UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A Generative Graph Template Toolkit (GraTe-Tk) for C++

By Theodore Koopman

Supervisor — Bruce Watson
Co-supervisor — Derrick Kourie

Submitted in partial fulfilment of the requirements for the degree

Magister Scientia (Computer Science)

in the Faculty of Computer Science

University of Pretoria

15 September 2009

© University of Pretoria

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A Generative Graph Template Toolkit (GraTe-Tk) for C++

Abstract

This study provides an investigation and discussion into the construction of a graph toolkit using generative programming techniques. It defines and analyses directed graphs, representations and identifies different techniques that may be used to discover new representations.

Each representation is classified according to a unique classification system. Doing this enables us to uniquely identify a particular type of graph representation. A naming convention is used when identifying each graph representation which is a direct by product of the classification system.

Details of how to implement the graph toolkit is presented and analysed. The toolkit is discussed and critically analysed with other major toolkits currently available today such as the Boost and Leda graph toolkits.

Acknowledgements

I would like to acknowledge:

- Firstly, the Higher Power who has given me the strength and courage to persevere.
- The love and support from my family and friends.
- All researchers whom I have referenced.
- My supervisor and co-supervisor for their unending guidance and support.
- All work colleagues who provided me with their support.

Theodore Koopman, 15th September 2009, South Africa

Contents

1 Introduction **10**
1.1 Purpose and Background 10
1.2 Objectives 11
1.2.1 Primary objectives 11
1.2.2 Secondary objectives 11
1.3 Terminology 11
1.3.1 Generic Functions and Classes 11
1.3.2 Generative Classes 12
1.3.3 Policies 12
1.4 Document Overview 12
1.4.1 Discussion layout 12
1.4.2 Chapter overview 13

2 Directed Graphs **15**
2.1 Chapter Overview 15
2.2 Directed Graphs Defined 15
2.3 The nature of graphs 16
2.3.1 A graph as vertex and edge tables 16
2.3.2 A graph as an adjacency matrix 17
2.3.3 A graph as a set of triples 17
2.4 Isomorphisms 18
2.4.1 Transpose 18
2.4.2 Reverse 18
2.4.3 Left Associate 19
2.4.4 Right Associate 19
2.5 Graph Operations 20
2.5.1 Definition of variables 20
2.5.2 Insertion 20
2.5.3 Removal 21
2.5.4 Querying 21
2.5.5 In-Domain 23
2.5.6 Size Determination 23
2.6 Effects Of Isomorphisms 23
2.7 Summary 24

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

3 Policies **25**
3.0.1 Function pointers 26
3.0.2 Pointers to members 28
3.0.3 Functors 31
3.1 Insertion policies 34
3.1.1 Container type requirements 34
3.1.2 Element type 34
3.1.3 Insert at-head 41
3.1.4 Insert at-tail 41
3.2 Lookup policies 41
3.2.1 On-found lookup 41
3.2.2 Move to front lookup policy 42
3.2.3 Migrate forward lookup policy 42

4 Linear Graph Representations **43**
4.1 Chapter Overview 43
4.2 Standard structure 44
4.2.1 Feature layout 44
4.3 Vector graph representation 45
4.3.1 Type 45
4.3.2 Fixed 45
4.3.3 Size 46
4.3.4 Insert policy 46
4.3.5 Lookup policy 46
4.3.6 Vector graph representation advantages 46
4.3.7 Vector graph representation disadvantages 47
4.4 Linked list graph representation 48
4.4.1 Type 48
4.4.2 Link type 48
4.4.3 Insert policy 49
4.4.4 Lookup policy 49
4.4.5 Linked list graph representation advantages 49
4.4.6 Linked list graph representation disadvantages 49
4.5 Set graph representation 50
4.5.1 Type 50
4.5.2 Representation 50
4.5.3 Insert policy 50
4.5.4 Lookup policy 50
4.5.5 Set graph representation advantages and disadvantages . 51
4.6 Classification 51
4.6.1 The way that the classification system works 51
4.6.2 T-CA-LC-V-IAH-LOF 52
4.6.3 T-CA-LC-S-IAH-LOF 52
4.6.4 T-CA-LC-L-IAH-LOF 53
4.6.5 T-CA-LC-V-IAT-LOF 53
4.6.6 T-CA-LC-S-IAT-LOF 53
4.6.7 T-CA-LC-L-IAT-LOF 53
4.7 Summary 54

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5 Binary Graph Representations 55
5.1 Chapter Overview . . . 55
5.2 Binary tree graphs . . . 56
5.2.1 Type . . . 56
5.2.2 Traversal . . . 56
5.2.3 Binary tree graph advantages . . . 57
5.2.4 Binary tree graph disadvantages . . . 58
5.3 Linear sorted graph representations . . . 58
5.3.1 Type . . . 58
5.3.2 Container type . . . 59
5.3.3 Insertion policy . . . 59
5.3.4 Lookup policy . . . 59
5.3.5 Sorted linear graph representation advantages . . . 60
5.3.6 Sorted linear graph disadvantages . . . 60
5.4 Classification . . . 60
5.4.1 T-CA-BC-BT-PRT . . . 61
5.4.2 T-CA-BC-BT-POT . . . 61
5.4.3 T-CA-BC-BT-IOT . . . 61
5.4.4 T-CA-SC-V-AI-LL . . . 61
5.4.5 T-CA-SC-V-DI-LL . . . 62
5.4.6 T-CA-SC-V-AI-BL . . . 62
5.4.7 T-CA-SC-V-DI-BL . . . 62
5.5 Summary . . . 62

6 Mapped Graph Representations 64
6.1 Chapter Overview . . . 64
6.2 Single map graph representations . . . 65
6.2.1 Key type . . . 65
6.2.2 Value type . . . 66
6.2.3 Comparison . . . 66
6.2.4 Allocator . . . 66
6.2.5 Advantages of using single mapped graph representations 66
6.2.6 Disadvantages of using single mapped graph representations 67
6.3 Map-of-map graph representations . . . 68
6.3.1 Key type . . . 68
6.3.2 Value type . . . 68
6.3.3 Comparison . . . 69
6.3.4 Allocator . . . 69
6.3.5 Map-of-map graph representation advantages . . . 69
6.3.6 Map-of-map graph representation disadvantages . . . 69
6.4 Classification . . . 69
6.4.1 T-RA-CA-S-M . . . 70
6.4.2 T-LA-CA-S-M . . . 70
6.4.3 T-RA-CA-D-M . . . 70
6.4.4 T-LA-CA-D-M . . . 71
6.5 Summary . . . 71

