DESIGN AND IMPLEMENTATION OF A GENERIC GRAPH CONTAINER IN JAVA™

by

David E. Goldschmidt

A Project Submitted to the

Graduate Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements of the Degree of

MASTER OF SCIENCE

in Computer Science

Approved:

____

Dr. Mukkai Krishnamoorthy
Project Advisor

Rensselaer Polytechnic Institute
Troy, New York

April 1998
(For Graduation May 1998)

Table of Contents

1 Project Overview and Motivation

1.1 Java as a Programming Language of Choice

With the continued expansion and development of the Internet, Java has established itself as a versatile and enjoyable tool for creating Java *applets*, miniature Java applications that are usually designed to execute in browser applications on client machines. Although Java has been heralded as an Internet programming language that is ideal for creating applets that fill World Wide Web pages with interactive and “cool” graphical interfaces and games, Java should also be recognized as a very powerful object-oriented programming language that has become widely accepted by programmers, schools and universities, the Internet community, and major computer companies including Netscape Communications Corp. and Microsoft Corp. (not to mention the language's creators at Sun Microsystems, Inc.).

Java’s syntax is very similar to C and C++; however, a majority of the programming pitfalls that exist in C and C++ are not present in Java.[1] Although Java borrows a large number of concepts and keywords from the C and C++ languages, the semantics of a few of these concepts and keywords differ from that of their C and C++ counterparts.[2] In general, the developers of the Java language excluded many of the “extraneous” features of the C and C++ languages in an effort to create a robust and concise programming language.[3]

As with other prominent programming languages such as C and C++, there are a number of software libraries available for use to the Java programmer (these are called *packages* in Java). Libraries typically consist of numerous functions and reusable programming components that are to be utilized and manipulated to match the given problem domain. For ANSI C programmers, the standard C libraries mandated by the ANSI and ISO Standards contain simple components for working with printable characters, character strings, numbers and numeric limits, signals and sockets supported by the local operating system, date and time-related constructs, basic input and output functions, and so on.[4]

1. The most common example of this is the lack of pointer arithmetic and pointer manipulation in the Java language. Another good example is the automatic garbage collection that occurs in Java; memory leaks are not present in Java. Yet another example is the lack of multiple inheritance amongst classes in Java, a feature of C++ that many feel is a design flaw of the C++ language.
2. A good example of this involves the use of the `public`, `protected`, and `private` modifiers on members of a class. Specifically, members that are modified with the `protected` keyword in C++ are visible in the given class and its subclasses. In Java, members that are modified with the `protected` keyword are visible in the given class, its subclasses, and *in all other classes of the same package*.
3. Example “extraneous” features of the C and C++ languages include operator overloading, multiple inheritance, default arguments in function prototypes, passing objects to functions by value (thus automatically invoking a copy constructor), and so on.
4. Refer to **[Plau92]** for a description of these standards, as well as implementation strategies in C.

1.1.1 The C++ Standard Template Library

Moving forward from C to C++, the recently standardized language contains libraries similar to that of C, with improved libraries for manipulating input and output *streams*.[5] The Standard Template Library (STL) is also included in the set of standardized C++ libraries. STL provides a number of generic *container* classes that are designed to maintain an organized set of arbitrary objects of the same type; example STL container classes include `vector`, `list`, `set`, and `map`. The "genericity" of these container classes is achieved by implementing these classes as template classes in C++.

A number of additional programming components complement the generic container classes of the Standard Template Library. *Iterators* are used to traverse through the objects stored in the container class instances. *Adaptors* are used to modify the interface of an existing component. *Function objects* provide a useful mechanism for defining generic operations on objects to be stored in container class instances; for example, a function object may define a comparison function for a given object type. Finally, numerous *generic algorithms* exist in STL that provide such functionality as sorting, searching, merging, locating, rotating, and partitioning components of container class instances. In general, the generic STL algorithms operate on the objects stored in the container class instances.[6]

1.1.2 Standard Java Packages

Moving forward from C++ to Java, the standard Java Development Kit that is currently available from Sun (JDK1.1.5) contains over 20 packages consisting of over 500 classes (a substantial increase from the 8 packages and roughly 200 classes found in Java 1.0). Many of these Java packages provide programming components similar to those found in the aforementioned C and C++ libraries. For example, the `java.lang` package provides fundamental math and string operations, as well as other primary classes that are most central to the Java language. As another example, the `java.io` package provides the basic input and output stream manipulators and classes.

Because Java is a relatively new language, many of the new networking, World Wide Web, and Internet-related concepts have been integrated into the language via standard Java packages. Specifically, the `java.net`, `java.security`, and `java.applet` packages provide the networking and security-related components that pertain to the Internet and the World Wide Web. Further, a platform-independent graphical interface package called the Abstract Windowing Toolkit (AWT) is part of the standard JDK distribution (i.e. the `java.awt` package and related subpackages). Other

5. Refer to **[Stro97]**, **[Elli90]**, **[Plau98]**, and other C++ resources for detailed information on the C++ language and its standard set of libraries.
6. Refer to **[Muss96]** for a comprehensive study of the C++ Standard Template Library, as well as a set of useful examples and tutorials that showcase the capabilities of STL. Also refer to **[Plau98]**.

features of the Java language include the *Java Beans API*, *Internationalization (I18N)*, the *Java Event Model*, *Object Serialization*, *Reflection*, *Remote Method Invocation (RMI)*, and the *Java Database Connectivity (JDBC) API*.[7]

1.1.3 Generic Container Classes in Java

Compared to the programming components provided by the C++ Standard Template Library, the standard Java 1.1 packages provide only a very limited set of generic container classes and related generic programming components. Presented as only a part of the `java.util` class, Java 1.1 contains two primary container classes: the `Hashtable` class and the `Vector` class.[8] Two additional container classes exist as subclasses to these primary container classes: the `Properties` class and the `Stack` class (which are subclasses of the `Hashtable` and `Vector` classes, respectively). Similar in concept to the STL iterator, the `Enumeration` interface provides a fairly limited mechanism for iterating through the objects stored in a container class instance.[9]

As part of this project, a more extensive `rpi.goldsd.container` package has been designed and implemented.[10] The `rpi.goldsd.container` package includes a linked list and a more flexible hashtable, as well as numerous generic interfaces to define sequences, comparable objects, hashable objects, associative containers, and so on. At the time of writing (April 1998), an initial beta version of Java 1.2 became available from Sun's Java website. In Java 1.2, the generic container classes and interfaces of Java 1.1 are extended and complemented with additional interfaces and classes. It is interesting to note that some of the strategies used in designing and structuring the `rpi.goldsd.container` package for this project have also been used independently in the design and implementation of the collection classes and interfaces of Java 1.2.

1.2 Implementing Graphs Using Java

The *graph* is one of the most useful and powerful data structures in the field of Computer Science. The scope of this data structure spreads well beyond the realm of Computer Science and even academia; application areas include various fields of science, neural network applications, electrical engineering, operations research, sociology, communications and networking, transportation sys-

7. Refer to **[Flan97]** for a comprehensive study of the Java language and the language features listed above. Also refer to **[Hors97]**.
8. Note that the `Hashtable` class is actually a subclass of the abstract `Dictionary` class that is also part of the `java.util` package. The abstract `Dictionary` class contains seven abstract methods that define the basic set of operations on a hashtable-like associative container.
9. "Fairly limited" is used to describe the `Enumeration` interface since the only operation allowed on the given enumeration is to move to the next element. Iterators in STL allow a much more flexible and comprehensive means of traversing and iterating through the objects of a container class instance.
10. In an attempt to avoid conflicting class and interface names, the container package (as well as the graph package) is implemented in the `rpi.goldsd` package namespace. Refer to **[Flan97]**, **[Hors97]**, and other Java resources for more information on globally unique package names.

tems, scheduling systems, simulation systems, and so on. The primary advantage to using the graph data structure is its flexibility. An extremely wide spectrum of problems may be represented using a sound implementation of the graph data structure.

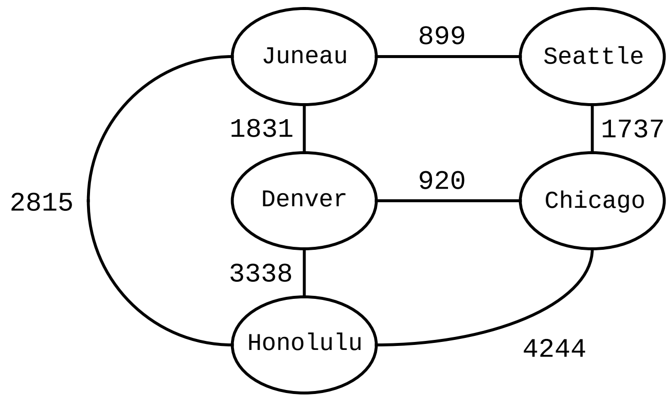
The fundamental goal of this project is to provide a generic graph package that abstracts the graph data structure and related algorithms in Java. The `rpi.goldsd.graph` package has been implemented and tested using Java 1.1 (specifically JDK1.1.5 under Windows 95 and JDK1.1.4 under Solaris).[11] This graph package utilizes many of the interfaces and classes of the aforementioned `rpi.goldsd.container` package to provide a robust and efficient implementation of the graph data structure and many graph-related algorithms.

1.2.1 Graphical Representation of Graphs Using the Java AWT

An aspect of graphs and graph theory that this project does not cover is the actual graphical representation of graphs using the Java AWT. All is not lost however, since the `rpi.goldsd.graph` package successfully interfaces with the *Graph Draw* applet developed by Úlfar Erlingsson and Mukkai Krishnamoorthy at Rensselaer.[12]

1.2.2 The `Cities.java` Example Program

As an introductory example, consider the following Java program called `Cities.java`. This program makes use of the `rpi.goldsd.graph` package to construct a small graph that represents cities in the United States (the five cities represented in the constructed graph are Chicago, Denver, Honolulu, Juneau, and Seattle). Each vertex represents a single city, and each undirected edge represents an existing path between two cities. The weight of each edge represents the approximate distance between the two corresponding cities (measured in miles). The graph is constructed as follows:



11. Neither the `rpi.goldsd.graph` package nor the `rpi.goldsd.container` package has been ported to the new beta version of Java 1.2.

12. Refer to `http://www.cs.rpi.edu/projects/pb/graphdraw` for more information on the interactive Graph Draw applet. Also refer to §3.10.7 of this document.

The `Cities.java` program is presented as follows:

```
import rpi.goldsd.graph.*;
public class Cities
{
   public static final void main( String[] args )
   {
      // Create a Vertex object for each city ...
      Vertex chicago = new Vertex( “Chicago” );
      Vertex denver = new Vertex( “Denver” );
      Vertex honolulu = new Vertex( “Honolulu” );
      Vertex juneau = new Vertex( “Juneau” );
      Vertex seattle = new Vertex( “Seattle” );

      // Create weighted Edge objects between some cities ...
      Edge E1 = new Edge( chicago, denver, 920.0 );
      Edge E2 = new Edge( chicago, honolulu, 4244.0 );
      Edge E3 = new Edge( denver, honolulu, 3338.0 );
      Edge E4 = new Edge( denver, juneau, 1831.0 );
      Edge E5 = new Edge( honolulu, juneau, 2815.0 );
      Edge E6 = new Edge( seattle, chicago, 1737.0 );
      Edge E7 = new Edge( seattle, juneau, 899.0 );

      // Construct a named graph ...
      Graph USMap = new Graph( “US Cities Graph” );

      // Add the vertices to the graph ...
      USMap.add( chicago );   USMap.add( denver );
      USMap.add( honolulu );  USMap.add( juneau );
      USMap.add( seattle );

      // Add the weighted edges to the graph ...
      USMap.add( E1 );  USMap.add( E2 );  USMap.add( E3 );
      USMap.add( E4 );  USMap.add( E5 );  USMap.add( E6 );
      USMap.add( E7 );

      USMap.printSummary();

      // Find the shortest path between Chicago and Juneau ...
      Path P = Algorithms.findShortestPath( chicago, juneau );
      System.out.print( “Shortest Path from ” + chicago );
      System.out.println( “ to ” + juneau + “ is: ” + P );
      System.out.println( “Overall distance is ” + P.weight() + “ miles.” );

      // Construct a BFS Tree rooted at Honolulu ...
      Tree T = Algorithms.breadthFirstTraversal( honolulu );
      T.printSummary();
   }
}
```

1.2.2.1 Initial Graph Construction

In the `Cities.java` program, after the `Vertex` and `Edge` objects are created, they are added to an instance of the `Graph` class called `USMap` via the `add()` method. Note that the `add()` method of the `Graph` class accepts either a `Vertex` or an `Edge` object as an argument (i.e. the method is overloaded). Making use of the `printSummary()` method that is defined in the abstract `GraphBase` class that the `Graph` class is a subclass of, the following summary is then displayed:

```
US Cities Graph:
  # of vertices: 5
  # of undirected edges: 7
  Vertices and incident edges:
    Denver: 1[W920.0] (Chicago), 3[W3338.0] (Honolulu), 4[W1831.0] (Juneau).
    Seattle: 6[W1737.0] (Chicago), 7[W899.0] (Juneau).
    Chicago: 1[W920.0] (Denver), 2[W4244.0] (Honolulu), 6[W1737.0] (Seattle).
    Honolulu: 2[W4244.0] (Chicago), 3[W3338.0] (Denver), 5[W2815.0] (Juneau).
    Juneau: 4[W1831.0] (Denver), 5[W2815.0] (Honolulu), 7[W899.0] (Seattle).
```

As shown in the output above, the `printSummary()` method displays the name of the graph, the number of vertices in the graph, and the number of edges in the graph. An adjacency-list format is used for displaying all of the vertices and their incident edges. From the output above, the vertex representing Denver has three incident edges that lead to Chicago, Honolulu, and Juneau with weights of 920, 3338, and 1831, respectively. Since the graph is undirected, the line of output pertaining to the Chicago vertex contains an incident edge back to Denver with the same weight of 920 (as well as two other incident edges that lead to Honolulu and Seattle).

1.2.2.2 Determining the Shortest Path Between Two Cities

As a component of the `rpi.goldsd.graph` package, the `Path` class represents a *path* or *walk* through an instance of the `Graph` class. In the `Cities.java` program, an instance of the `Path` class is created by invoking the `findShortestPath()` method of the static `Algorithms` class. Note that the constructed `Path` object is essentially an array of alternating references to `Vertex` and `Edge` objects in the `USMap` class (i.e. no cloning of objects occurs). The following output is displayed after the shortest path is determined using Dijkstra's Shortest Path algorithm:

```
Shortest Path from Chicago to Juneau is: Chicago->Seattle->Juneau.
Overall distance is 2636.0 miles.
```

Since the weights used in the `Cities.java` program do approximate the actual distances between cities, the resulting path geographically seems correct. There is no direct path in the graph from Chicago to Juneau, thus at least one additional vertex needs to be visited en route to Juneau. By simple inspection of the graph, it should be clear that Seattle is the correct intermediate vertex choice: Honolulu is out of the question, whereas Denver would result in a path that is 115 miles longer according to this graph.

1.2.2.3 Constructing a Breadth-First Search Tree of Cities

Another fundamental class of the `rpi.goldsd.graph` package is the `Tree` class that is used to represent a *tree*. At the end of the `Cities.java` program, an instance of the `Tree` class is created by invoking the `breadthFirstTraversal()` method of the static `Algorithms` class. This method constructs a subgraph of the `USMap` graph that is a *breadth-first search tree* (*BFS tree*) rooted at the `Vertex` object that represents Honolulu. In contrast to the `Path` class introduced above, the constructed `Tree` object contains clones (or copies) of the `Vertex` and `Edge` objects of the original `USMap` graph.[13] Thus, changes to the original graph do not affect the BFS tree, and vice versa. The `printSummary()` method is then used to display the BFS tree, as follows:

```
BFS Tree:
  # of vertices: 5
  # of undirected edges: 4
  Vertices and incident edges:
    Denver: 3[W3338.0] (Honolulu).
    Seattle: 6[W1737.0] (Chicago).
    Chicago: 2[W4244.0] (Honolulu), 6[W1737.0] (Seattle).
    Honolulu: 2[W4244.0] (Chicago), 3[W3338.0] (Denver), 5[W2815.0] (Juneau).
    Juneau: 5[W2815.0] (Honolulu).
```

Although this is a very simple and somewhat contrived example, the `Cities.java` program reveals some of the capabilities and possibilities of the `rpi.goldsd.graph` package.[14]

1.2.3 Project Availability on the World Wide Web

The current website for this project is `http://www.cs.rpi.edu/projects/pb/jgb`, where `jgb` stands for *Java Graph Base*. All of the source code for the container and graph packages are available from this website, as well as the source code for the example and test programs (including the `Cities.java` program described in the previous section). The source code should be considered a *beta* release since the packages have not yet been exhaustively tested, nor are these packages full-scale and complete. Since this project has been only a semester-long project in which I have also learned the Java programming language, more functionality will be added to these packages as time goes on. This document is also available from the given website, and will be updated appropriately as the packages continue to evolve. If you happen to find errors or inconsistencies in the package implementations, or if you have developed interesting applications making use of these packages, please contact me at `goldsd@cs.rpi.edu` with the details.

13. The `Edge` class does not actually implement the `Cloneable` interface and is therefore not able to be cloned via the `clone()` method; however, the `Edge` objects are still copied in the `breadthFirstTraversal()` and other methods. Refer to §3.4.1 and §3.10.1.1 for more details.
14. Refer to §3 for a thorough discussion of the `rpi.goldsd.graph` package, as well as §4 for additional example applications.

2 Implementing Generic Java Containers

As mentioned in the previous section, Java 1.1 provides only a limited set of generic container classes in the standard `java.util` package. As a building block for the generic graph package, the `rpi.goldsd.container` package has been designed and implemented as part of this project. The following sections describe the interfaces and classes of the `rpi.goldsd.container` package. Note that the descriptions that follow are not meant to be used as a comprehensive reference, but rather as a discussion of how the interfaces and classes were implemented and why many of the design decisions were made.[15] Also note that the `rpi.goldsd.container` package is by no means on par with the design of STL in terms of completeness or flexibility.

2.1 Interfaces of the Container Package

The Java *interface* provides a concise and extensible mechanism for defining the behavior of a generic programming component. The following table contains a brief summary of each of the interfaces that are defined in the `rpi.goldsd.container` package:

Interface Name	**`extends`**	**Interface Description**
`AssociativeContainer`	-	The `AssociativeContainer` interface defines the methods required to implement a mapping or hash-table-like container object.
`Comparable`	-	The `Comparable` interface defines the methods required by any object that can be compared to other objects of the same type.
`Hashable`	`Comparable`	The `Hashable` interface defines two versions of the `hash()` method for those objects that are to be stored in a hashtable or other associative container.
`Linkable`	-	The `Linkable` interface defines the methods required by any object that can be linked into a doubly linked list or other such bidirectionally linked structure.
`Sequence`	-	The `Sequence` interface defines two `sequence()` methods to be used to generate an enumeration of sequential values of a given type.

2.1.1 The `AssociativeContainer` Interface

Similar to the abstract `java.util.Dictionary` class, the `AssociativeContainer` interface defines the methods required to implement a mapping or hashtable-like container class. The elements contained within an associative container may be arbitrary heterogeneous Java objects; however, a specific key must be used to identify each element of the container. This required key

15. For a comprehensive reference of the interfaces and classes of the `rpi.goldsd.container` package, refer to the generated `javadoc` webpages at `http://www.cs.rpi.edu/projects/pb/jgb/jdoc`.

must be a Java class that implements the `Comparable` interface so that an `isEqualTo()` method is defined on the given objects in the container. Most classes that implement the `AssociativeContainer` interface will likely use a key that is *unique*; however, this restriction is *not* imposed or enforced by this interface.

The `AssociativeContainer` interface differs from the abstract `Dictionary` class by requiring the explicit use of the `Comparable` interface in the class used as the key. Although the fundamental root `java.lang.Object` class contains an `equals()` method that is similar in concept to the `isEqualTo()` method of the `Comparable` interface, the `Comparable` interface forces the package-user to create an explicit `isEqualTo()` method rather than rely on an `equals()` method that may or may not be overridden from the root `Object` class. In other words, a more robust environment is maintained in using the `Comparable` interface (and other interfaces and components of the `rpi.goldsd.container` package).

As described in greater detail below, the `Comparable` interface also provides an `isLessThan()` method such that a total ordering is defined, which is a requirement of some associative containers.[16] Also note that the `AssociativeContainer` interface contains more methods than the abstract `Dictionary` class of the `java.util` package, and provides a more complete and robust definition of an associative container.

Methods defined in the `AssociativeContainer` interface include: `add()` and `remove()` to add and remove objects from an associative container; `clear()` to remove all elements from an associative container; `isEmpty()` to determine if an associative container contains any elements at all; `getSize()` to obtain the number of elements in an associative container; `map()` to map a specific key to an object stored in an associative container; and so on.[17]

2.1.2 The `Comparable` Interface

The `Comparable` interface defines the methods required by any object that can be compared to other objects of the same type. As mentioned in the previous section, the root `Object` class of the standard `java.lang` package contains a fundamental `equals()` method that is similar in concept to the `isEqualTo()` method of the `Comparable` interface. Because `Object` is a base class as opposed to an interface, users are *not* required to override the `equals()` method; however, as part of the `Comparable` interface, the `isEqualTo()` method must be *explicitly* implemented by the user. Such a requirement forces a much more robust and complete implementation in the classes developed by the `rpi.goldsd.container` package-users.

16. Specifically, the associative containers of STL are implemented using red-black trees that maintain a total ordering of the keys of the container. Refer to **[Muss96]** for more information.
17. For a complete set of methods, refer to `http://www.cs.rpi.edu/projects/pb/jgb/jdoc`.

