes use of graphs.
- **Analysis:** graphs can be used to analyze and study the interaction and structure of information represented in the graph.

The TRIP Graph Database

Design

TRIP defines a specific database design for the storage of graph data; edges and simple vertices. This database design is automatically applied when creating a new database as a graph database using TRIPmanager, TRIPnpx or TRIPjxp.

One TRIP installation can have more than one graph database. Using several graph databases in the same graph operation can be done by clustering the graph databases.

A graph database can be customized by adding more fields to the design once the graph database has been created. This allows for additional properties to be specified for the edges and vertices in that database.

Vertices

In many graph databases, a vertex is a very simple object, often with only one single property. This is supported by TRIP, but it is also possible to use any local TRIP record as a vertex in a TRIP graph database.

Using a regular TRIP record as a graph vertex may often make sense. Store all properties intrinsic to the object that the record represents in the record itself. Then put all relations to and from that object into a graph database as edges.

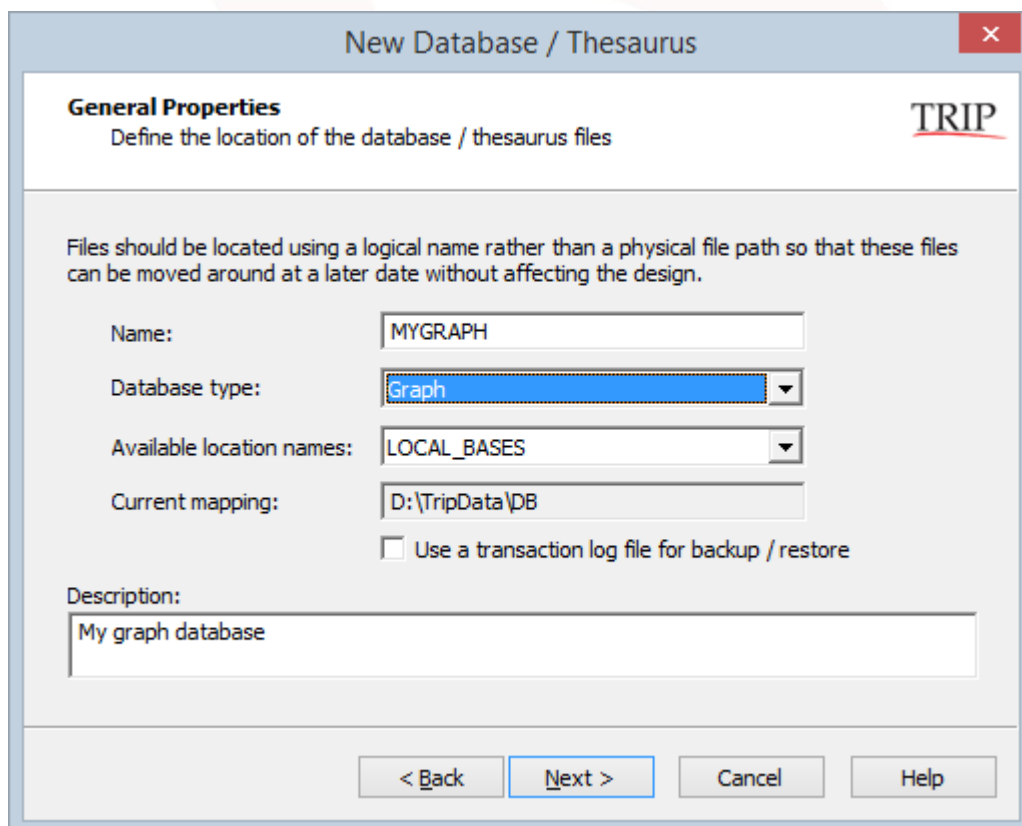
Edges

A graph edge in TRIP is stored in a record in a TRIP graph database. The edge, which always is unidirectional, must have a name. This name identifies the relation.

Create

TRIPmanager

To create a graph database in TRIPmanager, start the New Database wizard and select "graph" in the database type dropdown list.



You do not have to specify any fields. All fields required for the graph database will be automatically created by TRIP.

Programmatically

Graph databases can also be created programmatically using TRIPnpx, TRIPjxp, or the TRIPtoolkit (C API). The procedure is the same as for the creation of any other database, except that no fields must be explicitly created and the Graph property of the database design must be set to true. For example, using TRIPjxp:

```
TdbDatabaseDesign db = new TdbDatabaseDesign(session);
db.setBafFile(location + ":MYGRAPH.BAF");
db.setBifFile(location + ":MYGRAPH.BIF");
db.setVifFile(location + ":MYGRAPH.VIF");
db.setGraph(true);
db.put("MYGRAPH");
```

An example using the C programming language and the TRIPtoolkit can be found further on in this document.