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7 Hashed Graph Representations 72
7.1 Chapter Overview 72
7.2 Hash function graph representations 73
7.2.1 Type 73
7.2.2 Hash function 73
7.2.3 Bucket type 73
7.2.4 Hash function graph representation advantages 73
7.2.5 Hash function graph representation disadvantages 74
7.3 Direct hashed graph representations 75
7.3.1 Direct hashed graph representation advantages 75
7.3.2 Direct hashed graph representation disadvantages 76
7.4 Classification 76
7.4.1 T-CA-HT-F-BC-L-IAH-LOF 76
7.4.2 T-CA-HT-F-BC-B-T 77
7.4.3 T-CA-HT-F-BC-S-M 77
7.4.4 T-CA-HT-D-BC-L-IAH-LOF 77
7.4.5 T-CA-HT-D-BC-B-T 78
7.4.6 T-CA-HT-D-BC-S-M 78
7.5 Summary 79

8 Meta Programming 80
8.1 Chapter Overview 80
8.2 Partial Specialisation 80
8.3 Traits 82
8.3.1 Specialisation 82
8.3.2 Parameterisation 83
8.3.3 Trait lists 83
8.4 Selectors 83
8.5 Assertions 84
8.6 Summary 86

9 Toolkit 87
9.1 Chapter Overview 87
9.2 Features 87
9.2.1 A configurable toolkit 88
9.2.2 Efficiency 88
9.2.3 A common interface 88
9.2.4 Different queries 88
9.3 Software architecture 89
9.3.1 Algorithms 89
9.3.2 Properties 89
9.3.3 Identification constants 90
9.3.4 Error codes, assembler, compile time assertions and the product 90
9.4 Design decisions 91
9.4.1 First consideration 91
9.4.2 Second consideration 92
9.5 Graph construction 95
9.5.1 Defining the problem 95
9.5.2 Define the objective 96

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

9.5.3 Creating the graph 96
9.6 Summary 98

10 Toolkit Comparisons 99
10.1 Chapter Overview 99
10.2 LEDA 99
10.2.1 Simple data types 100
10.2.2 Dictionaries and priority queues 102
10.2.3 Graphs 103
10.3 The Boost Graph Library 105
10.3.1 Types of graphs 106
10.3.2 Construction and usage 108
10.4 The GraTe-Tk vs. LEDA 109
10.4.1 Architecture 110
10.4.2 Representations 110
10.4.3 Usability 111
10.5 The GraTe-Tk vs. the BOOST Graph Library 111
10.5.1 Architecture 111
10.5.2 Representations 112
10.5.3 Usability 113
10.6 Summary 113

11 Conclusion 115
11.1 Retrospection 115
11.2 Evaluation of Objectives 115
11.2.1 A different approach 116
11.2.2 Implementation techniques 116
11.2.3 Different representations 116
11.2.4 Naming conventions 116
11.2.5 Secondary objectives 117
11.3 Summary 117

List of Figures

1.1 Unit conversion using a Strategy pattern 13

3.1 On found lookup policy 42
3.2 Move to front lookup policy 42
3.3 Migrate forward lookup policy 42

4.1 Standard feature definition for vectors 44
4.2 High level architecture diagram 45
4.3 Alternative feature definition for vectors 46
4.4 Alternative list features 48
4.5 Alternative feature definition for sets 51
4.6 Linear graph classification machine 52

5.1 Feature diagram for a binary tree graph 56
5.2 Binary tree traversals 57
5.3 Feature diagram for sorted containers 58
5.4 Binary search algorithm for linear containers 59
5.5 Binary graph classification machine 60

6.1 Single mapping illustration 64
6.2 Map-Of-Map illustration 65
6.3 Standard single map features 66
6.4 Example of mapping to a collection of values 67
6.5 Standard map-of-map structure 68
6.6 Map-of-map depicting right and left associations respectively . . . 69
6.7 Mapped graph classification machine 70

7.1 Direct hashing example 75
7.2 Hashed graph classification system 76

9.1 General generative solution architecture 89
9.2 Generative pipeline solution architecture 91
9.3 Neural network for OR operation 95

10.1 Common Point heterogeneous Graph 104

List of Tables

3.1 Not grouped triple: (s,e,d) . . . 35
3.2 Not grouped triple: (e,s,d) . . . 35
3.3 Not grouped triple: (d,e,s) . . . 36
3.4 Not grouped triple: (e,d,s) . . . 36
3.5 Not grouped triple: (d,s,e) . . . 36
3.6 Not grouped triple: (s,d,e) . . . 36
3.7 Left grouped (Left associated) triple: ((s,e),d) . . . 37
3.8 Left grouped (Left associated) triple: ((e,s),d) . . . 37
3.9 Left grouped (Left associated) triple: ((d,e),s) . . . 37
3.10 Left grouped (Left associated) triple: ((e,d),s) . . . 37
3.11 Left grouped (Left associated) triple: ((d,s),e) . . . 38
3.12 Left grouped (Left associated) triple: ((s,d),e) . . . 38
3.13 Right grouped (Right associated) triple: (s,(e,d)) . . . 39
3.14 Right grouped (Right associated) triple: (e,(s,d)) . . . 39
3.15 Right grouped (Right associated) triple: (d,(e,s)) . . . 39
3.16 Right grouped (Right associated) triple: (e,(d,s)) . . . 40
3.17 Right grouped (Right associated) triple: (d,(s,e)) . . . 40
3.18 Right grouped (Right associated) triple: (s,(d,e)) . . . 40

9.1 Graph element types . . . 97

Chapter 1

Introduction

1.1 Purpose and Background

Graph theory has been around for a number of years. It is a discipline in mathematics with important applications to computer science. Applications such as:

- Directed Acyclic Graphs that are used in compiler construction to identify common sub-expressions in an expression. [ASU88]
- Finite automata that may be used in pattern matching applications e.g. regular expressions [ASU88].
- Artificial neural networks (used in artificial intelligence) that consist of a number of processing elements (neurons) connected with adjustable weights forming the edges between the processing elements. [ESD96]
- Networking fields such as transport e.g. Global Positioning Satellite, telecommunication, computer network applications e.g. the Internet consisting of interconnected web pages.

With a variety of graph applications it would be useful to have a reusable software library that facilitates the construction of the applications. The software library/toolkit allows the user of the toolkit to manage and manipulate the nodes and edges forming the graph. Even though a variety of graph toolkits are already in existence, it remains interesting to explore different methods of implementing such a toolkit. One of several approaches may be followed when building the toolkit. One also needs to consider the target programming language that will provide the toolkit. In this dissertation we have chosen to build a graph toolkit that is both generic and generative (these terms are explained in section 1.3) and deviates from classical graph implementation methods. The target programming language was chosen from a variety of languages. Selection was based on the ability of the programming language to support generic and generative programming techniques i.e. meta-programming (described in chapter 8) and its popularity in the software development domain. At the start of this project, C++ stood out from Ada95, Modula-3 and Java as a good candidate and was adopted as the target language for this software toolkit. However,

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

other languages and improvements to existing languages are emerging and may be considered as possible candidates for future research.

1.2 Objectives

This dissertation has several objectives that it hopes to achieve. These objectives have been separated into primary and secondary objectives.

