The Distributed Graph Storage System: A User's Manual for Application Programmers

TR93-003

January 27, 1993

Douglas E. Shackelford

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175
919-962-1792
jbs@cs.unc.edu

A TextLab/Collaboratory Report

Portions of this work were supported by the National Science Foundation (Grant #IRI-9015443 and by IBM Corporation (SUR Agreement #866).

Abstract

As the title suggests, this is a manual for people who will be using the Distributed Graph Storage (DGS) system to write applications. Since the focus of the manual is on the Application Programming Interface (API), it does not include specific information about the implementation, nor does it give specific details about how to compile an application. This information is provided in other documents.

Contents

List of Figures

1 An Introduction to the DGS

The Artifact-Based Collaboration (ABC) environment is being developed as a part of a long-term, multi-disciplinary program of research on computer support for collaboration by groups of people. Specifically, ABC supports groups that collaborate to create and maintain complex conceptual artifacts (e.g., research proposals and plans, network architecture specifications, software system designs) expressed in text (natural and programming languages), drawings, images, and other computer-based media. The environment includes several system-level components, including hypermedia browsers and applications, a shared window conferencing facility, real-time audio/video, and a hypermedia storage service called the Distributed Graph Storage (DGS) system.

This document is a tutorial and reference manual for the DGS application programming interface (API). As such, it is a user's manual for application developers. No knowledge of the DGS or of hypermedia is assumed. The implementation of the DGS is not discussed since these details can be found in other documents.

The manual is organized as follows. A section on "Basic Concepts" lays the background for the data model and system architecture of the DGS. Next, the "Getting Started" section introduces the rudiments of API programming. The next section is a reference guide for the C++ functions provided by the API. Finally, the manual concludes with an extended tutorial section that takes the reader step-by-step through the creation of a small, non-trivial application.

2 Basic Concepts

2.1 Data Model

2.1.1 Attributes and Content

The most basic element of storage is the *node*, which usually contains the expression of a single thought or idea. For example, a node might contain a paragraph of text or a figure from a document. Structural relationships (e.g., between chapters within the same document) and semantic relationships (e.g., references between different documents) are represented explicitly as *links* between nodes.

The data model provides two mechanisms for storing information within a node: node attributes and node content. *Attributes* are typed, named variables for storing fine-grained information (approximately 1-100 bytes). Some attributes (such as creation time and size) are maintained automatically by the system. There may also be an unlimited number of application-defined and maintained attributes.

In contrast to attributes, node *content* is designed to reference large amounts of information. This content can take one of two forms: (1) a stream of bytes (accessed using a file metaphor) or (2) a higher-level composite object (accessed using a graph metaphor). Applications control whether the content of a particular node is of Type 1 or of Type 2. Since Type 1 content obeys the standard file metaphor, it can be used to store the same types of information as files, e.g., text, bitmaps, line drawings, digitized audio and video, spreadsheets, and other binary data. Applications that can read and write conventional files can read and write Type 1 content with no changes.

Whereas Type 1 content is based on the file metaphor, Type 2 content is based on the metaphor of graph theory. Thus, the content of a Type 2 node is defined to be a composite object called a subgraph. For convenience, the term *artifact* will be used to refer to the set of all nodes and links being stored by the DGS. Using this definition, a DGS *subgraph* is defined as a consistent subset of the nodes and links in the artifact.

Subgraph consistency is maintained by the DGS. For example, all subgraphs satisfy the condition that if a link belongs to a subgraph, then so do the link's source node and target node. Nodes and links may belong to multiple subgraphs at the same time, but every node and link must belong to at least one subgraph. The DGS also provides *strongly typed* subgraphs (e.g., trees and lists) that are always guaranteed to be consistent with their type.

Hereafter, Type 1 content will be referred to as *file content* and Type 2 content will be called *subgraph content.*

2.1.2 Structure and Hyper-Structure

Figure 1A shows an artifact with 18 nodes and 24 links. In theory, this artifact could be decomposed into hundreds of different subgraphs. However, in practice, nodes and links are grouped into a few meaningful subgraphs that serve specific purposes. For instance, a user might construct two subgraphs (see Figures 1B and 1C): one containing the underlying tree structure and another containing the set of links that are external to the tree. In a real application, the tree might represent a document, and the external links could represent a path that a particular reader has taken through the document.

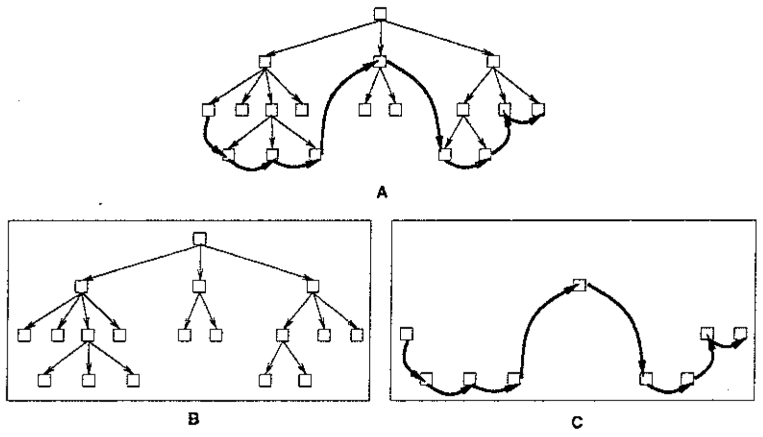


Figure 1: Two Subgraphs and the Underlying Artifact

In fact, these subgraphs represent two qualitatively different ways of using a subgraph. In the case of the tree, the subgraph is used to represent the essential structure and organization of the material. In contrast, the second subgraph is a representation of semantic relationships that are supplementary to that structure. Using the terminology of the data model, the tree in Figure 1B represents *structure* whereas the subgraph in Figure 1C represents *hyper-structure*. Other examples of hyper-structure include: references in a document to glossary entries, figures, or related sections; private annotations made by a reader but not intended to be part of a document; and references in a specification document to a requirements document. In each case, the relationship cuts across normal structural boundaries. To underscore this distinction, the data model defines two classes of subgraphs: structural subgraphs (*S-subgraphs*) and hyper-structural subgraphs (*HS-subgraphs*). Nodes can belong to both S-subgraphs and HS-subgraphs, but the existence of a node is dependent on its membership in at least one S-subgraph. The DGS automatically garbage collects a node when it is removed from the last S-subgraph that contains it, regardless of the number of HS-subgraphs that contain it.

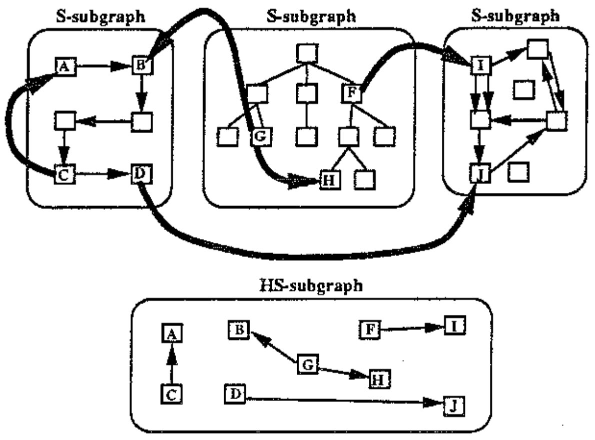


Figure 2: Examples of Hyper-structural Linking

Links are similarly divided into two classes: *S-links* and *HS-links.* HS-links are used to represent associations between nodes in different S-subgraphs or non-structural relationships between nodes within the same S-subgraph (See Figure 2). Both types of links are directed, although traversal is supported in either direction. Like nodes, links store information in attributes and content.

2.1.3 Using the Data Model to Organize Information

The data model encourages users to decompose a large artifact into small S-subgraphs related by composition. This organization improves human comprehension of the artifact and increases the potential for concurrent access to individual components. However, the best way to understand these mechanisms is by example.

Figure 3 illustrates one way to organize the public and private materials associated with a large research project (node content is indicated by dashed lines). First of all, notice that the data model forces every subgraph (except the root subgraph) to be the content of exactly one node. A second observation is that the figure includes several subgraphs that are devoid of links. The significance of this is that the subgraphs are being used to represent sets of nodes that do not have any structural relationships. Another aspect that

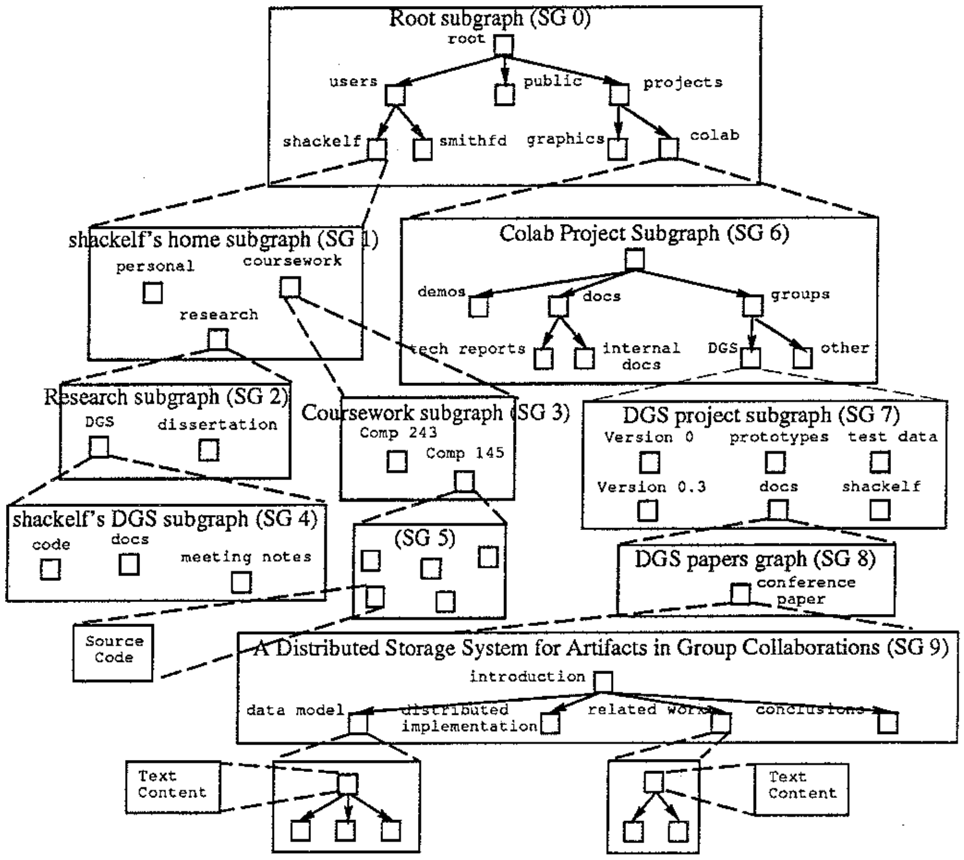


Figure 3: Organizing the Public and Private Pieces of an Artifact

is more difficult to see is that a node can appear in more than one place within the artifact. For example, not only does the "conference paper" node appear in *SG 8*, but it might also appear under the "tech reports" node in *SG 6* and under the "docs" node in *SG 4*. Also missing from the figure are the HS-links that would normally cut across the natural structure, linking nodes in different S-subgraphs.

Finally, one can observe that the example shown in Figure 3 subsumes the organization of data in a conventional file system (consider subgraphs as directories and nodes with file content as files). In fact, the DGS goes a step beyond file systems by providing new mechanisms for storing meta-information about files (in attributes) and for representing semantic and structural relationships between files (in links).

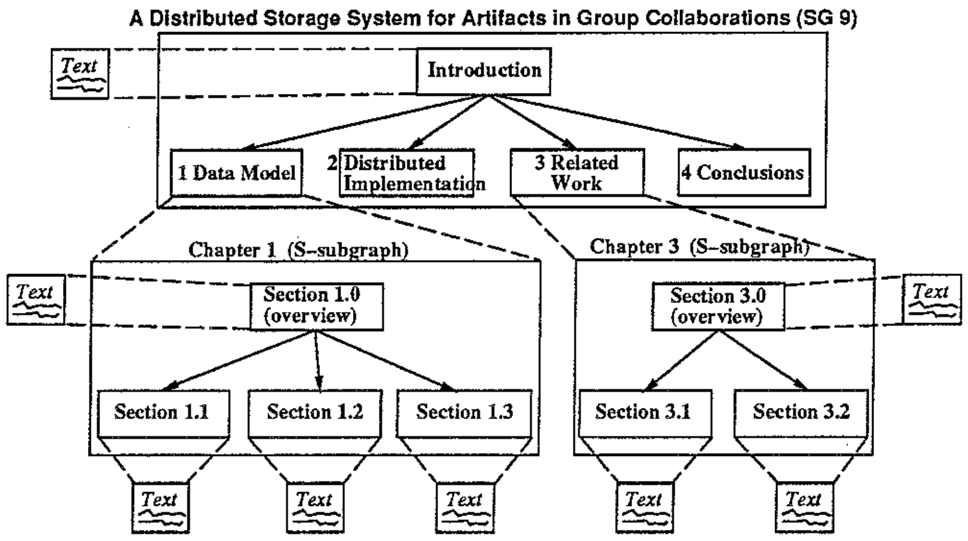


Figure 4: Using Node Content and Structure to Represent a Document

Figure 4 shows the conference paper from Figure 3 in more detail. A useful exercise is to compare Figure 4 with the way that the conference paper would be stored in a file system. The most striking difference is the number and size of the nodes that compose the document. Whereas a conventional document would be stored in a single monolithic file, the DGS data model encourages a user to divide documents into many smaller nodes and subgraphs. This maximizes the benefits of hyper-structural linking because each node expresses a single concept or idea. Consequently, when a node is linked, it is clearer which idea the link is referencing. An additional benefit is that documents can be structured according to the way that they will be used. For example, collaborative writing is much easier when authors are able to structure a document so that they are working in different parts of it.

2.1.4 Fine-Grained Linking Using Anchors

Although nodes are finer-grained than traditional files, there are still times when one would like to reference information at an even lower level. For example, an application might want to create a link that points to a specific word within a node, rather than to the node itself. Consider that a group

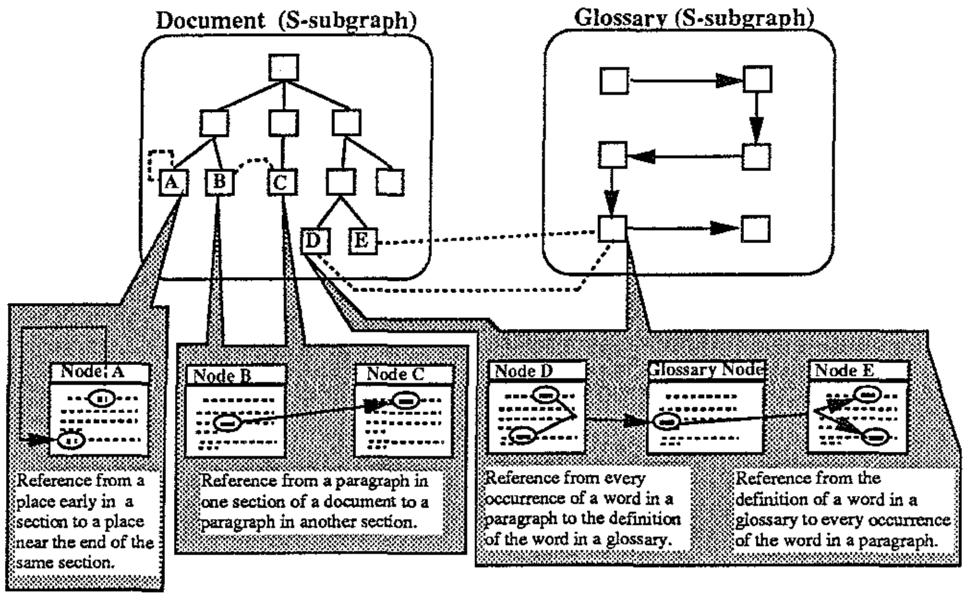

Document (S-subgraph)
Glossary (S-subgraph)
A
B
C
D
E
Node A
Node B
Node C
Node D
Glossary Node
Node E
Reference from a place early in a section to a place near the end of the same section.
Reference from a paragraph in one section of a document to a paragraph in another section.
Reference from every occurrence of a word in a paragraph to the definition of the word in a glossary.
Reference from the definition of a word in a glossary to every occurrence of the word in a paragraph.

Figure 5: Examples of Anchored HS-links

of users could use a node to store the glossary of terms that are common to their project. Given this, a user might like to create an HS-link from the occurrence of a term in a document node to its definition in the glossary node. To achieve fine-grained linking like this, the data model provides the concept of an anchor within a node. An anchor identifies part of a node's content, such as a function declaration in a program module, a definition in a glossary text, or an element of a line drawing. An anchor can be used to focus an HS-link onto a specific place within the content of a node. When an HS-link is paired with one or more anchors in its source or target nodes, it is called an *anchored HS-link.* The relationship between anchors and HS-links is many-to-many (see Figure 5).

2.1.5 Common Attributes and Graph Attributes

Some attributes are called *common attributes* because their values are independent of the context from which they are accessed. All objects—nodes, links, and subgraphs—can have common attributes. In addition, nodes and links can have context-sensitive attributes whose value may be different depending on the context from which they are accessed. This second type of attribute is called a *graph attribute* because a subgraph provides the context. For example, consider a node that is contained in two subgraphs: a list and a tree. This node can be viewed in two different contexts. A user could find the node while browsing the list, or the user could encounter the node while browsing the tree. In both cases, the node will be displayed as an icon on the screen, but the position, size, and other visual properties of the icon may depend on which subgraph is being viewed. If these properties were stored as common attributes of the node, then we would have to give them names such as "position in List #43564" and "size in Tree #6723", but if we store them as graph attributes of the node, then we can use shorter names such as "position" and "size", since the subgraph provides the missing context.

Application developers sometimes confuse common and graph attributes, but a graph attribute of a node or link is not the same as a common attribute of a subgraph. In the previous example, the tree could have had one or more common attributes such as a title or a short description of the tree's purpose. These would have been attributes of the tree and, thus, would not qualify as graph attributes. To summarize, nodes and links can have both common (context-independent) and graph attributes (context-dependent), whereas subgraphs can only have common attributes.