Customize

Custom fields may be added to the design of a graph database. These fields will have no impact on the graph traversal procedures implemented in TRIP, but may be used by the application to prepare sets of edges and vertices over which to perform traversal. Custom fields can therefore have an indirect, but very tangible, effect on graph traversal.

If done programmatically, custom fields can only be added to an existing graph database. Do not attempt to add custom fields to a graph database until it has been successfully saved once.

Graph Construction and Maintenance with TRIPnpx and TRIPjpx

Add Vertices

Any record in TRIP can act as a graph vertex without modification. But it is also possible to add vertex records to the graph database itself. These so called "simple vertices" are associated with a single label. Additional properties may be added as custom fields.

A simple vertex may reference another TRIP record. This reference is optional and assigned by the application. The referenced record may be used to provide additional, more rarely used data on the vertex. This separation can improve efficiency when processing large number of vertices in graph traversal and analysis operations.

To create a simple vertex, use the `TdbGraphRecord` class of TRIPnpx (for .NET) or TRIPjpx (for Java) like in the example below.

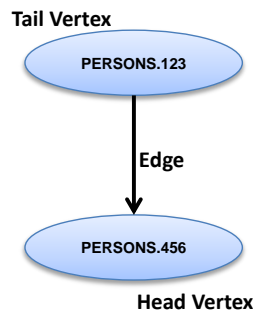
```
TdbGraphRecord gr = new TdbGraphRecord(session,graphdesign);
gr.setGraphRecordType(TdbGraphRecordType.Vertex);
gr.setName("Vertex Label #1");
gr.commit();
```

In order for an existing, regular TRIP record to act as a graph vertex, no action has to be taken. However, it is possible to create a simple vertex record that acts as a proxy to the regular one. This proxy vertex contains a reference to the regular record, which is useful for very large records. To do this, use the `setReference` method on the `TdbGraphRecord` instance before you commit it:

```
TdbGraphRecord gr = new TdbGraphRecord(session,graphdesign);
gr.setGraphRecordType(TdbGraphRecordType.Vertex);
gr.setName("Vertex Label #1");
gr.setReference("REGULARDB", 123, null);
gr.commit();
```

Add Edges

An edge associates two pre-existing vertices with each other. One vertex is the tail and the other is the head of the unidirectional relation that the edge represents.



To create an edge, use the `TdbGraphRecord` class in `TRIPnpx` (for .NET) or `TRIPjxp` (for Java) like in the example below.

```

TdbGraphRecord gr = new TdbGraphRecord(session, graphdesign);
gr.setGraphRecordType(TdbGraphRecordType.Edge);
gr.setName("FRIEND");
gr.setSourceRecord("PERSONS", 123);
gr.setTargetRecord("PERSONS", 456);
gr.setWeight(1.00);
gr.setProperty("CUSTOMPROPERTY", "CUSTOMVALUE");
gr.commit();
  
```

Note that weight and properties are optional. The only required data to set is the edge name, the source record and the target record.

Vertex Deletion

The current graph database implementation in TRIP does not restrict the deletion of vertices even if they are referenced to by one or more vertices. The removal of a vertex does not affect traversal over that vertex at all, but it is impossible to get any information about the vertex once it is deleted.

Dealing with Broken References

Some operations in TRIP may result in changed record numbers in a database. If such records are used as vertices, the edges from which these records are references will be broken. The same applies to simple vertices that have references to other TRIP records.

The best practices for how to prevent broken references are:

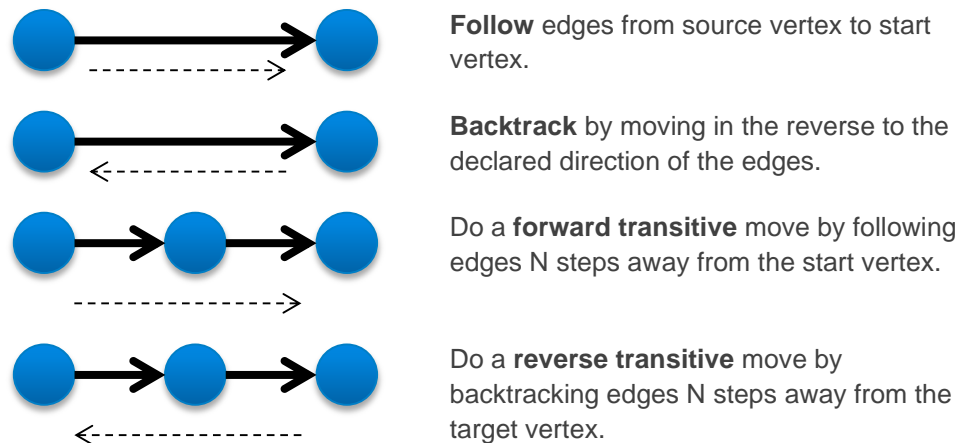
- When running `PACKIT`, use the `--keep-rid` option. Not using this option may cause record ID renumbering, which will break the graph.
- Use record names in the databases that contain vertices used in the graph. Record names are being kept track of by the graph, which means that - because a record name is immutable - it is possible to fix broken references.