Aside from the `isEqualTo()` method, the `Comparable` interface also defines the `isLessThan()` method for determining if one object is less than another. Classes implementing the `Comparable` interface should aim to implement a *strict total ordering* of elements via the `isEqualTo()` and `isLessThan()` methods. Specifically, every pair of elements in a given set `S` must be related by the < operator (via the `isLessThan()` and `isEqualTo()` methods) in the following manner:

- For every `x`, `y`, and `z` in `S`, if `x<y` and `y<z`, then `x<z` (*transitivity*).
- For every `x` and `y` in `S`, exactly one of the following is true: `x<y`, `y<x`, or `x==y` (*trichotomy*).[18]

All basic comparative operations (i.e. <, >, <=, >=, and ==) may be performed using the two methods defined by the `Comparable` interface; for example, the >= operator may be implemented on two `Comparable` objects `A` and `B` as follows:

```
if ( A.isEqualTo(B) || B.isLessThan(A) )
{
   // A is greater than or equal to B ...
}
```

Given such a definition, the `Comparable` interface may be used by numerous container classes such that sorting, searching, and other order-dependent operations may be performed in a generic fashion.

2.1.3 The `Hashable` Interface

The `Hashable` interface defines two versions of the `hash()` method for those objects that are to be stored in a hashtable or other such associative container. Both versions of the `hash()` method produce an integer *hash value* that identifies the object implementing the `Hashable` interface. Note that the `Hashable` interface extends the `Comparable` interface such that the `isEqualTo()` method is defined.

The default no-argument version of the `hash()` method must produce a hash value that is a valid Java integer. The algorithm used to generate such a hash value must also adhere to the following three rules:

- The `hash()` method must perform in constant time. It is very important for a hashing function to perform efficiently, regardless of the input.
- For a given object, the hash value must not change during the lifetime of that object. More specifically, each call to the `hash()` method for a given object must always return the same hash value.

18. Refer to §5.4 of **[Muss96]** for more details on defining relations that are *strict total orderings* and when and how these requirements may be relaxed.

- Two objects that are deemed equal via the `isEqualTo()` method that is defined by the `Comparable` interface must have identical hash values. In other words, independent calls to the `hash()` method of the equal objects must produce the same hash value.

Ideally, the `hash()` method should generate hash values that are fairly random; i.e., to take advantage of the performance gains of associative hashtable structures, the hash values should be evenly distributed over the size of the hashtable.[19]

When the size of the hashtable is known, the second `hash()` method should be used. The single argument to this version of the `hash()` method is an integer value representing the overall size of the associative container that will make use of the generated hash value. This `tableSize` argument should be used in the generation of the hash value for the given object.

The generation of the hash value using this second version of the `hash()` method must adhere to the same three requirements of the default no-argument `hash()` method described above; however, the second and third requirements described above should only be adhered to for calls to the `hash()` method in which the single integer argument is the same. For example, if this `hash()` method is repeatedly called with a `tableSize` value of 101, the resulting hash values must all be the same. If a `tableSize` value of 203 is then used, the resulting hash value may be different from the one obtained with a `tableSize` value of 101.

A fourth requirement also exists when using the single-argument version of the `hash()` method:

- The resulting hash value must be in the inclusive range `0..(tableSize-1)`.

If nothing else, this method should simply call the default no-argument `hash()` method and divide the absolute value of the resulting hash value by the given `tableSize` argument. The remainder of such a division is the resulting hash value in the valid range.

As an example, the single-argument `hash()` method may be implemented as follows:

```
public int hash( int tableSize )
{
   // restrict hash value to 0..(tableSize-1)
   return ( Math.abs( hash() ) % tableSize );
}
```

In just about all classes implementing the `Hashable` interface, the single-argument `hash()` method will likely be implemented as shown above.

19. Refer to **[Baas88]**, **[Corm92]**, **[Knut98b]**, **[Sedg88]**, or just about any other book on computer algorithms for detailed information on hashing and the advantages (and disadvantages) of using hashing.

2.1.4 The `Linkable` Interface

The `Linkable` interface defines the methods required of any object that can be linked into a doubly linked list or other such bidirectionally linked structure. Individual nodes of a bidirectionally linked structure will invariably contain a reference to the *next* node in the structure and a reference to the *previous* node in the structure. This interface simply defines four methods for obtaining and changing the next and previous references of such a node; the methods are: `getNext()`, `getPrevious()`, `setNext()`, and `setPrevious()`.

2.1.5 The `Sequence` Interface

The `Sequence` interface defines two `sequence()` methods to be used to generate an enumeration of sequential values of a given non-primitive type. The first no-argument `sequence()` method provides a Java `Enumeration` object that generates sequential values starting from a default value (e.g. a sequence of integers may begin at 1 by default). The second `sequence()` method also provides a Java `Enumeration` object; however, the sequence of values begins at the start-value specified by the single argument to the method (the single argument is called `startValue` and is of type `Object`). Thus a generic mechanism for starting a sequence at an explicitly specified value is provided in the `Sequence` interface.

Both `sequence()` methods return an actual instance of the Java `Enumeration` object. Since the `nextElement()` method of the standard `Enumeration` interface returns a Java `Object` instance, a sequence of the primitive types `int`, `char`, `double`, and so on cannot be generated by the implemented `sequence()` method. Further, the single-argument version of the `sequence()` method requires an `Object` argument; thus, the primitive types also cannot be used here.

To alleviate this problem, Java provides numerous *wrapper-classes* in the `java.lang` package for managing the primitive types; standard Java wrapper-classes include `Integer`, `Double`, `Character`, `Long`, `Short`, and so on. Thus to create a sequence of integers, an `IntegerSequence` class may implement the `Sequence` interface and provide an enumeration of `Integer` objects. As another example, consider an `AlphabetSequence` class that implements the `Sequence` interface and provides a sequence of `Character` objects.

In relation to generic container classes, consider an associative container of arbitrary objects. The unique key values that are assigned to each element of such a container may be generated using the example `IntegerSequence` class just mentioned. A simple call to the `nextElement()` method of the generated `Enumeration` object would produce the next unique key in a consecutive sequence of integer values; however, for associative containers and other generic container classes of the `rpi.goldsd.container` package, the use of the standard Java wrapper-classes is not sufficient since these wrapper-classes do not implement the `Comparable` or `Hashable` interfaces. To address this issue, a new set of wrapper-classes has been designed and implemented as part of this project.[20]

As another example, an enumeration of prime numbers may be generated by implementing the `Sequence` interface. Such a `PrimeNumberSequence` class may be of interest only in a classroom setting as an exercise for students, but there are many other "real-life" applications of generic sequences in computer science, engineering, statistical analysis, financial forecasting, and so on.

Another potential use of the `Sequence` interface involves sequences of *random numbers*. When generated by a computer, random numbers are not truly random; instead, *pseudo-random* numbers are generated using mathematical formulae. In most cases, a *linear congruential method* is used that generates a pseudo-random number based on the previously generated pseudo-random number. To start this process, an initial *seed* value is needed, which is typically calculated by manipulating the current date and time obtained from the local computer's system clock.

As an interesting excursion involving the `Sequence` interface, the `rpi.goldsd.container` package contains a class that implements a simple linear congruential method to generate pseudo-random numbers.[21] The name of this class is `RandomIntSequence` and is described in §2.2.9.

2.2 Classes of the Container Package

The table that follows contains a brief summary of the classes that are currently defined in the `rpi.goldsd.container` package. Included in this table is a summary of the interfaces each class implements, if any.

Class Name	**`implements`**	**Class Description**
`AlphabetSequence`	`Sequence`	The `AlphabetSequence` class provides an enumeration of consecutive `Char` objects in the ranges '`a`'-'`z`' and '`A`'-'`Z`'.
`Bool`	`Hashable`	The `Bool` class is a wrapper-class for the primitive type `boolean`.
`Char`	`Hashable`	The `Char` class is a wrapper-class for the primitive type `char`.
`DuplicateElementException`	-	The `DuplicateElementException` class extends the standard `RuntimeException` class and is used to indicate a unique index violation.

20. Refer to §2.2 and §2.2.1 for detailed information on the new set of wrapper-classes that are defined in the `rpi.goldsd.container` package.
21. For this and other *excursions* in Computer Science, refer to **[Dewd89]**, or the updated edition of this reference **[Dewd93]**. Also refer to §3.2.1 of **[Knut98a]** for details on the linear congruential method.

Class Name	**`implements`**	**Class Description**
`EmptyEnumeration`	`Enumeration`	The `EmptyEnumeration` class provides a default Java `Enumeration` object that contains no elements. Currently, this class is used in the `Table` class by the `elements()` method that expects a single argument.[a]
`Int`	`Hashable`	The `Int` class is a wrapper-class for the primitive type `int`.
`IntSequence`	`Sequence`	The `IntSequence` class provides an enumeration of consecutive `Int` objects.
`LinkedList`	-	The `LinkedList` class provides a generic doubly linked list of `ListNode` objects.
`ListNode`	`Comparable, Linkable`	The `ListNode` class represents a node of the `LinkedList` container.
`PrimeNumberSequence`	`Sequence`	The `PrimeNumberSequence` class is used to generate sequences of prime numbers beginning at either 2 or a specified prime number.
`RandomIntSequence`	`Sequence`	The `RandomIntSequence` class is used to generate simple sequences of random numbers using a basic linear congruential method as described in **[Dewd89]**, **[Dewd93]**, and **[Knut98a]**.
`Real`	`Hashable`	The `Real` class is a wrapper-class for the primitive type `double`.
`Str`	`Hashable`	The `Str` class is a wrapper-class for the fundamental `String` class provided by the `java.lang` package.
`Table`	`AssociativeContainer`	The `Table` class provides a generic associative hashtable by implementing the `AssociativeContainer` interface. Note that the `Hashable` elements that are stored in an instance of the `Table` class must be uniquely identified by the default `hash()` method.

a. Refer to §2.2.10 for more information on the `Table` class and the use of the `EmptyEnumeration` class.

2.2.1 Wrapper-Classes of the Container Package

Using the `Sequence` interface on primitive types in conjunction with the container classes defined in the `rpi.goldsd.container` package requires a new set of wrapper-classes. Using the integer sequence example introduced in §2.1.5, the suggested `IntegerSequence` class returns an enumeration of `Integer` objects, which do not implement the `Comparable` or `Hashable` interfaces. Objects that do not implement the `Comparable` interface may not be used as a key to an associative container that implements the `AssociativeContainer` interface of the `rpi.goldsd.container` package. To circumvent this problem, the following wrapper-classes have been designed to implement the `Hashable` interface:[22]

Wrapper-Class Name	**Primitive Data Type**
`Bool`	`boolean`
`Char`	`char`
`Int`	`integer`
`Real`[a]	`double`
`Str`	`String`[b]

a. Pascal programmers will be happy to see "`Real`" as a data type once again.

b. Of course the `String` class is not a primitive type; however, the appropriate wrapper-class does exist for the `String` class.

2.2.2 The `AlphabetSequence` Class

The `AlphabetSequence` class implements the `Sequence` interface by providing `sequence()` methods that generate enumerations of `Char` objects in the inclusive ranges '`a`'-'`z`' and '`A`'-'`Z`'. Specifically, the default no-argument version of the `sequence()` method returns an enumeration of `Char` objects beginning at the '`a`' character and ending at the '`z`' character. The single-argument versions of the `sequence()` method return enumerations of `Char` objects starting from the specified character and ending at either the '`z`' or '`Z`' character, depending upon the given start character of the generated sequence.

Although a maximum of only 26 distinct values are generated by a given `AlphabetSequence` instance, this class may be used to provide a sequence-generator for unique key-values in hashtables and other associative containers.

2.2.3 The `DuplicateElementException` Class

The `DuplicateElementException` class represents an exception that should be thrown when a method fails due to the occurrence of a duplicate element. This will typically be brought on by a violation of a unique constraint on a key of an associative container. The generic `add()` methods

22. Remember that the `Hashable` interface extends the `Comparable` interface, thus the defined wrapper-classes are both *comparable* and *hashable*.

of the `AssociativeContainer` interface are defined to throw the `DuplicateElementException` exception, if deemed necessary by the associative container class designer.

The generic `Table` class implements the `AssociativeContainer` interface and implements the multiple `add()` methods, each of which may throw this exception since the `Table` class maintains objects identified by a unique key. Other classes that implement the `AssociativeContainer` interface may contain an `add()` method that does not throw this exception if the key values are not required to be unique (e.g. a class similar to the `multimap` class of STL in which duplicate keys are allowed).

2.2.4 The `EmptyEnumeration` Class

The `EmptyEnumeration` class provides a default Java `Enumeration` object that contains no elements. This is accomplished by simply implementing the `hasMoreElements()` method to always return `false`. Methods that are to return an `Enumeration` object may opt to return an instance of the `EmptyEnumeration` class instead of returning `null` or throwing an exception when a noncritical error condition occurs.

Consider the following block of code, which is very common in the use of Java `Enumeration` objects (note that `ContainerInstance` is an arbitrary container object that contains an `elements()` method that returns a Java `Enumeration` object):

```
java.util.Enumeration e = ContainerInstance.elements();
while ( e.hasMoreElements() )
{
   // ...
}
```

In the above block of code, the user relies solely on the `hasMoreElements()` method to determine if elements exist in the `ContainerInstance` class. Requiring the user to check whether `e` is `null` or to catch an exception is far too burdensome for the user. Consider the following code in which the `elements()` method potentially returns `null` to indicate that no elements exist:

```
java.util.Enumeration e = ContainerInstance.elements();
if ( e != null )
{
   while ( e.hasMoreElements() )
   {
      // ...
   }
}
```

In this case, a `null` reference must always be avoided with an explicit `if` statement.

Next, consider this version of the code segment in which the `elements()` method potentially throws a `NoSuchElementException` (or similar) exception:

```
boolean containsElements = true;

try
{
   java.util.Enumeration e = ContainerInstance.elements();
}
catch ( java.util.NoSuchElementException E )
{
   containsElements = false;
}

if ( containsElements )
{
   while ( e.hasMoreElements() )
   {
      // ...
   }
}
```

In the above code, a `try/catch` clause is necessary, as well as an explicit `if` statement based on an additional `boolean` variable.

Currently, the `EmptyEnumeration` class is used in the `Table` class by the `elements()` method that expects a single argument. This version of the `elements()` method requires a single argument that specifies the hash-code bucket of interest. The `Enumeration` object that is returned is an enumeration of all objects that hash to the given hash value. If the given hash value happens to fall on a nonexistent bucket (i.e. a bucket with zero entries), an instance of the `EmptyEnumeration` class is returned.[23]

2.2.5 The `IntSequence` Class

The `IntSequence` class implements the `Sequence` interface by providing `sequence()` methods that generate enumerations of `Int` objects. Specifically, the default no-argument version of the `sequence()` method returns an enumeration of `Int` objects beginning at a default value of 1. Note that when constructing an `IntSequence` instance, the default start and end integer values may be explicitly specified as arguments to the constructor. Thus, the default no-argument `sequence()` method may begin at a default value aside from 1.

When an enumeration created by one of the `sequence()` methods comes to the end of the sequence, the `nextElement()` method of the `Enumeration` object will begin to return values

23. For a more detailed description of the `Table` class, refer to §2.2.10.

starting at the the initial start value once again (i.e., either 1 or a specified start value). Thus, a sequence of integers in the inclusive range [1000, 1099] may be generated as follows:

```
IntSequence I = new IntSequence( 1000, 1099 );
java.util.Enumeration e = I.sequence();
```

The single-argument versions of the `sequence()` method return enumerations of `Int` objects starting from the specified integer arguments. Thus, a sequence of integers beginning at 200 may be generated from a default `IntSequence` instance as follows:

```
IntSequence J = new IntSequence();
java.util.Enumeration e = J.sequence( 200 );
```

This class is generally used to provide a sequence-generator for unique key-values in hashtables and other associative containers. The `Vertex` and `Edge` classes of the `rpi.goldsd.graph` package make use of the default `IntSequence` class in many of their default constructors.[24]

2.2.6 The `LinkedList` Classes

The `LinkedList` class provides a generic linked list container by managing a doubly linked list of `ListNode` objects. Described in more detail in the following section, the `ListNode` class provides a generic node class that contains arbitrary objects that implement the `Comparable` interface such that the `isEqualTo()` method is defined.

The following table provides a description of each method of the `LinkedList` class. Note that the worst-case run-time performance using "Big-Oh" notation is also specified for each method.

Method Name	**Order**	**Method Description**
`addAfter()`	`O(1)`	Adds a new `ListNode` object to the linked list after the given `ListNode` object argument. This method assumes that the existing node does indeed exist in the linked list (verifying this assumption would change the running time to **O(n)** instead of constant time order).
`addToHead()`	`O(1)`	Adds a new `ListNode` object to the beginning of the linked list.
`addToTail()`	`O(1)`	Adds a new `ListNode` object to the end of the linked list.
`contains()`	`O(n)`	Tests whether the `Comparable` object argument exists in the linked list by making use of the `isEqualTo()` method of the given `Comparable` object.
`containsReferenceTo()`	`O(n)`	Tests whether the arbitrary `Object` argument exists in the linked list by comparing *object references* via the == operator.

24. Refer to §3.3.1 and §3.4.1 for a detailed description of the constructors of the `Vertex` and `Edge` classes, respectively.

Method Name	**Order**	**Method Description**
`elements()`	`O(1)`	Provides an enumeration or traversal of the linked list starting at the beginning of the list. The `nextElement()` method of the instantiated `Enumeration` object returns the next `ListNode` object in the linked list. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(n)**.
`find()`	`O(n)`	Traverses the linked list to find the *first* occurrence of the given `Comparable` object by using the `isEqualTo()` method. The `find()` method is used by the boolean `contains()` method that is described above.
`getSize()`	`O(1)`	Returns the number of elements in the linked list.
`isEmpty()`	`O(1)`	Tests whether the linked list contains any elements.
`remove()`	`O(n)`	Removes and returns a reference to the *first* occurrence of the given `ListNode` or `Comparable` object argument. To find the given object in the linked list, the `isEqualTo()` method is applied to the given argument and each node of the linked list. Note that the reference that is returned refers to the `ListNode` object that is removed (or `null` if not found).
`removeFromHead()`	`O(1)`	Removes and returns a reference to the `ListNode` object that is currently the head of the list.
`removeFromTail()`	`O(1)`	Removes and returns a reference to the `ListNode` object that is currently the tail of the list.
`toString()`	`O(n)`	Overrides the `toString()` method of the fundamental `java.lang.Object` class to provide a mechanism for displaying all nodes of the linked list.

2.2.7 The `ListNode` Class

The `ListNode` class is used to represent a single node of the `LinkedList` container class by implementing the `Linkable` interface. An arbitrary `Comparable` object must be associated with every instance of the `ListNode` class, since the `ListNode` class implements the `Comparable` interface by simply invoking the underlying interface of the `Comparable` object that is associated with the `ListNode` object.

2.2.8 The `PrimeNumberSequence` Class

The `PrimeNumberSequence` class implements the `Sequence` interface by providing `sequence()` methods that generate enumerations of `Int` objects representing prime numbers. The default no-argument `sequence()` method begins at the initial prime number of 2. As each successive call to the `nextElement()` method of the generated `Enumeration` object is made, the next prime number is determined via the `isPrime()` method of the `Int` class.[25]

25. Refer to the project website for more information regarding the `isPrime()` method of the `Int` class.

2.2.9 The `RandomIntSequence` Class

The `RandomIntSequence` class implements the `Sequence` interface by providing `sequence()` methods that generate enumerations of pseudo-random `Int` objects. The implemented `sequence()` methods make use of a simple linear congruential method in which the previously generated pseudo-random number is used to calculate the new pseudo-random number. To begin this recursive process, an initial *seed* value is needed.[26]

The default no-argument `sequence()` method contains an arbitrary default seed value of 25; the single-argument `sequence()` method allows the user to specify the seed value explicitly as an `Int` object.

By no means is this class appropriate for generating reliable random numbers in an application program. The basic linear congruential method provides a sequence that will typically repeat itself after only a small set of numbers is generated. A much more extensive algorithm would need to be designed and implemented to replace the simple linear congruential method that is currently used. As mentioned previously, this class has been implemented more as a short excursion in Computer Science rather than as a fundamental class component.

2.2.10 The `Table` Class

The `Table` class provides a generic associative hashtable by implementing the `AssociativeContainer` interface. Objects are identified by a *unique* key object that is also used in determining the corresponding hash values. The current implementation of the hashtable consists of an array of references to `LinkedList` objects. When a new object is added to the hashtable, it is added to either an existing `LinkedList` object or to a newly constructed `LinkedList` object. In other words, all `LinkedList` references are initially `null`; each `LinkedList` object is instantiated only when necessary.

The following table provides a description of each method of the `Table` class. Note that the worst-case run-time performance using “Big-Oh” notation is also specified for each method.

Method Name	**Order**	**Method Description**
`add()`	`O(n)`	Adds a new associative entry to the hashtable. This method performs in linear time in the worst-case due to the detection of a duplicate entry (specifically, the `contains()` method of the `LinkedList` class). Performance is substantially better in the average case, provided that the hash function provides an even distribution of hash values.
`clear()`	`O(1)`	Removes all elements from the hashtable.

26. Note that the implemented `sequence()` method is not recursive; rather, the formula that is used to calculate the next pseudo-random number is recursive.