2.2 Concurrency and Access Protection for Groups

2.2.1 Concurrency Semantics

Since the DGS data model is object-oriented, the objects of the data model—nodes, links, and subgraphs—exist as distinct entities within the storage system. Before a user's application can access the data within a particular object, the application must explicitly open the object using the `Open()` function. `Open()` will fail if the user lacks the proper access authorizations and if the request is in conflict with other requests in progress.

Conflict can occur when users try to access the same object concurrently. To specify allowable concurrent accesses, the DGS defines three access modes for nodes, links, and subgraphs: DGS_READ, DGS_WRITE, and DGS_READ_NO_ANCHOR. Applications must specify one of these modes as a parameter to `Open()`. DGS_READ *access* allows operations that do not change subgraph membership, linking information, or attribute or content values. In the case of nodes, DGS_READ access also allows anchor creation and deletion, but only when the application has read_write authorization on the HS-link that is being anchored. DGS_READ_NO_ANCHOR *access* is defined only for nodes and allows all operations of DGS_READ access except anchor creation and deletion. DGS_WRITE *access* allows all operations.

The following rules govern concurrent access to an object:

- For links and subgraphs, multiple opens with DGS_READ access and a single open with DGS_WRITE access are allowed (as is the weaker case of multiple DGS_READ opens alone).
- For nodes, multiple opens with DGS_READ_NO_ANCHOR access and a single open with DGS_WRITE access are allowed (as is the weaker case of multiple DGS_READ and/or DGS_READ_NO_ANCHOR opens alone).
- No other cases of opens for concurrent access are allowed. Browsers and other applications must implement measures for deadlock avoidance in case an access request is denied.

Changes to an object are not visible to any applications with overlapping opens of the object until it is closed by the writer and then only to applications that open it after the close completes.

2.2.2 Access Control Lists

Groups can control access to parts of the artifact by specifying access authorizations for node, link, and subgraph objects. Authorizations are expressed in an access control list that is stored with each object. An *access control list* maps names of users or groups of users to categories of operations that they are allowed to perform on the associated object. No user is allowed to access an object unless the user has proper authorization for operations implied by the access mode specified in the `Open()` function. Two categories of authorizations are defined: access and administer. *Access authorizations* give users permission to access the data associated with a particular object. *Administer authorizations* give users permission to perform operations such as changing the object's access control list.

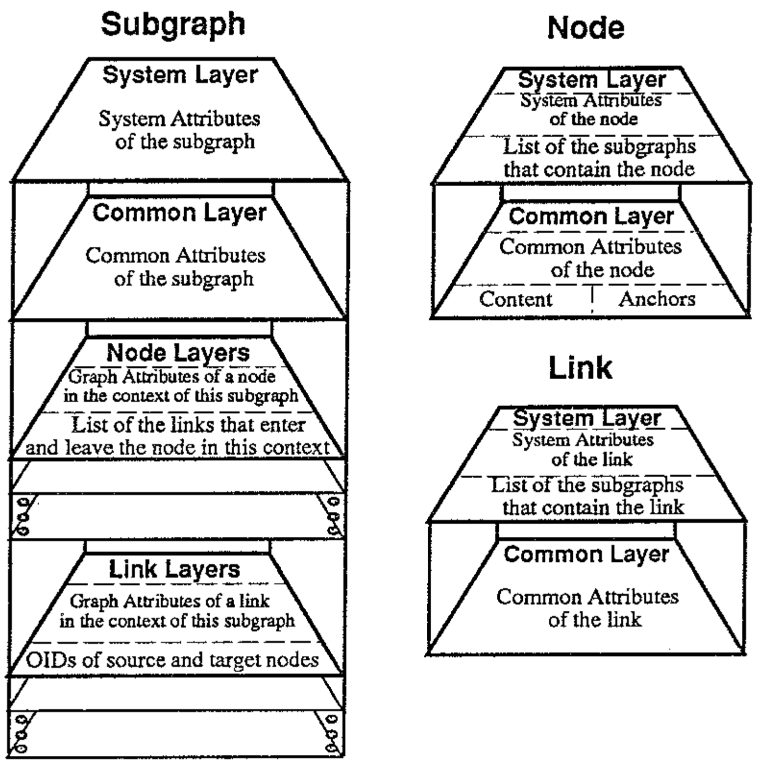

Subgraph
System Layer
System Attributes of the subgraph
Common Layer
Common Attributes of the subgraph
Node Layers
Graph Attributes of a node in the context of this subgraph
List of the links that enter and leave the node in this context
Link Layers
Graph Attributes of a link in the context of this subgraph
OIDs of source and target nodes
Node
System Layer
System Attributes of the node
List of the subgraphs that contain the node
Common Layer
Common Attributes of the node
Content
Anchors
Link
System Layer
System Attributes of the link
List of the subgraphs that contain the link
Common Layer
Common Attributes of the link

Figure 6: Logical Structure of Nodes, Links, and Subgraphs

2.2.3 Object Data and Access Authorizations

Figure 6 summarizes the data that is associated with each of the three types of objects. Table 1 lists the most common types of operations and the access authorization that is needed to perform them.

For example, suppose that an application wants to add a node to a subgraph. Since this operation would change the structure of the subgraph, we see from Table 1 that read_write authorization is required. First, the user who owns the application must have read_write authorization for the subgraph. Next, the application must open the subgraph in DGS_WRITE mode. Then, the application can perform the "add node" operation. Finally, when the application is finished, it should explicitly close the subgraph to make the changes permanent.

The process of anchor creation is more complicated because it requires

Table 1: Access Authorizations and What They Provide

Object Type	**read**	**read_write**
Node	read anchors	change anchor values
	read node attributes and content	change node attributes and content
Link	read link attributes and content	change link attributes and content
		create/delete anchors associated with the link
Subgraph	read common attributes of subgraph	change common attributes of subgraph
	read graph attributes of nodes and links in the subgraph	change graph attributes of nodes and links in the subgraph
	read structure of subgraph	change structure of subgraph

two authorizations (the same applies to anchor deletion). First, the user must have read authorization on the node that will contain the anchor and read_write authorization for the link that will be associated with the anchor. Next, the application must open the node in either DGS_READ or DGS_WRITE mode. Then, the application can create the anchor. The node must be explicitly closed to save the changes.

2.3 System Architecture

As shown in Figure 7, the DGS has a layered architecture that can be configured in a number of different ways. The *Application Layer* contains the user interface and other code that is application-specific. The top layer of the DGS is the *Application Programming Interface (API) Layer* which exports a graph-oriented data model to applications. An overview of this data model was presented in Section 2.1. Most of the DGS is implemented in the bottom two layers: the Graph-Semantics Layer and the Storage Layer. The *Graph-Semantics Layer* implements the data model and performs local caching; the *Storage Layer* is responsible for permanently storing results.

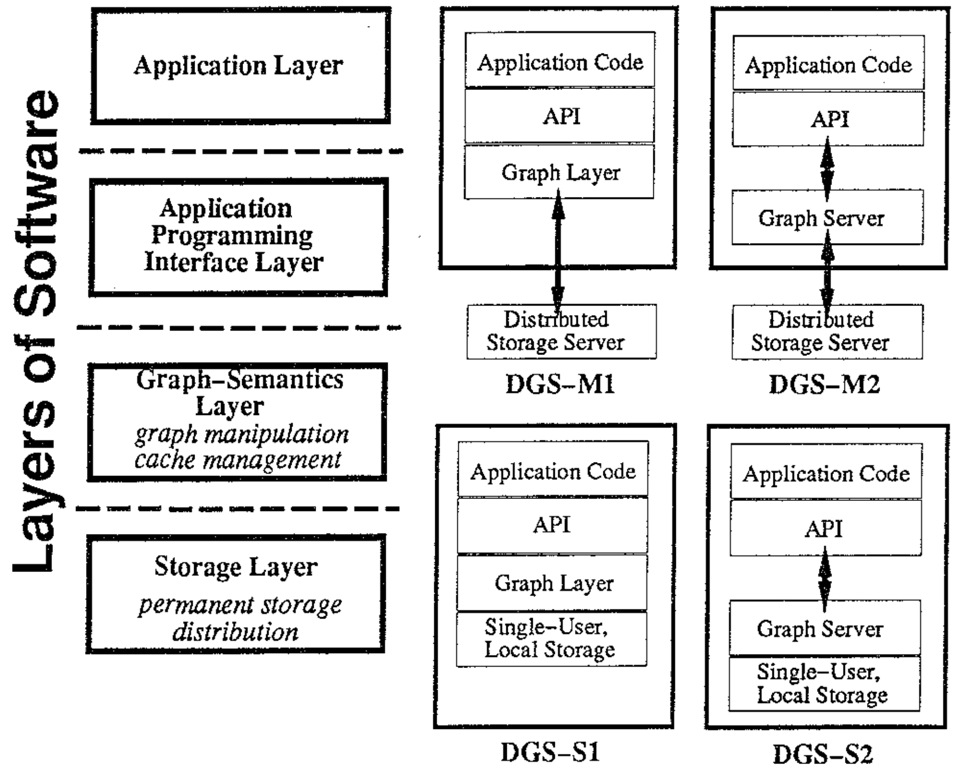


Figure 7: Four Architectures for the DGS

Since the API Layer isolates the Application Layer from the rest of the DGS, application code is portable across different implementations of the bottom two layers. Ultimately, the DGS will provide at least two different implementations of the storage layer and two different methods for connecting the API Layer with the Graph-Semantics Layer. This will result in the four architectures that are shown in Figure 7. In the DGS-M2 architecture, the Application Layer and the Graph-Semantics Layer will be running in different processes on the same machine; the Storage Layer will be implemented as a multi-user, distributed storage server. The DGS-M1 architecture will be the same except that the Graph-Semantics Layer will be linked with the application to become a single process. The advantage of this architecture is better local response time due to reduced Inter-Process Communication (IPC). A disadvantage is that it increases the size of application executables. The DGS-S1 and DGS-S2 architectures follow a similar pattern except that the distributed storage server is replaced by a single-user, non-distributed storage layer. This centralized architecture requires much less overhead than the distributed version but is unsuitable for users who need to share information with others.

At the present time (December 1992), DGS-M2 and DGS-S2 are unsupported, although we expect to support all of the architectures by Summer 1993.

All node, link, and subgraph objects are identified by an object identifier (OID) that is universal and unique. Once an object is created by the DGS, its OID is never changed and the value is never reused even if the object is deleted. To applications, an OID is an "opaque" (uninterpreted) key that can be used to retrieve the corresponding object. However, we encourage application programmers to use OIDs sparingly. The next section lists some alternatives to using OIDs to retrieve objects.

3 Getting Started

Figure 8 is a small program that uses the DGS. The essential lines are 2, 6, and 7. `CSDgsConnection::Initialize()` must be called before any other DGS functions. After this, the application is free to open, access, and close objects. Objects must be explicitly opened (line 12) before they can be accessed and should be explicitly closed (line 14) afterwards.

Figure 8: "Hello, world!" Program

```
1:   // Program: hello.c
2:   #include <csdgs_include.h>
3:
4:   main()  {
5:
6:      CSDgsConnection dgs;
7:      dgs.Initialize();    // Establish connection with the DGS
8:
9:      CSDgsTree *root_subgraph = dgs.GetRootSubgraph();
10:     if (root_subgraph != nil)
11:        cout << "The root subgraph exists!\n";
12:     root_subgraph->Open(DGS_READ);
13:     ...
14:     root_subgraph->Close();
15:
16:  }
```

An application must have a pointer to an object before it can open it. In Figure 8, the program uses the `CSDgsConnection::GetRootSubgraph()` function to get an object pointer (line 9). In fact, there are four ways for an application to get a pointer to an object:

- CSDgsConnection::GetRootSubgraph() returns a pointer to the root subgraph of the artifact. For example, *SG 0* is the root subgraph of the artifact in Figure 3.

- CSDgsConnection::GetHomeSubgraph() returns a pointer to the home subgraph associated with a particular user. For example, the operation `dgs.GetHomeSubgraph("shackelf")` would return a pointer to *SG 1* in Figure 3. If the user name is omitted, then the owner of the current process is assumed.

- Objects contain references to other objects. For example, the operation `root_subgraph->Root()` would return a pointer to the root node of the root subgraph.

- CSDgsConnection::GetObjectByOid() returns a pointer to the object that has a particular OID. For example, the operation `dgs.GetObjectByOid( oid_1 )` would return a pointer to the object that has the OID *oid_1*.

Once the application has a pointer to an object, it can open the object and access it using functions that are described in the following sections.

Although all four of the methods can be used to obtain pointers, the last method uses a data structure that is internal to the DGS. For this reason, we strongly discourage applications from using OIDs to retrieve objects. In some cases applications can avoid using OIDs by storing a reference to an object as an attribute of type `CSDgsObjectAttr` (see Section 4.2.6). In the rare event that an application must reference an OID, the following comments apply:

- The OID of an object can be obtained by executing the function `CSDgsObject::GetOid()`, e.g., `UUID oid_1 = root_subgraph->GetOid()`.

- Given the OID of an object, an application can retrieve a pointer to that object using the function `CSDgsConnection::GetObjectByOid()` as described above.

- It is better if the OID is used quickly and then discarded. For example, it is acceptable for a process to send an OID to another process for immediate use. When possible, applications should avoid embedding OIDs in long-term storage such as source code and other files.

4 API Reference Guide

4.1 Overview of the API Class Hierarchy

The Application Programming Interface (API) for the DGS is a C++ class library. Figure 9 shows the major classes in the inheritance hierarchy. The `Class CSDgsObject` defines operations that are common to all objects, such as `Open, Close`, and functions to manipulate common attributes. Subclasses inherit the API of their parent class and extend the inherited API with more specialized functions.

This section introduces the functions associated with each class and illustrates each one with a fragment of C++ code.

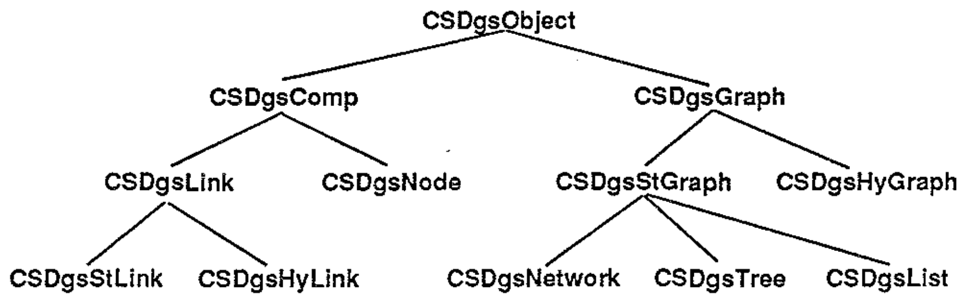


Figure 9: API Class Hierarchy

4.2 CSDgsObject

The root of the object hierarchy is the class `CSDgsObject`. In addition to general functions such as `Open` and `Close`, the class also provides functions to create, access, and delete common attributes.

4.2.1 Open/Close

int CSDgsObject::**Open**(int **mode**);
int CSDgsObject::**Close**();

ARGUMENTS:

mode: Access Mode (DGS_READ, DGS_WRITE, or DGS_READ_NO_ANCHOR)

COMMENT:

With few exceptions, an application must explicitly open an object before it executes any of the object's member functions. The exceptions are limited to meta-level operations such as `==`, `!=`, and `Status`. The `mode` argument determines the subset of member functions that the application will be allowed to execute (see Section 2.2.1). `Open` will fail (return non-zero) if the user lacks the proper access authorizations for the requested mode and if the object has been previously opened (but not closed) by the application. In all cases, `Open` should be followed by an explicit `Close`. Failure to close an object could result in lost data since changes are not considered permanent until after a successful `Close`.

USAGE:

```
main() {
   CSDgsConnection dgs;
   dgs.Initialize();
   CSDgsTree *root_subgraph = dgs.GetRootSubgraph();
   root_subgraph->Open(DGS_READ);
   ...
   root_subgraph->SomeReadOperation();
   ...
   root_subgraph->Close();
}
```

4.2.2 Object Equality

dgs_bool CSDgsObject::operator==(const CSDgsObject*) const;
dgs_bool CSDgsObject::operator!=(const CSDgsObject*) const;
dgs_bool CSDgsObject::operator==(const CSDgsObject&) const;
dgs_bool CSDgsObject::operator!=(const CSDgsObject&) const;

ARGUMENTS:

another object

COMMENT:

These functions determine whether or not two CSDgsObjects refer to the same physical object in permanent storage. Beware; direct comparison of pointer values is not sufficient to determine whether two CSDgsObject*s refer to the same object. It is possible for two pointers to have different values but refer to the same object. To determine equality, you should call the `operator==` or `operator!=` function of CSDgsObject.

USAGE:

```
CSDgsConnection dgs;
dgs.Initialize();
CSDgsTree *root_subgraph = dgs.GetRootSubgraph();
CSDgsTree *same_tree = dgs.GetRootSubgraph();
if (root_subgraph != same_tree) {
    }  // BAD! Do not compare pointers directly!
if (*root_subgraph != *same_tree) {
    }  // GOOD! This is the proper way to compare two objects.
```

4.2.3 Status Information

DgsStatus CSDgsObject::Status();

COMMENT:

`Status` returns an instance of the class `DgsStatus`. This function can be called before `Open`. DgsStatus responds to a number of predicates such as `IsOpen`, `NotOpen`, `IsWriteable`, `NotWriteable`, `IsReadable`, `NotReadable`, `IsAnchorable`, and `NotAnchorable`.