The `GRACHK` tool can be used to analyze a graph database for broken references. This tool can also fix broken references if the referenced record uses a record name and remains within the same database but with a different record ID.

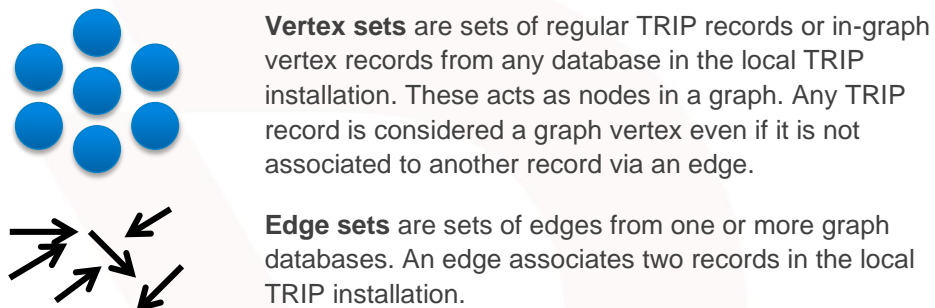
Using Graphs from `TRIPnpx` and `TRIPjxp`

Traversing the Graph

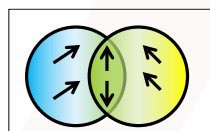
There are four basic graph traversal operations:



Since TRIP works in a set-based manner when searching, it follows that a graph implementation in TRIP will do the same. When working with graph data in TRIP, there are two types of sets involved:



Graph search and navigation operations in TRIP are all essentially operations on edge sets.



To best utilize TRIP's capabilities, graph querying and navigation works strictly with edge sets. This is possible because the edges contain information on the vertices it connects. This leads to performance gains because no vertex records will normally have to be accessed during graph navigation.

Find Vertices and Edges via the Graph

Normally, some of the vertex records that are involved in the search scenario are known. These may be:

- Records acting as source vertices for edges to follow.
- Records acting as target vertices for edges to follow.
- Records to restrict graph navigation to, essentially a subgraph.

While it is possible to query a graph without specifying any known vertices, standard use cases involve at least one of these types of vertices as a known factor.

To find the vertices to use in the graph search, a regular TRIP query is involved. This does not have to do anything with the graph functionality in itself, even if it could involve searching a graph database for in-graph vertices. The steps and commands involved are covered in depth in other TRIP documentation.

Scenario: Follow

This scenario uses an imagined social network database, where there are vertices of PERSONs and edges like "FRIEND". To know the friends for a particular person, a search set that contains that person vertex is needed.

```
S=2      <1>      FIND NAME='John Doe'
```

Note that because all operations are set based, this vertex set can contain any number of vertices – it does not have to contain only one.

Once we have the vertex set we are ready for graph search. Start with vertex set having search ID #2 and follow FRIEND edges outwards one step away. We could choose additional restrictions on target vertices (e.g. persons with particular interests), but in this case we simply want to list all friends. In the graph APIs, set number zero signifies the universal set, i.e. all records in the local TRIP installation. Expressed in TRIPjxp this becomes:

```
allv2edges = graphset.source(2);
v2friendEdges = allv2edges.follow("FRIEND",0);
```

As previously noted, all graph operations in TRIP works with sets of edge records. So in this case we first must create an edge set *allv2edges* with all the edges in the current graph that has the vertex in set #2 as source. The second line narrows this down further to select all the FRIEND edges.

If we from this last edge set want to get a vertex set consisting of the set of friends, we need to resolve the edge set. Resolving is a type of operation that gives a set of vertices based on either the edge sources or the edge targets. In this case we want the targets.

```
allFriends = v2friendEdges.resolveTargets();
```

And there we have it – a regular TRIP set represented by a TdbSearchSet instance with person records for the friends of our John Doe.