Method Name	**Order**	**Method Description**
`contains()`	`O(n)`	Tests whether the `Comparable` object argument exists in the hashtable by making use of the `map()` method that is described below. Although this method performs in linear time in the worst-case, performance is substantially better in the average case, provided that the hash function provides an even distribution of hash values.
`elements()`	`O(1)`	Provides an enumeration of the elements in the hashtable. The `nextElement()` method of the instantiated `Enumeration` object returns the next object in the hashtable. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(n)**. A second version of the `elements()` method provides an enumeration of the elements in a specific *bucket* of the hashtable. This second version of the `elements()` method takes a single argument that is the hash value to use. As noted earlier, this method may make use of the `EmptyEnumeration` class if necessary.
`getSize()`	`O(1)`	Returns the number of elements in the hashtable.
`getTableSize()`	`O(1)`	Returns the size of the array used to represent the hashtable (i.e. the number of available hash buckets).
`isEmpty()`	`O(1)`	Tests whether the hashtable contains any elements.
`keys()`	`O(1)`	Provides an enumeration of the keys in the hashtable. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(n)**.
`map()`	`O(n)`	Provides a mechanism for mapping a given key to its corresponding data. This method performs in linear time in the worst case; however, inherent in hashing, the average performance is substantially better since the linked list used to store the data should be very small with an evenly distributed hash function.
`remove()`	`O(n)`	Removes and returns a reference to the object identified by the given key argument. Note that a `null` reference is returned if the key does not map to an object in the hashtable. Although this method performs in linear time in the worst-case, performance is substantially better in the average case, provided that the hash function provides an even distribution of hash values.
`toString()`	`O(n)`	Overrides the `toString()` method of the fundamental `java.lang.Object` class to provide a mechanism for displaying all key/data pairs of the hashtable.

3 The Generic Graph Package

3.1 An Overview of the Generic Graph Package

Building on the generic containers and interfaces of the `rpi.goldsd.container` package, the `rpi.goldsd.graph` package provides a flexible and extensible graph container. Both undirected and directed graphs, as well as unweighted and weighted graphs may be represented using this package. A wide variety of graph-related components and algorithms are also included in the `rpi.goldsd.graph` package. For example, the `Vertex`, `Edge`, and `DirectedEdge` classes are the fundamental building blocks of the `Graph` and `Tree` container classes of this package.

Although the standard Java `Enumeration` interface is somewhat limited, a number of instance methods provide an `Enumeration` object to iterate through sets of vertices and edges. Further, multiple instances of the `Path` class may be associated with a `Graph` or `Tree` instance; the `Path` class provides a means of iterating through a graph across an alternating series of `Vertex` and `Edge` objects.

3.1.1 Graph-Related Algorithms

The static `Algorithms` class provides a set of fundamental graph-related algorithms. Static methods of the `Algorithms` class include: `breadthFirstTraversal()` to generate a breadth-first search tree; `depthFirstTraversal()` to generate a depth-first search tree; `complement()` to generate the complement of a graph; `findBicomponents()` to determine biconnected components of a graph; `findEulerPath()` to find an Euler Path in a given graph; `findShortestPath()` to determine the shortest path between two vertices using Dijkstra's Shortest Path algorithm; and `findLongestPath()` and `findLongestPath2()` to attempt to find the longest path between two vertices[27] using a simple greedy algorithm and a more complicated method called *stratified greed*, respectively.[28]

3.1.2 Sharing Vertices and Edges Amongst Multiple Graphs

In general, vertices and edges are constructed before being associated with a specific graph object. Thus, there is a separation of vertex and edge creation and the inclusion of such objects in a graph. Such a separation lends itself nicely to a sharing of vertex and edge objects amongst multiple graphs; however, the `rpi.goldsd.graph` package does *not* support such sharing. Sharing vertices and edges amongst multiple graphs is not necessary, since the set of multiple graphs may be created as a single graph with numerous walks or paths defined representing subgraphs. Further,

27. Determining the longest path between two vertices of a graph is an NP-complete problem; the longest path methods of the static `Algorithms` class find paths that *approximate* the longest path.
28. Methods of the static `Algorithms` class are described in detail in §3.10.

there may be problems involving speed and efficiency when manipulating graphs, due to the overhead involved in maintaining graph, vertex, and edge objects in which sharing occurs.

If applied incorrectly, the application developer may alter the vertices and edges of a graph and create adverse side-effects in an unrelated graph that shares some of the altered vertices and edges. Although the consistency is maintained internally and would not be a responsibility of the application developer, the side-effect problem described may occur and surprise the unwary application developer.

To further evaluate the sharing concept, consider a simpler data structure: the linked list. Would it make sense to share nodes in multiple linked lists? Depending on the purpose of the multiple linked lists, the same potential for unpredictable side-effects exists if a set of nodes is altered in one of the linked lists. In other words, changing the node in one linked list may result in an immediate change to other linked lists in which the altered node also exists. Although it depends on the problem domain, this is probably not a desirable feature.

Next, consider the Java `String` constants that are defined in the following block of code:

```
String S1 = “This is a simple String constant.”;
String S2 = “This is a simple String constant.”;

System.out.println( S1 );
System.out.println( S2 );
System.out.println( S1 + “  ” + S2 );
```

The Java compilers generally attempt to share as many `String` constants as possible in an effort to reduce the size of the resulting *byte-code*. Thus in the above code segment, `S1` and `S2` probably refer to the same memory location. This could be extremely dangerous if one of the `String` constants is subsequently changed; however, since `String` objects may not be altered in Java as they may be in C or C++, the potential for corruption does not exist in Java. In short, sharing `String` constants is acceptable and safe since the shared `String` objects are constant and immutable.

Returning to the graph problem domain, vertices and edges are *not* immutable objects, and therefore it is safer and more robust to *not* allow the sharing of such objects amongst multiple graph objects.

3.1.3 A Summary of the Graph Package Classes

The following table contains a description of the fundamental classes of the `rpi.goldsd.graph` package. Also included for each class is the class or classes that each class *extends* (if any), and the interface or interfaces that each class *implements* (if any):

Class Name	**`extends`**	**`implements`**	**Class Description**
`Algorithms`	-	-	The static `Algorithms` class provides a set of static algorithmic methods that operate on graphs and other graph-related objects.
`DirectedEdge`	`Edge`	-	The `DirectedEdge` class represents a directed edge object that is associated with either zero or one graph objects.
`Edge`	-	`Hashable`	The `Edge` class represents an undirected edge object that is associated with either zero or one graph objects.
`Graph`	`GraphBase`	-	The `Graph` class represents a graph object that contains zero or more `Vertex` objects and zero or more `Edge` or `DirectedEdge` objects. A `Graph` object may not contain a mix of `Edge` and `DirectedEdge` objects, nor may a `Graph` object contain a mix of weighted and unweighted edges.
`GraphBase`	-	`Cloneable`, `Drawable`	The abstract `GraphBase` class provides many of the fundamental methods of the graph container. The `Graph` and `Tree` classes are subclasses of the abstract `GraphBase` class.
`Path`	-	`Cloneable`, `Drawable`	The `Path` class is essentially an iterator class that is used to represent a path or walk through a graph object.
`Tree`	`GraphBase`	-	The `Tree` class represents a graph object that contains *one* or more `Vertex` objects and zero or more `Edge` or `DirectedEdge` objects. As with the `Graph` class, a `Tree` object may not contain a mix of `Edge` and `DirectedEdge` objects, nor may it contain a mix of weighted and unweighted edges. The `Tree` class is a graph container that must contain a *root* `Vertex` object, and must be a *connected*, *acyclic* graph. Multiple `Tree` objects may make up a *forest*, which may be implemented as a linked list (or other such container) of `Tree` object references.
`Vertex`	-	`Cloneable`, `Hashable`	The `Vertex` class represents a vertex object that is associated with either zero or one graph objects.

As with the previous descriptions of the `rpi.goldsd.container` package, the sections that follow regarding the `rpi.goldsd.graph` package are not meant to be used as a comprehensive reference, but rather as a discussion of how the classes were implemented and why many of the design decisions were made.[29]

3.2 Exception and Error Classes of the Graph Package

Before discussing the fundamental classes of the `rpi.goldsd.graph` package, a brief description of each of the exception and error classes of the package is presented in the table that follows:

Exception or Error Class Name	**`extends`**	**Exception or Error Class Description**
`EdgeNotFoundException`	`RuntimeException`	The `EdgeNotFoundException` class represents an exception that should be thrown when a method fails because a required edge was not found. As an example, the standard `remove()` method of the `Graph` class may throw this exception if the given `Edge` or `DirectedEdge` argument is not in the graph object.
`InAnotherGraphException`	`RuntimeException`	The `InAnotherGraphException` class represents an exception that should be thrown when a vertex or edge is in a given graph, yet the method attempts to associate the vertex or edge with a different graph. As an example, the `add()` method of the `Graph` class may throw this exception if the given vertex of edge object being inserted already exists as part of another graph.
`InternalGraphError`	`Error`	The `InternalGraphError` class is used to provide a mechanism for the graph package methods to abort program execution due to an error condition that is not represented by another exception class.
`PathDoesNotExistException`	`RuntimeException`	The `PathDoesNotExistException` class represents an exception that should be thrown when a method fails because a path does not exist. Examples of methods that throw this exception include `findShortestPath()` and `findLongestPath()`.
`VertexNotFoundException`	`RuntimeException`	The `VertexNotFoundException` class represents an exception that should be thrown when a method fails because a required vertex was not found. For example, the standard `remove()` method of the `Graph` class may throw this exception if the given `Vertex` argument is not in the graph object.

29. For a comprehensive reference of the interfaces and classes of the `rpi.goldsd.graph` package, refer to the generated `javadoc` web pages at `http://www.cs.rpi.edu/projects/pb/jgb/jdoc`.

3.3 The `Vertex` Class

The `Vertex` class represents a vertex object that is associated with either zero or one graph objects. Every instance of the `Vertex` class must be associated with a `Hashable` object that will be used as a *key* when the vertex is associated with a graph.[30] Each instance of the `Vertex` class also contains a reference to the associated graph object (if the vertex is part of a graph), a `Vector` of incident edges, a `Vector` of incoming edges, and numerous utility fields used by the static `Algorithms` class.

For undirected graphs, the `Vector` of incoming edges is not used; however, for digraphs, the `Vector` of incoming edges contains `DirectedEdge` objects that lead to the given `Vertex` instance, and the `Vector` of incident edges contains only outgoing `DirectedEdge` objects. This consistency is maintained via the mutator methods of the `GraphBase`, `Graph`, and `Tree` classes.

The associated graph reference, the incident and incoming edge `Vector` references, the `Hashable` object, and the utility fields are all `protected` data fields. Therefore, class users cannot directly modify these attributes. The only mutator method that is `public` is the `setData()` method that is used to change the `Hashable` object associated with the given vertex.[31]

3.3.1 Constructors of the `Vertex` Class

As with all instance classes of the `rpi.goldsd.graph` package, a basic set of constructors have been designed and implemented to provide multiple means of construction. The default no-argument constructor may be used to create an instance of the `Vertex` class with an automatically assigned `Hashable` key value (specifically, this is an instance of the `Int` class). Aside from the default no-argument constructor, the `Vertex` class also provides a constructor that allows the user to explicitly specify the `Hashable` object to be associated with the `Vertex` class instance.

The `Vertex` class implements the `Cloneable` interface; a *copy constructor* is also available that performs just as the `clone()` method does. A cloned or copied `Vertex` object only clones the original vertex's `Hashable` data object. The `Vertex` object created via the `clone()` method or via the copy constructor does not contain any incident or incoming edges and is not associated with any graph. Currently, the cloning process does not copy the utility fields of the `Vertex` class.[32]

30. As described in detail in §3.6, a graph object consists of a hashtable of `Vertex` objects and a hashtable of `Edge` or `DirectedEdge` objects, thus the need for a unique `Hashable` object exists.
31. In the future, the utility fields should probably be changed to be `public` such that users may utilize these fields in creating their own generic graph-related algorithms.
32. Along the same lines as the previous footnote, the utility fields should probably be copied during the cloning process.

3.3.2 Accessor Methods of the `Vertex` Class

The following table provides a brief description of each of the accessor methods of the `Vertex` class. Also included is the worst-case run-time performance using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`data()`	`O(1)`	Returns a reference to the `Hashable` object that is associated with the given `Vertex` instance.
`degree()`	`O(1)`	Returns the *degree* of the given `Vertex` instance. For a vertex in an undirected graph, this method returns the number of incident edges; for a vertex in a digraph, this method returns the sum of the number of outgoing incident edges and the number of incoming incident edges.
`graph()`	`O(1)`	Returns a reference to the `Graph` object that contains the given `Vertex` instance.
`hash()`	`O(1)`	Since the `Vertex` class implements the `Hashable` interface, two `hash()` methods are required. These methods simply call the corresponding `hash()` methods of the underlying `Hashable` data that is associated with the given `Vertex` instance.
`incidentEdges()`	`O(1)`	Returns an enumeration of the edges incident to the given `Vertex` instance. For a vertex in an undirected graph, this method returns an enumeration of incident edges; for a vertex in a digraph, this method returns an enumeration of outgoing incident edges.
`incomingEdges()`	`O(1)`	Returns an enumeration of the directed edges that are directed to the given `Vertex` instance. For an undirected graph, this method returns an empty enumeration.
`inDegree()`	`O(1)`	Returns the *in-degree* of the given `Vertex` instance. For a vertex in an undirected graph, this method returns the number of incident edges; for a vertex in a digraph, this method returns the number of incoming incident edges.
`isEqualTo()`	`O(1)`	Since the `Vertex` class implements the `Hashable` interface, methods of the `Comparable` interface must also be implemented. Thus, the `isEqualTo()` method simply calls the `isEqualTo()` method of the underlying `Hashable` data associated with the given `Vertex` instance.
`isIncident()`	`O(e)`	Tests whether the given `Edge` or `DirectedEdge` object is incident with the given `Vertex` instance. This method performs in linear time **O(e)**, where `e` represents the total number of edges that are incident with the given `Vertex` instance.
`isInGraph()`	`O(1)`	Tests whether the given `Vertex` instance is in either any graph or a specified graph.

Method Name	Order	Method Description
`isLessThan()`	`O(1)`	Since the `Vertex` class implements the `Hashable` interface, methods of the `Comparable` interface must also be implemented. Thus, the `isLessThan()` method calls the `isLessThan()` method of the underlying `Hashable` data that is associated with the given `Vertex` instance.
`numIncidentEdges()`	`O(1)`	Returns the number of edges that are incident with the given `Vertex` instance by simply calling the `degree()` method (see above description).
`outDegree()`	`O(1)`	Returns the *out-degree* of the given `Vertex` instance. For a vertex in an undirected graph, this method returns the number of incident edges; for a vertex in a digraph, this method returns the number of outgoing incident edges.
`toString()`	`O(1)`	Returns a string representation of the given `Vertex` instance by simply calling the `toString()` method of the underlying `Hashable` data object that is associated with the given `Vertex` instance.

3.3.3 Mutator Methods of the `Vertex` Class

The following table provides a brief description of each of the mutator methods of the `Vertex` class. Also included is the worst-case run-time performance using “Big-Oh” notation. Note that all methods are dynamic instance methods, and that all methods are `protected` except for the `setData()` method, which is defined as `public`. Thus, the class user may not alter the graph-related state of the `Vertex` via methods of the `Vertex` class itself.

Method Name	Order	Method Description
`addIncidentEdge()`	`O(1)`[a]	Adds an incident edge to the given `Vertex` instance.
`addIncomingEdge()`	`O(1)`[a]	Adds an incoming directed edge to the given `Vertex` instance.
`removeAllIncidentEdges()`	`O(1)`	Removes all incident edges associated with the given `Vertex` instance.
`removeAllIncomingEdges()`	`O(1)`	Removes all incoming edges associated with the given `Vertex` instance.
`removeIncidentEdge()`	`O(e)`	Finds and removes an incident edge from the given `Vertex` instance. In the worst-case, this method performs in linear time **O(e)**, where `e` represents the number of `Edge` objects that are incident with the given `Vertex` instance.
`removeIncomingEdge()`	`O(f)`	Finds and removes an incoming directed edge from the given `Vertex` instance. This method performs in linear time **O(f)** in the worst-case, where `f` represents the number of incoming `DirectedEdge` objects that lead to the given `Vertex` instance.

Method Name	Order	Method Description
`setData()`	`O(1)`[b]	Sets the `Hashable` object associated with the given `Vertex` instance to the given `Hashable` argument.
`setInGraph()`	`O(1)`	Sets the associated graph reference for the given `Vertex` instance to show that the given `Vertex` instance is part of a specific `GraphBase` object.

a. Since the underlying data structure is a Java `Vector`, it is possible that this method may not perform in constant time if the capacity of the `Vector` needs to be increased.

b. If the hash value of the `Hashable` object changes and the given `Vertex` instance is associated with a graph, the given `Vertex` instance will have to be removed from and then added back to the graph's underlying hashtable of `Vertex` objects. In such an instance, the performance will not be constant.

3.4 The `Edge` Class

The `Edge` class represents an undirected edge object that is associated with either zero or one graph objects. Directed edges may be represented via the `DirectedEdge` class, a subclass of the `Edge` class. Since the `DirectedEdge` class inherits all attributes and methods of the base `Edge` class, most of the discussion regarding the `Edge` class also pertains to the `DirectedEdge` class.[33]

Like the `Vertex` class, every instance of the `Edge` class must be associated with a `Hashable` object that will used as a *key* when the edge is added to a graph.[34] Each instance of the `Edge` class must also be associated with two `Vertex` objects that are the endpoint vertices of the edge. Note that these two `Vertex` object references may refer to the same `Vertex` object; in other words, self-loops are allowed. The two `Vertex` object references are `protected` attributes of the `Edge` class and are called `startVertex` and `endVertex`.

For instances of the `Edge` class, the order of the two endpoint vertices is arbitrary; i.e., use of the `startVertex` attribute as opposed to the `endVertex` attribute is arbitrary. For instances of the `DirectedEdge` class, the `startVertex` attribute refers to the *tail* of the directed edge, and the `endVertex` attribute refers to the *head* of the directed edge.

Both directed and undirected edges may have a weight or cost associated with them. The `weight` attribute is a `protected` attribute of type `double`. An edge is constructed as either a weighted or an unweighted edge. Once created, an edge may not change from being weighted to unweighted, or vice versa; however, during the lifetime of a weighted edge object, the value of `weight` may be modified via the `setWeight()` method.[35]

33. The `DirectedEdge` class is described in §3.5.
34. As described in detail in §3.6, a graph object consists of a hashtable of `Vertex` objects and a hashtable of either `Edge` or `DirectedEdge` objects.
35. As described in §3.6, a graph object cannot contain a mix of weighted and unweighted edges.

3.4.1 Constructors of the `Edge` Class

Numerous constructors are available for creating instances of the `Edge` class. Since each instance of the `Edge` class must be associated with two endpoint `Vertex` objects, the default no-argument constructor is modified with the `protected` keyword such that it is not available to the class user. The first two arguments of all `public` constructors are the two endpoint `Vertex` object references (i.e., `startVertex` and `endVertex`).

As with the `Vertex` class, an instance of the `Edge` class may be constructed with an automatically assigned `Hashable` key value (specifically, this is an instance of the `Int` class), or this `Hashable` key may be explicitly specified as an argument of the constructor. As mentioned above, the weight or cost of the constructed edge may also be specified in the constructor.

The `Edge` class does *not* implement the `Cloneable` interface. Similar to the cloning process of the `Vertex` class, the cloning process of an `Edge` object would not involve copying the associated graph reference, or the two endpoint `Vertex` references. Using this approach, cloning an `Edge` object is not feasible since two `Vertex` objects must invariably be associated with every `Edge` object instance.[36]

3.4.2 Accessor Methods of the `Edge` Class

The following table provides a description of each of the accessor methods of the `Edge` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`data()`	`O(1)`	Returns a reference to the `Hashable` object that is associated with the given `Edge` instance.
`endVertex()`	`O(1)`	Returns a reference to one of the two endpoint `Vertex` objects (specifically, the `endVertex` attribute).
`graph()`	`O(1)`	Returns a reference to the graph object that contains the given `Edge` instance.
`hash()`	`O(1)`	Since the `Edge` class implements the `Hashable` interface, two `hash()` methods are required. These methods simply call the corresponding `hash()` methods of the underlying `Hashable` data object that is associated with the given `Edge` instance.
`isEqualTo()`	`O(1)`	Since the `Edge` class implements the `Hashable` interface, the methods of the `Comparable` interface must also be implemented. Thus, the `isEqualTo()` method simply calls the `isEqualTo()` method of the underlying `Hashable` data associated with the given `Edge` instance.

36. Refer to §3.10.1.1 for a description of how to correctly copy an `Edge` or `DirectedEdge` object.

Method Name	**Order**	**Method Description**
`isInGraph()`	`O(1)`	Tests whether the given `Edge` instance is in either any graph or a specified graph.
`isLessThan()`	`O(1)`	Since the `Edge` class implements the `Hashable` interface, the methods of the `Comparable` interface must also be implemented. Thus, the `isLessThan()` method calls the `isLessThan()` method of the underlying `Hashable` data that is associated with the given `Edge` instance.
`isSelfLoop()`	`O(1)`	Tests whether the given `Edge` instance is a *self-loop* edge in which both endpoint vertices refer the same `Vertex` instance.
`isWeighted()`	`O(1)`	Tests whether the given `Edge` instance is a weighted edge.
`startVertex()`	`O(1)`	Returns a reference to one of the two endpoint `Vertex` objects (specifically, the `startVertex` attribute).
`toString()`	`O(1)`	Returns a string representation of the given `Edge` instance by simply calling the `toString()` method of the underlying `Hashable` data object that is associated with the given `Edge` instance.
`traverseFrom()`	`O(1)`	Traverses the given `Edge` instance starting from a given `Vertex` reference. The return value is therefore a reference to the other endpoint `Vertex` object, or `null` if the given `Vertex` reference is not an endpoint vertex of the given `Edge` instance.
`weight()`	`O(1)`	Returns the weight or cost associated with the given `Edge` instance.

3.4.3 Mutator Methods of the `Edge` Class

The following table provides a brief description of each of the mutator methods of the `Edge` class. Also included is the worst-case run-time performance of each method using “Big-Oh” notation. Note that all methods are `public` instance methods, except for the `setInGraph()` method, which is defined as `protected`.