USAGE:

```
// "root_subgraph" is the CSDgsTree* from previous examples.
DgsStatus status = root_subgraph->Status();
if (status.IsOpen() == dgs_true)
    // the object is already open
else
    // the object is not open
if (status.IsWriteable() == dgs_true)
    // the object is open in DGS_WRITE mode.
```

4.2.4 Object Type

int CSDgsObject::**TypeOfObject();**
dgs_bool CSDgsObject::**IsGraph();**
dgs_bool CSDgsObject::**IsStGraph();**
dgs_bool CSDgsObject::**IsHyGraph();**
dgs_bool CSDgsObject::**IsComponent();**
dgs_bool CSDgsObject::**IsNode();**
dgs_bool CSDgsObject::**IsLink();**
dgs_bool CSDgsObject::**IsStLink();**
dgs_bool CSDgsObject::**IsHyLink();**
dgs_bool CSDgsObject::**IsTree();**
dgs_bool CSDgsObject::**IsNetwork();**
dgs_bool CSDgsObject::**IsList();**

COMMENT:

`TypeOfObject` returns the type of the object. The following types are defined: `DGS_NODE`, `DGS_SLINK`, `DGS_HLINK`, `DGS_HGRAPH`, `DGS_TREE`, `DGS_NETWORK`, and `DGS_LIST`. The functions that begin with "Is" return `dgs_true` if the object satisfies the predicate and `dgs_false` otherwise. For example, `IsStGraph` returns `dgs_true` if the object is an S-subgraph. That is, it returns true if the object is a tree, list, or network.

All of these functions can be called even if the object has not been opened.

USAGE:

```
// CSDgsConnection dgs;  is the same as in previous examples.
CSDgsTree *root_graph = dgs.GetRootSubgraph();
if ( root_graph->TypeOfObject() != DGS_TREE )
     cerr << "Error: expected a tree\n";
if ( root_graph->IsGraph() == dgs_false )
     cerr << "error: should have returned dgs_true\";
```

4.2.5 GetOid

UUID CSDgsObject::**GetOid**();

COMMENT:

GetOid returns the unique object identifier (OID) of the object. The function CSDgsConnection::GetObjectByOid() can be used to retrieve an object based on its OID. However, in most cases this is discouraged (see Section 3). The OID interface is provided primarily as a method for two processes to communicate the identity of an object via Inter-Process Communication (IPC).

USAGE:

```
PROCESS A
  // CSDgsConnection dgs;  // is the same as in previous examples.
  CSDgsTree *root_graph = dgs.GetRootSubgraph();
  UUID tree_oid = root_graph->GetOid();
  // PROCESS A sends "tree_oid" to PROCESS B via IPC

PROCESS B
  // UUID oid;              // is received via IPC from another process
  CSDgsObject *object = dgs.GetObjectByOid(oid);
  object->Open(DGS_READ);
  switch ( object->TypeOfObject() )
     case DGS_TREE: // it is a tree
        CSDgsTree *tree = (CSDgsTree *)object;
        CSDgsNode *rnode = tree->Root();
        // "rnode" is a pointer to the root of the tree
        break;
     default:
        cerr << "Error: expected a tree\n";
  }
  object->Close();
```

4.2.6 Common Attributes

An attribute is a typed, named variable that applications can define at run-time. Some attributes are called *common attributes* because their values are the same in all contexts. Other attributes are called *graph attributes* because they are dependent on the context of a particular subgraph. Section 2.1.5 discusses the difference between common attributes and graph attributes in more detail. Common attributes are the focus of this section, although most of the discussion applies equally to graph attributes. The API for common attributes is provided by the `CSDgsObject` class. All objects (nodes, links, and subgraphs) can have common attributes. The interface for graph attributes is provided by the `CSDgsGraph` class (see Section 4.6.2). Only nodes and links can have graph attributes.

The *name* of an attribute is a text string that must be unique within the scope of the object that owns the attribute. The *value* of an attribute must be one of the types shown in Figure 10. Most of these attribute types are self-explanatory; for example, `CSDgsIntAttr` is an integer attribute and `CSDgsFloatAttr` is a floating point attribute. The class `DgsAttr` defines a function called `TypeOfAttr` that is inherited by all attribute types. `TypeOfAttr` returns an integer with one of the following values: DGS_INT, DGS_STRING, DGS_FLOAT, DGS_DOUBLE, DGS_OBJECT, or DGS_BYTE_ARRAY. For example, a string attribute returns DGS_STRING.

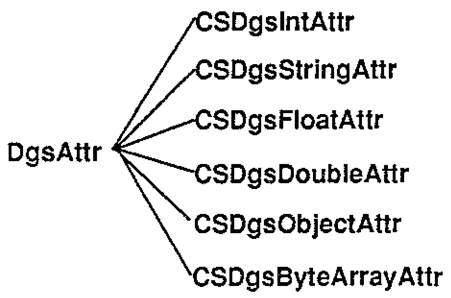


Figure 10: Attribute Classes

An attribute of type `CSDgsObjectAttr` has a value that is a pointer to a `CSDgsObject`. For example, suppose that we are writing an application to execute finite automata. A finite automaton could be stored in the DGS as a subgraph. Nodes in the subgraph could represent states and the links could represent transitions from one state to the next. We could use attributes on the links to store the conditions under which the transitions will be followed. Given this application, an attribute of type `CSDgsObjectAttr` is ideal for indicating the starting and final states of the automaton. For example, we might create a common attribute on the subgraph (call the attribute "Start State") and set its value to be a pointer to the node that represents the starting state. An alternative solution would be to store the OID of the starting node directly; however, we prefer to use a `CSDgsObjectAttr` for reasons that are enumerated in Section 3.

An attribute of type `CSDgsByteArrayAttr` is used to store an arbitrarily large array of bytes. This is useful if the application has complicated data structures that cannot be easily represented using the other types of attributes. Normally, when an object is accessed on a different machine architecture from the one on which it was created, the DGS transparently changes the byte-order of the data to match the new machine environment. However, the DGS cannot perform this service for byte-array attributes because it does not know anything about the structure of the data within the attribute. Consequently, applications must perform their own byte-reordering for data that is stored in a `CSDgsByteArrayAttr`.

The following functions are used to manipulate common attributes:

int CSDgsObject::**CreateAttr**(const DgsString& **name**);
int CSDgsObject::**CreSetAttr**(const DgsString& **name**, const DgsAttr& **value**);

ARGUMENTS:

`name`:	the text name of the attribute
`value`:	the value of the new attribute

COMMENT:

Use these functions to create a *new* common attribute. These functions fail (return non-zero) if an attribute already exists with the same name and if the object is not open in DGS_WRITE mode. `CreSetAttr` creates an attribute and sets its value in a single operation. `CreateAttr` creates the attribute and gives it a null value.

int CSDgsObject::**ChangeAttr**(const DgsString& **name**, const DgsAttr& **value**);
int CSDgsObject::**SetAttr**(const DgsString& **name**, const DgsAttr& **value**);

ARGUMENTS:

`name`:	the text name of the attribute
`value`:	the new value of the attribute

COMMENT:

Use `ChangeAttr` to set the value of a pre-existing common attribute. It will fail (return non-zero) if the attribute does not exist. Use `SetAttr` when you are not sure whether the attribute exists or not. If the attribute does not exist, `SetAttr` will automatically create it. Both functions fail if the object has not been previously opened in DGS_WRITE mode.

DgsAttr& CSDgsObject::**GetAttr**(const DgsString& **name**);
dgs_bool CSDgsObject::**HasAttr**(const DgsString& **name**);

int CSDgsObject::**DeleteAttr**(const DgsString& **name**);

ARGUMENTS:

`name`:	the text name of the attribute

COMMENT:

`GetAttr` returns the value of an attribute or a null value if the attribute does not exist. `HasAttr` returns `dgs_true` if the attribute exists and `dgs_false` if it does not exist. `DeleteAttr` deletes a common attribute. `GetAttr` and `HasAttr` fail if the object has not been opened. `DeleteAttr` fails if the object has not been opened in DGS_WRITE mode.

We conclude this section by discussing some C++ code that uses the attribute functions. Our example begins by opening a tree in DGS_WRITE mode.

```
// 'root_subgraph' is a CSDgsTree*.
 root_subgraph->Open(DGS_WRITE);  // always open the object first.
```

Next, we create two integer attributes in the object. Notice that there are two ways to specify the value. In the first method, an instance of the class CSDgsIntAttr is explicitly created and then passed to the SetAttr call. In the second case, the program casts an integer to CSDgsIntAttr just before passing it to SetAttr.

```
// create an int attribute with the name "age of tree" and the value 26
 CSDgsIntAttr age(26);
 root_subgraph->SetAttr( "age of tree", age );
// create an int attribute with the name "height of tree" and the value 14
 root_subgraph->SetAttr( "height of tree", (CSDgsIntAttr) 14 );
```

Once the attributes have been created, we can use GetAttr to read their values. Notice that the program does not explicitly cast the CSDgsIntAttr back to an integer. The DGS defines conversion functions that do this automatically. The DGS also defines automatic conversion functions for CSDgsStringAttr, CSDgsFloatAttr, CSDgsDoubleAttr, and CSDgsObjectAttr.

```
// get back the value and verify it.
 CSDgsIntAttr height = root_subgraph->GetAttr( "height of tree" );
 if (height != 14) cerr << "error\n";
```

The next piece of code creates a string attribute with the name "title" and the value "The Root Subgraph".

```
// Create an attribute with the name "title".
// Note: CreateAttr gives it a null value by default.
 root_subgraph->CreateAttr( "title" );
// Verify that the value is null.
 if ( (root_subgraph->GetAttr("title)).IsNull() == dgs_true )
    cout << "the value of the attribute is null.\n";

// Now, give the attribute a string value and verify it.
 root_subgraph->ChangeAttr( "title", (CSDgsStringAttr) "The Root Subgraph" );
```

Then, it verifies that the correct value was stored. Notice that C++ automatically converts the second parameter of strcmp() to a char*.

```
CSDgsStringAttr value = root_subgraph->GetAttr( "title" );
if ( strcmp("The Root Subgraph", value) != 0 )
   cerr << "error: expected a different value\n";
```

Next, we demonstrate the use of HasAttr and DeleteAttr.

```
if ( root_subgraph->HasAttr( "age of tree" ) == dgs_false )
   cerr << "the attribute should exist\n";
root_subgraph->DeleteAttr( "age of tree" );
if ( root_subgraph->HasAttr( "age of tree" ) == dgs_true )
   cerr << "the attribute should not exist!\n";
```

Now, we will look more closely at CSDgsObjectAttr. We create an an attribute with an object as its value, then, we verify its value. Notice that the class CSDgsObjectAttr can convert itself to any type of object pointer, e.g., CSDgsTree*, CSDgsGraph*, etc.

```
// "me" is an object attribute that points to 'root_subgraph'
 root_subgraph->SetAttr( "me", (CSDgsObjectAttr) root_subgraph );
 CSDgsTree *me = (CSDgsObjectAttr) root_subgraph->GetAttr( "me" );
 if ( *root_subgraph != *me )
    cerr << "they should be pointers to the same object\n";
```

Finally, we illustrate how to use CSDgsByteArrayAttr and then close the subgraph to make our changes permanent.

```
// "blob" is a byte-array attribute
 char *array = new char[80];  // allocate 80 bytes
 root_subgraph->SetAttr( "blob", CSDgsByteArrayAttr(80,array) );
 delete array;  // don't forget to free the memory when you are through
 CSDgsByteArrayAttr bytes = root_subgraph->GetAttr( "blob" );
 array = bytes.CharPtr();
 int size = bytes.Size();
 if ( size != 80 ) cerr << "the attribute should be 80 bytes\n";

// don't forget to close the subgraph to make the changes permanent
 root_subgraph->Close();
```

In the example, the application had to know the name of an attribute before it could access it. Next, we present three functions that are useful when the name of the attribute is not known.

int CSDgsObject::**NumOfAttrs**();
DgsString CSDgsObject::**FirstAttrName**();
DgsString CSDgsObject::**NextAttrName**(const DgsString& **last_name**);

ARGUMENTS:

`last_name`:	the name that was returned by the previous invocation of `NextAttrName`

COMMENT:

`NumOfAttrs` returns the number of common attributes that an object has. `FirstAttrName` and `NextAttrName` can be used to read the names of the common attributes one-by-one. These names could then be used as input to other attribute functions such as `GetAttr`. `NextAttrName` returns `dgs_nullstr` when there are no more attribute names to read. Before calling any of these functions, the object must have been successfully opened.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*.
cout << "Number of Common Attributes: ";
cout << root_subgraph->NumOfAttrs() << endl;

// Print the values of root_subgraph's string attributes.
for ( DgsString name = root_subgraph->FirstAttrName();
      name != dgs_nullstr;
      /* this space intentionally blank */          ) {
    DgsAttr *attr = root_subgraph->GetAttr( name );
    switch ( attr->TypeOfAttr() )  {
       case DGS_STRING:
          cout << "String(" << *( (CSDgsStringAttr *) attr ) << ")\n";
          break;
       default:
          break;
    }
    name = root_subgraph->NextAttrName( name );
}
```

int CSDgsObject::**CopyAttrsFrom**(CSDgsObject* **another_object**);

ARGUMENTS:

another_object: a pointer to an object with common attributes

COMMENT:

This function copies all of the common attributes of another_object to the object that is invoking the function (the receiver). The function will fail (return non-zero) if the receiver is not open in DGS_WRITE mode and if the user does not have read authorization for another_object. If the two objects have attributes with the same name, then the values of the receiver's attributes will be overwritten.

USAGE:

```
// 'root_subgraph' is a CSDgsTree*.
root_subgraph->Open(DGS_WRITE);
CSDgsNode *root_node = root_subgraph->Root();

// Copy all of the common attributes from 'root_node' to
// 'root_subgraph'.
root_subgraph->CopyAttrsFrom( root_node );

// do not forget to close the subgraph.
root_subgraph->Close();
```

CSDgsObject has two subclasses: CSDgsComp (components) and CSDgsGraph (subgraphs). These classes and their descendants inherit all of the functions that were described in this section. They also extend these basic functions by specializing their API. The following sections describe these extensions in detail.

4.3 CSDgsComp (components)

A *component* is a node or a link that belongs to at least one subgraph. Functions that are common to both nodes and links are implemented in the class `CSDgsComp`. This includes functions to create and delete content as well as functions to test for subgraph membership. When a component is removed from the last subgraph that contains it, the component is automatically garbage-collected by the DGS and ceases to exist. When a component is destroyed, its content also ceases to exist.

The following sections describe the API of `CSDgsComp` in more detail.

4.3.1 Subgraph Membership

dgs_bool CSDgsComp::**IsContainedIn**(CSDgsGraph *subgraph);
int CSDgsComp::**NumOfStGraphs();**
int CSDgsComp::**NumOfHyGraphs**();

COMMENT:

`IsContainedIn` returns `dgs_true` if the component is contained in the subgraph and `dgs_false` otherwise. `NumOfStGraphs` and `NumOfHyGraphs` return the number of S-subgraphs and the number of HS-subgraphs that contain the component, respectively. The application must open the component before invoking any of these functions.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*
CSDgsNode *root_node = root_subgraph->Root();

root_node->Open(DGS_READ);
if ( root_node->IsContainedIn( root_subgraph ) == dgs_false )
   cerr << "expected the node to be contained in the tree\n";

int S_subgraphs = root_node->NumOfStGraphs();
int HS_subgraphs = root_node->NumOfHyGraphs();
cout << "'root_node' is contained in " << S_subgraphs << " S-subgraphs\n";
cout << "and " << HS_subgraphs << " HS-subgraphs.\n";

root_node->Close();  // do not forget to close the node
```

CSDgsStGraph* CSDgsComp::**FirstStGraph();**
CSDgsHyGraph* CSDgsComp::**FirstHyGraph();**
CSDgsStGraph* CSDgsComp::**NextStGraph(** const CSDgsStGraph* **last**);
CSDgsHyGraph* CSDgsComp::**NextHyGraph(** const CSDgsHyGraph* **last**);

ARGUMENTS:

`last`: the subgraph that was returned by the previous invocation

COMMENT:

These functions can be used to find the subgraphs that contain a particular component. `FirstStGraph` or `FirstHyGraph` must be called first, then `NextStGraph` or `NextHyGraph` can be called iteratively until all of the subgraphs have been found. All four functions return `nil` when there are no more subgraphs to be found. Before using any of these functions, the application must open the component.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*.
CSDgsNode *root_node = root_subgraph->Root();
root_node->Open(DGS_READ);

// Read all of the S-subgraphs that contain 'root_node' and print the
// type of the subgraphs.
for ( CSDgsStGraph *StG = root_node->FirstStGraph();
      StG != nil;
      /* this space intentionally blank */            )  {
    switch ( StG->TypeOfObject() )  {
       case DGS_TREE:
          cout << "Tree\n";                        break;
       case DGS_LIST:
          cout << "List\n";                        break;
       case DGS_NETWORK:
          cout << "Network\n";                     break;
       default:
          cerr << "Unknown S-subgraph type!\n";   break;
    }
    StG = root_node->NextStGraph( StG );
}
root_node->Close();  // do not forget to close the node
```

4.3.2 Testing for the Presence of Content

int CSDgsComp::**FormOfContent();**
int CSDgsComp::**TypeOfContent();**
dgs_bool CSDgsComp::**IsEmpty();**
dgs_bool CSDgsComp::**NotEmpty();**

COMMENT:

These functions provide information about the presence or absence of content. `FormOfContent` describes the basic form of the content by returning one of three values: DGS_EMPTY_CONTENT, DGS_FILE_CONTENT, or DGS_GRAPH_CONTENT. If the form of the content is DGS_EMPTY_CONTENT then `IsEmpty` returns `dgs_true`, otherwise `dgs_false`. When the content is a subgraph, `TypeOfContent` is useful since it returns the type of the subgraph: DGS_FILE, DGS_EMPTY, DGS_TREE, DGS_NETWORK, DGS_LIST or DGS_HGRAPH. Note: these functions only work on an open component.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*
CSDgsNode *root_node = root_subgraph->Root();

root_node->Open(DGS_READ);
cout << "The node has ";
switch ( root_node->TypeOfContent() )  {
   case DGS_FILE:
     cout << "File"; break;
   case DGS_EMPTY:
     cout << "Empty"; break;
   case DGS_TREE:
     cout << "Tree"; break;
   case DGS_NETWORK:
     cout << "Network"; break;
   case DGS_LIST:
     cout << "List"; break;
   case DGS_HGRAPH:
     cout << "HS-subgraph"; break;
}
cout << " Content\n";

root_node->Close();  // do not forget to close the node
```