1.2.1 Primary objectives

The primary objective of this thesis is to discover a different approach to implementing the graph data structure. The focus shall be on directed graphs and how to implement them in C++ using static meta-programming techniques. We concentrate on providing a generic and generative toolkit that provides access to directed graph representations. We also aim to find a method of naming the graph representations, such that it is easy to identify the graph representation as well as being able to configure the exact representation by looking at the name.

1.2.2 Secondary objectives

Instead of implementing a set of algorithms immediately, we shall provide a set of querying functions. These functions will later be used when developing the graph algorithms. A querying function would typically provide a set of nodes, edges or a combination thereof while requiring a given node, edge or a combination of the two. The toolkit shall be designed to be easy to use in terms of construction. The user of the toolkit shall therefore be required to provide a set of requirements necessary to construct a graph with a specific representation.

1.3 Terminology

We assume the reader is familiar with basic programming concepts from object oriented programming. Here we describe a list of terms used throughout the document pertaining to generic and generative programming techniques.

1.3.1 Generic Functions and Classes

We use the term generic to classify types of functions and classes (object types).

A function (also referred to as a method or operation in other literature) or a class can be defined as being generic when it relies on its use of generic types for its implementation [CE00]. A generic function or class facilitates reuse by minimizing code duplication.

In general, a function implements an algorithm. A generic function implements its algorithm in such a way that the algorithm is type independent, because it operates on a generic type.

When this logic is applied to a class, the methods of the class become type independent. All the methods of the class operate in a generic way in response to the underlying type or types provided as inputs to the class. The class is therefore generic because it operates on any type.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

1.3.2 Generative Classes

A generative class or generative classes relates to the constructing of software system families using loosely coupled data or software components at a high level of abstraction. A **generative** toolkit differs from a **generic** toolkit in a certain way: The generic toolkit focuses on letting an algorithm work for arbitrary types whereas a generative toolkit provides specific algorithms that work with arbitrary types and are selectable by the compiler at compile time. Due to the two concepts differing, they may be combined to be more constructive.

1.3.3 Policies

A policy is defined as the specialisation unit for generic classes that allows the methods of the class to be customised at compile time. A policy emphasises behaviour rather than type and is therefore similar to the **Strategy** design pattern (with the exception that a policy is bound at compile time whereas the traditional mechanism allows for runtime binding).

Figure 1.1 depicts four classes: **Translator**, **FeetToMeters**, **MetersToFeet** and **Converter**. This example briefly illustrates the **Strategy** design pattern implemented using runtime binding (also referred to as dynamic or late binding in other literature). The Converter class provides the service of converting from one unit to another such as meters to feet and feet to meters. It does this by means of a **Translator**. The **Translator** provides an interface (the **translate** method) for doing the conversion. Subclasses inheriting from the **Translator** class provide a specialised service. The two concrete classes (**MetersToFeet** and **FeetToMeters**) inherit from **Translator** and each provide a specialised conversion routine. Through aggregation the **Converter** can change its translation policy via its **set** method thereby allowing it to translate a value from one unit to another at runtime.

A policy is implemented in a similar way except that the translation units (**MetersToFeet** and **FeetToMeters**) are provided as input to the template class[1]. The policy is bound at compile time to the template class and therefore cannot change during runtime. While the trade-off here is flexibility the advantages are greater efficiency and reliability [2].

In chapter 3, policies are described in more detail with C++ examples illustrating the various forms of policies. Thereafter a list of the different policies used in our toolkit are described.

1.4 Document Overview

This section briefly describes the layout of the remainder of the document.

1.4.1 Discussion layout

Hereafter we discuss different types of containers (a collection of types). Each chapter has a common layout and discussion pattern. This pattern is defined

[1] A template function/class is a generic function/class in C++

[2] A policy is more efficient because the class size could be reduced by removing the **set** method. Reliability is improved through composition of the translation unit instead of aggregation because the need for safety checks on the dynamic object can be removed

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

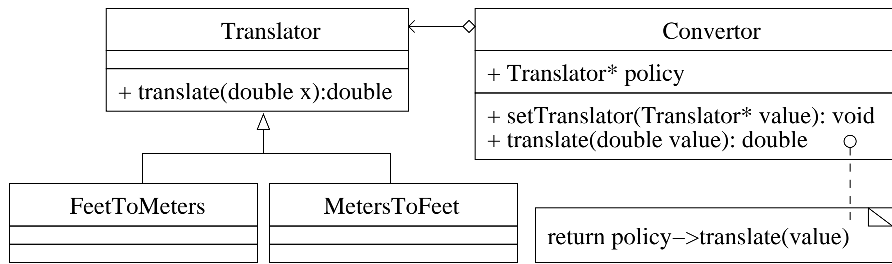


Figure 1.1: Unit conversion using a Strategy pattern

as follows:

1. Providing a proposed structural layout of the generic container,
2. Defining the structural layout of the generic container employed by our toolkit,
3. Defining the characteristics of the containers discussed in the chapter in terms of advantages and disadvantages,
4. Defining the classification system for the graph representations applicable to the containers discussed in the chapter and
5. An extract of possible graph configurations applicable to the chapter.

1.4.2 Chapter overview

Chapter 2 provides a definition of directed graphs (also referred to as digraphs Interchangeably in this dissertation). In chapter 3, we discuss policies again with using C++ to illustrate different types of policies. In chapter 4, linear containers are discussed. Our attention here is on vectors, sets and linked lists. Binary containers are dealt with in chapter 5 , specifically binary trees are discussed here. Chapter 6 deals with mapped graph representations (A dictionary implementation of graphs). Two types of mapped graph representations are examined in the context of the toolkit, namely:

- Single maps and
- Map-of-maps (also known as two-maps).

Chapter 7 provides a discussion on hashed graph representations. Here two types of representations are discussed:

- Hashed graph representations (graphs using a hash function) and
- Direct hashed graph representations (graphs that do not require hash functions).

Chapter 8 provides a brief discussion on meta-programming. The techniques that are outlined are explained using general examples. These techniques shown are used by the toolkit and thus illustrate how the objectives of the toolkit may be achieved by using meta-programming technology.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

In chapter 9 we discuss the toolkit. Here the toolkit shall be analysed in terms of its features, architectural layout and the graph construction process shall be explained by means of an example. In essence, this section focuses on how to use the toolkit.

In chapter 10, two popular graph toolkits are identified in terms of what each provides and possible improvements that could be implemented in future. We also highlight the differences between these two toolkits and our toolkit identifying extension points in our toolkit. The libraries being compared are:

- Boost and
- LEDA.

Finally, the document shall be concluded with an examination of the objectives set and the those objectives that were met. We also provide a summary of lessons learned.