Method Name	**Order**	**Method Description**
`setData()`	`O(1)`[a]	Sets the `Hashable` object associated with the given `Edge` instance to the given `Object` argument.
`setInGraph()`	`O(1)`	Sets the associated graph reference for the given `Edge` instance to show that the given `Edge` instance is part of a specific `GraphBase` object.
`setWeight()`	`O(1)`	Sets the weight or cost of the given `Edge` instance.

a. If the hash value of the `Hashable` object changes and the given `Edge` instance is associated with a graph, the given `Edge` instance will have to be removed from and then added back to the graph’s underlying hashtable of `Edge` objects. In such an instance, the performance will not be constant.

3.5 The `DirectedEdge` Class

As a subclass of the `Edge` class, the `DirectedEdge` class represents a directed edge object that is associated with zero or one graph objects. This section describes the additional and overridden methods that the `DirectedEdge` class provides; refer to §3.4 for a full description of the parent `Edge` class before proceeding with this section.

3.5.1 Accessor Methods of the `DirectedEdge` Class

The following table provides a description of each of the accessor methods of the `DirectedEdge` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`head()`	`O(1)`	Returns a reference to the `Vertex` object at the head of the directed edge (i.e. the vertex the directed edge is pointing to or leads to). This method simply calls the `endVertex()` method of the base `Edge` class.
`tail()`	`O(1)`	Returns a reference to the `Vertex` object at the tail of the directed edge. This method simply calls the `startVertex()` method of the base `Edge` class.
`traverseFrom()`	`O(1)`	The `traverseFrom()` method of the parent `Edge` class is overridden in the `DirectedEdge` class to allow a traversal only from the tail of the directed edge to the head. Thus, this method returns `null` if a traversal is attempted from the `Vertex` object referenced by the head of the directed edge.

3.6 The Abstract `GraphBase` Class

The abstract `GraphBase` class provides many of the fundamental attributes and methods of the generic graph container. The `Graph` and `Tree` classes are non-abstract subclasses of the abstract `GraphBase` class.[37] The basic attributes that are provided in the abstract `GraphBase` class include the name of the graph object (i.e., a Java `String` object), a hashtable containing references to all `Vertex` objects in the graph, and a hashtable containing references to all `Edge` or `DirectedEdge` objects in the graph.

Since instances of the `Table` class of the `rpi.goldsd.container` package are used to represent the two hashtables of vertices and edges in a given graph, duplicate vertices and edges are not allowed. The hashtable implemented by the `Table` class provides an associative container based on *unique* keys. Thus, two `Vertex` objects of a graph cannot share the same unique key; this is also true of `Edge` and `DirectedEdge` object instances of a given graph.

37. The `Graph` and `Tree` classes are described in §3.7 and §3.8, respectively.

The methods of the abstract `GraphBase` class also enforce the following rules regarding edges:

- A graph object may contain only `Edge` or `DirectedEdge` objects, but not a mix of the two. In other words, a graph cannot contain a mix of undirected and directed edges.
- A graph object may contain only weighted or unweighted edges, but not a mix of the two. A class-user may interpret a weight of 0.0 to be an unweighted edge, if such a mix is deemed necessary.

3.6.1 Constructors of the Abstract `GraphBase` Class

A variety of constructors are available in the abstract `GraphBase` class. Aside from the default no-argument constructor, constructors exist that allow the class-user to specify the name of the graph, as well as the overall sizes of the two hashtables that are associated with each graph object.

A generic cloning mechanism is also available via the standard `clone()` method and a standard copy constructor. Cloned graph objects are created by performing a *deep copy* of all vertices and edges of the source graph. In other words, all vertices and edges of the graph are individually cloned. Thus, the target graph and the source graph do not share any components after the cloning process is complete.

3.6.2 Accessor Methods of the Abstract `GraphBase` Class

The following table provides a brief summary of each of the accessor methods of the `GraphBase` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`contains()`	`O(v)` or `O(e)`	Tests if a given `Vertex` or `Edge` object exists in the given graph object. Although this method performs in linear time in the worst-case (where `e` represents the number of edges and `v` represents the number of vertices in the given graph object), performance is substantially better in the average case, provided that the given hash function provides an even distribution.
`edgeBetween()`	`O(v+f)`	Finds an edge of this graph that is between the two given `Vertex` arguments. The two vertices and the corresponding edge, if present in the graph, must be part of the given graph object for this method to return a valid `Edge` object. Otherwise, `null` is simply returned. In the worst case, the order of this method is **O(v+f)**, where `v` is the number of vertices in this graph and `f` is the number of edges that are incident with the two given `Vertex` object arguments. In the average case, the `v` term will be substantially smaller, provided that the given hash function used to store the `Vertex` objects is evenly distributed.

Method Name	**Order**	**Method Description**
`edges()`	`O(1)`	Provides an enumeration of the edges in the given graph object. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(e)**, where `e` is the number of edges in the given graph object.
`isDigraph()`	`O(1)`	Tests whether the given graph contains directed edges.
`isEdgeBetween()`	`O(v+f)`	Tests whether an edge exists between the two given `Vertex` object arguments by invoking the `edgeBetween()` method (see above description).
`isEmpty()`	`O(1)`	Tests whether the given graph contains any vertices.
`isWeighted()`	`O(1)`	Tests whether the given graph contains weighted edges.
`mapToEdge()`	`O(e)`	Maps the given key to an `Edge` object in the given graph, if such a mapping exists in the underlying hashtable of `Edge` objects. This method performs in linear time **O(e)** in the worst-case, where `e` is the number of edges in the given graph object; however, inherent in hashing, the average performance is substantially better when the hash function applied to the `Edge` objects provides a set of evenly distributed hash values.
`mapToVertex()`	`O(v)`	Maps the given key to a `Vertex` object in the given graph, if such a mapping exists in the underlying hashtable of `Vertex` objects. This method performs in linear time **O(v)** in the worst-case, where `v` is the number of vertices in the given graph object; however, as with the `mapToEdge()` method, performance is substantially better when the given hash function presents an evenly distributed set of hash buckets.
`name()`	`O(1)`	Returns the name associated with the given graph object.
`numEdges()`	`O(1)`	Returns the number of edges in the given graph object.
`numVertices()`	`O(1)`	Returns the number of vertices in the given graph object.
`printSummary()`	`O(v+e)`	Outputs information regarding the given graph object, including the name of the graph, the number of vertices and edges in the graph, and a full adjacency-list presentation with incident edge information and which vertices each incident edge leads to.
`toString()`	`O(1)`	Returns a string representation of the given graph object. Specifically, this method returns the name of the graph; for more detailed output, the `printSummary()` method should be used (see above description).
`vertices()`	`O(1)`	Provides an enumeration of the vertices in the given graph object. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(v)**, where `v` is the number of vertices in the given graph object.

3.6.3 Mutator Methods of the Abstract `GraphBase` Class

The following table provides a brief description of each of the mutator methods of the `GraphBase` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods, except for the standard `addBase()` method and the `rehash()` method, which are defined as `protected` methods.

Method Name	**Order**	**Method Description**
`addBase()`	`O(v+e)` or `O(v)`	Two versions of the overloaded `addBase()` method exist: one for adding a `Vertex` object to the given graph object, the other for adding an `Edge` object. Since this method is `protected`, the class-user cannot make use of this method directly; instead, classes that use the abstract `GraphBase` class as a base-class should provide `add()` methods that are `public` that make use of this `addBase()` method.[a] Adding an `Edge` object to the given graph occurs in **O(v+e)** in the worst-case, where `v` is the number of vertices in the graph and `e` is the number of edges. As with many of the methods involving hashtables, the average performance is substantially better when the given hash functions that are applied to the `Vertex` and `Edge` objects present evenly distributed sets of hash buckets. Adding a `Vertex` object to the given graph object occurs in linear time **O(v)** in the worst-case, where `v` is the number of vertices in the graph.[b] The average performance is substantially better when the hash function that is applied to the `Vertex` objects presents an evenly distributed set of hash buckets.
`clear()`	`O(v+e)`	Removes all vertices and edges from the given graph object. Clearing the underlying hashtables occurs in constant time **O(1)**; however, each `Vertex` and `Edge` object must be modified to show that it is no longer part of a graph object. Thus, the overall running time is **O(v+e)**, where `v` is the number of vertices in the given graph object, and `e` is the number of edges.
`rehash()`	`O(n)`	Rehashes an existing `Edge` or `Vertex` object in the given graph object. This method is `protected` and is currently only invoked from the `setData()` methods of the `Vertex` and `Edge` classes. This method performs in linear time based on the `remove()` and `add()` methods of the `Table` class.
`setName()`	`O(1)`	Sets the name of the given graph object.

a. Refer to §3.7.3 and §3.8.2 for a description of the `add()` methods of the `Graph` and `Tree` subclass, respectively.

b. Adding a `Vertex` object to a graph occurs in linear time as opposed to constant time due to the duplication detection process that is performed by the `add()` method of the `Table` class. Refer to §2.2.10.

3.7 The `Graph` Class

The `Graph` class represents a graph object that contains zero or more `Vertex` objects and zero or more `Edge` or `DirectedEdge` objects. Since the `Graph` class is a subclass of the abstract `GraphBase` class, refer back to §3.6 for a full description of the `GraphBase` class before proceeding with this section.

3.7.1 Constructor Methods of the `Graph` Class (Constructing Random Graphs)

In addition to the fundamental set of constructors that are inherited from the abstract `GraphBase` class, the `Graph` class contains a series of constructors for creating *random graphs*. Each of these constructors requires an `int` argument specifying the number of vertices the constructed graph is to contain, as well as a `double` argument indicating the probability of an undirected edge existing between each pair of distinct vertices.[38] Given such argument requirements, constructing a random graph may be done with the following Java statement:

```
Graph G1 = new Graph( “Random Graph”, 80, 0.25 );
```

In this first example, a `Graph` object named “Random Graph” is created with 80 vertices. For every distinct pair of vertices from the set of 80, there is a 25% chance that an edge is constructed and added to the graph.

Constructing a random graph of weighted edges is also possible via another `Graph` class constructor. In addition to the arguments of the previous example, two `double` values indicating the inclusive range of weight values are also expected. Consider the following Java statement:

```
Graph G2 = new Graph( “Random Weighted Graph”, 80, 0.25, 1.0, 25.0 );
```

As in graph `G1`, the `G2` graph is created with 80 vertices and a 25% probability of an edge existing between each distinct pair of vertices. For every edge that is constructed and added to this graph, a random weight in the inclusive range [1.0, 25.0] is generated and assigned. Thus, the resulting graph is a weighted, undirected graph of 80 vertices with weights in the range [1.0, 25.0].

In graphs `G1` and `G2`, the `Vertex` and `Edge` objects that are constructed make use of the default `Vertex` and `Edge` class constructors. Thus, the `Hashable` key that is associated with each graph component is generated from a default `IntSequence` object. It is also possible to construct random graphs with specific `Sequence` objects for generating the unique key values for the constructed vertices and edges.[39]

38. Randomly generated graphs do not currently contain directed edges or edges that are self-loops.
39. Refer to §2.1.5 and §2.2.5 for more information on the `Sequence` interface and the `IntSequence` class, respectively.

Consider the following Java statements:

```
AlphabetSequence S1 = new AlphabetSequence();
PrimeNumberSequence S2 = new PrimeNumberSequence();
Graph G3 = new Graph( "Random Graph 3", 26, 0.15, S1, S2 );
```

In this third example, graph `G3` is constructed by creating 26 vertices, uniquely identified by a sequence of `Hashable` objects (in this case, `Char` objects) that are provided by instance `S1` of the `AlphabetSequence` class. For every distinct pair of vertices, an edge is constructed and added to the graph 15% of the time. The `Edge` objects that are constructed and added to the graph are uniquely identified by a sequence of `Hashable` objects (in this case, `Int` objects) that are provided by instance `S2` of the `PrimeNumberSequence` class.[40]

3.7.2 Accessor Methods of the `Graph` Class

The following table provides a brief description of each of the accessor methods of the `Graph` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`isClique()`	$O(v^2)$	Tests if the given graph object is a *clique* by constructing the complement of the given graph and checking whether the number of edges in the constructed complement-graph is zero. A *clique* or *complete graph* is a graph in which there is an edge between every distinct pair of vertices.
`isComplete()`	$O(v^2)$	Tests whether the given graph object is a *complete* graph by invoking the `isClique()` method (see above description).
`isConnected()`	`O(v+e)`	Tests whether the given graph object is a *connected* graph by first constructing a depth-first search (DFS) tree via the `depthFirstTraversal()` method of the static `Algorithms` class. If the constructed `Tree` object contains the same number of vertices as the given graph object, then the graph is indeed connected.[a] A *connected graph* is a graph in which there is a path between every pair of distinct vertices. When considering a digraph, a *strongly connected digraph* is a connected digraph in which directed edges are traversed only from tail to head. For digraphs, the `isConnected()` method tests whether the digraph is a strongly connected graph.

a. Refer to §3.10.1.2 for a description of the `depthFirstTraversal()` method of the static `Algorithms` class; also refer to §3.8 for a description of the `Tree` class.

40. Refer to §2.2.2 and §2.2.8 for a discussion of the `AlphabetSequence` and `PrimeNumberSequence` classes, respectively. Also refer to §2.1.5 for a general discussion of the `Sequence` interface.

3.7.3 Mutator Methods of the `Graph` Class

The following table provides a brief description of each of the mutator methods of the `Graph` class. Also included is the worst-case run-time performance of each method using “Big-Oh” notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`add()`	`O(v+e)` or `O(v)`	Two versions of the overloaded `add()` method exist: one for adding a `Vertex` object to the given graph, the other for adding an `Edge` to the given graph. Both versions make use of the standard `addBase()` method of the base `GraphBase` class. The `add()` method used to add a `Vertex` object to the given graph performs in linear time **O(v)** in the worst-case, where `v` is the number of vertices in the graph. The `add()` method used to add an `Edge` object to the given graph performs in linear time **O(v+e)** in the worst-case, where `v` is the number of vertices in the graph and `e` is the number of edges in the graph.[a]
`remove()`	`O(v+e)`	Two versions of the overloaded `remove()` method exist: one for removing a specified `Vertex` object from the given graph object, the other for removing a specified `Edge` object.
`removeAllEdges()`	`O(v+e)`	Removes all edges from the given graph object. This method performs in linear time **O(v+e)** based on both `v`, the number of vertices in the given graph object, and `e`, the number of edges in the given graph.

a. Refer back to the description of the `addBase()` method in §3.6.3 for more details regarding the worst-case run-time performance measures.

3.8 The `Tree` Class

The `Tree` class represents a graph object that contains *one* or more `Vertex` objects and zero or more `Edge` or `DirectedEdge` objects. As with the `Graph` class, a `Tree` object may not contain a mix of `Edge` and `DirectedEdge` objects, nor may it contain a mix of weighted or unweighted edges. Since the `Tree` class is a subclass of the abstract `GraphBase` class, refer back to §3.6 for a full description of the `GraphBase` class before proceeding with this section.

The `Tree` class is a graph container that adheres to the following set of requirements:

- The graph must be rooted at a specific `Vertex` object.
- The graph must be *connected*.
- The graph must be *acyclic*.

Note that the `isEmpty()` method that is inherited from the abstract `GraphBase` class is useless when applied to the `Tree` class since this method will always return `true`, as there must be at least one vertex in every instance of the `Tree` class (i.e. the root vertex).

3.8.1 Accessor Methods of the `Tree` Class

The following table provides a brief description of each of the accessor methods of the `Tree` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`root()`	`O(1)`	Returns a reference to the `Vertex` object that is designated as the *root* of the given `Tree` object.

3.8.2 Mutator Methods of the `Tree` Class

The following table provides a brief description of each of the mutator methods of the `Tree` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`add()`	`O(v+e)`	Adds an `Edge` object and corresponding `Vertex` object to the given `Tree` object. To maintain the properties of a tree, an edge and corresponding vertex must be added as an atomic operation. Adding an unattached `Vertex` object to a `Tree` object is not allowed. The given `Edge` object to be added must contain an end-point `Vertex` object that is part of the given `Tree`. This method performs in linear time **O(v+e)** in the worst-case, where `v` is the number of vertices in the given tree and `e` is the number of edges in the tree. Fundamentally, this method calls the `addBase()` method of the base `GraphBase` class twice: once for the added edge, and once for the added vertex.
`clear()`	`O(v+e)`	This method overrides the `clear()` method of the abstract `GraphBase` class to provide a method that removes all vertices and edges from the given `Tree` object, *except for the root vertex*. This method essentially calls the corresponding `clear()` method of the base `GraphBase` class, thus this method performs in linear time **O(v+e)**, where `v` represents the number of vertices in the given tree and `e` represents the number of edges in the tree.

3.9 The `Path` Class

The `Path` class is essentially an iterator class used to represent a path or walk through a graph object. Since the `Path` class is essentially an iterator, new `Vertex` and `Edge` objects are *not* created when working with methods of the `Path` class. A *path* is defined as a series of alternating vertices and edges of a single graph or digraph, and is represented in the `Path` class via a Java `Vector`. A path must begin and end with a `Vertex` object, thus the size of the underlying Java `Vector` should always be odd, unless the path contains no vertices or edges.

As an example of a `Path` class instance, consider the following series of alternating `Vertex` and `Edge` object references:

```
V1 -- E1 -- V2 -- E2 -- V3 -- E3 -- V4 -- E4 -- V5 -- E5 -- V6 -- E6 -- V7
```

In this example, the path begins with vertex `V1` and ends with vertex `V7`. Note that vertices `V1` and `V7` may in fact refer to the same `Vertex` object in which case the path is called a *cycle*. Although the definition of a *path* in graph theory requires that no duplicate vertices are present, repeating `Vertex` references are supported by the `Path` class.

The *length* of a path is defined as the number of edges in the path. Thus the length of the given example path is 6.

3.9.1 Constructors of the `Path` Class

An instance of the `Path` class may be created via a number of constructors. The default no-argument constructor simply creates an empty path by creating a `Vector` class instance with default *capacity*.[41] A `Path` class constructor exists in which the `Vector` capacity may be explicitly specified when the number of vertices and edges that will be added to the path is known.

A `Path` class instance may also be constructed by specifying the `Vertex` object that begins the path. The `Path` class also implements the `Cloneable` interface and therefore implements the `clone()` method, as well as a copy constructor.

41. For more information on the standard Java `Vector` class and how the *capacity* value is utilized, refer to **[Flan97]** or other Java resources (including the source code of the `java.util.Vector` class).

3.9.2 Accessor Methods of the `Path` Class

The following table provides a brief description of each of the accessor methods of the `Path` class. Also included is the worst-case run-time performance of each method using "Big-Oh" notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`contains()`	`O(e)` or `O(v)`	Tests whether a given `Vertex` or `Edge` exists in the given `Path` object. Since the underlying data structure used to store the alternating series of vertices and edges is a Java `Vector`, both versions of the overloaded `contains()` method perform in linear time based on either the number of vertices or the number of edges in the given `Path` object.
`edges()`	`O(1)`	Provides an enumeration of the edges in the given `Path` object. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(e)**, where `e` is the number of edges in the given `Path` object.
`elements()`	`O(1)`	Provides an enumeration of the vertices and edges in the given `Path` object. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(v+e)**, where `e` is the number of edges in the given `Path` object, and `v` is the number of vertices.
`firstVertex()`	`O(1)`	Returns a reference to the first `Vertex` object in the given path.
`graph()`	`O(1)`	Returns a reference to the graph object that contains the vertices and edges of the given `Path` object.
`isCycle()`	`O(1)`	Tests whether the given `Path` object is a cycle. Specifically, this method returns `true` if the first and last vertices in the given path are the same (i.e. refer to the same `Vertex` object).
`isReachable()`	`O(v+e)`	Tests if the given end `Vertex` is *reachable* from the given start `Vertex` in the `Path` object. Note that if the path is a cycle, then if the two given vertices exist in the path, this method returns `true` immediately. This method performs in linear time **O(v+e)** in the worst-case, where `v` is the number of vertices in the given `Path` object, and `e` is the number of edges in the given `Path` object.
`lastVertex()`	`O(1)`	Returns a reference to the last `Vertex` object in the given path.
`length()`	`O(1)`	Returns the *length* of the given `Path` object, which is defined to be the number of edges in the path.
`name()`	`O(1)`	Returns the name of the graph that the given `Path` object is associated with.

Method Name	**Order**	**Method Description**
`occurrences()`	`O(e)` or `O(v)`	Two versions of this overloaded method exist: one for determining the number of times a specific `Edge` object appears in the given `Path` object, the other for determining the number of times a specific `Vertex` object appears in the given path. In the worst-case, this method performs in either **O(v)** or **O(e)**, where `v` is the number of vertices in the given `Path` object, and `e` is the number of edges in the given path.
`toString()`	`O(v)`	Returns a string representation of the given `Path` object by displaying each `Vertex` object in the path in a traversal from the first vertex to the last vertex.
`vertices()`	`O(1)`	Provides an enumeration of the vertices in the given `Path` object. This method performs in constant time; however, note that an algorithm making use of the resulting `Enumeration` will probably perform in **O(v)**, where `v` is the number of vertices in the given `Path` object.
`weight()`	`O(e)`	Calculates the overall *weight* of the given `Path` object by summing up all individual edge weights.

3.9.3 Mutator Methods of the `Path` Class

The following table provides a description of each of the mutator methods of the `Path` class. Also included is the worst-case run-time performance of each of the methods using “Big-Oh” notation. Note that all methods are `public` instance methods.

Method Name	**Order**	**Method Description**
`add()`	`O(1)`	Two versions of this overloaded method exist: one that takes an `Edge` object as an argument, and one that takes a `Vertex` object as an argument. Regardless of the specific method that is used, both an `Edge` and a `Vertex` object are added to the given path, unless the edge does not originate from the last vertex of the path.
`addStartVertex()`	`O(1)`	Initializes the given `Path` object and adds the given start `Vertex` as the first vertex of the path. This method removes all vertices and edges from the existing path, if any.
`backtrack()`	`O(1)`	Removes the last `Vertex` and `Edge` objects from the given `Path` object. Thus, this method essentially *backtracks* along the path by moving back to the second-to-last vertex in the path.
`concat()`	`O(m)`	Concatenates two `Path` objects together. Note that the last vertex of the given path must be the same as the first `Vertex` object of the given `Path` argument. This method performs in **O(m)**, where `m` is the number of edges and vertices in the path argument.
`insertAfter()`	`O(n+m)`	Inserts one `Path` object into another `Path` object after a specified `Vertex` object. This method performs in **O(n+m)**, where `n` is the number of edges and vertices in the original `Path` object, and `m` is the number of edges and vertices in the path being inserted.