Scenario: Shortest Path

Shortest path analysis is a classic type of graph problem. In version 7.1 TRIP uses a breadth-first set-oriented approach that moves outwards, one step away from the starting point until all paths (in the entire graph or in a specified subgraph) have been explored and the shortest one determined. Using a traditional vertex based technique would lead to poor performance because of the need to load each vertex record from its database. But since the edge records in a TRIP graph database contains information on the vertices, the vertex records themselves are not needed. Only the index (BIF) of the graph database is utilized.

Locating a shortest path requires that we know the starting point and the destination. For instance, finding the best way to get to the beach:

```
S=2      <1>      FIND NAME='123 Home Road'
S=3      <1>      FIND NAME='7 Sunny Beach'
```

Now all we have to do is to use these set numbers in a call to the *shortestPath* method.

```
edges = new string[]{"ROAD","STREET","HIGHWAY"};
path = graphset.shortestPath(2,edges,3);
```

The first statement creates an array of the names of all the edge types we want to traverse. If we wanted to use all edges, we could just set this to an empty array or null. The second statement sends the path analysis request to the server, and returns a single TdbGraphPath instance that represents the shortest path found. If no path could be found, a null value is returned instead.

TRIP uses two metrics to determine the shortest path:

- The number of edges between start and destination
- The total weight of the edges in the path.

If two candidate paths have equal weight, the one with the least number of edges in it is considered shortest.

Read Vertices and Edges

Records in a graph database are readable to applications just like records in any other database. A database can therefore read and inspect edge and in-graph vertex records as needed.

Graph Functionality and the TRIPtoolkit API

The server-side C API known as TRIPtoolkit is what exposes the graph functionality from the TRIP kernel. These functions are used by the TRIPjxp and TRIPnxx APIs, but can also be used directly from server-side applications.

Create a Graph Database

Creating a graph database using TRIPtoolkit is similar to the creation of a regular database. The difference is that no fields must be added to the design, and that the function `TdbDefineGraph` must be called prior before the design is committed using `TdbPutBaseDef`. For example (error handling excluded for clarity):

```
base_spec_rec bsr;
TdbGetBaseDef("MYGRAPH", &bsr);
memset(bsr.baf_file, ' ', sizeof(bsr.baf_file));
memset(bsr.bif_file, ' ', sizeof(bsr.bif_file));
memset(bsr.vif_file, ' ', sizeof(bsr.vif_file));
memset(bsr.baf_file, "DB_LOCATION:MYGRAPH.BAF", 23);
memset(bsr.bif_file, "DB_LOCATION:MYGRAPH.BIF", 23);
memset(bsr.vif_file, "DB_LOCATION:MYGRAPH.VIF", 23);
TdbDefineGraph(&bsr, true);
TdbPutBaseDef(&bsr);
```

Custom fields may be added once the graph database design has been saved, since custom fields cannot be added to a graph database that does not yet exist.

Adding Edges and Vertices

The TRIPtoolkit API in TRIPsystem has two functions that simplify the process of creating edges and in-graph vertices. These are `TdbGraphAddVertex` and `TdbGraphRecAssoc`.

In-Graph Vertices

To create an in-graph vertex, use the `TdbGraphAddVertex` function. In the simplest form, this only involves specifying the label of the vertex:

```
TdbGraphAddVertex(NULL, NULL, "PERSON.123", NULL);
```

The above requires that the graph database into which to store the record currently open for search. To specify another database, you need to create and initialize a record control yourself:

```
int recordId = 0;
char graphdb[32] = "MYGRAPH";
TdbHandle recordControl = (TdbHandle)0;
TdbCreateRecordControl(&recordControl);
TdbSetBase(recordControl, graphdb, MODE_WRITE);
TdbGraphAddVertex(&recordControl, NULL, "PERSON.123", NULL);
TdbPutRecord(recordControl, &recordId);
```

As indicated, creating the record control yourself means that you will be responsible for committing the

record using `TdbPutRecord`. The same holds if you pass a record control handle variable that is set to `NULL`. In this case, the `TdbGraphAddVertex` function will create the record control but leave it up to the calling code to commit. The reason for this is to allow the code to modify the record (e.g. by adding values to custom fields) before the commit.

Edges

To create an edge record, use the `TdbGraphRecAssoc` function. This requires record controls referring to both vertex records being associated. The example below assumes that 'sourceRecord' and 'targetRecord' are record controls that refer to pre-existing TRIP records.

```
int recordId = 0;
char graphdb[32] = "MYGRAPH";
TdbHandle recordControl = (TdbHandle)0;
TdbCreateRecordControl(&recordControl);
TdbSetBase(recordControl, graphdb, MODE_WRITE);
TdbGraphRecAssoc(&recordControl, NULL, "FRIEND", sourceRecord,
                targetRecord, 0.00);
TdbPutRecord(recordControl, &recordId);
```

Graph Search

The two main functions for performing graph search operations via the TRIPtoolkit are `TdbGraphFind` and `TdbGraphResolve`.