Chapter 2

Directed Graphs

2.1 Chapter Overview

In this chapter we define directed graphs[1]. We also use this chapter to form a basis for discussions in future chapters. Reading this chapter is important because it is the focal point of the research that lays the foundation to understanding subsequent chapters. We begin by defining what exactly a digraph is and what we mean when discussing digraphs. Alternative approaches for representing digraphs are communicated. These alternative approaches relate to how directed graphs are represented in terms of its implementation method. There is an introduction to and explanation of isomorphisms. This effectively explains what is considered as an isomorphism in the context of this dissertation. We also provide a list of supported isomorphisms and describe each of these. Thereafter the operational interface of the generic graph toolkit is provided. The operational interface is described in terms of pre-conditions and post-conditions for each accessible operation provided by the toolkit. We finally conclude the chapter by an examination and evaluation of the chapter.

2.2 Directed Graphs Defined

Traditionally, digraphs are defined as a nonempty set of vertices with pairs of directed edges [CL86]. This approach separates the vertices (referred to hereafter as nodes) from the edges (also known as arcs). Consider for example a graph G, with a set of nodes N(G)={x, y, z}. Nodes are connected by node pairs (forming an edge), for example: E(G)={(x,z), (x,y)}. Thus the graph is defined by the set of nodes N and the set of edges E. This mathematical representation may be implemented in a programming language, which in our case this would be C++.

Another way of representing graphs is through an adjacency matrix. The representation as the name suggests, makes use of a matrix consisting of source and destination nodes labeling columns and rows. The order of these labels is of little importance. For instance source nodes may be represented by columns and destination nodes may be represented by rows. Indeed the opposite is also

[1]Directed graphs are also known as digraphs in some literature however we will refer to them interchangeably as digraphs or simply graphs

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

possible, in which case the source nodes are represented by rows instead of columns and the destination nodes are represented by columns instead of rows. The intersection between the column (source node) and row (destination node) result in the formation of an edge between the source node and destination node.

Alternatively, a graph may be represented as a set of triples. The triples contain the edge and node data that define a single entry in the graph. A triple is defined by two nodes [2] (a source node and a destination node) and an edge that connects the two nodes.

$$T=\{s, e, d\} \tag{2.1}$$

Where **s**, represents the source node, **e** represents the edge and **d** represents the destination node. The graph is defined as a finite set of these triples limited by n:

$$G = \{T_0, ..., T_{n-1}\} \tag{2.2}$$

2.3 The nature of graphs

A word or two is necessary regarding the nature of directed graphs. In the previous section we outlined three type of representations for graphs namely:

- Graphs as vertex and edge tables,
- Graphs as an adjacency matrix and
- Graphs as a set of triples.

2.3.1 A graph as vertex and edge tables

In the first form we notice that graphs are represented by their vertices and edges by using separate tables. It is possible that this form of graph is represented as either a vector or linked list or perhaps a set implemented as either a vector, linked list or binary tree. This form of graph may be dense or sparse in nature depending on the number of elements being stored and the physical representation thereof.

If for instance the graph is represented in terms of a vector, a fixed amount of memory is allocated for the graph. Whether the number of nodes and vertices in the graph occupy the entire memory that has been allocated is of great significance as it indicates the nature of the graph. If all the memory has been occupied i.e. there is no resource waste occurring then the graph is said to be dense in nature. If however only a certain amount of memory is occupied and there are gaps within the data collection (also known as container) either due to the way that the graph was constructed or the way in which the elements were removed from the graph, the graph is said to be sparse.

[2] The implications for using this triple as a basis for taxonomizing graph representations has been explored in [BS03] and [BSWK04]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A method to ensure that the graphs are always dense is to represent them as either binary trees or linked lists. However there may be certain trade-offs when dealing with these representations.

2.3.2 A graph as an adjacency matrix

In the second instance, the graph is represented as an adjacency matrix. The word matrix suggests the use of a vector (also known traditionally as an array) as the primary container used to store information since the vector data structure is capable of providing direct access using direct element access methods through pointer arithmetic and special programming language operators. From the first type of graph we noticed that vectors are very easily subjected to being sparse in nature — sparse either through construction or through some form of post modification such as element removal. As a consequence of this data structure, the adjacency matrix possesses this sparse nature. In very few cases however it is possible to have a dense adjacency matrix. Consider, for example, a travel chart representing distances between cities where the names of the cities are the nodes that are listed by column and row. The place where the row and column coincide and they are the same (the source node is the same as the destination node or the starting city is the same as the end city) the cell is kept blank. This would serve either as an indication that the cell is not used or has a distance of zero. Having a distance of zero indicates that the cell is used (as it contains valid data). In the case where the cell is blank, a condition arises where the matrix is defined as sparse (as it contains wasted space). However if the cells were instead occupied with data (filled with zero) the matrix would be categorised as being dense because it hasn't wasted any space.

Does the matrix not always contains values anyway? The answer is yes. So then why is it considered sparse? We consider it to be sparse because those values are not part of the graph and only valid values that are part of the graph (nodes or edges in this case with valid values) are considered. Other values would indicate empty space (an unused cell). For example, when specifically dealing with a distance matrix, if empty spaces are indicated by the value *-1* then the distance between Pretoria and Pretoria is indicated as *-1*, implying that the cell is not in use as opposed to the distance between Pretoria and Johannesburg indicated by the value *50*.

2.3.3 A graph as a set of triples

In this type of graph, we cater for sparse graphs and dense graphs through the different types of representations (as will be seen later). In some cases it is necessary and quite unavoidable to have sparse graphs. An example of this is a hash table implemented as a growing vector. A hash function may in fact produce an index that specifies an entry at position 100. Later the hash function may produce an index that specifies an entry at position three. This hash function has now created a sparse graph. It is possible however to represent the graph as a dense collection by using a linked list. However this would remove the fast random access ability that is a key ingredient to hash tables. This trade-off may be acceptable in some cases. However the representation would then be more adequately defined as a hash list instead of a hash table since the hash

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

function then produces an index into a linked list that has to be searched for in a linear manner.

2.4 Isomorphisms

The triple may be represented in several ways. At this point isomorphisms are introduced. An isomorphism is defined as *having an exact corresponding form* [Dig94]. From this we introduce four isomorphic operations that may be applied to triples:

1. Transpose,
2. Reverse,
3. Left associate and
4. Right associate

These isomorphisms are applied to some form of triple or graph layout to produce a new form of triple or graph layout. They change the representation in terms of memory layout. They also change the manner in which the triple component is referenced during searches.

Isomorphisms are a compile time notion — that is to say that they affect the structure of the graph when the program is compiled. Later in this chapter we shall look at pitfalls of using isomorphisms.

2.4.1 Transpose

Synopsis

Swaps the first two components of the triple **T** (refer to 2.1) .

$$I_t(T) = \{e, s, d\} \tag{2.3}$$

Characteristics

This isomorphism has the following notable properties:

- A subsequent application of the transpose operation cancels out the effect of the first transpose operation.

2.4.2 Reverse

Synopsis

Swaps the first and the last component of the triple **T** (refer to 2.1).

$$I_r(T) = \{d, e, s\} \tag{2.4}$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Characteristics

Characteristics of this isomorphism is equivalent to the characteristics of the transpose isomorphism.