3.10 The Static `Algorithms` Class

The static `Algorithms` class provides a set of algorithmic methods that operate on graphs and graph-related objects. The following table provides a brief description of each of the `public` methods. Also included is the worst-case run-time performance of each method using "Big-Oh" notation:

Method Name	**Order**	**Method Description**
`breadthFirstTraversal()`	`O(v+e)`	Constructs a `Tree` object that represents a breadth-first search (BFS) tree rooted at either a specified or arbitrary `Vertex` object. Refer to §3.10.1 and §3.10.1.1 for a description of this method.
`cloneEdge()`	`O(1)`	Constructs an `Edge` or `DirectedEdge` object by essentially cloning an existing `Edge` or `DirectedEdge` object. Since the `Edge` class does not implement the `Cloneable` interface, this static method is available to copy an existing edge, provided a new pair of endpoint `Vertex` objects are also given as arguments. Refer to §3.10.1.1 for more details regarding the `cloneEdge()` method.
`complement()`	$O(v^2)$	Constructs a `Graph` object that is the *complement* of a given `GraphBase` object. Refer to §3.10.2 for a description of the `complement()` method.
`depthFirstTraversal()`	`O(v+e)`	Constructs a `Tree` object that represents a depth-first search (DFS) tree rooted at either a specified or arbitrary `Vertex` object. Refer to §3.10.1 and §3.10.1.2 for a description of the `depthFirstTraversal()` method.
`displayBicomponents()`	`O(v+e)`	After invoking the `findBicomponents()` method, this method simply displays the *biconnected components* of the given `Graph` object. Refer to §3.10.4 for information regarding this method and the `findBicomponents()` method.
`findBicomponents()`	`O(v+e)`	Finds the *biconnected components* of the given `Graph` object by using a depth-first traversal and the utility fields of the `Vertex` class. Refer to §3.10.4 for a description of this method.
`findCycle()`	`O(v+e)`	Attempts to find a *cycle* in a given graph. This recursive method is used by the `findEulerPath()` method that is described next. Refer to §3.10.3 for details regarding the `findCycle()` and `findEulerPath()` methods.
`findEulerPath()`	`O(v+e)` `[Average]`	Attempts to find an *Euler Path* in a connected graph starting from either a specified or arbitrary `Vertex` object. Refer to §3.10.3 for a description of this method. Note that this method is identified as performing in **O(v+e)** in the average case rather than the worst-case.

Method Name	**Order**	**Method Description**
`findLongestPath()`	`O(v`^{2}`+ve)`	Using a simple greedy approach, this method attempts to approximate the longest path between two vertices. Refer to §3.10.6 and all of its subsections for detailed information regarding the `findLongestPath()` and more involved `findLongestPath2()` method.
`findLongestPath2()`	`O(v`^{2}`+ve)` `O(w`^{2}`)`	Using an algorithm involving *stratified greed*, this method attempts to approximate the longest path between two vertices. This method typically out-performs the original `findLongestPath()` method in terms of the path produced. Refer to §3.10.6 and all of its subsections for information regarding the `findLongestPath2()` method. Refer specifically to §3.10.6.4 for a description of how well this method performs in relation to `v`, the number of vertices in the graph, `e`, the number of edges in the graph, and `w`, the given *width* parameter.
`findMaxWeightEdge()`	`O(e)`	Performs a linear search through the edges of the given graph to find the edge with the maximum weight.
`findMinWeightEdge()`	`O(e)`	Performs a linear search through the edges of the given graph to find the edge with the minimum weight.
`findShortestPath()`	`O(v`^{2}`)`	Using Dijkstra's Shortest Path algorithm, this method constructs a `Path` object that represents the shortest path between two vertices. Refer to §3.10.5 for a description of the `findShortestPath()` method.
`toGraphDraw()`	`O(v+e)`	To view a `GraphBase` object using the *Graph Draw* applet, the `toGraphDraw()` method provides a mechanism for converting an existing graph to an `HTML` format suitable as input to the Graph Draw applet. Refer to §3.10.7 and §3.11 for more details regarding the `toGraphDraw()` methods and the `Drawable` interface, respectively.

As indicated in the given table, the following sections describe each of these methods in more detail.[42] In some cases, Java source code is also provided to demonstrate how the algorithm has been implemented, and at the same time show how flexible and powerful the components of the `rpi.goldsd.graph` package are. Since not all of the source code is included in this document, refer to the project website to obtain the complete set of source code.

42. Note that the `findMinWeightEdge()` and `findMaxWeightEdge()` are not described in a separate section of this document. Refer to the webpage associated with this project for more information regarding these utility methods.

3.10.1 Constructing Breadth-First and Depth-First Traversal Trees

The `breadthFirstTraversal()` and `depthFirstTraversal()` methods of the static `Algorithms` class both construct and return `Tree` objects. Both of these methods are overloaded such that each method expects either a `Graph` object or a specific `Vertex` object. When a `Graph` object is used as an argument, an arbitrary `Vertex` from the graph is chosen as the root of the `Tree` object to be constructed. When a specific `Vertex` object is explicitly used as an argument, this vertex is used as the root of the `Tree` object to be constructed.

Both the `breadthFirstTraversal()` and `depthFirstTraversal()` methods accept arbitrary `Graph` objects that may contain either directed or undirected edges. Weighted and unweighted graphs are also successfully processed by these methods. This flexibility is achieved primarily by making use of the `cloneEdge()` and `traverseFrom()` methods.[43]

3.10.1.1 Constructing a Breadth-First Search Tree

When invoked with a single `Graph` object as an argument, the `breadthFirstTraversal()` method chooses an arbitrary `Vertex` object as the root of the BFS Tree and calls the other overloaded version of the `breadthFirstTraversal()` method, using the chosen `Vertex` object as the single argument. If the given `Graph` object contains no vertices, the method returns `null`. The code is presented as follows:

```
public static final Tree breadthFirstTraversal( Graph G )
  throws IllegalArgumentException
{
   Enumeration e = G.vertices();
   if ( e.hasMoreElements() )
      return ( breadthFirstTraversal( (Vertex)e.nextElement() ) );
   else
      throw new IllegalArgumentException( “Graph contains no vertices.” );
}
```

Note that this method may throw an `IllegalArgumentException` if the graph contains no vertices or edges whatsoever.

43. A full description of the `cloneEdge()` method is provided in §3.10.1.1. Since the implementation of the `traverseFrom()` method differs between the base `Edge` class and the `DirectedEdge` class, refer back to §3.4.2 and §3.5.1 for information on this accessor method of the `Edge` and `DirectedEdge` classes, respectively.

The second version of the `breadthFirstTraversal()` method makes use of a *queue* construct to maintain an ordered list of `Vertex` objects whose *children* have not yet been added to the BFS Tree. The algorithm for constructing a BFS Tree is described as follows:[44]

```
(1) Add the chosen root Vertex to the BFS Tree.
(2) Add the root Vertex to the queue.
(3) While the queue is not empty, do steps (a) and (b):
    (a) Remove the next Vertex from the queue.
    (b) For each edge that is incident to the dequeued Vertex, do step (i):
        (i) If the destination Vertex of this edge is not in the BFS Tree, add this
            edge and destination Vertex to the BFS Tree.  Then add this destination
            Vertex to the queue such that its children will be processed.
```

All `Vertex` and `Edge` objects of the constructed `Tree` object are clones of their original counterparts from the original `Graph` object. Note that since `Edge` objects do not implement the `Cloneable` interface, the `clone()` method is not used to copy each edge; instead, a utility method that is also available in the static `Algorithms` class is used. The following `cloneEdge()` utility method may be used to construct `Edge` or `DirectedEdge` objects based on an original `Edge` or `DirectedEdge` object, as well as a new pair of vertex endpoints. The code is presented as follows:

```
public static final Edge cloneEdge( Vertex startVertex, Vertex endVertex,
                                    Edge originalEdge )
{
   Hashable data = originalEdge.data();
   Edge E = null;

   if ( originalEdge instanceof DirectedEdge )
   {
      if ( originalEdge.isWeighted() )
         E = new DirectedEdge( startVertex, endVertex, data,
                               originalEdge.weight() );
      else
         E = new DirectedEdge( startVertex, endVertex, data );
   }
   else  /* undirected edge */
   {
      if ( originalEdge.isWeighted() )
         E = new Edge( startVertex, endVertex, data, originalEdge.weight() );
      else
         E = new Edge( startVertex, endVertex, data );
   }

   return E;
}
```

44. Refer to **[Baas88]**, **[Corm92]**, **[Knut93]**, **[Sedg88]**, or just about any other book on algorithms for details on the breadth-first search and other algorithms of the static `Algorithms` class.

The implementation of the second `breadthFirstTraversal()` method makes use of the `cloneEdge()` utility method, and is presented as follows:

```
public static final Tree breadthFirstTraversal( Vertex startVertex )
{
   // Construct a new root Vertex by cloning the given startVertex ...
   Vertex root = (Vertex)startVertex.clone();

   // Construct a new Tree with the given root Vertex ...
   Tree T = new Tree( root, “BFS Tree” );

   // Construct a queue for the BFS algorithm and add the startVertex
   // and the cloned root Vertex to this queue ...
   LinkedList queue = new LinkedList();
   queue.addToTail( new ListNode( startVertex ) );
   queue.addToTail( new ListNode( root ) );

   // Iterate through all vertices ...
   while ( ! queue.isEmpty() )
   {
      // Remove the next graph and tree vertices from the queue ...
      Vertex graphV = (Vertex)queue.removeFromHead().getData();
      Vertex treeV = (Vertex)queue.removeFromHead().getData();

      // For each incident Edge and destination Vertex that is not yet
      // in the BFS Tree, add the Edge and Vertex to the BFS Tree ...
      Enumeration e = graphV.incidentEdges();
      while ( e.hasMoreElements() )
      {
         Edge graphE = (Edge)e.nextElement();
         Vertex graphW = graphE.traverseFrom( graphV );

         if ( ! T.contains( graphW ) )
         {
            Vertex treeW = (Vertex)graphW.clone();
            Edge treeE = cloneEdge( treeV, treeW, graphE );
            T.add( treeE, treeW );

            // Add new Vertex to the queue such that its children
            // are subsequently processed ...
            queue.addToTail( new ListNode( graphW ) );
            queue.addToTail( new ListNode( treeW ) );
         }
      }
   }

   return T;
}
```

3.10.1.2 Constructing a Depth-First Search Tree

When invoked with a `Graph` object as an argument, the `depthFirstTraversal()` method chooses an arbitrary `Vertex` object as the root of the DFS Tree and calls the other overloaded version of the `depthFirstTraversal()` method, using that `Vertex` object as the single argument. If the given `Graph` object contains no vertices, the method returns `null`. The code is almost identical to the corresponding `breadthFirstTraversal()` method and is presented as follows:

```
public static final Tree depthFirstTraversal( Graph G )
  throws IllegalArgumentException
{
   Enumeration e = G.vertices();
   if ( e.hasMoreElements() )
      return ( depthFirstTraversal( (Vertex)e.nextElement() ) );
   else
      throw new IllegalArgumentException( “Graph contains no vertices.” );
}
```

The second version of the `depthFirstTraversal()` method is also fairly concise. Since the depth-first search algorithm is inherently recursive, the implementation consists of a recursive *helper* method called `DFS()`. Thus, the second version of the `depthFirstTraversal()` method must simply start the recursive process by calling the `DFS()` method. The code is presented as follows:

```
public static final Tree depthFirstTraversal( Vertex startVertex )
{
   // Construct a new root Vertex by cloning the given startVertex ...
   Vertex root = (Vertex)startVertex.clone();

   // Construct a new Tree with the given root Vertex ...
   Tree T = new Tree( root, “DFS Tree” );

   // Recursively construct the DFS Tree via the DFS() helper method ...
   DFS( startVertex, root, T );

   return T;
}
```

The recursive `DFS()` method is also fairly concise (recursive methods typically result in concise and compact programs). Just as in the `breadthFirstTraversal()` method, the `DFS()` method makes use of the `cloneEdge()` utility method to copy the attributes of the graph’s `Edge` or `DirectedEdge` object when creating a new `Edge` or `DirectedEdge` object for the DFS Tree.

Since the `DFS()` method is a recursive helper method and not very useful to the average class-user, it is a `private` method, and therefore not available outside of the static `Algorithms` class. The code of the recursive `DFS()` method is presented as follows:

```
private static final void DFS( Vertex graphVertex, Vertex treeVertex, Tree T )
{
   // For each incident Edge and destination Vertex that is not yet
   // in the DFS Tree, add the Edge and Vertex to the DFS Tree.  Then
   // recursively call the DFS() method with the destination Vertex ...
   Enumeration e = graphVertex.incidentEdges();
   while ( e.hasMoreElements() )
   {
      Edge E = (Edge)e.nextElement();
      Vertex dest = E.traverseFrom( graphVertex );

      if ( ! T.contains( dest ) )
      {
         Vertex V = (Vertex)dest.clone();
         Edge newE = cloneEdge( treeVertex, V, E );
         T.add( newE, V );
         DFS( dest, V, T );
      }
   }
}
```

The `depthFirstTraversal()` method is actually used by the `isConnected()` method of the `Graph` class. This method determines if the given `Graph` object is a connected graph by constructing a depth-first search `Tree` object via the `depthFirstTraversal()` method. If the number of vertices in the given `Graph` object is equal to the number of vertices in the constructed `Tree` object, then the graph is considered connected.

3.10.2 Constructing the Complement of a Graph

The `complement()` method of the static `Algorithms` class constructs a new `Graph` object that is the *complement* of a given `GraphBase` object. The complement of an existing graph `H` is defined as a new unweighted graph `G` that is constructed as follows:

- All vertices of existing graph `H` are cloned and added to graph `G`. Thus, graphs `G` and `H` will always contain the same number of `Vertex` objects.
- A new `Edge` object is constructed between each pair of distinct vertices in graph `G`, if and only if no such edge exists in the original graph `H`. Note that edges that are self-loops are ignored in graph `H`. Further, edges that are self-loops do not appear in the constructed graph `G`.

The `isClique()` and `isComplete()` methods of the `Graph` class make use of the `complement()` method to determine if the given `Graph` object is a *clique* or *complete* graph. In general, the complement of a clique is a graph that contains no edges, thus the `isClique()` and `isComplete()` methods construct the complement-graph of the given `Graph` object and return `true` if the number of edges in the constructed complement-graph is zero.

Implementing the `complement()` method consists of iterating through all distinct pairs of `Vertex` objects in the original `GraphBase` object. For each pair of vertices, if an edge does not exist between the vertices, a new `Edge` object is added to the constructed complement `Graph` object.

One shortcoming of the `complement()` method is its current inability to handle digraphs. The current implementation of this method produces a complement-graph of an undirected graph only. The implementation of the `complement()` method is presented as follows:

```
public static final Graph complement( GraphBase H )
{
   Graph G = new Graph( “Complement of ” + H.name() );

   // Clone all vertices of the original GraphBase object ...
   Enumeration e = H.vertices();
   while ( e.hasMoreElements() )
   {
      Vertex j = (Vertex)e.nextElement();
      G.add( (Vertex)j.clone() );
   }

   // Iterate through each distinct pair of vertices of the
   // original GraphBase object H ...
   Enumeration v = H.vertices();
   while ( v.hasMoreElements() )
   {
      Vertex q = (Vertex)v.nextElement();

      Enumeration w = H.vertices();
      while ( w.hasMoreElements() )
      {
         Vertex r = (Vertex)w.nextElement();
         if ( q != r && ! H.isEdgeBetween( q, r ) )
         {
            // Since there is no Edge between Vertex q and Vertex r
            // of the original GraphBase object H, add a new Edge to
            // the complement Graph object G by first finding the
            // proper endpoint vertices in G ...
            Vertex qq = G.mapToVertex( q.data() );
            Vertex rr = G.mapToVertex( r.data() );
            if ( ! G.isEdgeBetween( qq, rr ) )
               G.add( new Edge( qq, rr ) );
         }
      }
   }

   // Return the constructed complement-graph ...
   return G;
}
```

3.10.3 Finding an Euler Path in a Graph

The `findEulerPath()` method of the static `Algorithms` class constructs a new `Path` object that represents an *Euler Path* through a given graph. An Euler Path (also called an *Euler Tour* or an *Eulerian Circuit*) is defined as a path through a given graph that contains each `Edge` or `DirectedEdge` object *exactly once*.[45] Further, the start and end vertices of the path must be the same vertex; i.e., the path must end up at the same vertex in which it began. Note that vertices of the given graph may appear in the constructed path multiple times, if necessary.

From the given definition, it should be clear that the resulting path contains at least one cycle, and is itself a cycle. More specifically, the generated `Path` object consists of a common start and end `Vertex` object (i.e., the `isCycle()` method will return `true` for all Euler `Path` objects). The generated path may also contain multiple cycles. A recursive utility method of the static `Algorithms` class is used by the `findEulerPath()` method to find the set of one or more cycles making up the Euler Path. This `findCycle()` method is presented as follows:

```
public static final boolean findCycle( Path P )
{
   // Extend the given Path by attempting to traverse an Edge that
   // is incident with the last Vertex of the given Path ...
   Enumeration e = P.lastVertex().incidentEdges();
   while ( e.hasMoreElements() )
   {
      // Add the next available UNSEEN Edge to the given Path ...
      Edge E = (Edge)e.nextElement();
      if ( E.status == UNSEEN )
      {
         P.add( E );
         E.status = IN_TREE_OR_PATH;

         // Recursively continue to add available Edge objects to the
         // given Path until hopefully a cycle is found ...
         if ( P.isCycle() || findCycle( P ) )
            return true;

         // If a cycle is not found, backtrack by removing the last
         // Edge object from the given Path and marking this Edge as
         // being UNSEEN again ...
         P.backtrack();
         E.status = UNSEEN;
      }
   }

   return false;
}
```

45. Refer to **[Evan92]** and **[Knut97]** for more information on finding Euler Paths in a given graph.

The `findCycle()` method presented above attempts to find a cycle in a given graph by recursively traversing available edges in a depth-first search fashion. If at any point, a cycle is found, the recursion is “unraveled” by simply returning a value of `true`. Note that the `backtrack()` method of the `Path` class is used to backtrack along a path that has hit a “dead end” in which a cycle has not been found and no more incident edges are available at that vertex.

Before the actual cycle determination process is started, however, a few necessary pre-conditions are verified at the onset of the `findEulerPath()` method. First, the `isConnected()` method of the `Graph` class is used to verify that the given graph is indeed a connected graph. By definition, a graph that is not connected does not contain an Euler Path.[46]

The second pre-condition that is verified is based on whether the graph is a directed or undirected graph. For directed graphs, the *in-degree* and *out-degree* values associated with each `Vertex` object must adhere to the following requirement:

- For each distinct vertex *V* of the given graph, the *in-degree* of *V* must be equal to the *out-degree* of *V*. In other words, each vertex must contain the same number of incoming edges as it does outgoing edges. A graph adhering to such a requirement is typically called a *balanced graph* or a *symmetric graph*.[47]

Similarly, for undirected graphs, the *degree* of each `Vertex` object must adhere to the following requirement:

- For each distinct vertex *V* of the given graph, the *degree* of *V* must be a positive even number. Thus, for each edge traversed that leads to *V*, there exists a second edge that leads out of *V*.

Although not proven in this document, the above requirements are both necessary and sufficient for a given graph to contain an Euler Path.[48] Intuitively, this conjecture should make sense. For each incoming edge of each `Vertex` object, there must be a corresponding outgoing edge, or else the traversal would become “stuck” at such a vertex. For both undirected and directed graphs, this fundamental necessity is met by the requirements listed above.

46. Strictly adhering to the definition would allow lone vertices that have no incident edges (i.e., vertices with a degree of zero); however, the `isConnected()` method does not take this into account, and therefore, a graph that is connected aside from a set of lone vertices will be rejected by the current version of the `findEulerPath()` method.
47. Refer to §2.3.4.2 of **[Knut97]** for more details.
48. Refer to **[Evan92]** and **[Knut97]** for more information regarding these pre-conditions and proof as to why these pre-conditions are necessary and sufficient in order for an Euler Path to exist in a given graph.

These primary pre-conditions are presented in the code as follows:[49]

```
public static final Path findEulerPath( Vertex startVertex )
   throws IllegalArgumentException, PathDoesNotExistException
{
   ...

   // Determine the Graph object that the given startVertex is
   // currently associated with ...
   Graph G = (Graph)startVertex.graph();

   // Ensure that G is a connected graph ...
   if ( ! G.isConnected() ) throw
      new PathDoesNotExistException( “Graph must be connected.” );

   // Ensure that an Euler Path exists in the given graph by verifying
   // that the degrees of each Vertex of an undirected graph are even, or
   // that the in-degree and out-degree values of each Vertex are the same
   // in a directed graph ...
   Enumeration e = G.vertices();
   if ( G.isDigraph() )
   {
      while ( e.hasMoreElements() )
      {
         // For directed graphs, make sure that the in-degree of each
         // Vertex object is equal to the out-degree ...
         Vertex V = (Vertex)e.nextElement();
         if ( V.inDegree() != V.outDegree() ) throw
            new PathDoesNotExistException();
      }
   }
   else  /* undirected graph */
   {
      while ( e.hasMoreElements() )
      {
         // For undirected graphs, make sure that the degree of each
         // Vertex object is an even number ...
         Vertex V = (Vertex)e.nextElement();
         if ( V.degree() % 2 != 0 ) throw
            new PathDoesNotExistException();
      }
   }

   ...
```

49. Note that there are a few additional pre-conditions that may result in an `IllegalArgumentException` being thrown. Such pre-conditions are fairly trivial and uninteresting. As an example, the given `startVertex` must be associated with an existing `Graph` object.