4.3.3 Creating and Deleting Content

DgsString CSDgsComp::**CreateFileContent();**
DgsString CSDgsComp::**CopyFileContent(** DgsString **fname**);

ARGUMENTS:

`fname`: an absolute pathname for a pre-existing file

COMMENT:

Both functions initialize the content of an empty component by creating a new file in the application's file space. `CreateFileContent` creates an empty file, whereas `CopyFileContent` copies a pre-existing file. Both functions fail (return `dgs_nullstr`) if the component already has content and if the component is not open in DGS_WRITE mode.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*
CSDgsNode *root_node = root_subgraph->Root();

root_node->Open(DGS_WRITE);
DgsString new_name = root_node->CreateFileContent();

if ( new_name == dgs_nullstr )

   cerr << "Could not create content.\n";

else  {

   ofstream new_file( new_name );  // open the file
   if (!new_file)  cerr << "Could not open file.\n";
   ...
   // perform write operations on file.
   ...
   new_file.close();    // do not forget to close the file
}

root_node->Close();  // do not forget to close the node
```

CSDgsNetwork* CSDgsComp::**CreateNetworkContent**(CSDgsGraph* **neighbor**);
CSDgsTree* CSDgsComp::**CreateTreeContent**(CSDgsGraph* **neighbor**);
CSDgsList* CSDgsComp::**CreateListContent**(CSDgsGraph* **neighbor**);
CSDgsHyGraph* CSDgsComp::**CreateHyGraphContent**(CSDgsGraph* **neighbor**);

ARGUMENTS:

`neighbor`: a pointer to an open graph that contains the node.

COMMENT:

These functions create a new subgraph and make it the content of the component. The DGS stores the new subgraph in a physical location that is near the `neighbor` object that is passed as a parameter to each function. The functions fail (return `nil`) if the component already has content and if the component is not open in DGS_WRITE mode.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*
CSDgsNode *root_node = root_subgraph->Root();

root_node->Open(DGS_WRITE);
CSDgsTree *new_tree = root_node->CreateTreeContent();

if ( new_tree == nil )

   cerr << "CreateTreeContent() failed.\n";

else  {

   new_tree->Open(DGS_WRITE);
   ...
   // perform write operations on the tree.
   ...
   new_tree->Close();

}

root_node->Close();  // do not forget to close the node
```

int CSDgsComp::**DeleteContent();**

COMMENT:

DeleteContent destroys the content of the component. If the content is a file, then the file is deleted. If the content is a subgraph, then subgraph must be empty. DeleteContent will fail if the subgraph contains any nodes or links. DeleteContent will also fail (return non-zero) if there is no content and if the component is not open in DGS_WRITE mode.

USAGE:

```
// 'root_subgraph' is an open CSDgsTree*
CSDgsNode *root_node = root_subgraph->Root();

root_node->Open(DGS_WRITE);
CSDgsNetwork *new_network = root_node->CreateNetworkContent();

if ( new_network == nil )
   cerr << "CreateNetworkContent() failed.\n";
else
   root_node->DeleteContent();

root_node->Close();  // do not forget to close the node
```

CSDgsComp has two subclasses: CSDgsNode and CSDgsLink. These classes and their descendants inherit all of the functions that were described in this section. They also extend these basic functions by specializing their API to a more narrow purpose. The following sections describe these extensions in detail.

4.4 CSDgsNode (nodes)

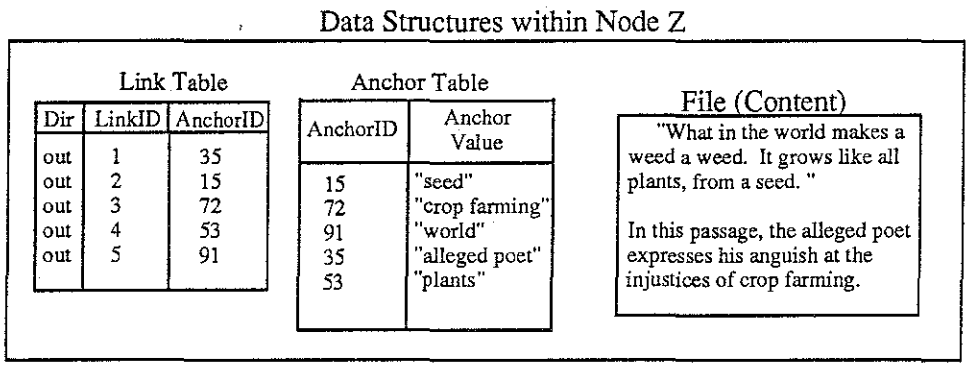
A *node* is a component of a subgraph whose primary purpose is to store a chunk of information. Three mechanisms are provided for storing this information: common attributes, graph attributes, and content. The APIs for common attributes and graph attributes are defined in other classes (see Sections 4.2.6 and 4.6.2). In addition, the `CSDgsComp` class provides some but not all of the content functions. Consequently, the primary responsibility of the `CSDgsNode` class is to provide the remainder of the API for content-related functions. This includes functions to access file and subgraph content as well as functions for creating and maintaining anchor points within that content.

As we illustrated in Section 2.1.4, users sometimes need to create links that have fine-grained endpoints such as a single word within a paragraph of text. Unfortunately, content is not a first-class object within the DGS data model. Thus, although the DGS can manipulate nodes, links, and subgraphs directly, it is forced to treat file content as an uninterpreted black box. To put it another way, the DGS provides concurrency control, security, and permanence to content, but only the application that created the content has the knowledge to manipulate it. As a result, only an application that understands the content of a node can define and maintain the consistency of a fine-grained endpoint (called an *anchor*). The DGS can store the description of the anchor (called the anchor's *value*), but applications are solely responsible for the validity of this information. This division of responsibility between the DGS and its applications underlies all of the mechanisms that we describe next.

Along with the content of a node, the DGS also stores two tables: an anchor table and a link table. The anchor table has entries of the form: (AnchorID, Value). The *AnchorID* is an identifier generated by the DGS (returned by the `CSDgsNode::AnchorAtTarget()` and `CSDgsNode::AnchorAtSource()` functions) that is unique within the context of the node that contains the anchor table. The *value* is an application-defined description of a fine-grained endpoint within the content of the node. An entry in the anchor table is called an *anchor*.

The link table is used to associate a particular anchor with the set of links that reference it. An entry in the link table is a tuple of the form: (Direction, LinkID, AnchorID). *LinkID* is the OID of an anchored link that is associated with the anchor specified by AnchorID. *Direction* indicates whether the anchor is associated with the source node of the link or with the target node of the link. Obviously, if the direction is "incoming", then the node that contains the anchor must be the target node of the associated link. Similarly, if the direction is "outgoing", then the node must be the source node of the link. The association between anchors and links is many-to-many. Every link *can* be associated with zero or more anchors and every anchor *must* be associated with one or more links.

Figure 11 shows the anchor and link tables associated with a node (call it Node Z). Node Z has five anchors that are fine-grained endpoints for five different links. Since Node Z is the source node of these links, the direction of the anchors is outward. We can use the tables to answer questions such as "Which anchor is associated with Link 5?" For example, in the link table, we see that Link 5 is associated with AnchorID 91. Next, the anchor table tells us that AnchorID 91 has the value "world". Thus, the fine-grained endpoint of Link 5 is the word "world" in the first paragraph of the node's file content.


Data Structures within Node Z

Link Table

Dir	LinkID	AnchorID
out	1	35
out	2	15
out	3	72
out	4	53
out	5	91

Anchor Table

AnchorID	Anchor Value
15	"seed"
72	"crop farming"
91	"world"
35	"alleged poet"
53	"plants"

File (Content)

"What in the world makes a weed a weed. It grows like all plants, from a seed. "

In this passage, the alleged poet expresses his anguish at the injustices of crop farming.

Figure 11: A Node with File Content and Five Anchors

Most hypertext applications highlight anchors when they display the content of a node. For instance, a hypertext application might display Node Z with the word "world" highlighted. This emphasis would indicate that the word is the endpoint for at least one link. By double-clicking on "world", a user could follow Link 5 and see the content of the node at the other end of the link.

Because content and anchor values are application-specific, an application should not be allowed to change an anchor unless it knows the structure of the content as well as the format of the anchor value. For example, we would not want to edit a Microsoft Word document with a simple text editor (emacs or vi), since it would not understand the special format of the Word file. For the same reason, an application should not modify the content or anchors associated with a node unless it understands the format of both. To prevent accidental corruption of the anchor table, the DGS requires applications to register the format of the anchor table. The DGS will not allow an application to modify the content and anchors of a node unless the application knows the registered format of the node's anchor table.

When an application gets the content of a node, it also receives a copy of the node's anchor table and a copy of the link table. If the node was opened in DGS_WRITE mode, then the application is responsible for keeping these tables up-to-date with the content. Consequently, if the application changes the content, it must also update the anchor table.

To illustrate, let's consider a text editor that encodes its anchor values as byte offsets into the text file. Thus, instead of using "world" as the value for AnchorID 91 (in Figure 11), the text editor would encode the value of the anchor as 16 (since "world" begins on the 16th byte of the file). Now, suppose that a user opens the node in DGS_WRITE mode and begins editing the content with the text editor. Next, the user changes the phrase "What in the world" to "What in this world" and closes the node. Obviously, the text editor is responsible for updating the content to reflect the new phrasing. However, the application is also responsible for updating the values of *all* of the anchors that appear after the modified phrase. More specifically, the application must add 1 to the value of each anchor in the anchor table. For example, the anchor value of AnchorID 91 should be changed from 16 to 17. Just before closing the node, the application must pass the updated anchor table to the DGS as a parameter to the `CSDgsNode::PutContent()` function.

Applications have fewer responsibilities when they open a node in DGS_READ or DGS_READ_NO_ANCHOR mode. In both cases, the DGS prevents applications from changing the content or the values of pre-existing anchors. However, DGS_READ mode does not prevent applications from creating new anchors using the `CSDgsNode::AnchorAtSource()` and `CSDgsNode::AnchorAtTarget()` function calls, nor does it prevent an application from removing old anchors using the `CSDgsNode::UnAnchor()` function. When a DGS_READ or DGS_WRITE application creates a new anchor it must transfer the value of the new anchor to the DGS as a parameter to the `CSDgsNode::PutContent()` function just before closing the node.

The following sections describe the API of `CSDgsNode` in more detail.

4.4.1 Accessing Content and Anchors

```
CSDgsContent CSDgsNode::GetContent( DgsLinkTable*& link_tbl,
                                    DgsAnchorTable*& anchor_tbl,
                                    [ int password ]              );
```

ARGUMENTS:

`link_tbl`:	returns a pointer to the node's link table
`anchor_tbl`:	returns a pointer to the node's anchor table
`password`:	used to prevent accidental damage to anchor information

COMMENT:

`GetContent` retrieves the content, link table, and anchor table of a node. When the node is opened in DGS_READ or DGS_WRITE mode, then the `password` parameter must match the type of the node's anchor table (as registered in a previous `PutContent` function call). The parameter is optional when the node is opened in DGS_READ_NO_ANCHOR mode and if the type of the node's anchor table is zero (the default value). The password is intended to prevent naive applications from accidentally damaging the anchor table. `GetContent` will fail (return empty content) if the correct password is not specified (when required) and if the node is not open.

`GetContent` returns an instance of the class `CSDgsContent`. This class responds to the `FormOfContent`, `TypeOfContent`, `IsEmpty`, and `NotEmpty` functions that are described in Section 4.3.2. Moreover, `CSDgsContent` will convert itself automatically to other classes. For example, `CSDgsContent` defines a conversion function that allows file content to convert itself to a `char*` that points to the name of the file. Similarly, subgraph content can convert itself to an appropriate object pointer such as `CSDgsTree*`.

Accessing the content of a node is a three step process:

1. Execute `GetContent` to obtain an instance of `CSDgsContent`.
2. Convert the `CSDgsContent` to a filename or object pointer, depending on whether the content is a file or a subgraph.
3. Open the file or object using a UNIX iostream or the DGS `Open` function.

For example,

```
{
    ...
    DgsLinkTable *link_table;
    DgsAnchorTable *anchor_table;
    // We assume that the anchor table type is zero, so that the
    //    password is not required by GetContent().
    CSDgsContent content = node->GetContent( link_table, anchor_table );
    switch ( content->TypeOfContent() )  {
       case DGS_FILE:
          char *filename = content;
          ifstream in( filename );
          // read from file
          in.close();
          break;
       case DGS_TREE:
          CSDgsTree *tree = content;
          tree->Open( DGS_READ );
          ...
          tree->Close();
          break;
       case DGS_LIST:
          break;
       case DGS_NETWORK:
          break;
       default:
    }
    delete link_table;
    delete anchor_table;
    ...
}
```

int CSDgsNode::**PutContent**(DgsAnchorTable* **old_anchors**,
DgsAnchorTable* **new_anchors** [, int **table_type**]);

ARGUMENTS:

`old_anchors`:	a pointer to an anchor table that contains the new values of the pre-existing anchors
`new_anchors`:	a pointer to an anchor table that contains the newly-created anchors
`table_type`:	used to register the type of the node's anchor table

COMMENT:

`PutContent` transfers a modified anchor table from the application to the DGS for storage. Applications should call it before `Close` if they have modified the anchor table in any way. The optional parameter `table_type` is used to register the format of the anchor table with the DGS. All anchor-aware applications should register the table_type when they create new content. If the table_type parameter to `PutContent` is omitted, then table_type retains its previous value (Note: table_type defaults to zero). Applications should **not** specify a table_type if they do not intend to support anchors. When table_type has a non-zero value, then applications may be required to supply it as a parameter to `GetContent`. This mechanism prevents naive applications from accidentally damaging the anchor table.

`PutContent` fails (returns non-zero) if the node is not open in DGS_READ or DGS_WRITE mode. When the node is opened in DGS_WRITE mode, then the `old_anchors` parameter to `PutContent` should contain a pointer to an updated copy of the anchor table that was returned by `GetContent`. That is, the values of the anchors should reflect any changes that were made to the content. Also, the application should remove any anchors that were deleted. The `new_anchors` parameter should be a pointer to a table containing all of the newly-created anchors. If one or both of the parameters is `nil`, then the DGS will assume that the tables are empty. The DGS ignores `old_anchors` when the node is opened in DGS_READ mode.

Usage:

```
node->Open( DGS_WRITE );

node->CreateFileContent();  // the file is created by the DGS
DgsLinkTable *link_table;
DgsAnchorTable *anchor_table;
CSDgsContent content = node->GetContent( link_table, anchor_table );

char *filename = content;      // convert content to a char*
ofstream cfile( filename );    // open an iostream on the file
// write to file
cfile.close();                 // close the file

// register the type of the anchor table (table_type = 26)
node->PutContent( nil, nil, 26 );

CSDgsContent content2, content3;
content2 = node->GetContent( link_table, anchor_table );
content3 = node->GetContent( link_table, anchor_table, 3 );
content = node->GetContent( link_table, anchor_table, 26 );

if ( content2.NotEmpty() == dgs_true ||
     content3.NotEmpty() == dgs_true || content.IsEmpty() == dgs_true )  {
   cerr << "The first two GetContent's should have failed";
   cerr << " because they did not specify the correct password.\n";
   cerr << "The last GetContent should have succeeded.\n"
}

node->Close();  // the file is stored and deleted by the DGS
```

4.4.2 Creating and Deleting Anchors

This section describes the node API for creating and deleting anchors. For now, we will just describe the functions. The following section provides concrete examples and introduces the anchor-related classes such as `DgsAnchorTable` and `DgsLinkTable`.

AnchorID CSDgsNode::**AnchorAtSource**(CSDgsLink* **link** [, AnchorID **anchor**]);
AnchorID CSDgsNode::**AnchorAtTarget**(CSDgsLink* **link** [, AnchorID **anchor**]);
int CSDgsNode::**UnAnchor**(CSDgsLink* **link**, AnchorID **anchor**);

ARGUMENTS:

`link:`	a pointer to a link
`anchor:`	the AnchorID of a pre-existing anchor

COMMENT:

Applications must call `GetContent` before using any of these functions. `AnchorAtSource` and `AnchorAtTarget` tell the DGS to create an association between an anchor and link by adding a new entry in the node's link table. Applications should use `AnchorAtSource` when the node is the source node of the link and `AnchorAtTarget` when it is the target node of the link. The `UnAnchor` function tells the DGS to remove an entry from the link table. Note: these functions do not change the application's local copy of the link table; however, they do change the copy of the link table that is maintained by the DGS. The application should also keep its local copy up-to-date. Furthermore, none of these functions affect the anchor table. Thus, the application is responsible for adding new anchors to its anchor table and it should transfer the new anchors to the DGS as a parameter to the `CSDgsNode::PutContent()` function. `PutContent` should be called just before `Close`.

The optional argument `AnchorID` is used to associate a link with a pre-existing anchor. If it is omitted, then a new anchorID is allocated by the DGS and its value is returned by the `AnchorAt` function. These functions will fail (return a null anchor or non-zero) if the node is not open in DGS_READ or DGS_WRITE mode. They will also fail if `GetContent` failed.

4.4.3 Anchor-related Classes

In addition to `CSDgsNode`, a number of other classes are required to create and manipulate anchors. The most important of these are `DgsLinkTable`, `DgsLinkEntry`, `DgsAnchorTable`, and `DgsAnchorEntry`. In this section, we will describe the API for these classes and provide a detailed example of their use.