- A Transpose operation followed by a reverse operation followed by a transpose operation followed by a reverse operation does not necessarily hold the original configuration. In order to cancel each operation out, each type of isomorphism needs to be applied in direct succession or in equal reverse order. For example: Transpose followed by reverse followed by reverse and followed again by transpose would yield the original configuration.

2.4.3 Left Associate

Synopsis

An association is formed with the first two components (normally the source node and edge) of the triple **T** (refer to 2.1), resulting in a combination of the association and the remaining component.

$$I_{la}(T) = \{(s, e), d\} \tag{2.5}$$

Characteristics

This isomorphism has the following notable properties:

- Subsequent applications of this isomorphism does not result in the original configuration.
- It is not possible to return to the original configuration through this isomorphism.

2.4.4 Right Associate

Synopsis

An association is formed with the last two components (normally the edge and destination node) of the triple **T** (refer to 2.1), resulting in a combination of the association and the remaining component.

$$I_{ra}(T) = \{s, (e, d)\} \tag{2.6}$$

Characteristics

This isomorphism has the following notable property:

- It is possible to apply with effect the isomorphisms: transpose, reverse and left associate.

In addition to the above property, this isomorphism has the same characteristics as specified by the left associate isomorphism.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2.5 Graph Operations

The graph toolkit provides basic methods for:

- inserting,
- removing,
- data lookup,
- data querying and
- size determination.

The graph operations are generic as they operate on any underlying representation. The underlying algorithm for each operation differs depending on the underlying digraph representation. For this reason a specification shall be used to define an operation due to the inputs (pre-conditions) and the outputs (post-conditions) remaining independent of the underlying algorithm.

2.5.1 Definition of variables

In the following sections we define each operations contract in terms of pre and post conditions. The specification language used is a form of predicate calculus as described in [Mor98]. The general form of these specifications is defined as:

$$x_1 ... x_n : [Pre, Post] \tag{2.7}$$

Where $x_1 ... x_n$ are so-called frame variables and Pre and $Post$ are the pre- and postcondition of the specification respectively. The specification indicates that if the precondition is true prior to the relevant operation, then the post-condition will be true after it is applied and that to meet the postcondition, frame variables may be altered.

2.5.2 Insertion

Element insertion deals with adding elements to the graph. Since the graph may be represented using a variety of methods (as shall be seen in subsequent chapters), the addition of triples become container type specific. For instance the method of inserting elements into a vector differs from inserting elements into a map. This is why generative programming plays such a crucial role in having a generic graph class with these numerous representations (see section 8). Although the insertion of methods may differ, the inputs and outputs remain fixed. The following specification defines the requirement of the graph insertion facility:

$$s, e, d, G : [\forall x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,e,d} \neq y_{s,e,d}, \exists x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,e,d} = y_{s,e,d}] \tag{2.8}$$

Where $x_{s,e,d}$ and $y_{s,e,d}$ are triples not part of the graph G and triples part of the graph G respectively.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2.5.3 Removal

Like insertion, removal of elements from the different types of containers may also be container specific. The resulting output stays constant, which is to have the desired element taken out of the underlying container. This allows us to use the following specification to define the remove operation that is applicable to an arbitrary container housing the graph elements.

$$s, e, d, G : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,e,d} = y_{s,e,d}, \forall x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,e,d} \neq y_{s,e,d}] \tag{2.9}$$

2.5.4 Querying

A useful set of operations applicable to digraphs would be to determine the list of edges from a particular source node or destination node. The querying operations provide this functionality. The set of queries that would be useful are described and defined below:

Set of Edges and Destination Nodes

This query allows the retrieval of edges and destination nodes where there is a common source node. This particular type of query would be useful in a finite automaton that needs to determine what the next possible transitions are going to be and where they lead. A specification defining such a query is defined here.

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_s = y_s, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_{e,d} = x_{e,d} \wedge z_s = x_s] \tag{2.10}$$

Set of Source and Destination Nodes

This query allows the retrieval of source and destination nodes where there is a common edge. This query is useful when a particular transition (transition from one node to another i.e. the transition from a source node to a destination node) is known and the set of nodes that apply to that transition is required. An example of this would be a street guidance system that allows the user to specify a street name and in return receive a list of starting addresses that lead to the ending address with that street name.

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_e = y_e, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_{s,d} = x_{s,d} \wedge z_e = x_e] \tag{2.11}$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Set of Source Nodes and Edges

This query allows the retrieval of source nodes with adjacent edges. This type of query is particularly useful when the destination is known and the transition and the source of the transition is not known. In a flight management system, the final destination may be known but how to get there has to be determined. This query serves to resolve this problem. The specification defining this operation is defined here:

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_d = y_d, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_{s,e} = x_{s,e} \wedge z_d = x_d] \tag{2.12}$$

Set of Source Nodes

This query retrieves a set of all source nodes that belong to a certain edge and destination node. This type of query is useful when a particular method of transition with its outcome is known and the cause of the outcome needs to be determined. An example of this would be a chemistry system that maintains a cause and effect relationship based on some chemical reaction. The reaction (transition) is known as well as the effect (destination). The query is used to retrieve a number of possible causes for the reaction-effect relationship. The specification governing this query is defined as:

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_{e,d} = y_{e,d}, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_s = x_s \wedge z_{e,d} = x_{e,d}] \tag{2.13}$$

Set of Edges

This query retrieves a set of edges common to both a source and destination node. This type of query is useful in street navigation system to determine the path to take to get from a start point to an end point. The path (edge) is unknown and is determined by finding the common source and destination nodes. The specification for this query is defined by:

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,d} = y_{s,d}, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_e = x_e \wedge z_{s,d} = x_{s,d}] \tag{2.14}$$

Set of Destination Nodes

This query retrieves a set of destination nodes based on a source and an edge. This query is useful in an illness classification system where the symptom coupled with a verification flag determines the illness. The specification for this query is defined by:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

$$s, e, d, G, S : [\exists x_{s,e,d}, y_{s,e,d} : G \cdot x_{s,e} = y_{s,e}, \exists x_{s,e,d} : G \cdot \exists z_{s,e,d} : S \cdot z_d = x_d \wedge z_{s,e} = x_{s,e}] \tag{2.15}$$

2.5.5 In-Domain

The in-domain query determines the existence of a particular graph element. It returns true if the element is within the domain and false otherwise. This query is used to determine if at least one graph element exists in the graph. A subsequent call to the relevant type of retrieval query would extract a set of data that may become useful information. The specification for this query is defined by:

$$s, e, d, G : [true, (\exists x_{s,e,d} \cdot G(x_{s,e,d})) \vee \neg(\exists x_{s,e,d} \cdot G(x_{s,e,d}))] \tag{2.16}$$

2.5.6 Size Determination

The size determination operation determines the size of the underlying data container. It determines the size of the graph by measuring the number of graph elements it stores. The specification for this query is defined by:

$$G : [true, \sum_{i=0}^{n} G = i + (i+1) + ... + (i+n)] \tag{2.17}$$

2.6 Effects Of Isomorphisms

Although isomorphisms have this automatic configuration transformation ability, they have the following impact:

> It affects the order in which the elements are specified in the above operations. Take for example an original configuration consisting of the types $int, string, int$. After a transpose operation the configuration looks as follows: $string, int, int$. This is not a problem but it does require that the user of the toolkit knows what the final representation configuration is which is in this case $string, int, int$ and not $int, string, int$. This is important as it affects every graph operation since the graph operation signatures are based on generic type definitions.