Once these pre-conditions are met, the process of finding an Euler Path may begin. First, a `Path` object is constructed with the given `startVertex` argument. The `findCycle()` method is then called to find an initial cycle in the given graph starting from the `startVertex` argument. In most graphs, finding a single cycle will not typically produce an Euler Path. Thus, the process of finding a cycle must be repeated until all edges have been traversed exactly once. The following algorithm summarizes the necessary steps to constructing an Euler Path:[50]

```
(1) Construct a Path object called Euler.
(2) Mark all Edge objects of the given Graph object as being unseen.
(3) Find a cycle in the given Graph object via the findCycle() method.  The
    Vertex and Edge objects of this cycle should be added to Euler.
(4) While the number of edges in Euler is less than the number of edges in
    the given graph, do steps (a)-(d):
    (a) Since the given graph is connected, there must be an Edge incident
        to one of the Vertex objects in Euler that is marked as being unseen.
        Find such an Edge and call it E.  Also, call the corresponding
        Vertex from Euler the Vertex V.
    (b) Construct a new Path object called newEuler with the initial Vertex V.
    (c) Find a cycle in the given Graph object that does not include any of the
        Edge objects in Euler.  To do so, make use of the findCycle() method.
    (d) Splice together the two Path objects Euler and newEuler by inserting
        newEuler into Euler after Vertex V.
(5) The resulting Path object Euler is a valid Euler Path.
```

It is important to realize that this algorithm would fail were it not for the given pre-conditions; however, assuming the pre-conditions are successfully satisfied, the above algorithm is essentially guaranteed to produce an Euler Path. Implementing the given algorithm proves to be fairly straightforward, especially with the existence of the `findCycle()` method that has already been described earlier in this section. The first three steps of the given algorithm are implemented in the `findEulerPath()` method as follows:

```
// Instantiate a new Path object and add the initial startVertex ...
Path Euler = new Path( startVertex );

// Mark all edges as unseen ...
e = G.edges();
while ( e.hasMoreElements() )
   ((Edge)e.nextElement()).status = UNSEEN;

// Find a cycle in G ...
findCycle( Euler );

if ( verbose )
   System.out.println( "Found cycle: " + Euler );
```

50. Refer to §8.2 of **[Evan92]** for a more general description of the algorithm.

Once the initial cycle is found, all additional cycles are located via successive calls to the given `findCycle()` method in the `while` loop of step (4). During each iteration, each new cycle is spliced together with the existing `Euler` path. The implementation is presented as follows:

```
      while ( Euler.length() < G.numEdges() )
      {
         // Find an untraversed edge that is incident with a Vertex
         // of the current Euler path ...
         e = Euler.vertices();
         Edge E = null;
         Vertex V = null;
         boolean foundEdge = false;

         while ( ! foundEdge && e.hasMoreElements() )
         {
            V = (Vertex)e.nextElement();
            Enumeration f = V.incidentEdges();
            while ( ! foundEdge && f.hasMoreElements() )
            {
               E = (Edge)f.nextElement();
               if ( E.status == UNSEEN )
                  foundEdge = true;
            }
         }

         // Construct a new Path object that contains both Vertex V
         // and Edge E.  Mark E as being part of a path ...
         Path newEuler = new Path( V );
         newEuler.add( E );
         E.status = IN_TREE_OR_PATH;

         // Find a new cycle from Vertex V traversing Edge E ...
         findCycle( newEuler );

         if ( verbose )
            System.out.println( "\n\nNext cycle: " + newEuler );

         // Splice together Euler and newEuler paths ...
         Euler.insertAfter( newEuler, V );
      }
```

When the `while` loop terminates, finish the algorithm by simply returning the constructed `Path` object to the calling method (i.e., step (5) of the given algorithm):

```
      // Return the Euler path ...
      return Euler;
   }
```

3.10.4 Determining the Biconnected Components of a Graph

Based on the `BOOK_COMPONENTS` program that is presented in **[Knut93]**, the static `Algorithms` class provides methods for finding and displaying *biconnected components* (or *bicomponents*) of a given `Graph` object. A bicomponent of a graph `G` is defined to be a maximal subgraph `H` of graph `G` in which if a vertex and its incident edges are removed from `H`, the resulting subgraph `H'` is still a connected graph. In other words, a valid path still exists between each distinct pair of vertices that remain in graph `H'`.

An *articulation point* is a vertex of a connected graph or subgraph that, when removed from the graph, leaves an unconnected graph. Formally, a vertex `v` is an articulation point of a graph if there are distinct vertices `w` and `x` (that are also distinct from `v`) such that `v` is in every path from `w` to `x`; thus, removing `v` from the graph results in an unconnected graph since `x` is no longer reachable from `w`, and vice versa. A connected graph is a bicomponent if and only if it contains no such articulation points.[51]

The `findBicomponents()` method of the static `Algorithms` class implements the algorithm presented in the `BOOK_COMPONENTS` program from **[Knut93]** by determining the bicomponents and articulation points of a given `Graph` object. Making use of numerous utility fields of the `Vertex` class, this method performs a non-recursive depth-first traversal rooted at either an arbitrary `Vertex` object or a specified `Vertex` object to determine the bicomponents of the given `Graph` object.

During this process, when a bicomponent is recognized, its root `Vertex` object is added to a standard Java `Stack` object called the *settled stack*.[52] When all bicomponents are located and added to this settled stack, the `Stack` object is returned, and should contain a `Vertex` object that identifies all other `Vertex` objects in the given bicomponent via utility fields of the `Vertex` class.

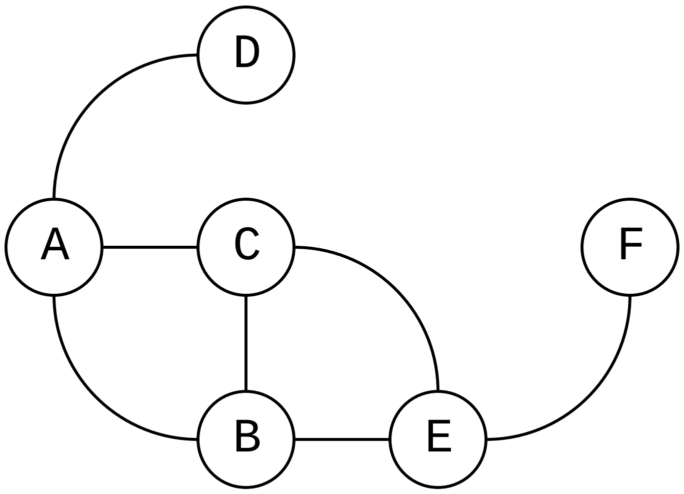
Note that the `findLongestPath2()` method makes use of the `findBicomponents()` method in maximizing the number of edges traversed from a given start vertex to a given goal vertex.

The `displayBicomponents()` method is available primarily for debugging and demonstrative purposes. After first invoking the `findBicomponents()` method, the `displayBicomponents()` method processes the `Vertex` object references from the settled stack that is returned from the `findBicomponents()` method call. For each `Vertex` object from the settled stack, the `displayBicomponents()` method displays the vertex and all other `Vertex` objects that are part of the corresponding bicomponent.

---

51. Refer to §4.5 (specifically §4.5.1) of **[Baas88]** for more information on bicomponents and articulation points, and their related properties. Also refer to **[Knut93]**.
52. Use of the term *settled stack* comes directly from **[Knut93]**.

Consider the following graph that consists of 6 vertices and 7 undirected edges:



Constructing this graph and subsequently invoking the `displayBicomponents()` method on this graph produces the following output in which 3 bicomponents are described:

```
There are 3 bicomponents in the given graph:
  Bicomponent D (rank 1): A, D.
  Bicomponent B (rank 3): A, B, C, E.
  Bicomponent F (rank 4): E, F.
```

The given output clearly specifies how many bicomponents exist, as well as how each vertex is distributed amongst the various bicomponents. Articulation points are the vertices that exist in more than one biconnected component. Thus, from the given output, vertices `A` and `E` are clearly the articulation points of the given example graph.

Since the code would span many pages, it is not included in this document; instead, refer to **[Knut93]** and the source code located on the website that is associated with this project.

3.10.5 Finding the Shortest Path Between Two Vertices

Devising an algorithm that may be used to determine the shortest path between two vertices of a graph is a common problem posed to many students of Computer Science. The `findShortest-Path()` method of the static `Algorithms` class provides an implementation of Dijkstra's classic Shortest Path algorithm, as presented in **[Baas88]**.[53] Specifically, this method constructs a `Path` object representing the shortest weighted path between the given start and end `Vertex` objects. If the shortest path cannot be determined due to the nonexistence of such a path in the given graph, the `PathDoesNotExistException` exception is thrown.

53. Refer to §4.3 of **[Baas88]** for more information. The actual algorithm appears on page 171 of **[Baas88]** and is labeled **Algorithm 4.3**.

The `findShortestPath()` method correctly processes both directed and undirected graphs. This "genericity" is obtained through the simple use of the `traverseFrom()` method that appears in both the `Edge` and `DirectedEdge` classes.[54] A shortcoming of this method, however, involves the requirement that the weights associated with each `Edge` or `DirectedEdge` object must be positive. In other words, graphs with edges of negative or zero weights are not processed by the `findShortestPath()` method; instead, an `IllegalArgumentException` exception is thrown. Although not presented in **[Baas88]**, Dijkstra's fundamental shortest path algorithm may be modified to handle edges with negative or zero weights; however, such modifications have not currently been implemented in the static `Algorithms` class.[55]

As with the aforementioned `findBicomponents()` method, the code that implements the `findShortestPath()` method is not included in this document due to the length of the algorithm; instead, refer to **[Baas88]** and source code located on the website that is associated with this project.

3.10.6 Approximating the Longest Path Between Two Vertices

Determining the longest path between two vertices of a graph is a much more complex problem than that of determining the shortest path between two vertices of a graph. Fundamentally, an algorithm may be developed to exhaustively search all possible paths of a given graph to determine which path or paths consist of the longest sum of weights. Such an algorithm is clearly NP-complete, and therefore algorithms that only approximate the longest path between two vertices are considered and implemented in the static `Algorithms` class. Specifically, two *greedy* approaches to the longest path problem have been considered and implemented in the `findLongestPath()` and `findLongestPath2()` methods.[56]

Throughout the following sections, `startVertex` is the argument that refers to the start vertex, and `endVertex` is the argument that refers to the end or goal vertex. Many of the utility fields of the `Vertex` class are used to maintain various state information regarding each vertex during the longest path determination process.

3.10.6.1 Implementing A Simple Greedy Algorithm

The `findLongestPath()` algorithm is an implementation of a simple greedy algorithm in which each iteration of the algorithm essentially consists of choosing an edge of maximum weight between the last vertex of the current path and an adjacent vertex that eventually leads to the goal

54. Refer back to §3.4.2 and §3.5.1 for more information on the `traverseFrom()` method.
55. Refer to page 88 of **[Evan92]** for a complete description of how Dijkstra's algorithm may be modified to form Ford's algorithm, which correctly handles edges with negative weights.
56. Both approaches are based on the `GB_GAMES` and `FOOTBALL` programs of **[Knut93]**. Also refer to §4.2 for an implementation of the `FOOTBALL` game that makes use of the `rpi.goldsd.graph` package.

vertex. This method begins by constructing a `Path` object that consists only of the `startVertex` argument. The `Vertex` object referred to by the `startVertex` argument is marked as being *visited*; all other `Vertex` objects of the associated graph are marked as being *unseen*.

Once the initialization process is complete, the iteration steps may be performed. At each step of the iteration process, an edge is chosen that connects the last vertex of the current path to an adjacent vertex that is not yet in the path. The latter vertex must eventually lead to the goal vertex, or else there would be no reason to choose such a vertex. To determine which vertices eventually lead to the specified goal vertex, a modified breadth-first traversal is performed, starting from the goal vertex and visiting only those vertices that are marked as *unseen*.

All vertices that are visited during this breadth-first search are marked as *reachable* by setting the `parent` utility field of the `Vertex` object to refer to the last vertex of the current path. Thus, when this breadth-first traversal is complete, all vertices that are adjacent to the last vertex of the current path are considered as potential candidate edges as long as they have `parent` fields that refer to this last vertex. During the edge selection process, an edge leading directly to the goal vertex is chosen only if no other suitable incident edge exists. Here is a concise summary of the algorithm:

```
(1) Mark all vertices of the associated Graph object as unseen.
(2) Construct a Path object and add startVertex to this Path, marking
    startVertex as visited.
(3) Set currentVertex to refer to the last Vertex of the constructed Path,
    which will simply be startVertex.
(4) While currentVertex and endVertex do not refer to the same Vertex object,
    perform steps (a)-(d):
    (a) Perform a breadth-first traversal starting from endVertex, only visiting
        Vertex objects that are marked as unseen.  At each Vertex, set the
        parent utility field to refer to currentVertex.
    (b) Choose the edge from the incident edges of currentVertex with the maximum
        weight that does not directly lead to endVertex.  If no such edge exists,
        choose the edge that leads to endVertex.  If an edge does not lead to
        endVertex, throw the PathDoesNotExistException exception and abort.
    (c) Add the selected edge and the corresponding destination Vertex object to
        the Path.
    (d) Set currentVertex to refer to the last Vertex of the constructed Path.
```

Note that the `findLongestPath()` method constructs `Path` objects for both undirected and directed graphs. The more complex `findLongestPath2()` method that is described in the next section does not currently work with directed graphs, unless the graph is completely bidirectional (i.e. for each directed edge that leads from vertex `v` to vertex `w`, there also exists a directed edge that leads from vertex `w` to vertex `v`). Since the code that implements the `findLongestPath()` method would span many pages, it is not included in this document; instead, refer to **[Knut93]** and the source code located on the website that is associated with this project.

3.10.6.2 Measuring the Performance of the Simple Greedy Algorithm

The graph-related algorithms presented in the previous sections are introduced and examined in just about any text covering computer algorithms.[57] The worst-case run-time performance measures that are given in the table of §3.10 are also included in such texts and are therefore not discussed in this document; however, the methods used to approximate the longest path between two vertices of a graph and their corresponding run-time performance measures are not usually part of the typical computer algorithms textbook. Thus, this section and §3.10.6.4 provide a summary of how well the `findLongestPath()` and `findLongestPath2()` methods perform.

Based on the algorithm presented in the previous section, the worst-case run-time performance of the `findLongestPath()` method is $\mathbf{O(v^2+ve)}$, where `v` is the number of vertices in the graph and `e` is the number of edges in the graph. The analysis begins with the first three initialization steps of the given algorithm: step (1) is clearly **O(v)**[58]; steps (2) and (3) occur in constant time **O(1)**. Thus, the initialization steps of the algorithm perform in linear time **O(v)**, based only on the number of vertices in the graph. These steps may be ignored when measuring the run-time performance due to the complexity of step (4), which is described next.

The core of the algorithm is step (4), which is further broken down into four additional steps that are labeled (a), (b), (c), and (d). The outer loop of step (4) performs in **O(v)** in the worst-case.[59] Within this loop, the breadth-first search that occurs in step (a) performs in **O(v+e)** in the worst-case. Combining these nested terms results in $\mathbf{O(v^2+ve)}$, which turns out to be the overall performance of the algorithm.

Before this conclusion is accepted, we must first consider the remaining steps (b), (c), and (d) of step (4). When the breadth-first traversal of step (a) is complete, a maximally weighted edge is selected from the set of incident edges of the current vertex (this is step (b) of the algorithm). Since the outer loop causes the algorithm to visit every vertex, all edges of the graph will be candidate edges in step (b) at some point during the execution of the algorithm. The *overall* run-time performance of step (b) is therefore **O(e)** in the worst-case. Clearly, step (b) may be ignored due to its insignificance in comparison to the $\mathbf{O(v^2+ve)}$ performance measure that was given above involving the breadth-first traversals that occur during each iteration of the algorithm.

Finally, steps (c) and (d) clearly perform in constant time **O(1)** and may therefore be ignored in this analysis.

57. Refer to **[Baas88]**, **[Corm92]**, **[Sedg88]**, and so on.
58. Step (1) is actually Θ**(v)** since each vertex is visited exactly once. Refer to **[Baas88]** for a full description of the difference between **O(*f*)** and Θ**(*f*)**, as well as Ω**(*f*)**.
59. Which may actually be the *best-case* from the perspective of the user attempting to find the longest path, since such a path would cover a maximum number of edges of the graph.

3.10.6.3 Implementing A Stratified Greedy Algorithm

The `findLongestPath2()` algorithm is an implementation of a greedy algorithm involving an approach called *stratified greed* associated with a *backtrack tree* that is rooted at the initial start vertex.[60] The fundamental concept behind the stratified greed algorithm is the partitioning of the vertices of the graph into a number of *strata* based on a *stratification function h* that has the following properties:

- When applied to the goal vertex, *h(goal) == 0*.
- When generally applied to vertices (including the goal vertex), *h(child) < h(parent)*.

The chosen stratification function will typically provide values that are not unique; i.e., there usually will not be a unique one-to-one correspondence between a vertex *V* and the corresponding stratification function value *h(V)*. In short, multiple vertices will typically reside at each defined stratum. Therefore, implementing the algorithm consists of maintaining a linked list of candidate vertices and corresponding edges for each stratum. Note that *v* such linked lists are necessary, where *v* is the number of vertices in the given graph.

A potentially large backtrack tree is also constructed as part of the algorithm; this backtrack tree is rooted at the initial start vertex. Vertices and edges are added to this tree based on where they exist in the linked lists associated with each stratum. More specifically, vertices and edges from the highest possible stratum are added to the backtrack tree during each iteration of the algorithm. Further, each linked list associated with a stratum is maintained in a sorted order according to the total weight of the path from the start vertex to the given vertex.

To limit the number of possible paths added to the backtrack tree, a *width* parameter exists that specifies the number of vertices and edges to maintain at each stratum.[61] In other words, each linked list has a maximum size specified by the *width* parameter.

During each iteration of the algorithm, vertices and corresponding edges are added to specific strata based on the following stratification function from **[Knut93]**: "Given a node *x* of the backtrack tree for longest paths, corresponding to a path from *start* to a certain vertex *u = u(x)*, we will let *h(x)* be the number of vertices that lie between *u* and *goal* (in the sense that the simple path from *start* to *u* can be extended until it passes through such a vertex and then all the way to *goal*)."[62] In the actual implementation of the `findLongestPath2()` method, strata lists are updated by performing a depth-first traversal from the goal vertex to determine the biconnected components of the graph, thus maximizing the number of edges that are traversed.

60. For a full description of the algorithm, refer to the `FOOTBALL` program of **[Knut93]**.
61. The *width* parameter is actually the third parameter of the `findLongestPath2()` method.
62. Directly quoted from page 114 (program item 19) of **[Knut93]**.

In general, the backtrack tree grows in a breadth-first manner, the *breadth* of the growth depending on the given width parameter.[63] Once a branch of this tree reaches the goal vertex, the iteration stops and the resulting path from the start vertex to the goal vertex is constructed by traversing the backtrack tree *backwards* from the goal vertex to the start vertex.

3.10.6.4 Measuring the Performance of the Stratified Greedy Algorithm

The worst-case run-time performance of the simple greedy algorithm that is implemented in the `findLongestPath()` method was calculated in §3.10.6.2 to be **$O(v^2+ve)$**, where `v` is the number of vertices in the graph, and `e` is the number of edges in the graph. The `findLongestPath2()` method has a similar structure and therefore may be initially approximated as **$O(v^2+ve)$**[64]; however, the performance of this method also greatly depends on the given *width* parameter (which will subsequently be called `w`).

Consider the operations necessary to maintain the linked lists of vertices and edges at each stratum. Since there are `v` strata, and potentially `w` vertices and edges at each stratum, the overall space requirement is **$O(vw)$**. Maintaining the sorted order in a linked list of maximum size `w` is clearly **$O(w)$** for each insertion that occurs (in the worst-case). Such insertions occur repeatedly for many edges of the graph, depending on the characteristics of the graph and the value of `w`. It is therefore difficult to analytically determine how `w` is related to the initial approximation **$O(v^2+ve)$**.

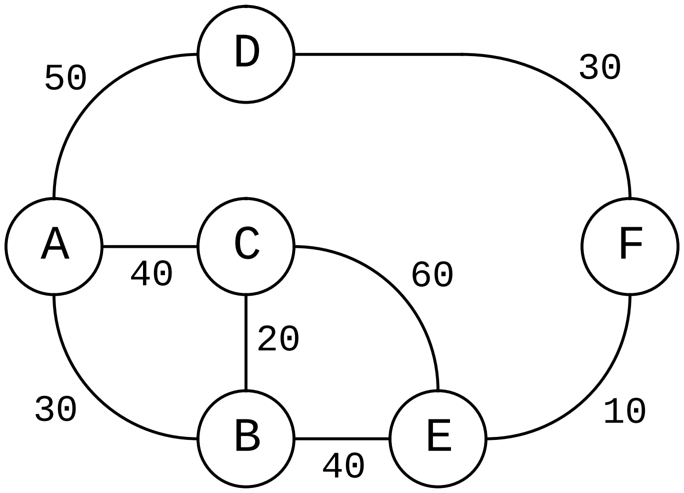
Knuth concludes through empirical methods that "the running time is roughly proportional to `w` when `w` is small, although it eventually becomes proportional to w^2. If more sophisticated data structures were substituted for the linear lists in the present implementation, we could bring this asymptotic behavior down to **$O(w\log w)$**. But the algorithm appears to reach a point of diminishing returns, after which feasible increases in `w` do not improve the solutions found, so it probably should never be run with extremely large `w`."[65]

Concluding that substantially larger values of `w` will not typically produce longer paths makes sense. Since each stratum contains a *sorted* linked list of vertices and edges based on the total weight "so far," adding such *low-value* vertices and edges to the backtrack tree seems somewhat useless. These vertices and edges will likely *not* be part of the constructed path.