The class `DgsLinkTable` is a table of `DgsLinkEntry`'s. Each entry is a tuple of the form: (Direction, Link, AnchorID). The meaning of these tuples has been described previously. Here, we will describe the programming interface. `DgsLinkTable` defines the following functions:

- Adding and Deleting Entries
  - void AddEntry(DgsLinkEntry *entry);
  - DgsLinkEntry *AddEntry(char direction, CSDgsLink *link, AnchorID anchor);
  - void RemoveEntry(DgsLinkEntry *entry);
  - void RemoveEntry(CSDgsLink *link, AnchorID anchor [, char direction]);
- Enumerating Entries
  - int NumOfEntries();
  - DgsLinkEntry *FirstEntry();
  - DgsLinkEntry *NextEntry(DgsLinkEntry *last_entry);
- Finding Entries and Using Selection to Create SubTables
  - DgsLinkEntry *FindEntry(UUID linkID, AnchorID anchor);
  - DgsLinkTable *Select(CSDgsLink *link [, char direction]);
  - DgsLinkTable *Select(AnchorID anchor [, char direction]);

Most of these functions are self-explanatory or have been seen in other contexts. For example, `FirstEntry/NextEntry` is a construct that has appeared previously in the context of Section 4.3.1's `FirstStGraph/NextStGraph` functions. The `Select` function is new, but it is analogous to the select function of relational database models. `Select` returns the subset of the entries in the original table that satisfy the selection criteria. More specically, it allows us to isolate all of the table entries that involve a particular link or a particular anchor. The direction parameter is optional.

Each entry in a `DgsLinkTable` is an instance of the class `DgsLinkEntry` which has the following API:

- CSDgsLink *Link();
- AnchorID AnchorID();
- char Direction();
- dgs_bool IsIncoming();
- dgs_bool IsOutgoing();

The first three functions return the values of the three parts of the tuple. `Direction` returns either DGS_INCOMING or DGS_OUTGOING. The last two functions are shortcuts for testing the direction.

The class `DgsAnchorTable` is a table of `DgsAnchorEntry`'s. Each entry is a tuple of the form: (AnchorID, Value_Length, Value). The meaning of these tuples has been described previously. `Value` is a byte array of length `Value_Length`. The API for `DgsAnchorTable` is very similar to the API for `DgsLinkTable`.

- Adding and Deleting Entries
  - void AddEntry(DgsAnchorEntry *entry);
  - DgsAnchorEntry *AddEntry(AnchorID anchor, int value_length, char *value);
  - void RemoveEntry(DgsAnchorEntry *entry);
  - void RemoveEntry(AnchorID anchor);
- Enumerating Entries
  - int NumOfEntries();
  - DgsAnchorEntry *FirstEntry();
  - DgsAnchorEntry *NextEntry(DgsAnchorEntry *last_entry);

- Finding Entries
  - DgsAnchorEntry *FindEntry(AnchorID anchor);
  - DgsAnchorEntry *FindEntry(int value_length, char *value);

Since this API does not differ significantly from `DgsLinkTable`, we will not discuss `DgsAnchorTable` further.

Each entry in a `DgsAnchorTable` is an instance of the class `DgsAnchorEntry` which defines two functions for accessing the contents of the entry:

- AnchorID AnchorID();
- void Value(int &length, char *&value);

4.4.4 Examples of How to Use Anchors

We conclude this section with an example that illustrates all the anchor-related functions and classes. In the example, we assume that the node has pre-existing anchors.

First, we open the node and get its content.

```
node->Open( DGS_READ );

DgsLinkTable *link_table;
DgsAnchorTable *anchor_table;
// we assume that the type of the anchor table is 26.
CSDgsContent content = node->GetContent( link_table, anchor_table, 26);

if ( content.IsEmpty() ) cerr << "No Content!\n";
```

Next, we will read all of the anchors and print the OID's of the links that are associated with them. After running this section of code, the screen should look something like this:

```
AnchorID(24) is associated with 1 links:
        Link(32.25.0)
AnchorID(11) is associated with 3 links:
        Link(32.114.0)
        Link(32.54.0)
        Link(32.213.1)
```

```
for ( DgsAnchorEntry *aEntry = anchor_table->FirstEntry();
      aEntry != nil;
      /* this space is intentionally blank */  ) {

   AnchorID aID = aEntry->AnchorID();
   DgsLinkTable *links = link_table->Select( aID );

   cout << "AnchorID (" << aID << ") is associated with ";
   cout << links->NumOfEntries() << " links:\n"
   for ( DgsLinkEntry *lEntry = links->FirstEntry();
         lEntry != nil;
         /* this space is intentionally blank */ )  {

      UUID linkID = ( lEntry->Link() )->GetOid();
      cout << "   Link(" << linkID << ")\n";
      lEntry = links->NextEntry( lEntry );
   }
   aEntry = anchor_table->NextEntry( aEntry );
}
```

Now, we will break the association between an anchored link and its corresponding anchor. This is a multi-step process. First, we unanchor the link. Then, we remove the (link, anchor) entry from the node's link table. Finally, we check to see if the anchor is associated with links other than the one that we unanchored. If it is, then we leave the anchor in the anchor table, otherwise we remove it.

```
// find the anchor and link
DgsAnchorEntry *anchor = anchor_table->FirstEntry();
CSDgsLink *anchored_link = anchor->Link();
AnchorID aID = anchor->AnchorID();

// unanchor the link
node->Unanchor( anchored_link, aID );

// remove the entry from the link table
link_table->RemoveEntry( anchored_link, aID );

// remove the anchor from the anchor table if it is no longer
//  associated with any links.
DgsLinkTable table = link_table->Select( aID );
if ( table == nil ) anchor_table->RemoveEntry( aID );
```

We conclude the example by creating a new anchor to replace the one we just removed. Notice that we create a new anchor table to contain the new anchor. We do this so that we can pass the new anchors as a parameter to the PutContent function.

```
// we assume that the node is a source node.
DgsAnchorTable new_anchors;
AnchorID newID = node->AnchorAtSource( anchored_link );
new_anchors->AddEntry( newID, 0, nil );

// Transfer the new anchor back to the DGS.
node->PutContent( anchor_table, &new_anchors );

// Close the node to save the changes
node->Close();
```

4.5 CSDgsLink (links)

A *link* is a component of a subgraph whose primary purpose is to represent a semantic or structural relationship between two nodes. Like nodes, links can store information in common attributes, graph attributes, and content. The APIs for common attributes and graph attributes are defined in other classes (see Sections 4.2.6 and 4.6.2). In addition, the `CSDgsComp` class provides some but not all of the content functions. Consequently, the primary responsibility of the `CSDgsLink` class is to provide the remainder of the interface for content-related functions. Note: the API functions for finding the source node or target node of a link are defined in the `CSDgsGraph` class (see Section 4.6.1).

CSDgsContent CSDgsLink::**GetContent()**;

COMMENT:

`GetContent` retrieves the content of a link and will fail (return empty content) if the link is not open. `CSDgsLink::GetContent()` behaves like `CSDgsNode::GetContent()` except that it is parameterless.

`GetContent` returns an instance of the class `CSDgsContent`. This class responds to the `FormOfContent`, `TypeOfContent`, `IsEmpty`, and `NotEmpty` functions that are described in Section 4.3.2. Moreover, `CSDgsContent` will convert itself automatically to other classes. For example, `CSDgsContent` defines a conversion function that allows file content to convert itself to a `char*` that points to the name of the file. Similarly, subgraph content can convert itself to an appropriate object pointer such as `CSDgsTree*`. Refer to Section 4.4.1 for an example.

4.6 CSDgsGraph (subgraphs)

A *subgraph* is a collection of nodes and links such that if a link is a member of the subgraph, then the source and target node of the link are also members of the subgraph. Functions that are common to all subgraphs are implemented in the class `CSDgsGraph`. This includes functions to support the traversal of subgraphs as well as functions to access and manipulate the graph attributes of nodes and links.

4.6.1 Subgraph Traversal

Subgraph traversal is the process of finding nodes and following links within a subgraph. The simplest way to traverse a subgraph is to enumerate the components of the subgraph in random order. `CSDgsGraph` supports random enumeration through the `FirstNode/NextNode` and `FirstLink/NextLink` pairs of functions. `CSDgsGraph` also supports more sophisticated forms of traversal such as following links from node to node. For example, applications can follow a link by asking for the source node or the target node of the link. They can also ask for the unordered set of all of the links that are incident to a particular node.

Next, we describe the subgraph traversal functions in detail. This is the base upon which all subgraph classes are built.

CSDgsNode* CSDgsGraph::**Container**();

COMMENT:

Since every subgraph is the content of one and only one node, there is a unique node for every subgraph that is called its container. This function returns a pointer to the container of the subgraph. It will fail (return `nil`) if the subgraph has not been opened.

USAGE:

```
node->Open( DGS_WRITE );

// create a subgraph as the content of the node
CSDgsTree *tree = node->CreateTreeContent();

// open the tree and get its container
tree->Open( DGS_READ );
CSDgsNode *container = tree->Container();
tree->Close();  // close the tree when finished

if ( *node == *container )
   cout << "Correct!  The node is the container of the tree.";
node->Close();
```

dgs_bool CSDgsGraph::**Contains**(CSDgsComp* **component**);
int CSDgsGraph::**NumOfNodes**();
int CSDgsGraph::**NumOfLinks**();

ARGUMENTS:

`component`: a pointer to a node or link

COMMENT:

`Contains` returns `dgs_true` if the `component` is a member of the subgraph. `NumOfNodes` and `NumOfLinks` return the number of nodes and the number of links that are contained in the subgraph, respectively. These functions will fail if the object has not been opened first.

CSDgsNode* CSDgsGraph::**FirstNode**();
CSDgsNode* CSDgsGraph::**NextNode**(CSDgsNode* **last**);
CSDgsLink* CSDgsGraph::**FirstLink**();
CSDgsLink* CSDgsGraph::**NextLink**(CSDgsLink* **last**);

ARGUMENTS:

`last`: the pointer that was returned by the previous invocation of the function

COMMENT:

`FirstNode` and `NextNode` can be used to enumerate the nodes of the subgraph one-by-one. Similarly, `FirstLink` and `NextLink` can be used to read the links of the subgraph. These functions will fail (return `nil`) if the subgraph has not been opened.

USAGE:

```
tree->Open( DGS_READ );
// read the nodes one-by-one
for ( CSDgsNode *node = tree->FirstNode();
      node != nil;
      /* this space is intentionally blank */ )  {

   // do something with node

   node = tree->NextNode( node );
}
// read the links one-by-one
for ( CSDgsLink *link = tree->FirstLink();
      link != nil;
      /* this space is intentionally blank */ )  {

   // do something with link

   link = tree->NextLink( link );
}
tree->Close();
```

CSDgsNode* CSDgsGraph::**SourceNode**(CSDgsLink* **link**);
CSDgsNode* CSDgsGraph::**TargetNode**(CSDgsLink* **link**);

ARGUMENTS:

`link`: a pointer to a link that is a member of the subgraph

COMMENT:

`SourceNode` and `TargetNode` return a pointer to the source node and target node of the link, respectively. These functions fail (return `nil`) if the subgraph has not been opened and if the link is not a member of the subgraph.

CSDgsLink* CSDgsGraph::**FirstInLink**(CSDgsNode* **node**);
CSDgsLink* CSDgsGraph::**NextInLink**(CSDgsNode* **node**, CSDgsLink* **last_link**);
int CSDgsGraph::**NumOfInLinks**(CSDgsNode* **node**);

ARGUMENTS:

`node`:	a pointer to a node that belongs to the subgraph
`last_link`:	the link that was returned by the previous invocation of the function

COMMENT:

`NumOfInLinks` returns the number of links that have the node as their target node and which also belong to the subgraph. `FirstInLink` and `NextInLink` can be used to enumerate these links using the same technique as `FirstLink/NextLink`.

CSDgsLink* CSDgsGraph::**FirstOutLink**(CSDgsNode* **node**);
CSDgsLink* CSDgsGraph::**NextOutLink**(CSDgsNode* **node**, CSDgsLink* **last_link**);
int CSDgsGraph::**NumOfOutLinks**(CSDgsNode* **node**);

ARGUMENTS:

`node`:	a pointer to a node that belongs to the subgraph
`last_link`:	the link that was returned by the previous invocation of the function

COMMENT:

`NumOfOutLinks` returns the number of links that have the node as their source node and which also belong to the subgraph. `FirstOutLink` and `NextOutLink` can be used to enumerate these links using the same technique as `FirstLink/NextLink`.

4.6.2 Graph Attributes

An attribute is a typed, named variable that applications can define at run-time. Some attributes are called *common attributes* because their values are the same in all contexts. Other attributes are called *graph attributes* because they are dependent on the context of a particular subgraph. The reader is referred to Sections 2.1.5 and 4.2.6 for general information about attributes. We will confine the present discussion to aspects that are unique to graph attributes.

The term *graph attribute* is somewhat confusing because it implies that the attribute belongs to a subgraph. However, this is only partially true. It is better to think of a graph attribute as an attribute of a node or link that is accessed through the API of a subgraph that contains it. In some sense, the attribute really belongs to a node or link. The subgraph simply provides access to the attribute. Nevertheless, the attribute is bound to the subgraph in a significant way. Since a graph attribute is accessed through the API of a subgraph, the subgraph must be opened. Thus, the permission to access the attribute is associated with the subgraph rather than with the node or link.

The following functions are used to manipulate the graph attributes of nodes and links:

int CSDgsGraph::**CreateGAttr**(DgsString& **name**, CSDgsComp* **component**);
int CSDgsGraph::**CreSetGAttr**(DgsString& **name**, DgsAttr& **value**,
CSDgsComp* **component**);

ARGUMENTS:

`name:`	the text name of the attribute
`value:`	the value of the new attribute
`component:`	a pointer to the node or link that owns the attribute

COMMENT:

Use these functions to create a *new* graph attribute for a node or link. These functions fail (return non-zero) if an attribute already exists with the same name and if the subgraph is not open in DGS_WRITE mode. `CreSetGAttr` creates an attribute and sets its value in a single operation. `CreateGAttr` creates the attribute and gives it a null value.

int CSDgsGraph::**ChangeGAttr**(DgsString& **name**, DgsAttr& **value**,
CSDgsComp* **component**);
int CSDgsGraph::**SetGAttr**(DgsString& **name**, DgsAttr& **value**,
CSDgsComp* **component**);

ARGUMENTS:

`name:`	the text name of the attribute
`value:`	the new value of the attribute
`component:`	a pointer to the node or link that owns the attribute

COMMENT:

Use `ChangeGAttr` to set the value of a pre-existing graph attribute. It will fail (return non-zero) if the attribute does not exist. Use `SetGAttr` when you are not sure whether the attribute exists or not. If the attribute does not exist, `SetGAttr` will automatically create it. Both functions fail if the subgraph has not been previously opened in DGS_WRITE mode.

DgsAttr& CSDgsGraph::**GetGAttr**(DgsString& **name**, CSDgsComp* **component**);
dgs_bool CSDgsGraph::**HasGAttr**(DgsString& **name**, CSDgsComp* **component**);
int CSDgsGraph::**DeleteGAttr**(DgsString& **name**, CSDgsComp* **component**);

ARGUMENTS:

`name`:	the text name of the attribute
`component`:	a pointer to the node or link that owns the attribute

COMMENT:

`GetGAttr` returns the value of a graph attribute and a null value if the attribute does not exist. `HasGAttr` returns `dgs_true` if the attribute exists and `dgs_false` if it does not exist. `DeleteGAttr` deletes a graph attribute. `GetGAttr` and `HasGAttr` fail if the subgraph has not been opened. `DeleteGAttr` fails if the subgraph has not been opened in DGS_WRITE mode.

The next functions are useful when the names of the attributes are not known.

DgsString CSDgsGraph::**FirstGAttrName**(CSDgsComp* **component**);
DgsString CSDgsGraph::**NextGAttrName**(DgsString& **last_name**, CSDgsComp* **component**);
int CSDgsGraph::**NumOfGAttrs**(CSDgsComp* **component**);

ARGUMENTS:

`component`:	a pointer to a node or link that is a member of the subgraph
`last_name`:	the name that was returned by the previous invocation of `NextGAttrName`

COMMENT:

`NumOfGAttrs` returns the number of graph attributes that the component has in the context of this subgraph. `FirstGAttrName` and `NextGAttrName` can be used to read the names of the graph attributes of a component one-by-one. These names could then be used as input to other attribute functions such as `GetGAttr`. `NextGAttrName` returns `dgs_nullstr` when there are no more attribute names to read. Before calling any of these functions, the subgraph must have been successfully opened.

USAGE:

```
subgraph->Open( DGS_READ );
CSDgsNode *node = subgraph->FirstNode();

cout << "Number of Graph Attributes: ";
cout << subgraph->NumOfGAttrs( node ) << endl;

// Print the values of the node's string attributes.
for ( DgsString name = subgraph->FirstGAttrName( node );
      name != dgs_nullstr;
      /* this space intentionally blank */          ) {
    DgsAttr *attr = subgraph->GetGAttr( name, node );
    switch ( attr->TypeOfAttr() ) {
       case DGS_STRING:
          cout << "String(" << *( (CSDgsStringAttr *) attr ) << ")\n";
          break;
       default:
          break;
    }
    name = subgraph->NextGAttrName( name, node );
}
subgraph->Close();
```

int CSDgsGraph::**CopyGAttrsFrom**(CSDgsComp* **target_object**,
CSDgsComp* **source_object**, CSDgsGraph* **source_graph**);

ARGUMENTS:

`target_object`: a pointer to a node or link that belongs to this subgraph
`source_object`: a pointer to a node or link
`source_graph`: a pointer to a subgraph that contains `source_object`

COMMENT:

This function copies all of the graph attributes of `source_object` in the context of `source_graph` to `target_object` in the context of the subgraph that is invoking the function (the receiver). The function will fail (return non-zero) if the receiver is not open in DGS_WRITE mode and if the user does not have read authorization for `source_graph`. If the two objects have graph attributes with the same name, then the values of `target_object`'s attributes will be overwritten.