It is possible however to develop the graph toolkit with this in mind and maintain the original configuration in the operation signature however this results in a more complex and error prone design. Not only is the implementation more complicated but the toolkit becomes more difficult to maintain. In addition, a technique is required to track what the types are for the current configuration as well as how they relate to the original configuration.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2.7 Summary

In this chapter we defined the directed graph. We stated what it was and gave examples of three different forms and implementation techniques for directed graphs. We identified the more traditional approaches and branched off with a less common approach to dealing with the problem which is to implement graphs using a collection of triples. Next we took a look into the nature of graphs in terms of density and the cases that determine when the graph is dense and when it is sparse. The section that followed was a discussion on isomorphisms. A group of isomorphisms were listed and defined. These isomorphisms serve as graph mutation mechanisms. They change the underlying memory layout of the graph thereby in effect changing the representation of the graph. The isomorphisms behave on structure and was noted as being compile time based. We also defined the characteristics of the different isomorphisms emphasising behavioural aspects with regards to particular types of isomorphisms. We saw how chaining isomorphisms are possible and noticed the some of the pitfalls associated with mutating the representations. What followed was a discussion on the graph interface in terms of the operations. These operations were defined and discussed by means of operation specifications. It was shown that the graph queries were applicable and usable to several different application areas. In the chapters to follow we look at the different types of directed graphs in the context of the toolkit. The graph operations introduced in this chapter can be invoked using a generic method signature that is independent of the underlying representation.

Chapter 3

Policies

As a broad definition, the Oxford English dictionary defines a policy as a course of action adopted by a government, party or individual. In the current context, a policy is defined as the method of implementation provided by a policy class (PC). The PC is a class consisting of the method implementing the policy as well as attributes, helper methods and functions. A class that uses a PC is considered to be a policy user.

Policies are equivalent to the strategy design pattern as described in [GHJV94]. The strategy design pattern allows the construction of a family of algorithms i.e. a library or toolkit, making the algorithms interchangeable. In the C++ and Ada context, this is also achieved through the use of parameterisation. Making substitutable parts of the algorithm available as parameters to a component, allows the user of the component to construct an object with the desired functionality. This inherently satisfies a given set of requirements.

The important point to note is that the component being constructed through the use of these parameterised policies is statically bound at compile time. This means that the properties (particularly the configured policy) of the component may not change after the component has been constructed - i.e. the algorithms are not interchangeable during runtime but are interchangeable before compile time. One way in which runtime binding may be achieved, is to dynamically construct policy **users** instead of policy **classes**. The policy user is configured to use different policy classes based on runtime requirements i.e. runtime inputs.

Policies introduce the concept of functors[1]. A functor is a specific type of policy class that overloads the parenthesis operator that implements the policy. Invoking the parenthesis operator on an instance of the class, becomes the execution point of the class. Classes that provide this functionality are considered functors or function objects. Functors are useful as call-backs. They are more flexible and easier to implement than conventional call-back functions because only a reference to the instance of the class is needed.

Several methods may be used to implement call-back functions namely:

[1]In C++ specifically methods as well as operators (such as +, -, etc.) can be overloaded and overridden as ordinary methods. In this case we introduce functors that override the parenthesis operator "()". When this is done the instance of the class can be used like a function call, this is the advantage of using functors because the whole class can behave as a single function. This mechanism of making the class behave as a function we term the execution point of the class.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Function pointers
- Pointers to members
- Functors/policies

3.0.1 Function pointers

This approach is separated from the object oriented approaches in the sense that it uses global functions/procedures to achieve its objective. It may however access and manipulate objects through input parameters. As an example, consider two objects:

- A Teacher and
- A Student.

The Teacher teaches a Student. The Teacher gives a lesson and asks if the Student understands. The Student responds with *yes* or *no*. If the response is *yes*, the Teacher continues with the next part of the lesson. If the response is *no*, the Teacher repeats the lesson. In the source listing, the yes and no responses will be simulated using a random function by producing a random number between zero and ten and if that number is even the lesson proceeds otherwise the lesson is repeated. The C++ source listing for this example is provided:

```
#include <stdlib.h>
#include <iostream>

typedef bool (*response_type)(void);

class Teacher
{
  public:
    void teach( void )
    {
      do
        lesson1();
      while( !continue_response() );
      do
        lesson2();
      while( !continue_response() );
    }

    void listener( response_type a_response )
    { continue_response = a_response;
    }

  protected:
    response_type continue_response;
  private:
    void lesson1( void )
    { // lesson 1
      std::cout << "lesson1" << std::endl;
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
    }

    void lesson2( void )
    { // lesson 2
      std::cout << "lesson2" << std::endl;
    }
};

class Student
{
  public:
    bool any_questions( void )
    {
      if( ((rand() % 10) % 2) == 0 )
      {
        proceed();
      }
      else
      {
        repeat();
      }
      return understands;
    }

  protected:
    bool understands;

    void proceed( void )
    {
      understands = true;
      std::cout << "teacher I understand, please continue" << std::endl;
    }

    void repeat( void )
    {
      understands = false;
      std::cout << "teacher I don't understand" << std::endl;
    }

};

// students sitting outside of class
Student mary;
Student jack;

// type of student responses

bool student1Response( void )
{
  std::cout << "mary any questions? ";
  return mary.any_questions();
}

bool student2Response( void )
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
{
  std::cout << "jack any questions? ";
  return jack.any_questions();
}

bool classResponse( void )
{ return student1Response() && student2Response();
}

int main( int argc, char** argv )
{
  Teacher* blind_john = new Teacher();

  std::cout << "begining lesson:" << std::endl;
  blind_john->listener( student1Response );
  blind_john->teach(); // blind john teaches mary, but doesnt know it
  std::cout << "end of lesson" << std::endl << std::endl;

  std::cout << "begining lesson:" << std::endl;
  blind_john->listener( student2Response );
  blind_john->teach(); // now blind john teachs jack (hasn't a clue)
  std::cout << "end of lesson" << std::endl << std::endl;

  std::cout << "begining lesson:" << std::endl;
  blind_john->listener( classResponse );
  blind_john->teach(); // now blind john is teaching everyone...
  std::cout << "end of lesson" << std::endl << std::endl;

  delete blind_john;
  blind_john = NULL;

  return 0;
}
```

In the example, the Teacher gives two lessons. After each lesson there is an interruption where the Teacher expects a response from something. In the first and second instance, the Teacher receives a response from individual students. The third instance the Teacher is teaching a group of students. Each time the Teacher has a different Student base. Each Student's response may differ independently of the lesson being provided i.e. the Teacher's teaching policy changes. Thus the policy class in this example is the call back function type because it allows the teaching algorithm to change dynamically when the listener is changed. The policy user class is the Teacher class because it uses a policy class to change the behaviour of the teaching algorithm.