63. That is, the greater the value of the width parameter, the more vertices and edges of a given stratum are added to the backtrack tree.
64. The stratified greedy algorithm involves the use of a depth-first search instead of a breadth-first search; however, both traversal algorithms are **$O(v+e)$**. Further, the stratified greedy algorithm also visits each vertex at least once as it builds the backtrack tree, so providing a nested **$O(v)$** term seems reasonable.
65. Directly quoted from page 20 of **[Knut93]**. Note that these empirical results are obtained from the `FOOTBALL` program, which is described in §4.2 of this document.

3.10.6.5 A Simple Test of the Longest Path Algorithms

The following undirected graph has been specifically designed as a simple test graph for the `findLongestPath()` and `findLongestPath2()` methods:



Using the given graph, the posed problem is to find the longest path between vertex `A` and vertex `F`. Through simple inspection, it should be clear that the single solution is the path `A->B->C->E->F`, which produces an overall weight of 120. Not only does this solution maximize the number of vertices that are visited, this solution also maximizes the overall sum of the weights of the edges that are traversed. In other words, there is no such path in this graph from vertex `A` to vertex `F` that traverses more edges or has an overall weight greater than 120.

The limitations of the `findLongestPath()` method becomes clear when this method is applied to this simple graph of 6 vertices and 8 edges. In the simple greedy case, the `findLongestPath()` method produces the path `A->D->F`, which consists of an overall weight of 80 (and only two edges). The algorithm first chooses the edge between vertex `A` and vertex `D` since this edge is the edge incident with vertex `A` that has the maximum weight. This choice turns out to be a poor choice, since it forces the next edge chosen to be the final choice.

Upon applying the stratified greedy algorithm with a width of 1, the more complex `findLongestPath2()` method also produces the path `A->D->F`. The simple greedy algorithm is actually a special case of the stratified greedy algorithm when the width parameter is 1. With a width of 1, the backtrack tree that is constructed as part of the stratified greedy algorithm will contain only a single branch leading from the start vertex to the goal vertex. No other branches will be considered since the linked lists at each stratum will essentially contain only a single element.[66]

66. Refer to page 114 (program item 19) of **[Knut93]** for more information as to why this occurs.

Applying the stratified greedy algorithm with a width of 2 produces the path `A->C->E->F`, which has an overall weight of 110. Although this is an improvement over the simple greedy case, it is still not the correct solution. Increasing the value of the width parameter to 3 and invoking the `findLongestPath2()` method once again produces the correct path `A->B->C->E->F`, which consists of an overall weight of 120. In general, the resulting path that is generated by the `findLongestPath2()` method is substantially better than the path produced via the simple greedy approach when the width parameter is greater than 1.[67]

3.10.7 Interfacing with the Graph Draw Applet

Making use of the methods defined in the `Drawable` interface[68], graphs and paths may be displayed graphically in the *Graph Draw* applet via one of the two overloaded `toGraphDraw()` methods that are available in the static `Algorithms` class.

The fundamental `toGraphDraw()` method takes as an argument a single `Drawable` object, and returns a Java `String` object that is suitable as input to the Graph Draw applet. Specifically, this `String` object contains all edges of the given `Drawable` object, adhering to the *edges* format of the Graph Draw applet. An example `String` object representing a graph or path with three *directed* edges would appear as "`v1-v2,v2-v3,v3-v4`" in the required *edges* format. In this example, there is an edge directed from vertex `v1` to vertex `v2`, an edge directed from vertex `v2` to vertex `v3`, and an edge directed from vertex `v3` to vertex `v4`. An undirected edge may be represented as two directed edges in the Graph Draw applet; thus, the given `String` object "`Harvard-Stanford,Stanford-Harvard`" is a valid representation of an undirected edge between two vertices representing Stanford and Harvard.

The Graph Draw applet does not currently support the depiction of weighted edges, thus weights of edges are ignored by the `toGraphDraw()` method. For example, when applied to the weighted undirected graph of §3.10.6.5, the fundamental `toGraphDraw()` method may return the following `String` object (note that the order of edges is somewhat arbitrary):

```
A-B,B-A,A-C,C-A,A-D,D-A,B-C,C-B,B-E,E-B,C-E,E-C,D-F,F-D,E-F,F-E
```

Since the Graph Draw applet does not currently support edges that are self-loops, using the notation "`v5-v5`" does not indicate a self-loop; instead, if a given `Vertex` object does not contain any edges, including such a `String` object will produce a lone vertex in the Graph Draw applet (i.e., a vertex with no incident edges).[69]

67. Refer to §4.2 for a more conclusive example of how the `findLongestPath2()` method out-performs the `findLongestPath()` method.
68. Refer to §3.11 for more information regarding the `Drawable` interface of the static `Algorithms` class.
69. As mentioned way back in §1.2.1, the interactive Graph Draw applet is available for use at the following website: `http://www.cs.rpi.edu/projects/pb/graphdraw`.

A second version of the `toGraphDraw()` method is available that takes two arguments: the `Drawable` object to be displayed, and a Java `String` representing the name of the output `HTML` file to generate. This version of the `toGraphDraw()` method creates an `HTML` file that contains the *edges* representation of the given `Drawable` object, which is obtained from the single-argument version of the `toGraphDraw()` method described above. The resulting `HTML` file, when loaded into a browser application, loads the Graph Draw applet and presents the given `Drawable` object to the applet, thus displaying the `Drawable` object in graphical form.

Using the example graph of §3.10.6.5 once again, this version of the `toGraphDraw()` method produces the following `HTML` file:

```
<BASE HREF="http://www.cs.rpi.edu/projects/pb/graphdraw/">
<HTML>
<HEAD>
<TITLE>Graph Drawing Applet</TITLE>
</HEAD>
<BODY>
<APPLET code="GraphApplet.class"
        width=50
        height=30>
<PARAM name="edges"
       value="A-B,B-A,A-C,C-A,A-D,D-A,B-C,C-B,B-E,E-B,C-E,E-C,D-F,F-D,E-F,F-E">
<PARAM name="name"
       value="Section 3.10.5.5 Graph">
</APPLET>
</BODY>
</HTML>
```

Notice that the `name` parameter passed to the Graph Draw applet via the `<PARAM>` tag contains the actual name of the corresponding `Drawable` object. Note that this `String` object is obtained via the `name()` method of the `Drawable` interface.

3.11 The `Drawable` Interface

The `Drawable` interface of the static `Algorithms` class is used to identify graph-related objects that may be displayed in a graphical application (i.e., the *Graph Draw* applet). Three methods are currently defined in the `Drawable` interface: `edges()` to obtain an enumeration of the edges of the graph-related object; `vertices()` to obtain an enumeration of the vertices; and `name()` to obtain the name of the graph-related object. The abstract `GraphBase` class and the `Path` class implement the `Drawable` interface.[70]

70. Since the abstract `GraphBase` class implements the `Drawable` interface, the `Graph` and `Tree` classes are also considered *drawable*. Refer back to §3.10.7 for a description of the `toGraphDraw()` methods of the static `Algorithms` class that make use of classes that implement the `Drawable` interface.

4 Developing Applications with the Graph Package

Two demonstrative applications have been developed using the `rpi.goldsd.graph` package. These application programs are based on the `LADDERS` and `FOOTBALL` programs developed by Knuth in **[Knut93]**. The following sections describe each of the applications as implemented using the `rpi.goldsd.graph` package.

4.1 The Word Ladders Game (`Ladders.java`)

The `Ladders.java` program provides an interactive game in which the user is asked to input two 5-letter words, and the program attempts to construct a *word ladder* between the two words based on a dictionary of valid 5-letter words. Word ladders are sequences of words in which each adjacent word differs by only one letter at a given position. As an example, consider the following execution of the `Ladders.java` program in which two word ladders are generated (user input is highlighted in bold):

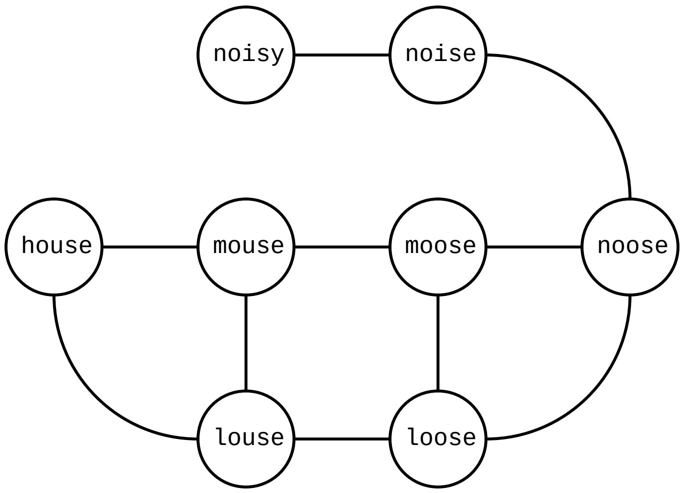
```
Please enter starting word:
flour
Please enter ending word:
bread

Attempting to construct word ladder from
flour to bread.

Found Path with 6 edges:
flour->floor->flood->blood->brood->broad->bread.


Please enter starting word:
chaos
Please enter ending word:
order

Attempting to construct word ladder from
chaos to order.

Found Path with 13 edges:
chaos->chats->coats->costs->hosts->hoses->roses->rises->rides->rider->
eider->elder->older->order.
```

In this example, a dictionary of 3300 common 5-letter words was utilized.[71]

71. The dictionary of 5-letter words is actually the `words.dat` file provided in the distribution that accompanies **[Knut93]**. This dictionary actually contains 5757 words; however, many of the words are somewhat obscure, thus a mechanism for obtaining the 3300 most common words is provided. Again, refer to **[Knut93]** for details.

4.1.1 Constructing the Graph of 5-Letter Words

The fundamental goal of the `Ladders.java` program is to construct an undirected graph representing the set of 5-letter words. Each vertex of the graph represents a distinct 5-letter word; thus, for the example in which 3300 words are considered, the constructed graph consists of 3300 vertices. An undirected edge exists between two vertices in the graph when the corresponding words differ by only a single letter at a common position. Consider a very small dictionary containing only the following 5-letter words: *house*, *loose*, *louse*, *moose*, *mouse*, *noise*, *noisy*, and *noose*. The constructed word ladder graph would appear as follows:



With the set of 3300 5-letter words provided by **[Knut93]**, the constructed graph consists of 6596 edges.[72] Constructing such a graph is a time-consuming process, especially if a poor algorithm is chosen to construct the graph. As a first attempt, consider the following algorithm:

```
(1) For each 5-letter word in the input file, do steps (a) and (b):
    (a) Add a new Vertex to the word ladder graph that represents the
        current 5-letter word.
    (b) For all existing Vertex objects in the graph, do step (i):
        (i) If the 5-letter word of the current Vertex differs from the
            new 5-letter word being added to the graph by only one letter,
            create a new undirected edge between the two vertices.
```

Clearly, this algorithm performs in **$O(n^2)$** in the worst-case, where `n` represents the number of words to be processed. Although this brute-force approach works, as the number of 5-letter words grows, the time waiting for the graph to be constructed grows quadratically. This quadratic algorithm is not acceptable since dictionaries will typically consist of a relatively large number of words.

72. Making use of the full set of 5757 words provided in the `words.dat` file from **[Knut93]**, the constructed graph consists of 14,135 edges.

A better algorithm for constructing the word ladder graph exists and is described and fully implemented in the `GB_WORDS` program of **[Knut93]**. Using the same approach implemented in Knuth's `GB_WORDS` program, the `Ladders.java` program creates an array of five hashtables. Each entry of each of the hashtables consists of a `LadderStr` object, which extends the `Str` wrapper-class of the `rpi.goldsd.container` package, and is defined as follows:

```
class LadderStr extends Str
{
   private int omitPosition;
   public Vertex associatedVertex;

   public LadderStr( String s, int omitPosition, Vertex V )
   {
      super( s );
      this.omitPosition = omitPosition;
      this.associatedVertex = V;
   }

   public int hash()
   {
      int result = 0;
      char[] chars = string.toCharArray();
      for ( int i = 0 ; i < chars.length ; i++ )
         if ( i != omitPosition )
            result += chars[i] - 'a';
      return result;
   }

   public boolean similar( LadderStr s )
   {
      char[] chars1 = string.toCharArray();
      char[] chars2 = s.string.toCharArray();
      for ( int i = 0 ; i < chars1.length ; i++ )
         if ( i != omitPosition && chars1[i] != chars2[i] )
            return false;
      return true;
   }
}
```

The `LadderStr` class consists of a Java `String` object that is inherited from the `Str` base class, an `omitPosition` value that indicates the index position to ignore in the associated `String` object, and a reference to the associated `Vertex` object that contains the corresponding 5-letter word. To efficiently match words that differ by only one letter in a common position, the `hash()` method of the `LadderStr` class sums up the *value* of each character in the word, except for the character in the omitted position that is specified by the `omitPosition` attribute. Thus, the two words *house* and *mouse* would hash to the same bucket with an `omitPosition` value of 0. Similarly, the two words *hosts* and *hoses* would hash to the same bucket with an `omitPosition` value of 3.

Applying this `hash()` method to a given 5-letter word five times with each possible `omitPosition` values provides five distinct hash values. Thus, five `LadderStr` objects are constructed for a given 5-letter word, each being stored in a separate hashtable. The index used to identify a hashtable from the array of five hashtables matches the `omitPosition` value of the corresponding `LadderStr` object. For example, the two `LadderStr` objects representing the words *house* and *mouse* are stored in the same hash bucket in the hashtable identified by index 0.

When the `Ladders.java` program executes, it begins by reading in all of the valid 5-letter words from the `words.dat` input file into an array of `String` objects, as follows:

```
int numWords = 0;
String strings[] = new String[5757];
try
{
   FileInputStream istream = new FileInputStream( “words.dat” );
   InputStreamReader isreader = new InputStreamReader( istream );
   BufferedReader in = new BufferedReader( isreader );
   while ( in.ready() )
   {
      String s = in.readLine();
      if ( s.charAt( 0 ) != ‘*’ && s.length() > 5 && s.charAt( 5 ) == ‘*’ )
         strings[numWords++] = new String( s.substring( 0, 5 ) );
   }

   System.out.println( “Encountered ” + numWords + “ words.” );
   in.close();
}
catch( IOException e )
{
   System.err.println( e );
   System.exit(0);
}
```

Once the array of `String` objects has been successfully read in from the `words.dat` input file, an empty graph is constructed, as well as an array of `Vertex` references. Each of the `String` objects from the `strings` array is used to construct a new `Vertex` object consisting of a single `Str` object that represents the 5-letter word. Also, a 2-dimensional array of `LadderStr` objects is constructed and populated with each of the five variations of the 5-letter word (i.e., with the first character omitted, with the second character omitted, and so on). The code continues as follows (note that the `LEN` variable is used to define the number of letters in each word, thus if a dictionary of 7-letter words was used, the `Ladders.java` program could still construct valid word ladders):

```
int LEN = 5;   // Number of letters in each word ...
Graph G = new Graph( “Ladders Graph” );
Vertex V[] = new Vertex[numWords];
LadderStr strs[][] = new LadderStr[numWords][LEN];

for ( int i = 0 ; i < numWords ; i++ )
{
   // Construct a new Vertex and add it to the graph ...
   V[i] = new Vertex( new Str( strings[i] ) );
   G.add( V[i] );

   // Create LEN number of LadderStr objects ...
   for ( int j = 0 ; j < LEN ; j++ )
      strs[i][j] = new LadderStr( strings[i], j, V[i] );
}
```

The next section of code constructs and populates the five hashtables. Since the resulting graph consists of undirected edges, the order of determining a match and constructing a new edge does not matter. Thus, during the population of the five hashtables, valid edges are added to the graph based on the `similar()` method of the `LadderStr` class. The code continues as follows:

```
// Construct LEN number of Table objects ...
Table tables[] = new Table[LEN];
for ( int i = 0 ; i < LEN ; i++ )
   tables[i] = new Table( 127 );

// For each String, add LEN number of LadderStr object references to
// each of the LEN hashtables.  If matches are encountered based on
// the similar() method of the LadderStr class, construct and add a
// new undirected edge to the graph ...
for ( int i = 0 ; i < strs.length ; i++ )
{
   for ( int j = 0 ; j < LEN ; j++ )
   {
      Enumeration e = tables[j].elements( strs[i][j].hash() );
      while ( e.hasMoreElements() )
      {
         LadderStr LS = (LadderStr)e.nextElement();
         if ( LS.similar( strs[i][j] ) )
         {
            Edge E = new Edge( V[i], LS.associatedVertex, 1.0 );
            G.add( E );
         }
      }

      tables[j].add( strs[i][j] );
   }
}
```

Note that an arbitrary weight of 1.0 is associated with each `Edge` object that is added to the word ladder graph. As described in the next section, Dijkstra's Shortest Path algorithm is utilized to construct a word ladder, and therefore each edge of the graph must be a weighted edge.

4.1.2 Constructing Word Ladders

Once the word ladders graph has been constructed, a word ladder may be created via Dijkstra's Shortest Path algorithm that is available via the `findShortestPath()` method of the static `Algorithms` class.[73] The following code is the final portion of the `Ladders.java` program. Note that the code used to input the words from the user is not included in this document due to its length. Essentially, the words that are entered by the user are represented by the `startWord` and `endWord` variables (two Java `String` objects). The code is presented as follows:

```
System.out.println( “Attempting to construct word ladder from ” );
System.out.println( startWord + “ to ” + endWord + “.” );

// Obtain references to the start and end vertices ...
Vertex startVertex = G.mapToVertex( startWord );
Vertex endVertex = G.mapToVertex( endWord );

boolean pathExists = true;
Path P = null;

try
{
   // Use Dijkstra’s Shortest Path algorithm to construct the
   // word ladder between the startWord and the endWord ...
   P = Algorithms.findShortestPath( startVertex, endVertex );
}
catch( PathDoesNotExistException E )
{
   System.out.println( “Sorry, a path does not exist ...” );
   pathExists = false;
}

if ( pathExists )
{
   System.out.println( “Found Path with ” + P.length() + “ edges:” );
   System.out.println( P );
}
```

Making use of a `try/catch` block, the `PathDoesNotExistException` exception is handled by setting the `pathExists` variable to `false`. Also note that the `toString()` method of the `Path` class is automatically invoked within the last `System.out.println()` method call.

73. Refer back to §3.10 for a description of the static `Algorithms` class; refer specifically to §3.10.5 for information regarding the `findShortestPath()` method.

4.2 The Football Scoring Game (`Football.java`)

The `Football.java` program provides a game in which the user inputs two football teams and an optional *width* argument, and the program attempts to determine the highest number of points the first team would score against the second team by using one of the longest path algorithms of the static `Algorithms` class. Based on the `FOOTBALL` program and the `games.dat` file designed and implemented by Knuth in **[Knut93]**, the `Football.java` program begins by constructing a graph of football games played by football teams of prominent college and universities of the United States during the 1990 season.