USAGE:

```
target_subgraph->Open( DGS_WRITE );
source_subgraph->Open( DGS_READ );

CSDgsLink *target_link = target_subgraph->FirstLink();
CSDgsLink *source_link = source_subgraph->FirstLink();

// Copy the graph attributes to target_link
target_subgraph->CopyGAttrsFrom( target_link, source_link, source_graph );

target_subgraph->Close();
source_subgraph->Close();
```

4.7 CSDgsStGraph (S-subgraphs)

A *structural subgraph* (S-subgraph) is a subset of the nodes and links that participate in the essential organization of an artifact. The canonical example of structure is the hierarchical organization of a document. One consequence of this type of organization is that the order of information is often important. It is not enough to know that two sections are in the same chapter; we also need to know which section comes first. This simply means that when a hierarchical document is stored in a tree subgraph, the tree must store the order of the out-going links for each node. This is not surprising, but it leads to an interesting generalization—that the order of the in-coming and out-going links associated with each node is important in all types of S-subgraphs, not just in trees.

For example, suppose that we represent a computer-communication network as a subgraph (an instance of the class `CSDgsNetwork`). Links could model the communication channels between computers. Channels could have different costs associated with them and some channels might be redundant. Obviously, low-cost channels would be preferred over higher-cost alternatives. One way to embed this information in the structure of the subgraph would be to order the out-going links of each node according to the order in which the channels should be tried. Thus, low-cost channels would be tried first and high-cost channels would not be used unless lower-cost channels were unavailable. This is somewhat simplistic but it shows the general usefulness of being able to change the order of the links in S-subgraphs.

Functions that are common to all S-subgraphs are implemented in the class `CSDgsStGraph`. As suggested by the previous discussion, these functions provide programmers with the ability to change the order of the links that are incident to each node. This ability is inherited by all of the S-subgraph classes: `CSDgsTree`, `CSDgsList`, and `CSDgsNetwork`. It is notably absent from the API of hyper-structural subgraphs.

CSDgsLink* CSDgsGraph::**FirstInLink**(CSDgsNode* **node**);
CSDgsStLink* CSDgsStGraph::**LastInLink**(CSDgsNode* **node**);
CSDgsStLink* CSDgsStGraph::**InLinkBefore**(CSDgsNode* **node**, CSDgsStLink* **link**);
CSDgsStLink* CSDgsStGraph::**InLinkAfter**(CSDgsNode* **node**, CSDgsStLink* **link**);

ARGUMENTS:

`node`: a pointer to a node that is a member of the S-subgraph
`link`: a pointer to a link that has the node as its target node

COMMENT:

These functions can be used to enumerate the in-coming links (inlinks) of a node in the context of a particular S-subgraph. Only those links that belong to this S-subgraph are returned. That is, the node can have inlinks in other S-subgraphs, and these are invisible unless the other subgraphs are explicitly opened and queried. `FirstInLink` and `LastInLink` return a pointer to the first and last inlink of the node, respectively. `InLinkBefore` returns the inlink that immediately precedes the link parameter in the list of the node's inlinks. `InLinkAfter` performs a similar function but returns the link that succeeds the link parameter. All of the functions return `nil` when an error condition exists and when there is no inlink that satisfies the request.

USAGE:

```
subgraph->Open( DGS_READ );
CSDgsNode *node = subgraph->FirstNode();

// enumerate all of the node's inlinks
for ( CSDgsStLink *link = subgraph->FirstInLink( node );
      link != nil;
      /* this space is intentionally blank */            ) {

    // do something with the link

    link = subgraph->InLinkAfter( node, link );
}
subgraph->Close();
```

int CSDgsStGraph::**PutInLinkFirst**(CSDgsNode* **node**, CSDgsStLink* **inlink**);
int CSDgsStGraph::**PutInLinkLast**(CSDgsNode* **node**, CSDgsStLink* **inlink**);
int CSDgsStGraph::**PutInLinkBefore**(CSDgsNode* **node**, CSDgsStLink* **inlink**,
CSDgsStLink* **beforelink**);
int CSDgsStGraph::**PutInLinkAfter**(CSDgsNode* **node**, CSDgsStLink* **inlink**,
CSDgsStLink* **afterlink**);

ARGUMENTS:

`node`:	a pointer to a node that is a member of the S-subgraph
`inlink`:	a pointer to a link that has the node as its target node
`beforelink`:	a pointer to a link that has the node as its target node
`afterlink`:	a pointer to a link that has the node as its target node

COMMENT:

These functions can be used to re-order the in-coming links (inlinks) of a node in the context of a particular S-subgraph. The inlinks must already belong to the subgraph and must have the node as their target node. For example, `PutInLinkBefore` will re-order the inlinks of the node so that `inlink` precedes `beforelink` in the node's list of inlinks. The functions return zero to indicate that the operation was successful.

USAGE:

```
subgraph->Open( DGS_WRITE );

CSDgsNode *node = subgraph->FirstNode();
CSDgsStLink *first = subgraph->FirstInLink( node );
CSDgsStLink *second = subgraph->InLinkAfter( node, first );

// exchange the first and second inlinks of the node
subgraph->PutInLinkAfter( first, second );

subgraph->Close();
```

CSDgsLink* CSDgsGraph::**FirstOutLink**(CSDgsNode* **node**);
CSDgsStLink* CSDgsStGraph::**LastOutLink**(CSDgsNode* **node**);
CSDgsStLink* CSDgsStGraph::**OutLinkBefore**(CSDgsNode* **node**, CSDgsStLink* **link**);
CSDgsStLink* CSDgsStGraph::**OutLinkAfter**(CSDgsNode* **node**, CSDgsStLink* **link**);

ARGUMENTS:

`node`:	a pointer to a node that is a member of the S-subgraph
`link`:	a pointer to a link that has the node as its source node

COMMENT:

These functions can be used to enumerate the out-going links (outlinks) of a node in the context of a particular S-subgraph. Only those links that belong to this S-subgraph are returned. That is, the node can have outlinks in other S-subgraphs, and these are invisible unless the other subgraphs are explicitly opened and queried. `FirstOutLink` and `LastOutLink` return a pointer to the first and last outlink of the node, respectively. `OutLinkBefore` returns the outlink that immediately precedes the link parameter in the list of the node's outlinks. `OutLinkAfter` performs a similar function but returns the link that succeeds the link parameter. All of the functions return `nil` when an error condition exists and when there is no outlink that satisfies the request. (See `CSDgsStGraph::FirstInLink()` for usage.)

int CSDgsStGraph::**PutOutLinkFirst**(CSDgsNode* **node**, CSDgsStLink* **outlink**);
int CSDgsStGraph::**PutOutLinkLast**(CSDgsNode* **node**, CSDgsStLink* **outlink**);
int CSDgsStGraph::**PutOutLinkBefore**(CSDgsNode* **node**, CSDgsStLink* **outlink**,
CSDgsStLink* **beforelink**);
int CSDgsStGraph::**PutOutLinkAfter**(CSDgsNode* **node**, CSDgsStLink* **outlink**,
CSDgsStLink* **afterlink**);

ARGUMENTS:

`node:`	a pointer to a node that is a member of the S-subgraph
`outlink:`	a pointer to a link that has the node as its source node
`beforelink:`	a pointer to a link that has the node as its source node
`afterlink:`	a pointer to a link that has the node as its source node

COMMENT:

These functions can be used to re-order the out-going links (outlinks) of a node in the context of a particular S-subgraph. The outlinks must already belong to the subgraph and must have the node as their source node. For example, `PutOutLinkBefore` will re-order the outlinks of the node so that `outlink` precedes `beforelink` in the node's list of outlinks. The functions return zero to indicate that the operation was successful. (See `CSDgsStGraph::PutInLinkFirst()` for usage.)

4.8 CSDgsNetwork (unconstrained S-subgraph)

A *network* is the typed S-subgraph with the fewest constraints. The API for network subgraphs is provided by the class `CSDgsNetwork`. This class extends its parent class `CSDgsStGraph` by providing functions to create, add, and remove nodes and links. `CSDgsNetwork` allows cycles and other structural features that are prohibited by its sibling classes `CSDgsTree` and `CSDgsList`. In fact, the only restriction that is placed on network subgraphs is that they conform to the general definition of a subgraph. That is, a network that contains a link must also contain the source node and target node of the link. Aside from this, networks are unconstrained.

CSDgsNode* CSDgsNetwork::**CreateNode();**
CSDgsStLink* CSDgsNetwork::**CreateLink**(CSDgsNode* **source**, CSDgsNode* **target**);

ARGUMENTS:

`source`: a pointer to a node that is a member of the network
`target`: a pointer to a node that is a member of the network

COMMENT:

These functions can be used to create new nodes and links in a network. `CreateLink` will fail (return `nil`) if the network does not contain both the source node and the target node. Both functions return `nil` if the network is not open in DGS_WRITE mode.

USAGE:

```
network->Open( DGS_WRITE );

// create two new nodes in the network.
CSDgsNode *node1 = network->CreateNode();
CSDgsNode *node2 = network->CreateNode();

// create a new link between the nodes
CSDgsStLink *link = network->CreateLink( node1, node2 );
```

```
// networks are unconstrained.  we can even create a link from a
//   node to itself.
CSDgsStLink *link2 = network->CreateLink( node2, node2 );

network->Close();
```

CSDgsStLink* CSDgsNetwork::**ChangeTargetOfLink**(CSDgsStLink* **old_link**,
CSDgsNode* **new_target**);
CSDgsStLink* CSDgsNetwork::**ChangeSourceOfLink**(CSDgsStLink* **old_link**,
CSDgsNode* **new_source**);

ARGUMENTS:

`old_link`:	a pointer to a link that is a member of the network
`new_target`:	a pointer to a node that is a member of the network
`new_source`:	a pointer to a node that is a member of the network

COMMENT:

These functions have an effect that is similar to changing the source node or the target node of a link. What they actually do is to create a new link, copy all of `old_link`'s attributes to the new link, and then remove `old_link` from the subgraph. In most cases, the new link will appear to be the old link. However, applications should be aware that the new link will have a different OID from the old link. This could lead to subtle bugs in advanced applications.

The functions fail (return `nil`) if the network has not been opened in DGS_WRITE mode.

USAGE:

```
network->Open( DGS_WRITE );

// Find a node and a link
CSDgsNode *node = network->FirstNode();
CSDgsStLink *old_link = (CSDgsStLink *) network->FirstLink();

// Change the source node of the link
```

```
CSDgsStLink *new_link = ChangeSourceOfLink( old_link, node );

// the new link should not be equal to the old link even though
//     they are carbon copies of each other.
if ( *old_link == *new_link )  cerr << "error!\n"

network->Close();
```

int CSDgsNetwork::**Add**(CSDgsNode* **component**);
int CSDgsNetwork::**Add**(CSDgsStLink* **component**);
int CSDgsNetwork::**Remove**(CSDgsNode* **component**);
int CSDgsNetwork::**Remove**(CSDgsStLink* **component**);

ARGUMENTS:

`component`: a pointer to a node or S-link

COMMENT:

These functions add and remove a node or S-link from a network. If the `component` is removed from the last S-subgraph that contains it, then it will cease to exist. The functions fail (return non-zero) when the network is not open in DGS_WRITE mode.

USAGE:

```
subgraph->Open( DGS_READ );
network->Open( DGS_WRITE );

// copy all of the nodes from an arbitrary subgraph to a network
for ( CSDgsNode *node = subgraph->FirstNode();
      node != nil;
      /* this space is intentionally blank */   )  {

   network->Add( node );
   node = subgraph->NextNode( node );
}
```

4.9 CSDgsTree (tree S-subgraph)

The class `CSDgsTree` is a typed S-subgraph that constrains its own structure to be in the form of a tree.

CSDgsNode* CSDgsTree::**CreateNodeAsRoot**();
int CSDgsTree::**AddNodeAsRoot**(CSDgsNode* **node**);

ARGUMENTS:

`node`: a pointer to a node that is not a member of the tree

COMMENT:

Create a new node or add a pre-existing node to the tree and make it the tree's root. Both functions fail if the tree already has a root and if the tree has not been opened in DGS_WRITE mode.

CSDgsNode* CSDgsTree::**Root**();
CSDgsNode* CSDgsTree::**Parent**(CSDgsNode* **node**);
CSDgsStLink* CSDgsTree::**ParentLink**(CSDgsNode* **node**);

ARGUMENTS:

`node`: a pointer to a node that is a member of the tree

COMMENT:

`Root` returns a pointer to the root node of the tree and `nil` if the tree is empty. `ParentLink` returns the same value as `CSDgsGraph::FirstInLink()`. `Parent` returns the immediate ancestor of node. The functions fail (return `nil`) if the tree is not open.

USAGE:

```
CSDgsNode *root = tree->Root();
CSDgsNode *child = tree->FirstChild( root );
CSDgsNode *parent = tree->Parent( child );

if ( *root != *parent )  cerr << "error!\n";
```

int   CSDgsTree::**NumOfChildren**(CSDgsNode* **parent**);
int   CSDgsTree::**NumOfSiblings**(CSDgsNode* **node**);
dgs_bool CSDgsTree::**IsLeaf**(CSDgsNode* **node**);
dgs_bool CSDgsTree::**NotLeaf**(CSDgsNode* **node**);

ARGUMENTS:

`parent`:	a pointer to a node that is a member of the tree
`node`:	a pointer to a node that is a member of the tree

COMMENT:

`NumOfChildren` returns the number of children that a particular node has. `NumOfSiblings` returns the number of other nodes that share the same parent as the node specified. `IsLeaf` and `NotLeaf` return `dgs_true` and `dgs_false` according to whether a particular node is a leaf node or not. All of these functions will fail if the tree has not been opened and if the specified node does not belong to the tree.

USAGE:

```
tree->Open( DGS_READ );
CSDgsNode *root = tree->Root();

// print the number of children that the root node has
int num_childs = tree->NumOfChildren;
cout << "The root has " << num_childs << " children.\";

if ( tree->NumOfSiblings( root ) != 0 )
   cerr << "error!  the root cannot have siblings!\n";

if ( num_childs > 0 && tree->IsLeaf( root ) == dgs_true )
   cerr << "error!  the root cannot be a leaf if it has children!\n"

tree->Close();
```

CSDgsNode* CSDgsTree::**FirstChild**(CSDgsNode* **parent**);
CSDgsNode* CSDgsTree::**LastChild**(CSDgsNode* **parent**);
CSDgsNode* CSDgsTree::**SibBefore**(CSDgsNode* **node**);
CSDgsNode* CSDgsTree::**SibAfter**(CSDgsNode* **node**);

ARGUMENTS:

`parent:`	a pointer to a node that is a member of the tree
`node:`	a pointer to a node that is a member of the tree

COMMENT:

These functions can be used to enumerate the children of a particular node. `FirstChild` and `LastChild` return the first and last child of a node in the context of this tree. `SibBefore` returns the sibling node that precedes the indicated node. `SibAfter` returns the sibling node that immediately succeeds the indicated node. These functions return `nil` if there is no node that satisfies the request, if the tree has not been opened, and if the node parameter does not belong to the tree.

USAGE:

```
tree->Open( DGS_READ );
CSDgsNode *root = tree->Root();

// enumerate the children of the root node
for ( CSDgsNode *node = tree->FirstChild( root );
      node != nil;
      /* this space is intentionally blank */     )  {

    // do something with node

    node = tree->SibAfter( node );
}
tree->Close();
```

```
CSDgsStLink* CSDgsTree::InsertFirstChild( CSDgsNode* new_child,
                                          CSDgsNode* parent      );
CSDgsStLink* CSDgsTree::InsertLastChild(  CSDgsNode* new_child,
                                          CSDgsNode* parent      );
```

ARGUMENTS:

`parent`:	a pointer to a node that is a member of the tree
`new_child`:	a pointer to a node that is not a member of the tree

COMMENT:

`InsertFirstChild` adds `new_child` to the tree, creates a link from `parent` to `new_child`, and re-orders `parent`'s out-going links so that `new_child` is first. `InsertLastChild` does exactly the same thing except that the node becomes the last child of `parent`. Both functions return `nil` if they fail and return a pointer to the new link if they succeed. They will fail if the `new_child` already belongs to the tree, if `parent` does not belong to the tree, and if the tree has not been opened in DGS_WRITE mode.

USAGE:

```
subgraph->Open( DGS_READ );
tree->Open( DGS_WRITE );

CSDgsNode *node = subgraph->FirstNode();
CSDgsNode *root = tree->Root();

// add the node to the tree as the first child of the root
tree->InsertFirstChild( node, root );

subgraph->Close();
tree->Close();
```

CSDgsStLink* CSDgsTree::**InsertSibBefore**(CSDgsNode* **new_node**,
CSDgsNode* **old_node**);
CSDgsStLink* CSDgsTree::**InsertSibAfter**(CSDgsNode* **new_node**,
CSDgsNode* **old_node**);

ARGUMENTS:

`old_node`:	a pointer to a node that is a member of the tree
`new_node`:	a pointer to a node that is not a member of the tree

COMMENT:

The syllable "Sib" in the name of these functions means "sibling". Thus, "InsertSibBefore" means "insert `new_node` as the sibling node that immediately precedes `old_node`." `InsertSibBefore` does this by adding `new_node` to the tree, creating a link from the parent of `old_node` to `new_node`, and then re-ordering the parent's out-going links so that `new_node` precedes `old_node`. `InsertSibAfter` performs a similar function. Both functions return a pointer to the link that is created when they succeed and a `nil` pointer when they fail. For them to succeed, the tree must be open in DGS_WRITE mode. Also, the tree must contain `old_node` but not `new_node`.

USAGE:

```
subgraph->Open( DGS_READ );
tree->Open( DGS_WRITE );

CSDgsNode *node = subgraph->FirstNode();
CSDgsNode *root = tree->Root();
CSDgsNode *first = tree->FirstChild( root );

// add the node to the tree as the first child of the root
tree->InsertSibBefore( node, first );

subgraph->Close();
tree->Close();
```

CSDgsNode* CSDgsTree::**CreateFirstChild**(CSDgsNode* **parent**);
CSDgsNode* CSDgsTree::**CreateLastChild**(CSDgsNode* **parent**);

ARGUMENTS:

`parent`: a pointer to a node that is a member of the tree

COMMENT:

`CreateFirstChild` creates a new node in the tree, creates a link from `parent` to the new node, and re-orders `parent`'s out-going links so that the new node is `parent`'s first child. `CreateLastChild` does exactly the same thing except that the new node becomes the last child of `parent`. Both functions return `nil` if they fail and return a pointer to the new node if they succeed. They will fail if `parent` does not belong to the tree, and if the tree has not been opened in DGS_WRITE mode. See `InsertFirstChild` for usage.