TdbGraphFind

Prototype:

```
int TdbGraphFind(int* resultset, int sourceSet,
                int edgeSet, int targetSet,
                int exclusionSet,
                const char* condition);
```

The `TdbGraphFind` method uses four existing search sets:

- **Source set:** a set of vertices that edges have as sources. If set to `GRAPH_SET_UNIVERSAL` (zero), the function will consider all possible records acting as sources for edges.
- **Edge set:** a set of edges that is to be traversed. If set to `GRAPH_SET_UNIVERSAL` (zero), the function will consider all edges in the current graph database.
- **Target set:** a set of vertices that edges have as targets. . If set to `GRAPH_SET_UNIVERSAL` (zero), the function will consider all possible records acting as targets for edges.
- **Exclusion set:** a set of edges that must not be included in the result set. If set to zero, no exclusion set will be used.

In addition, it is possible to specify a **condition** that is a CCL condition without the command keyword that will be appended to searches for edges in the graph database that the `TdbGraphFind` method performs based on the other input parameters. This can for instance be used to select edges of particular names or other attributes.

The example below searches for all mountain pictures taken in Europe:

```
int targetSet = 0, sourceSet = 0, edgeSet = 0;
TdbExecuteCcl("BASE IMAGES", &status);
TdbExecuteCcl("FIND KEYWORDS=mountains", &status);
TdbSearchResult(&sourceSet, &setSize, &hitcount, &ccl, &status);
TdbExecuteCcl("BASE LOCATIONS", &status);
TdbExecuteCcl("FIND CONTINENT=EUROPE", &status);
TdbSearchResult(&targetSet, &setSize, &hitCount, &ccl, &status);
TdbExecuteCcl("BASE MYGRAPHDB", &status);
TdbSearchResult(&edgeSet, &setSize, &hitcount, &ccl, &status);
TdbGraphFind(&edgeOutputSet, sourceSet, edgeSet,
             targetSet, 0, "");
```

The result of the above sequence is a set of edges that link images of mountains to information about their locations. Non-European locations are discarded.

TdbGraphResolve

The final step in a graph search operation is usually to obtain the set of vertices for the sources or targets of the set of edges a TdbGraphFind operation resulted in. This is where the TdbGraphResolve function comes in.

Prototype:

```
int TdbGraphResolve(int* resultset, int mode, int edgeSet,
                   const char* database);
```

The **mode** parameter indicates which type of vertex set to retrieve. Setting this to GRAPH_RESOLVE_SOURCE yields a set of vertices for which the edges in the specified edge set have as sources. Setting this to GRAPH_RESOLVE_TARGET yields a set of vertices for which the edges in the specified edge set have as targets.

The optional **database** parameter can be used to restrict output vertices to a particular database only. If this parameter is not set, the result set may contain vertex records from any local TRIP database.

The example below completes the TdbGraphFind example by obtaining a set of vertices that represent the European locations for which we have mountain images.

```
TdbGraphResolve(&vertexOutputSet, GRAPH_RESOLVE_TARGET,
               edgeOutputSet, "");
```

Graph Path Analysis

Graph path analysis using the TRIPtoolkit functions is quite a bit more involved than using the methods in TRIPnpx or TRIPjxp. The pseudo code for a path analysis is:

```
TdbGraphStartPathAnalysis
Do
    TdbGraphPathAnalysisStep
While Successful
TdbGraphAnalysisSnapshot
While TdbGraphNextPath
    Read Start Of Path
    While TdbGraphGetPath
        Read Next Edge and Vertex From Path
    Wend
    Report Path to User
Wend
TdbGraphClosePathAnalysis
```

The loop around the TdbGraphPathAnalysisStep function is what performs the actual analysis. Each call makes the analysis progress one edge further away from the origin vertex (or vertices) of the analysis.

TdbGraphAnalysisSnapshot returns a count of the paths found. This can also be called after every call to TdbGraphPathAnalysisStep in order to inspect how the analysis is progressing.

The analysis may result in any number of paths. The function TdbGraphNextPath is used to iterate through these paths. To get the complete details for a particular path, repeated calls to TdbGraphGetPath is needed.

Refer to the TRIPtoolkit API reference in the documentation set for TRIPsystem 7.1 or later for more details.