Each vertex of the constructed *football graph* represents a single football team. Each *pair* of directed edges is used to represent a game played between two teams. Making use of the data found in Knuth's `games.dat` file, the constructed graph consists of 120 vertices (teams), and 1276 directed edges (638 games).[74]

The `Football.java` program must be executed with two command-line arguments representing the *abbreviations* of the starting and ending teams. As an example, suppose the `Football.java` program was executed with `STAN` and `HARV` as its command-line arguments (in the given order). Given these two arguments, the `Football.java` program constructs the football graph and subsequently invokes the `findLongestPath()` method to approximate the longest path between the Stanford and Harvard vertices.[75] The following output is obtained:

```
Football Graph:
  # of vertices: 120
  # of directed edges: 1276

Finding “longest” path using the simple greedy algorithm.
  Sep 22:  Stanford Cardinal 37, Oregon State Beavers 3 (+34)
  Oct 13:  Oregon State Beavers 35, Arizona Wildcats 21 (+48)
  Sep 15:  Arizona Wildcats 25, New Mexico Lobos 10 (+63)
  Oct 06:  New Mexico Lobos 48, Texas-El Paso Miners 28 (+83)
  Oct 13:  Texas-El Paso Miners 12, Hawaii Rainbow Warriors 10 (+85)
  Dec 01:  Hawaii Rainbow Warriors 59, Brigham Young Cougars 28 (+116)
  Nov 03:  Brigham Young Cougars 54, Air Force Falcons 7 (+163)
  Oct 27:  Air Force Falcons 52, Utah Utes 21 (+194)
  Sep 01:  Utah Utes 19, Utah State Aggies 0 (+213)
  Nov 03:  Utah State Aggies 55, New Mexico State Aggies 10 (+258)
  Nov 17:  New Mexico State Aggies 43, Fullerton State Titans 9 (+292)
  Oct 27:  Fullerton State Titans 35, Long Beach State Forty-Niners 37 (+290)
  Sep 22:  Long Beach State Forty-Niners 28, Pacific Tigers 7 (+311)
```

74. For more information on the `games.dat` file, refer to program item 12 of the `GB_GAMES` program on page 227 of **[Knut93]**.
75. The `findLongestPath()` method makes use of a simple greedy algorithm that is described back in §3.10.6.1 and §3.10.6.2.

```
    Sep 29:  Pacific Tigers 28, Nevada-Las Vegas Rebels 37 (+302)
    Nov 03:  Nevada-Las Vegas Rebels 18, Fresno State Bulldogs 45 (+275)
    Sep 01:  Fresno State Bulldogs 41, Eastern Michigan Hurons 10 (+306)
    Sep 15:  Eastern Michigan Hurons 45, Ohio University Bobcats 18 (+333)
    Oct 06:  Ohio University Bobcats 10, Bowling Green Falcons 10 (+333)
    Sep 02:  Bowling Green Falcons 34, Cincinnati Bearcats 20 (+347)
    Sep 08:  Cincinnati Bearcats 34, Central Michigan Chippewas 0 (+381)
    Oct 06:  Central Michigan Chippewas 42, Kent State Golden Flashes 0 (+423)
    Sep 08:  Kent State Golden Flashes 38, Akron Zips 10 (+451)
    Oct 27:  Akron Zips 17, Rutgers Scarlet Knights 20 (+448)
    Sep 08:  Rutgers Scarlet Knights 24, Kentucky Wildcats 8 (+464)
    Nov 10:  Kentucky Wildcats 28, Vanderbilt Commodores 21 (+471)
    Sep 22:  Vanderbilt Commodores 24, Louisiana State Fighting Tigers 21 (+474)
    Sep 15:  Louisiana State Fighting Tigers 35, Miami of Ohio Redskins 7 (+502)
    Oct 06:  Miami of Ohio Redskins 24, Ball State Cardinals 10 (+516)
    Oct 20:  Ball State Cardinals 13, Western Michigan Broncos 14 (+515)
    Sep 15:  Western Michigan Broncos 27, Louisiana Tech Bulldogs 21 (+521)
    Sep 29:  Louisiana Tech Bulldogs 24, Southwestern Lsna Ragin' Cajuns 10 (+535)
    Sep 01:  Southwestern Louisiana Ragin' Cajuns 48, Tulane Green Wave 6 (+577)
    Sep 15:  Tulane Green Wave 43, Southern Methodist Mustangs 7 (+613)
    Nov 10:  Southern Methodist Mustangs 28, Rice Owls 30 (+611)
    Oct 20:  Rice Owls 42, Texas Tech Red Raiders 21 (+632)
    Nov 10:  Texas Tech Red Raiders 40, Texas Christian Horned Frogs 28 (+644)
    Oct 06:  Texas Christian Horned Frogs 54, Arkansas Razorbacks 26 (+672)
    Sep 15:  Arkansas Razorbacks 28, Tulsa Golden Hurricane 3 (+697)
    Sep 01:  Tulsa Golden Hurricane 3, Oklahoma State Cowboys 10 (+690)
    Oct 27:  Oklahoma State Cowboys 48, Missouri Tigers 28 (+710)
    Sep 29:  Missouri Tigers 30, Arizona State Sun Devils 9 (+731)
    Nov 10:  Arizona State Sun Devils 51, Washington State Cougars 26 (+756)
    Sep 22:  Washington State Cougars 41, California Golden Bears 31 (+766)
    Nov 10:  California Golden Bears 28, Oregon Ducks 3 (+791)
    Sep 08:  Oregon Ducks 42, San Diego State Aztecs 21 (+812)
    Oct 06:  San Diego State Aztecs 51, Wyoming Cowboys 52 (+811)
    Sep 01:  Wyoming Cowboys 38, Temple Owls 23 (+826)
    Nov 24:  Temple Owls 29, Boston College Eagles 10 (+845)
    Oct 13:  Boston College Eagles 41, Army Cadets 20 (+866)
    Oct 20:  Army Cadets 56, Lafayette Leopards 0 (+922)
    Nov 03:  Lafayette Leopards 59, Fordham Rams 14 (+967)
    Sep 29:  Fordham Rams 35, Brown Bears 28 (+974)
    Nov 17:  Brown Bears 17, Columbia Lions 0 (+991)
    Oct 27:  Columbia Lions 17, Princeton Tigers 15 (+993)
    Nov 03:  Princeton Tigers 34, Pennsylvania Red \& Blue 20 (+1007)
    Sep 15:  Pennsylvania Red \& Blue 16, Dartmouth Big Green 6 (+1017)
    Sep 22:  Dartmouth Big Green 33, Lehigh Engineers 14 (+1036)
    Nov 03:  Lehigh Engineers 52, Colgate Red Raiders 7 (+1081)
    Sep 22:  Colgate Red Raiders 59, Cornell Big Red 24 (+1116)
    Nov 03:  Cornell Big Red 41, Yale Bulldogs 31 (+1126)
    Nov 17:  Yale Bulldogs 34, Harvard Crimson 19 (+1141)

Path contains 61 edges and has weight 1141.0.
```

The output provides a summary of each game played, as well as a running total of the overall points accumulated due to each game. Although the date of each game appears in the output, the games are not played in chronological order according to the given dates.

It is interesting to note that the path generated by the `Football.java` program has an overall weight of 1141, whereas Knuth's simple greedy implementation results in a path of weight 781.[76] By no means does this imply that the `findLongestPath()` method used by the `Football.java` program produces longer paths than Knuth's implementation. Actually, it is by sheer luck that the `findLongestPath()` obtains an overall weight 360 points greater than Knuth's implementation. The inconsistencies occur during the edge selection process.[77] The set of candidate edges may in fact contain edges of equal weight, in which case an edge is chosen somewhat arbitrarily.

Referring to the output presented on page 19 of **[Knut93]**, the `Football.java` program chooses a different edge than Knuth's `FOOTBALL` program after 18 common edges (games) are chosen. Both of the differing edges have a weight of 14, yet the destination teams (i.e., the losing teams) differ. The `Football.java` program chooses the following edge:

```
        Sep 02:  Bowling Green Falcons 34, Cincinnati Bearcats 20 (+347)
```

The `FOOTBALL` program chooses a different edge:

```
        Sep 08:  Bowling Green Falcons 21, Virginia Tech Gobblers 7 (+347)
```

As luck would have it, the `Football.java` program ends up finding a path that turns out to be 360 points greater than Knuth's implementation, in the case of Stanford over Harvard. There are probably many other pairs of teams in which this, and the opposite case, is true.[78]

To further improve on the number of points one team may accumulate over another team, the `findLongestPath2()` method that implements a *stratified* greedy algorithm may also be used. An optional third command-line argument exists in the `Football.java` program; this argument is an integer used to specify the *width* value to use in the `findLongestPath2()` method.[79] Continuing the example in which we are trying to maximize the number of points Stanford may score over Harvard, the `Football.java` program may be invoked with the three arguments `STAN`, `HARV`, and 20. The following (abbreviated) output results from such an invocation:

76. Refer to program item 1 of the `FOOTBALL` program on page 106 of **[Knut93]** for a summary of some example results. Also refer to page 20 of **[Knut93]** for more results.
77. Refer back to the algorithm of §3.10.6.1 for a summary of the `findLongestPath()` method.
78. Swapping the example teams results in a case where the `FOOTBALL` program favors Harvard over Stanford by 1529 points; the `Football.java` results in a path of weight 1487 only.
79. Refer back to §3.10.6.3 for a summary of the `findLongestPath2()` method and a description of the stratified greedy approach that this method implements.

```
Constructing the Football graph ...
Football Graph:
  # of vertices: 120
  # of directed edges: 1276

Finding “longest” path using the stratified greedy approach (width 20).
  Sep 06:  Stanford Cardinal 17, Colorado Buffaloes 21 (-4)
  Nov 17:  Colorado Buffaloes 64, Kansas State Wildcats 3 (+57)
  Sep 15:  Kansas State Wildcats 52, New Mexico State Aggies 7 (+102)
  Nov 17:  New Mexico State Aggies 43, Fullerton State Titans 9 (+136)
  Sep 15:  Fullerton State Titans 13, Mississippi State Bulldogs 27 (+122)
  Oct 20:  Mississippi State Bulldogs 38, Tulane Green Wave 17 (+143)
  Oct 27:  Tulane Green Wave 49, Cincinnati Bearcats 7 (+185)
  Sep 08:  Cincinnati Bearcats 34, Central Michigan Chippewas 0 (+219)
  Oct 06:  Central Michigan Chippewas 42, Kent State Golden Flashes 0 (+261)
  Oct 20:  Kent State Golden Flashes 44, Ohio University Bobcats 15 (+290)
  Oct 06:  Ohio University Bobcats 10, Bowling Green Falcons 10 (+290)
  Sep 08:  Bowling Green Falcons 21, Virginia Tech Gobblers 7 (+304)
  Nov 24:  Virginia Tech Gobblers 38, Virginia Cavaliers 13 (+329)
  Sep 22:  Virginia Cavaliers 59, Duke Blue Devils 0 (+388)
  Nov 03:  Duke Blue Devils 57, Wake Forest Demon Deacons 20 (+425)
  Sep 29:  Wake Forest Demon Deacons 52, Army Cadets 14 (+463)
  Nov 10:  Army Cadets 3, Air Force Falcons 15 (+451)
  Sep 08:  Air Force Falcons 27, Hawaii Rainbow Warriors 3 (+475)
  Dec 01:  Hawaii Rainbow Warriors 59, Brigham Young Cougars 28 (+506)
  Oct 13:  Brigham Young Cougars 52, Colorado State Rams 9 (+549)
  ...
  ...
  Sep 01:  Temple Owls 23, Wyoming Cowboys 38 (+1659)
  Oct 06:  Wyoming Cowboys 52, San Diego State Aztecs 51 (+1660)
  Dec 01:  San Diego State Aztecs 28, Miami Hurricanes 30 (+1658)
  Oct 06:  Miami Hurricanes 31, Florida State Seminoles 22 (+1667)
  Dec 28:  Florida State Seminoles 24, Penn State Nittany Lions 17 (+1674)
  Sep 22:  Penn State Nittany Lions 28, Rutgers Scarlet Knights 0 (+1702)
  Sep 15:  Rutgers Scarlet Knights 28, Colgate Red Raiders 17 (+1713)
  Sep 22:  Colgate Red Raiders 59, Cornell Big Red 24 (+1748)
  Oct 13:  Cornell Big Red 38, Lafayette Leopards 16 (+1770)
  Nov 03:  Lafayette Leopards 59, Fordham Rams 14 (+1815)
  Sep 29:  Fordham Rams 35, Brown Bears 28 (+1822)
  Sep 15:  Brown Bears 21, Yale Bulldogs 27 (+1816)
  Nov 10:  Yale Bulldogs 34, Princeton Tigers 7 (+1843)
  Nov 03:  Princeton Tigers 34, Pennsylvania Red \& Blue 20 (+1857)
  Sep 15:  Pennsylvania Red \& Blue 16, Dartmouth Big Green 6 (+1867)
  Sep 22:  Dartmouth Big Green 33, Lehigh Engineers 14 (+1886)
  Oct 20:  Lehigh Engineers 22, Holy Cross Crusaders 34 (+1874)
  Nov 03:  Holy Cross Crusaders 43, Bucknell Bisons 14 (+1903)
  Sep 22:  Bucknell Bisons 41, Columbia Lions 16 (+1928)
  Sep 15:  Columbia Lions 6, Harvard Crimson 9 (+1925)

Path contains 108 edges and has weight 1925.0.
```

Not only did the number of edges in the resulting path increase in the above example, but also the total number of points increased from 1141 to 1925. It is interesting to note that the first game of this example actually is a losing game in which the accumulated point total drops immediately from its initial value of 0 to -4.

4.2.1 Results of the Stratified Greedy Algorithm

The following table summarizes the results of using the stratified greedy algorithm with varying values of `w` in both the `FOOTBALL` and `Football.java` programs, using Stanford as the starting team and Harvard as the ending team:

	`FOOTBALL`		**`Football.java`**	
`w`	**Edges in Path**	**Path Weight**	**Edges in Path**	**Path Weight**
1	103	1573	107	1692
2	109	1776	108	1806
3	107	1847	111	1859
4	106	1950	106	1931
5	105	1951	105	1951
10	113	1895	105	1949
20	108	1925	108	1925
50	108	2057	108	2057
100	113	2126	113	2126
200	109	2129	109	2129
500	111	2154	111	2154

As discussed in the previous section, the paths that are generated by the `FOOTBALL` and `Foot-ball.java` programs differ slightly based on minor implementation differences in the underlying stratified greedy algorithm and the graph data structure itself. In the above table, the overall average difference amounts to only 1.5 edges and 14 overall points.

The `FOOTBALL` program determines a path much more quickly than the `Football.java` program, which is to be expected. The `FOOTBALL` program is written in `CWEB` (which reduces to C), whereas the `Football.java` program is of course written in Java. The executable produced by an optimizing C compiler will easily out-perform the Java Virtual Machine interpreter by multiple orders of magnitude. In fact, the results of Knuth's `FOOTBALL` program that appear in the above table all appeared instantaneously (i.e., under 5 seconds), whereas the `Football.java` program started to slow down substantially when `w` was only 20. The following table summarizes the approximate time taken by the `Football.java` program based on the width `w` (note that these results were obtained on a SPARCstation Ultra2 named `dishwasher.cs.rpi.edu` in the middle of the night):

`w`	**Elapsed Time (in seconds)**
1	7
2	14
3	22
4	29
5	36
10	69
20	134
50	340
100	692
200	1431
500	3566

Note that the time measures are fairly proportional to the value of `w`. Referring back to §3.10.6.4 of this document, Knuth described his empirical findings in which the elapsed time remains approximately proportional to the value of `w` for awhile, and then changes to become approximately proportional to $\mathtt{w}^2$. As is evident in the above table, the time measures have not even entered the proportional to $\mathtt{w}^2$ phase, and already the program is taking 3566 seconds (1 hour) to execute with a width of 500.

4.2.2 Implementing the `Football.java` Program

The implementation of the `Football.java` program is fairly straightforward. The program reads in the `games.dat` file and constructs the corresponding football graph from the given data. Once constructed, the program calls either the `findLongestPath()` or `findLongestPath2()` method of the static `Algorithms` class to determine the longest path between two football team vertices.

The first section of the `games.dat` file contains a summary of each of the 120 football teams. To encapsulate the data provided, the `FootballTeam` class has been designed and implemented as part of the `Football.java` program. As described in the `GB_GAMES` program of **[Knut93]**, each of the first 120 lines of the `games.dat` file describes a single football team by identifying the team's name, nickname, corresponding abbreviation, and the conference to which it belongs.[80]

80. Note that there is other information included in each of the first 120 lines of the `games.dat` file that is not used by the `Football.java` program. Refer to **[Knut93]** for more information.

As an example, consider Stanford and Harvard again. The `games.dat` file represents Stanford and Harvard as follows:

```
STAN Stanford(Cardinal)Pacific Ten;4,;,
HARV Harvard(Crimson)Ivy;,;,
```

The corresponding `FootballTeam` class that is used to represent a single football team in the `Football.java` program is defined as follows:

```
class FootballTeam implements Hashable
{
   public Str abbr;
   public Str name;
   public Str nickname;
   public Str conference;

   public FootballTeam()  { }

   public FootballTeam( String abbr, String name, String nickname,
                        String conference )
   {
      this.abbr = new Str( abbr );
      this.name = new Str( name );
      this.nickname = new Str( nickname );
      this.conference = new Str( conference );
   }

   public int hash()  { return abbr.hash(); }
   public int hash( int tableSize )  { return abbr.hash( tableSize ); }

   public boolean isEqualTo( Comparable C )
   {
      return abbr.isEqualTo( ((FootballTeam)C).abbr );
   }

   public boolean isLessThan( Comparable C )
   {
      return abbr.isLessThan( ((FootballTeam)C).abbr );
   }

   public String toString()  { return name + “ ” + nickname; }
}
```

As indicated above, the `FootballTeam` class implements the `Hashable` interface since instances of this class will be associated with `Vertex` objects. The given `hash()` methods simply apply the standard `Str` class `hash()` methods to the `abbr` attribute. Since each of the 120 football team abbreviations is unique, each may be used as a unique key to identify each `Vertex` object.

The code used to parse the first 120 lines of the `games.dat` file and construct the 120 corresponding `FootballTeam` class instances is not very interesting and is therefore not included in this document.[81] Once these `FootballTeam` objects are constructed, they are associated with `Vertex` objects and added to the football graph. The next task of the `Football.java` program is to continue parsing the `games.dat` file to obtain information about each game that has been played.[82]

Although the game-parsing process is also fairly trivial and uninteresting and is therefore not included in this document, the `Game` class that has been designed and implemented as part of the `Football.java` program is defined as follows:

```
class Game implements Hashable
{
   private static Sequence S = new IntSequence();
   private static java.util.Enumeration e = S.sequence();
   public Int key;
   public Str date;
   public int startVertexScore, endVertexScore;
   public Vertex hometeam;

   public Game( String date, int startVertexScore, int endVertexScore,
                Vertex hometeam )
   {
      this.key = (Int)e.nextElement();
      this.date = new Str( date );
      this.startVertexScore = startVertexScore;
      this.endVertexScore = endVertexScore;
      this.hometeam = hometeam;
   }

   public int hash()  { return key.hash(); }
   public int hash( int tableSize )  { return key.hash( tableSize ); }
   public String toString()  { return "<" + key + ">"; }

   public boolean isEqualTo( Comparable C )
   {
      return key.isEqualTo( ((Game)C).key );
   }

   public boolean isLessThan( Comparable C )
   {
      return key.isLessThan( ((Game)C).key );
   }
}
```

81. As per usual, refer to the website associated with this project for the full source code.
82. Refer to program item 12 of the `GB_GAMES` program on page 227 of **[Knut93]** for detailed information regarding the format used to represent a single game between two football teams.

As with the `FootballTeam` class, the `Game` class implements the `Hashable` interface and therefore provides a default no-argument `hash()` method that will uniquely identify each directed edge of the associated football graph. To ensure that each constructed instance of the `Game` class consists of a unique `key` attribute, a static `IntSequence` instance and corresponding Java `Enumeration` object are part of the `Game` class.

The remaining code that actually calls either the `findLongestPath()` or `findLongestPath2()` method is fairly trivial and uninteresting and is therefore not included in this document.[83]

83. Again, refer back to §3.10.6.1 and §3.10.6.3 for a description of the `findLongestPath()` and `findLongestPath2()` methods, respectively.

5 Acknowledgments

My Masters Project has been “in the works” for the past three years and has experienced three separate incarnations. Attempting to complete, or even begin, a substantial and worthwhile Masters Project while working full-time in a software development department of an international company is an impossibility. I started at IFS International, Inc. as a part-time programmer back in June, 1994. In January of 1996, I decided to go full-time at IFS since I had already completed the necessary course-work for the Masters Degree at Rensselaer. Time passed. Masters Project ideas came and went.

Finally, I decided to terminate my employment with IFS in December of 1997. This decision and subsequent action was necessary for a number of reasons, one of which was to allow me the time and energy to dedicate to completing this project.

I’d like to thank Moorthy for always being so patient and forgiving as I repeatedly postponed meeting after meeting with him over the past few years. Always greeting me with a smile and providing different ideas and perspectives, Moorthy has continued to keep my morale and motivation levels high. Thanks Moorthy!

I’d like to also thank my parents, Richard and Christine, for always asking me about how my Masters Project was progressing, or when I was going to start! The emotional and financial support that they have given me throughout my years at Rensselaer dating as far back as 1990 when I began as a freshman is immeasurable. And just think, there are three more kids to put through college!

Most importantly, I’d like to thank Katrina, my wife, for the unending love and support that she’s given me during this semester and the past decade in which we’ve been together. She sure put up with a lot of my madness during the time I was working at IFS, especially the final year 1997. As this semester comes to an end, we’re looking forward to my taking a position at PSINet, Inc., which is actually right next door to IFS. After what I’ve learned about life over the past few years, my career at PSINet will always be lower on the totem pole than the time I spend with my wife and our family of five cats.

6 Bibliography

[Baas88] Sara Baase, Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley Publishing Company, 1988 2nd edition. ISBN 0201060353.

[Corm92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms. McGraw Hill Book Company, 1992 8th printing. ISBN 0070131430.

[Dewd89] A. K. Dewdney, The Turing Omnibus: 61 Excursions in Computer Science. Computer Science Press, 1989. ISBN 0716781549.

[Dewd93] A. K. Dewdney, The New Turing Omnibus: 66 Excursions in Computer Science. W. H. Freeman and Company, 1993. ISBN 0716782715.

[Elli90] Margaret A. Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual. Addison-Wesley Publishing Company, 1990. ISBN 0201514591.

[Evan92] James R. Evans and Edward Minieka, Optimization Algorithms for Networks and Graphs. Marcel Dekker, Inc., 1992 2nd edition, revised. ISBN 0824786025.

[Flan97] David Flanagan, Java in a Nutshell. O'Reilly & Associates, Inc., 1997 2nd edition. ISBN 156592262X.

[Hors97] Cay Horstmann and Gary Cornell, Core Java 1.1: Fundamentals (Volume 1). Prentice Hall, 1997. ISBN 0137669577.

[Knut93] Donald E. Knuth, The Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press, 1993. ISBN 0201542757.

[Knut97] Donald E. Knuth, The Art of Computer Programming: Fundamental Algorithms (Volume 1). Addison-Wesley Publishing Company, 1997 3rd edition. ISBN 0201896834.

[Knut98a] Donald E. Knuth, The Art of Computer Programming: Seminumerical Algorithms (Volume 2). Addison-Wesley Publishing Company, 1998 3rd edition. ISBN 0201896842.

[Knut98b] Donald E. Knuth, The Art of Computer Programming: Sorting and Searching (Volume 3). Addison-Wesley Publishing Company, 1998 2nd edition. ISBN 0201896850.

[Muss96] David R. Musser and Atul Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison-Wesley Publishing Company, 1996. ISBN 0201633981.

[Plau92] P. J. Plauger, The Standard C Library. Prentice Hall, Inc., 1992. ISBN 0131315099.

[Plau98] P. J. Plauger, Alexander Stepanov, Meng Lee, and A. Alexander, The Standard Template Libraries: A Definitive Approach to C++ Programming Using STL. Prentice Hall, 1998. ISBN 0134376331.

[Sedg88] Robert Sedgewick, Algorithms. Addison-Wesley Publishing Company, 1988 2nd edition. ISBN 0201066734.

[Stro97] Bjarne Stroustrup, The C++ Programming Language. Addison-Wesley Publishing Company, 1997 3rd edition. ISBN 0201889544.