CSDgsNode* CSDgsTree::**CreateSibBefore**(CSDgsNode* **old_node**);
CSDgsNode* CSDgsTree::**CreateSibAfter**(CSDgsNode* **old_node**);

ARGUMENTS:

`old_node`: a pointer to a node that is a member of the tree

COMMENT:

The syllable "Sib" in the name of these functions means "sibling". Thus, "CreateSibBefore" means "create a new node and insert it as the sibling node that immediately precedes `old_node`." `CreateSibBefore` does this by creating a new node in the tree, creating a link from the parent of `old_node` to the new node, and then re-ordering the parent's out-going links so that the new node precedes `old_node`. `CreateSibAfter` performs a similar function. Both functions return a pointer to the node that is created when they succeed and a `nil` pointer when they fail. For them to succeed, the tree must be open in DGS_WRITE mode. Also, the tree must contain `old_node`. See `InsertSibBefore` for usage.

CSDgsStLink* CSDgsTree::**ChangeSourceOfLink**(CSDgsStLink* **old_link**,
CSDgsNode* **new_source**);

ARGUMENTS:

`old_link`:	a pointer to a link that is a member of the tree
`new_source`:	a pointer to a node that is a member of the tree

COMMENT:

This function can be used to move a subtree from one part of the tree to another. To move a subtree, one simply changes the source node of the link that attaches the subtree to the tree. The function performs automatic checking to ensure that a cycle is not produced in the process.

Unfortunately, the function does not behave exactly as one might think. What it actually does is to create a new link, copy all of `old_link`'s attributes to the new link, and then remove `old_link` from the tree. In most cases, the new link will appear to be the old link. However, applications should be aware that the new link will have a different OID from the old link. This could lead to subtle bugs in advanced applications.

`ChangeSourceOfLink` returns a pointer to the new link when it succeeds and `nil` when it fails. The tree must be open in DGS_WRITE mode.

USAGE:

```
tree->Open( DGS_WRITE );
CSDgsNode *root = tree->Root();

CSDgsNode *child1 = tree->FirstChild();
CSDgsNode *child2 = tree->SibAfter( child1 );
CSDgsLink *parentlink = tree->ParentLink( child2 );

// move child2's subtree so that it is under child1
tree->ChangeSourceOfLink( parentlink, child1 );

// now, child1 is the parent of child2
```

```
if ( *child1 != tree->Parent( child2 ) )
   cerr << "error!\n";

tree->Close();
```

int CSDgsTree::**MoveSubtreeToRoot**(CSDgsNode* **root_of_subtree**);

ARGUMENTS:

`root_of_subtree`: a pointer to a node that is a member of the tree

COMMENT:

This function moves the subtree that is rooted at `root_of_subtree` to the top of the tree. `root_of_subtree` becomes the new root of the whole tree. The old root becomes the last child of the new root. The function fails (returns non-zero) if the tree is not open in DGS_WRITE mode and if the tree does not contain `root_of_subtree`.

USAGE:

```
tree->Open( DGS_WRITE );
CSDgsNode *old_root = tree->Root();
CSDgsNode *node = tree->FirstChild( old_root );

// promote 'node' to root.
tree->MoveSubtreeToRoot( node );

// at this point 'node' should be the new root and 'old_root'
//    should be its last child.
if ( *node != tree->Root() || *old_root != tree->FirstChild( node ) )
   cerr << "error!\n";

tree->Close();
```

int CSDgsTree::**Height();**
int CSDgsTree::**HeightOfSubtree(** CSDgsNode* **root_of_subtree);**

ARGUMENTS:

`root_of_subtree`: a pointer to a node that is a member of the tree

COMMENT:

These functions compute the height of the whole tree or of a subtree. The functions will fail (return a negative number) if the tree is not open.

Height is computed according to the definition in Aho, Hopcroft, and Ullman. Thus, a tree with one node has height 0 and a tree with two nodes has height 1. Height is undefined on an empty tree (-1 will be returned).

USAGE:

```
tree->Open( DGS_READ );

// print the height of the tree
cout << "Height = " << tree->Height() << ".\n";

tree->Close();
```

int CSDgsTree::**RemoveSubtree**(CSDgsNode* **root_of_subtree**);
int CSDgsTree::**RemoveNode**(CSDgsNode* **node**);

ARGUMENTS:

`root_of_subtree`:	a pointer to a node that is a member of the tree
`node`:	a pointer to a node that is a member of the tree

COMMENT:

`RemoveSubtree` removes all of the nodes and links in the subtree rooted at the node `root_of_subtree`. `RemoveNode` removes a single node and its parental link from the tree. Any children (and their parental links) are promoted to the level of the node being removed. Note: the root node cannot be removed using `RemoveNode` when it has more than one child because there is no way to promote its children. Both functions will fail (return non-zero) if the tree does not contain the node parameter and if the tree is not open in DGS_WRITE mode.

USAGE:

```
tree->Open( DGS_WRITE );
CSDgsNode *root = tree->Root();
CSDgsNode *child1 = tree->FirstChild( root );
CSDgsNode *child2 = tree->SibAfter( child1 );
CSDgsNode *child1.1 = tree->FirstChild( child1 );
CSDgsNode *child1.n = tree->LastChild( child1 );

// when child1 is removed, its children will be promoted
//    i.e., they will become children of the root
tree->RemoveNode( child1 );
if ( *child1.1 != tree->FirstChild( root ) ||
     *child1.n != tree->SibBefore( child2 )    )
   cerr << "error!\n";

// remove all of the nodes from the tree
tree->RemoveSubtree( root );
if ( tree->NumOfNodes() != 0 ) cerr << "error!\n";
tree->Close();
```

4.10 CSDgsList (list S-subgraph)

The class `CSDgsList` is a typed S-subgraph that constrains its own structure to be in the form of a list.

CSDgsNode* CSDgsList::**Head**();
CSDgsNode* CSDgsList::**Tail**();
CSDgsNode* CSDgsList::**After**(CSDgsNode* **node**);
CSDgsNode* CSDgsList::**Before**(CSDgsNode* **node**);

ARGUMENTS:

`node`: a pointer to a node that is a member of the list

COMMENT:

`Head` and `Tail` return a pointer to the first and last node in the list, respectively. They return `nil` if the list is empty. `Before` and `After` return a pointer to the node that comes before or after `node` in the list. All four functions fail (return `nil`) if the list has not been opened.

USAGE:

```
list->Open( DGS_READ );

CSDgsNode *first_node_in_list = list->Head();
CSDgsNode *second_node_in_list = list->After( first_node_in_list );
CSDgsNode *last_node_in_list = list->Tail();

list->Close();
```

CSDgsNode* CSDgsList::**CreateNodeAndPrepend();**
CSDgsNode* CSDgsList::**CreateNodeAndAppend();**
int CSDgsList::**Prepend(** CSDgsNode* **node);**
int CSDgsList::**Append(** CSDgsNode* **node);**

ARGUMENTS:

`node`: a pointer to a node

COMMENT:

`CreateNodeAndPrepend` creates a new node and adds it to the beginning of the list. `CreateNodeAndAppend` does the same thing except that it adds the new node to the end of the list. Both functions return a pointer to the new node when they succeed and a `nil` pointer when they fail. `Prepend` and `Append` add pre-existing nodes to the beginning and end of the list, respectively. They return zero when they succeed. All four functions fail if the list has not been opened in DGS_WRITE mode. `Prepend` and `Append` also fail if the `node` is already contained in the list.

USAGE:

```
subgraph->Open( DGS_READ );
list->Open( DGS_WRITE );

// add all of the nodes from subgraph to the list
for ( CSDgsNode *node = subgraph->FirstNode();
      node != nil;
      /* this space is intentionally blank */   )  {

   list->Append( node );
   node = subgraph->NextNode( node );
}

// create a new node and add it to the end of the list
CSDgsNode *new_node = list->CreateNodeAndAppend();

subgraph->Close();
list->Close();
```

```
CSDgsNode* CSDgsList::CreateNodeBefore( CSDgsNode* node );
CSDgsNode* CSDgsList::CreateNodeAfter( CSDgsNode* node );
int        CSDgsList::InsertNodeBefore( CSDgsNode* new_node,
                                        CSDgsNode* old_node   );
int        CSDgsList::InsertNodeAfter( CSDgsNode* new_node,
                                        CSDgsNode* old_node   );
```

ARGUMENTS:

`node`:	a pointer to a node that is a member of the list
`old_node`:	a pointer to a node that is a member of the list
`new_node`:	a pointer to a node that is not a member of the list

COMMENT:

`CreateNodeBefore` creates a new node and inserts it into the list so that it immediately precedes `node`. `CreateNodeAfter` does the same thing except that it inserts the new node after `node`. Both functions return a pointer to the new node if they succeed and a `nil` pointer if they fail. They fail when the list has not been opened in DGS_WRITE mode and when `node` is not contained in the list.

`InsertNodeBefore` inserts `new_node` into the list so that it immediately precedes `old_node`. `InsertNodeAfter` does the same thing except that it inserts `new_node` after `old_node`. Both functions fail (return non-zero) if the list already contains `new_node`, if the list does not contain `old_node`, and if the list has not been opened in DGS_WRITE mode.

USAGE:

```
subgraph->Open( DGS_READ );
list->Open( DGS_WRITE );

CSDgsNode *list_node = list->Tail();

// add all of the nodes from subgraph to the list
for ( CSDgsNode *node = subgraph->FirstNode();
      node != nil;
```

```
        /* this space is intentionally blank */    )  {

      list->InsertNodeAfter( node, list_node );
      list_node = node;
      node = subgraph->NextNode( node );
   }

   // create a new node and add it to the end of the list
   CSDgsNode *new_node = list->CreateNodeAfter( list_node );

   subgraph->Close();
   list->Close();
```

CSDgsStLink* CSDgsList::**MoveEndlistToHead**(CSDgsNode* **head_of_endlist**);
int CSDgsList::**MoveEndlistToMiddle**(CSDgsNode* **head_of_endlist**, CSDgsNode* **middle_node**);

ARGUMENTS:

`head_of_endlist`:	a pointer to a node that is a member of the list
`middle_node`:	a pointer to a node that is a member of the list

COMMENT:

`MoveEndlistToHead` moves the end of the list to the beginning of the list. The end of the list (endlist) is defined as the sublist that is composed by the nodes between `head_of_endlist` and the tail of the list, inclusive. For example, if the nodes were ordered 1-2-3-4-5 before calling `MoveEndlistToHead( 4 )`, they would be ordered 4-5-1-2-3 after calling it.

`MoveEndlistToMiddle` performs a similar function except that the endlist is not restricted to being moved to the head of the list. Instead, the endlist is inserted immediately after `middle_node`. For example, if the nodes were ordered 1-2-3-4-5 before calling `MoveEndlistToMiddle( 4, 2 )`, they would be ordered 1-2-4-5-3 after calling it.

int CSDgsList::**RemoveNode**(CSDgsNode* **node**);
int CSDgsList::**RemoveHead**();
int CSDgsList::**RemoveTail**();
int CSDgsList::**RemoveEndlist**(CSDgsNode* **head_of_endlist**);

ARGUMENTS:

`head_of_endlist`:	a pointer to a node that is a member of the list
`node`:	a pointer to a node that is a member of the list

COMMENT:

`RemoveHead` and `RemoveTail` remove the head node and the tail node of the list, respectively. `RemoveNode` removes `node` from the list. As a consequence, `node`'s in-coming link is also removed. `RemoveEndlist` removes all of the nodes from `head_of_endlist` to the tail of the list. All of these functions fail (return non-zero) if the list is not open in DGS_WRITE mode. `RemoveNode` and `RemoveEndlist` also fail if the list does not contain the specified node.

USAGE:

```
list->Open( DGS_WRITE );
// remove all of the nodes in the list (dumb method)
for ( CSDgsNode *node = list->Head();
      node != nil;
      /* this space is intentionally blank */  )  {

   list->RemoveHead();
   node = list->Head();
}
list->Close();

list2->Open( DGS_WRITE );
// remove all of the nodes in the list (better method)
CSDgsNode *head = list2->Head();
if ( head != nil )
   list2->RemoveEndlist( head );
list2->Close();
```

4.11 CSDgsHyGraph (HS-subgraph)

A hyper-structural subgraph (HS-subgraph) is a subgraph that contains links (HS-links) that supplement—and sometimes oppose—the basic structure of an artifact. The reader is referred to Section 2.1.2 for a general discussion of structure versus hyper-structure. This section describes HS-subgraphs from the point of view of the API defined by the class `CSDgsHyGraph`. In contrast to the typed S-subgraph classes, `CSDgsHyGraph` is extraordinarily simple. The design of the class emphasizes lightweight, flexible, unstructured linking with minimal overhead. In fact, it defines only three functions beyond those that it inherits from `CSDgsGraph`. Another difference is that the `CSDgsHyGraph` API does not permit HS-subgraphs to contain disconnected nodes. However, the most significant difference is that the existence of nodes is independent their membership in HS-subgraphs. When a node is removed from the last S-subgraph that contains it, then the node ceases to exist, regardless of the fact that it might still belong to one or more HS-subgraphs.

Now, we define the functions in the API.

CSDgsHyLink* CSDgsHyGraph::**HyperLink**(CSDgsNode* **source**, CSDgsNode* **target**);

ARGUMENTS:

`source`: a pointer to a node
`target`: a pointer to a node

COMMENT:

This function creates a new HS-link and puts it into the HS-subgraph. `HyperLink` differs from `CSDgsNetwork::CreateLink()` in that it does not require that the source and target already exist in the subgraph. `HyperLink` will add these nodes to the subgraph automatically. In fact, there is no way to create or add an individual node to an HS-subgraph. Nodes are *always* created in S-subgraphs, and then added to HS-subgraphs as side-effects of the `HyperLink` function.

`HyperLink` returns a pointer to the new HS-link when it succeeds and a `nil` pointer when it fails. It will fail if the HS-subgraph is not open in DGS_WRITE mode.

USAGE:

```
subgraph1->Open( DGS_READ );
subgraph2->Open( DGS_READ );

CSDgsNode node1 = subgraph1->FirstNode();
CSDgsNode node2 = subgraph2->FirstNode();

hygraph->Open( DGS_WRITE );

// create an HS-link between the two nodes
hygraph->HyperLink( node1, node2 );

hygraph->Close();
subgraph1->Close();
subgraph2->Close();
```

int CSDgsHyGraph::**Add**(CSDgsHyLink* **link**);
int CSDgsHyGraph::**Remove**(CSDgsNode* **component**);
int CSDgsHyGraph::**Remove**(CSDgsHyLink* **component**);

ARGUMENTS:

`link:`	a pointer to a HS-link
`component:`	a pointer to a node or HS-link that is a member of the HS-subgraph

COMMENT:

`Add` adds a pre-existing HS-link to an HS-subgraph. The source node and target node of the link are also added automatically. `Remove` removes nodes and links from an HS-subgraph. When a link is removed, the DGS automatically removes any nodes that are orphaned in the process. The DGS also removes any dangling links that are created when nodes are removed.

These functions fail (return non-zero) if the HS-subgraph has not been opened in DGS_WRITE mode.

USAGE:

```
hygraph->Open( DGS_WRITE );

// remove all of the nodes an links from the HS-subgraph
//    note: the nodes are removed automatically when the links are
//          removed.
for ( CSDgsLink *link = hygraph->FirstLink();
      link != nil;
      /* this space is intentionally blank */  )  {

   CSDgsHyLink *remove_link = (CSDgsHyLink *) link;
   link = hygraph->NextLink( link );
   hygraph->Remove( remove_link );
}

hygraph->Close();
```

5 Tutorial – Building a Simple Application

This section is a brief tutorial for new application programmers. The tutorial defines and then solves a small but non-trivial problem. The purpose of the tutorial is to illustrate key techniques and to present a DGS application in its entirety. The complete source code is provided in the appendix.

After reading this tutorial you should know how to:

- initialize the connection to the DGS and get the root subgraph
- create a new subgraph
- enumerate all of the nodes in a subgraph
- enumerate all of the attributes of a node
- read the value of an attribute when you don't know its type
- get the content of a node when you don't know its type
- determine the type of object that is pointed to by a CSDgsObjectAttr

Problem Statement:

> To display the attributes of all the nodes in the artifact and to find all of the nodes that have at least one integer attribute that has the value 42.

To solve the problem, the program will have to look at every attribute of every node that is stored by the DGS. The results of the search will be kept in a network subgraph so that they can be stored for future reference. Every time that the program finds a node that satisfies the search criterion, it will place the node in this subgraph.

The `main()` function looks something like this:

```
main() {
CSDgsTree *root_subgraph;  // Root subgraph
CSDgsNode *root_node;      // Root node of the root subgraph
CSDgsNode *new_node;       // A new node that will be created to hold
                           // the results in its subgraph content.

// Initialize data structures and create a subgraph to store the nodes
//     that match the search criteria
Initialize( &root_subgraph, &root_node );
CreateSubgraphForResults( root_subgraph, root_node, new_node );

// Start the recursive search on the root subgraph.
SearchSubgraph( root_subgraph );

// At this point, the global variable 'NodesThatMatch' will point to a
// subgraph that contains all of the nodes that match the search criteria.
...
}
```

First, `main` calls `Initialize` to initialize the connection to the DGS and to open the root subgraph of the artifact. Then, it calls `CreateSubgraphForResults` which creates a new subgraph to store the results of the search. Finally, the program begins a recursive search of the nodes in the artifact by calling `SearchSubgraph` on the root subgraph.

The program defines two global variables:

- CSDgsConnection dgs;
- CSDgsNetwork *NodesThatMatch;

When the program has finished, the subgraph pointed to by `NodesThatMatch` will contain all of the nodes that have an integer attribute equal to 42.