3.0.2 Pointers to members

The second example uses a more object-oriented approach. There may be two types of implementations here:

- Direct object communication (i.e. without the call-back /policy functionality)

- Object-oriented function pointers (also known as pointers to members [Amm95])

For the purposes of this discussion, we shall focus on object-oriented function pointers. The example is shown:

```
#include <stdlib.h>
#include <iostream>

class Student
{
  public:
    virtual bool any_questions( void )
    {
      if( ((rand() % 10) % 2) == 0 )
      { proceed();
      }
      else
      { repeat();
      }
      return understands;
    }

  protected:
    bool understands;

    void proceed( void )
    {
      understands = true;
      std::cout << "teacher I understand, please continue" << std::endl;
    }

    void repeat( void )
    {
      understands = false;
      std::cout << "teacher I don't understand" << std::endl;
    }

};

class StudentMary: public Student
{
  public:

  virtual bool any_questions( void )
  {
    std::cout << "any questions mary? ";
    return Student::any_questions();
  }
};

class StudentJack: public Student
{
  public:
```

```
  virtual bool any_questions( void )
  {
    std::cout << "any questions jack? ";
    return Student::any_questions();
  }
};

typedef bool (Student::*response_type)(void);

class Teacher
{
  public:
    void teach( void )
    {
      do
        lesson1();
      while( !((a_student->*continue_response)()) );
      do
        lesson2();
      while( !((a_student->*continue_response)()) );
    }

    void listener( response_type a_response )
    { continue_response = a_response;
    }

    void student( Student* child )
    { a_student = child;
    }

  protected:
    bool (Student::*continue_response)( void );
    Student*       a_student;
  private:
    void lesson1( void )
    { std::cout << "lesson1" << std::endl;
    }

    void lesson2( void )
    { std::cout << "lesson2" << std::endl;
    }
};


int main( int argc, char** argv )
{
  Teacher*      blind_john = new Teacher();
  StudentMary*  mary = new StudentMary;
  StudentJack*  jack = new StudentJack;

  std::cout << "begining lesson:" << std::endl;
  blind_john->listener( (response_type)(&StudentMary::any_questions) );
  blind_john->student( mary );
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
  blind_john->teach(); // blind john teaches mary, but doesnt know it
  std::cout << "end of lesson" << std::endl << std::endl;

  std::cout << "begining lesson:" << std::endl;
  blind_john->listener( (response_type)(&StudentJack::any_questions) );
  blind_john->student( jack );
  blind_john->teach(); // now blind john teachs jack (hasn't a clue)
  std::cout << "end of lesson" << std::endl << std::endl;

  delete blind_john;
  delete mary;
  delete jack;

  blind_john  = NULL;
  mary        = NULL;
  jack        = NULL;
  return 0;
}
```

This example differs slightly from the previous discussion in that the method of defining the policy/call-back has changed but the functionality and outcome remains the same. It also introduces more unnecessary complexity into the program that may make it difficult to maintain.

3.0.3 Functors

In the following example, we see a slight modification to the classes, by the introduction of an overloaded operator. This operator allows the class instance to be invoked as a function call as is seen in the Teacher object. The substitution process is slightly different too: all the students are now statically bound to the type of Teacher object when the Teacher is created.

```
#include <stdlib.h>
#include <iostream>

class Student
{
  public:
    virtual bool any_questions( void )
    {
      if( ((rand() % 10) % 2) == 0 )
      { proceed();
      }
      else
      { repeat();
      }
      return understands;
    }

    virtual bool operator()( void )
    {
      return any_questions();
    }
```

```
  protected:
    bool understands;

    void proceed( void )
    {
      understands = true;
      std::cout << "teacher I understand, please continue" << std::endl;
    }

    void repeat( void )
    {
      understands = false;
      std::cout << "teacher I don't understand" << std::endl;
    }

};

class TeachingMary: public Student
{
  public:

  virtual bool any_questions( void )
  {
    return Student::any_questions();
  }
};

class TeachingJack: public Student
{
  public:

  virtual bool any_questions( void )
  {
    return Student::any_questions();
  }
};

class TeachingClass: public Student
{
  public:

  virtual bool any_questions( void )
  {
    std::cout << "any questions class? ";
    return mary() && jack();
  }

  protected:
    TeachingMary mary;
    TeachingJack jack;
};
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
template< class StudentType >
class Teacher
{
  public:
    void teach( void )
    {
      do
        lesson1();
      while( !a_student() );
      do
        lesson2();
      while( !a_student() );
    }

  protected:
    StudentType  a_student;

  private:
    void lesson1( void )
    { std::cout << "lesson1" << std::endl;
    }

    void lesson2( void )
    { std::cout << "lesson2" << std::endl;
    }
};


int main( int argc, char** argv )
{
  Teacher< TeachingMary >   tutor_joe;
  Teacher< TeachingJack >   tutor_john;
  Teacher< TeachingClass >  class_teacher;

  std::cout << "begining lesson:" << std::endl;
  tutor_joe.teach(); // tutor teaches mary
  std::cout << "end of lesson" << std::endl << std::endl;

  std::cout << "begining lesson:" << std::endl;
  tutor_john.teach(); // tutor teaches john
  std::cout << "end of lesson" << std::endl << std::endl;

  std::cout << "begining lesson:" << std::endl;
  class_teacher.teach(); // teachers teaches class
  std::cout << "end of lesson" << std::endl << std::endl;

  return 0;
}
```

The reason why this is mentioned is that parameterised class functions are usually constructed using these functors. It is this approach that shall be used by the graph library as it ties in with the final objective: to be able to create a class with the exact user requirements. For example, the three teachers created

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

fulfill a specific purpose. This example illustrates the characteristics of policies in its simplest form. We shall use these features more extensively as we progress. Next we shall examine the different types of policies used by the toolkit.

3.1 Insertion policies

Insertion policies focus on providing mechanisms for adding elements to a particular container using a consistent approach. There are however a set of requirements that must be met before the insertion policy may be applied. These requirements pertain to the container type and the contained element type.