Since `main` begins by calling `::Initialize()`, we will begin by looking at that function in more detail.

```
void Initialize( CSDgsTree *&root_subgraph, CSDgsNode *&root_node );

   dgs.Initialize();
   root_subgraph = dgs.GetRootSubgraph();
   root_subgraph->Open(DGS_WRITE)
   root_node = root_subgraph->Root();
}
```

First, `::Initialize` calls `CSDgsConnection::Initialize()` to establish the connection with the DGS. Next, the function gets a pointer to the root subgraph and opens it. Then, the function asks for a pointer to the root node of the root subgraph. Note: the root subgraph always has a root node. Finally, the function passes both pointers back to `main()`.

`At this point, the program is half-way through the setup process. Now, it needs to create a subgraph to store the results in. This is the purpose of the function `CreateSubgraphForResults`.

```
void CreateSubgraphForResults( CSDgsGraph *root_subgraph,
                               CSDgsNode *root_node, CSDgsNode *&new_node )  {

// Create a new node in the root subgraph and close it.
   new_node = root_subgraph->CreateFirstChild( root_node );
   root_subgraph->Close();

// Open new_node in write mode
   new_node->Open( DGS_WRITE );

// Create a new subgraph to hold the nodes that match the query,
// and open it.
   NodesThatMatch = new_node->CreateNetworkContent( root_subgraph );
   NodesThatMatch->Open( DGS_WRITE );
}
```

`CreateSubgraphForResults` is passed pointers to the root subgraph and to the root node. First, the function creates a new node in the root subgraph. Normally, one has to open a subgraph in DGS_WRITE mode before creating nodes in it. In this case, the root subgraph was already open, so it was not necessary to do that here. It is also a good idea to close objects when they are no longer needed. Since the program is finished with the root subgraph, `CreateSubgraphForResults` closes it. Next, after opening the new node in DGS_WRITE mode, the function creates a new network subgraph as the node's content. The last thing that the function does is to open the new subgraph so that the program will be able to add nodes to it later.

This ends the setup phase of the program. The program has established a connection with the DGS. It has opened the root subgraph and created a new network subgraph to store the results of the search. Now, the program enters the search phase. Starting with the root subgraph, it will systematically read the attributes of every node in the artifact. The search is a recursive process because each node can have a subgraph as its content. Since these subgraphs are also a part of the artifact, the nodes that they contain must also be searched. Consequently, the search function must be called recursively on the content of every node.

The function `SearchSubgraph` is the top-level function of the search mechanism. `main()` begins the search by calling this function on the root subgraph.

```
void SearchSubgraph( CSDgsStGraph *subgraph )  {

   subgraph->Open(DGS_READ);
   for ( CSDgsNode *node = subgraph->FirstNode();
         node != nil;
         /* this space is intentionally blank */   )  {

         SearchNode( subgraph, node );
         node = subgraph->NextNode( node );
   }
   subgraph->Close();
}
```

The function uses `CSDgsGraph::FirstNode` and `CSDgsGraph::NextNode` to enumerate all of the nodes in the subgraph. First, the subgraph is opened. Then, the nodes are enumerated and `SearchNode` is called for each one. The function ends by closing the subgraph.

`SearchNode` is a mid-level function whose purpose is to test the attributes of a single node and to call the search function recursively on the node's content.

```
void SearchNode( CSDgsGraph *graph, CSDgsNode *node    ) {

   DgsStatus status = node->Status();
   //        do not continue if the node is already open.
   if ( status.IsOpen() == dgs_true ) return;

   node->Open( DGS_READ_NO_ANCHOR );
   SearchContentOfNode( node );

   cout << "Searching node.\n";  SearchAttrsOfNode( graph, node );

   node->Close();  cout << "Node closed.\n";
}
```

First, the function tests the status of the node to see if it has already been opened. If it is open, then the function terminates because the node is already being searched by another instance of the function. This prevents infinite recursion. Next, the function opens the node in DGS_READ_NO_ANCHOR mode. This access mode was chosen because this application does not have any knowledge of anchors. After opening the node, the function calls SearchContentOfNode which will get the content of the node and recursively search the nodes that it contains. SearchAttrsOfNode is called to search the attributes of the node, and then the node is closed.

An excerpt from SearchContentOfNode is presented next.

```
void SearchContentOfNode( CSDgsNode *node    ) {

   DgsLinkTable *lt; DgsAnchorTable *at;
   CSDgsContent content = node->GetContent( lt, at);

   switch ( content.FormOfContent() )  {
    case DGS_FILE_CONTENT:
      break;
    case DGS_GRAPH_CONTENT:
      CSDgsGraph *subgraph = content;
      SearchSubtree( subgraph );
      break;
    case DGS_EMPTY_CONTENT:
      break;
    default:
         cerr << "SearchContentOfNode(): unknown form of content\n"; exit(-1);
   }
}
```

The purpose of this function is to get the content of the node and to call the search function on it if the content is a subgraph. Initially, the application knows nothing about the content. It does not know what type of content the node has, nor does it know whether the node has any anchors. Moreover, it does not know the registered type of the anchor table. Thus, it is important that the node was opened in DGS_READ_NO_ANCHOR mode; otherwise, the application would be required to specify the type of the anchor table in the `GetContent` call. After getting the content, `SearchContentOfNode` determines whether or not the content is a subgraph. If it is, then it converts the `DgsContent` to a `CSDgsGraph*` and passes it to the search function.

After the content of the node is searched, `SearchAttrsOfNode` is called.

```
void SearchAttrsOfNode( CSDgsGraph *graph, CSDgsNode *node )  {

   for ( DgsString name = node->FirstAttrName();
         name != dgs_nullstr;                       )  {

      DgsAttr *attr = node->GetAttr( name );
      if ( TestAttr( attr, name ) == -1 )  NodesThatMatch->Add(node);
      name = component->NextAttrName( name );
   }

   for ( DgsString name = graph->FirstGAttrName( node );
         name != dgs_nullstr;                               )  {

      DgsAttr *attr = graph->GetGAttr( name, node );
      if ( TestAttr( attr, name ) == -1 )  NodesThatMatch->Add(node);
      name = graph->NextGAttrName( name, node );
   }
}
```

`SearchAttrsOfNode` is contains two `for` loops: one to search the common attributes of the node and another to search the graph attributes of the node. The construction of the two loops is similar, so we will only discuss the first one. The function uses `FirstAttrName` and `NextAttrName` to enumerate all of the attribute names. Then, for each name, it gets the value of the attribute with that name and calls a function to test whether the value is equal to 42. If it is, then the node is added to the subgraph `NodesThatMatch`.

`TestAttr` is described next.

```
int TestAttr( DgsAttr* attr, DgsString& name )  {

   switch ( attr->TypeOfAttr() )  {
      case DGS_INT:
         int num = (CSDgsIntAttr) *attr;
         cout << "Int(" << num << ")\n";

         // return -1 if the integer is equal to 42
         if ( num == 42 ) return -1;
         break;
      case DGS_STRING:
         cout << "String(" << *( (CSDgsStringAttr *) attr ) << ")\n"; break;
      case DGS_FLOAT:
         cout << "Float(" << *( (CSDgsFloatAttr *) attr ) << ")\n";   break;
      case DGS_DOUBLE:
         cout << "Double(" << *( (CSDgsDoubleAttr *) attr ) << ")\n"; break;
      case DGS_OBJECT:
         CSDgsObject *object = *( CSDgsObjectAttr *) attr );
         int object_type = object->TypeOfObject();
         cout << "Object(" << PrintObjectType( object_type ) << ")\n";
         break;
      case DGS_BYTE_ARRAY:
         CSDgsByteArrayAttr *battr = (CSDgsByteArrayAttr *) attr;
         int size = battr->Size()
         cout << "ByteArray(" << size << " bytes)\n";
         break;
      default:
         break;
   }
   return 0;  // otherwise
}
```

First, `TestAttr` determines the type of the attribute and then prints its value. If the attribute is an integer equal to 42, then the function returns -1, otherwise it returns 0. If the attribute is a `CSDgsObjectAttr` then the function prints the type of the object such as "TREE" or "LINK". It does this by converting the attribute to a `CSDgsObject*` and then calling the `TypeOfObject` function to determine the type. Since `TypeOfObject` returns integers, the function calls `PrintObjectType` to print out a meaningful string that corresponds to the number. The source code for `PrintObjectType` is in the appendix.

This concludes the tutorial. The commented source code for this program is in the appendix.

6 Appendix: Source Code for the Tutorial

```
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
// FILE: tutorial.c
//
// Purpose:
//
//   Searches through all of the nodes in the artifact, printing information
//   about what it finds and collecting all of the nodes that have an
//   integer attribute equal to 42.
//
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

#include <iostream.h>
#include <iostream.h>
#include <strings.h>
#include <csdgs_include.h>


extern "C"
{
        void exit(int);
}

/* @@@@@@@@@@@@@@@  GLOBAL VARIABLES  @@@@@@@@@@@@@@@ */

// class that controls the connection to the DGS
//
CSDgsConnection dgs;


// subgraph to store the results of the search
//
CSDgsNetwork *NodesThatMatch;

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
```

```
// ****************************************************
// void              PrintObjectType
//
// PURPOSE:  prints a string representation of the type.
//
// ****************************************************
void PrintObjectType( int object_type )  {

   switch ( object_type )  {
      case DGS_NODE:
         cout << "node";         break;
      case DGS_SLINK:
         cout << "S-link";       break;
      case DGS_HLINK:
         cout << "HS-link";      break;
      case DGS_TREE:
         cout << "tree";         break;
      case DGS_NETWORK:
         cout << "network";      break;
      case DGS_LIST:
         cout << "list";         break;
      case DGS_HGRAPH:
         cout << "HS-subgraph"; break;
      default:
         cout << "unknown!";    break;
   }
}
```

```
// **************************************************
// void           CreateSubgraphForResults
//
// PURPOSE:  Creates a network subgraph to store
//           the nodes that match the search criteria.
//
// **************************************************
void CreateSubgraphForResults( CSDgsGraph *root_subgraph,
                               CSDgsNode *root_node, CSDgsNode *&new_node )  {

// Create a new first child of the root node

   new_node = root_subgraph->CreateFirstChild( root_node );
   if ( new_node == nil ) {
      cerr << "CreateSubgraphForResults(): could not create first child\n";
      exit(-1);
   }

// Close root_subgraph. We are through with it.

   if (root_subgraph->Close() != 0) {
      cerr << "CreateSubgraphForResults(): could not close root_subgraph\n";
      exit(-1);
   }

// Open new_node in write mode

   if ( new_node->Open( DGS_WRITE ) != 0) {
      cerr << "CreateSubgraphForResults(): could not open new_node\n";
      exit(-1);
   }

// Create a new subgraph to hold the nodes that match the query,
// and open it.

   NodesThatMatch = new_node->CreateNetworkContent( root_subgraph );

   if ( NodesThatMatch->Open( DGS_WRITE ) != 0 ) {
      cerr << "CreateSubgraphForResults(): couldnt open subgraph\n";
      exit(-1);
   }

}
```

```
// ****************************************************
// void           Initialize
//
// PURPOSE:  Opens the root subgraph and creates a new
//           node that will store the results of the
//           search in its subgraph content.
//
// ****************************************************
void Initialize( CSDgsTree *&root_subgraph, CSDgsNode *&root_node );

// Establish the connection with the DGS

   dgs.Initialize();

// Get a pointer to the root subgraph

   root_subgraph = dgs.GetRootSubgraph();

// Open the root subgraph in read mode

   if ( root_subgraph->Open(DGS_WRITE) != 0 ) {
      cerr << "Initialize(): could not open root subgraph\n"; exit(-1);
   }

// Get the root node of the root subgraph

   root_node = root_subgraph->Root();
   if (root_node == nil) {
      cerr << "Initialize(): could not get root node\n"; exit(-1);
   }
}
```

```
// ************************************************
// void              TestAttr
//
// PURPOSE:  Prints the name and value of the attribute.
//           Returns -1 if the attribute is an integer equal to 42.
//           Returns 0, otherwise.
//
// ************************************************
int TestAttr( DgsAttr* attr, DgsString& name )  {

   cout << "Name: " << name << ", Value: ";

   switch ( attr->TypeOfAttr() )  {
      case DGS_INT:
         int num = (CSDgsIntAttr) *attr;
         cout << "Int(" << num << ")\n";

         // return -1 if the integer is equal to 42
         if ( num == 42 ) return -1;

         break;
      case DGS_STRING:
         cout << "String(" << *( (CSDgsStringAttr *) attr) << ")\n"; break;
      case DGS_FLOAT:
         cout << "Float(" << *( (CSDgsFloatAttr *) attr) << ")\n";   break;
      case DGS_DOUBLE:
         cout << "Double(" << *( (CSDgsDoubleAttr *) attr) << ")\n"; break;
      case DGS_OBJECT:
         CSDgsObject *object = *( (CSDgsObjectAttr *) attr);
         int object_type = object->TypeOfObject();
         cout << "Object(" << PrintObjectType( object_type ) << ")\n";
         break;
      case DGS_BYTE_ARRAY:
         CSDgsByteArrayAttr *battr = (CSDgsByteArrayAttr *) attr;
         int size = battr->Size()
         cout << "ByteArray(" << size << " bytes)\n";
         break;
      default:
         break;
   }
   return 0;  // otherwise
}
```

```
// **************************************************
// void             SearchAttrsOfNode
//
// PURPOSE:  Prints the common and graph attributes of the node.
//           Adds the node to the subgraph NodesThatMatch if
//           the node contains an integer attribute equal to 42.
//
// **************************************************
void SearchAttrsOfNode( CSDgsGraph *graph, CSDgsNode *node )  {

   cout << "...common attributes...\n";

   for ( DgsString name = node->FirstAttrName();
         name != dgs_nullstr;
         /* this space is intentionally blank */      )  {

      DgsAttr *attr = node->GetAttr( name );

      // print and test the attribute.
      if ( TestAttr( attr, name ) == -1 )
         NodesThatMatch->Add(node);

      name = component->NextAttrName( name );
   }

   cout << "\n...graph attributes...\n";

   for ( DgsString name = graph->FirstGAttrName( node );
         name != dgs_nullstr;
         /* this space is intentionally blank */      )  {

      DgsAttr *attr = graph->GetGAttr( name, node );

      // print and test the attribute.
      if ( TestAttr( attr, name ) == -1 )
         NodesThatMatch->Add(node);

      name = graph->NextGAttrName( name, node );
   }

}
```

```
// *************************************************
// void        SearchContentOfNode
//
// PURPOSE:  Does nothing if the content is a file and if the
//           content is empty.  If the content is a subgraph,
//           then it calls the recursive search function on the
//           content.
//
// *************************************************
void SearchContentOfNode( CSDgsNode *node    )  {

   void SearchSubtree( CSDgsGraph* );

   DgsLinkTable *lt; DgsAnchorTable *at;
   CSDgsContent content = node->GetContent( lt, at);

   switch ( content.FormOfContent() )  {
    case DGS_FILE_CONTENT:,
      break;
    case DGS_GRAPH_CONTENT:
      CSDgsGraph *subgraph = content;
      SearchSubtree( subgraph );
      break;
    case DGS_EMPTY_CONTENT:
      break;
    default:
         cerr << "SearchContentOfNode(): unknown form of content\n";
         cerr << "form = " << content.FormOfContent();
         cerr << endl;
         exit(-1);
   }
}
```

```
// ***************************************************
// void       SearchNode
//
// PURPOSE:  Calls other functions that print the attributes
//           of the node while looking for the number 42.
//           Recursively searches any nodes that can be
//           reached through the content of the node.
//
// ***************************************************
void SearchNode( CSDgsGraph *graph, CSDgsNode *node    )  {

   // First, check to see if we are already in the process of
   //        searching this node.
   DgsStatus status = node->Status();
   //        do not continue if the node is already open.
   if ( status.IsOpen() == dgs_true ) return;

   if ( node->Open( DGS_READ_NO_ANCHOR ) != 0 )  {
      cerr << "SearchNode(): could not open node" << endl;
      exit(-1);
   }

   SearchContentOfNode( node );

   cout << "Searching node.\n";

   SearchAttrsOfNode( graph, node );

   if ( node->Close() != 0 )  {
      cerr << "SearchNode(): could not close node." << endl;
      exit(-1);
   }

   cout << "Node closed.\n";
}
```

```
// *************************************************
// void            SearchSubgraph
//
// PURPOSE:  For each node in the subgraph, the function
//           causes the node's attributes to be printed.
//           Nodes that have an integer attribute equal to 42
//           are added to a special subgraph.  The function
//           is recursive on the content of the nodes.
//
// *************************************************
void SearchSubgraph( CSDgsStGraph *subgraph )  {

   if ( subgraph->Open(DGS_READ) != 0 )  {
      cerr << "SearchSubgraph(): could not open subgraph" << endl;
      exit(-1);
   }

   for ( CSDgsNode *node = subgraph->FirstNode();
         node != nil;
         /* this space is intentionally blank */   )  {

         SearchNode( subgraph, node );

         node = subgraph->NextNode( node );
   }

   if ( subgraph->Close() != 0 )  {
      cerr << "SearchSubgraph(): could not close the subgraph." << endl;
      exit(-1);
   }
}
```

```
//-----------------------------------------------------------------//
//         MAIN   ----------------------------   MAIN            //
//-------------------------    MAIN    ----------------------------//
//         MAIN   ----------------------------   MAIN            //
//-----------------------------------------------------------------//

main() {

CSDgsTree *root_subgraph;  // Root subgraph
CSDgsNode *root_node;      // Root node of the root subgraph
CSDgsNode *new_node;       // A new node that will be created to hold
                           // the results in its subgraph content.

// Initialize and create a subgraph to store the nodes that match
// the search criteria

Initialize( &root_subgraph, &root_node );
CreateSubgraphForResults( root_subgraph, root_node, &new_node );

// Start the recursive search on the root subgraph.

SearchSubgraph( root_subgraph );

// At this point, the subgraph 'NodesThatMatch' will contain all
// of the nodes that match the search criteria.


// Close the objects that are still open

   if (new_node->Close() != 0) {
      cerr << "MAIN error: could not close new_node" << endl;
      exit(-1);
   }
   if (NodesThatMatch->Close() != 0) {
      cerr << "MAIN error: could not close NodesThatMatch" << endl;
      exit(-1);
   }

}
```