3.1.1 Container type requirements

The container is required to be a linear container. We define a linear container as a collection of elements where each element is accessed in a sequential manner. Supported linear containers therefore include either a list, set or vector (array). Vectors are both linear and direct access (random access) collections. Our toolkit requires that the container being used should adhere to the Standard Template Library (STL) interface. This implies the following:

- The container provides as a minimum the methods: *insert, erase, begin, end* and *size*.
- The *insert* method allows inserting elements to the start of the list by using the method *begin* and inserting elements to the end of the list by using the method *end* .
- The *erase* method allows erasing elements from the container using an *iterator* type.
- The *begin* method provides an iterator that points to the start of the container.
- The *end* method provides an iterator that points to the end of the container.
- The *size* method returns the number of elements in the container.

Through the use of *begin and end* any of the supported containers can then be traversed in a uniform fashion.

3.1.2 Element type

We define an element to be a composite type consisting of a source node type, edge type and destination node type as in definition (2.1). Element components (source node type, edge type and destination node type) are either grouped or not grouped. Element components have a specific order that we refer to as the element configuration (or configuration for short). Two types of groupings are supported, namely: left and right. If an element configuration is left grouped we say the element has a left associated configuration. If an element configuration is grouped on the right, we say the element has a right associated configuration. Elements are the building blocks of our graphs. We call this composite type a **triple**.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Non-grouped triples

In the following tables we define the different types of triple configurations that do not have any grouping characteristics.

Definition	$c_1 \triangleq \{n_s, e, n_d\}$
Identification	Standard triple configuration.
Description	This type of configuration is the default triple configuration used by the toolkit. All subsequent triples can be derived by applying an isomorphism to this standard triple configuration.
c++ example	triple<string, int, char>

Table 3.1: Not grouped triple: (s,e,d)

Left associated triples

In the tables that follow, we list and describe the group of triples that have a left grouping.

Definition	$c_2 \triangleq \{e, n_s, n_d\}$
Identification	Transposed standard triple.
Description	This type of configuration is the result of the transpose isomorphism applied to c_1 .
c++ example	triple<int, string, char>

Table 3.2: Not grouped triple: (e,s,d)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Definition	$c_3 \triangleq \{n_d, e, n_s\}$
Identification	Reversed standard triple.
Description	This type of configuration is the result of the reverse isomorphism applied to c_1 .
c++ example	`triple<char, int, string>`

Table 3.3: Not grouped triple: (d,e,s)

Definition	$c_4 \triangleq \{e, n_d, n_s\}$
Identification	Transposed reversed triple.
Description	This type of configuration is the result of the transpose isomorphism applied to c_3 .
c++ example	`triple<int, char, string>`

Table 3.4: Not grouped triple: (e,d,s)

Definition	$c_5 \triangleq \{n_d, n_s, e\}$
Identification	Reversed transposed triple.
Description	This type of configuration is a result of the reverse isomorphism applied to c_2 .
c++ example	`triple<char, string, int>`

Table 3.5: Not grouped triple: (d,s,e)

Definition	$c_6 \triangleq \{n_s, n_d, e\}$
Identification	Transposed reversed transposed triple.
Description	This type of configuration is a result of the transpose isomorphism applied to c_5 .
c++ example	`triple<string, char, int>`

Table 3.6: Not grouped triple: (s,d,e)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Definition	$c_7 \triangleq \{(n_s, e), n_d\}$
Identification	Left associated standard triple.
Description	This type of configuration is the result of the left associate isomorphism applied to c_1
c++ example	`pair<pair<string, int>, char>`

Table 3.7: Left grouped (Left associated) triple: ((s,e),d)

Definition	$c_8 \triangleq \{(e, n_s), n_d\}$
Identification	Transposed left associated standard triple.
Description	This type of configuration is the result of the left associate isomorphism applied to c_2 .
c++ example	`pair<pair<int, string>, char>`

Table 3.8: Left grouped (Left associated) triple: ((e,s),d)

Definition	$c_9 \triangleq \{(n_d, e), n_s\}$
Identification	Reversed left associated standard triple.
Description	This type of configuration is the result of the left associate isomorphism applied to c_3 .
c++ example	`pair<pair<char, int>, string>`

Table 3.9: Left grouped (Left associated) triple: ((d,e),s)

Definition	$c_{10} \triangleq \{(e, n_d), n_s\}$
Identification	Transposed reversed left associated standard triple.
Description	This type of configuration is the result of the left associate isomorphism applied to c_4 .
c++ example	`pair<pair<int, char>, string>`

Table 3.10: Left grouped (Left associated) triple: ((e,d),s)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Definition	$c_{11} \triangleq \{(n_d, n_s), e\}$
Identification	Reversed transposed left associated standard triple.
Description	This type of configuration is a result of the left associate isomorphism applied to c_5 .
c++ example	`pair<pair<char, string>, int>`

Table 3.11: Left grouped (Left associated) triple: ((d,s),e)

Definition	$c_{12} \triangleq \{(n_s, n_d), e\}$
Identification	Transposed reversed transposed left associated standard triple.
Description	This type of configuration is a result of the left associate isomorphism applied to c_6 .
c++ example	`pair<pair<string, char>, int>`

Table 3.12: Left grouped (Left associated) triple: ((s,d),e)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Right associated triples

In the tables that follow, we list and describe the group of triples that have a right grouping.

Definition	$c_{13} \triangleq \{n_s, (e, n_d)\}$
Identification	Right associated standard triple.
Description	This type of configuration is the result of the right associate isomorphism applied to c_1
c++ example	`pair<pair<string, pair<int, char>>`

Table 3.13: Right grouped (Right associated) triple: (s,(e,d))

Definition	$c_{14} \triangleq \{e, (n_s, n_d)\}$
Identification	Transposed right associated standard triple.
Description	This type of configuration is the result of the right associate isomorphism applied to c_2 .
c++ example	`pair<int, pair<string, char>>`

Table 3.14: Right grouped (Right associated) triple: (e,(s,d))

Definition	$c_{15} \triangleq \{n_d, (e, n_s)\}$
Identification	Reversed right associated standard triple.
Description	This type of configuration is the result of the right associate isomorphism applied to c_3 .
c++ example	`pair<char, pair<int, string>>`

Table 3.15: Right grouped (Right associated) triple: (d,(e,s))

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Definition	$c_{16} \triangleq \{e, (n_d, n_s)\}$
Identification	Transposed reversed right associated standard triple.
Description	This type of configuration is the result of the right associate isomorphism applied to c_4 .
c++ example	`pair<int, pair<char, string>>`

Table 3.16: Right grouped (Right associated) triple: (e,(d,s))

Definition	$c_{17} \triangleq \{n_d, (n_s), e)\}$
Identification	Reversed transposed right associated standard triple.
Description	This type of configuration is a result of the right associate isomorphism applied to c_5 .
c++ example	`pair<char, pair<string, int>>`

Table 3.17: Right grouped (Right associated) triple: (d,(s,e))

Definition	$c_{18} \triangleq \{n_s, (n_d, e)\}$
Identification	Transposed reversed transposed right associated standard triple.
Description	This type of configuration is a result of the right associate isomorphism applied to c_6 .
c++ example	`pair<string, pair<char, int>>`

Table 3.18: Right grouped (Right associated) triple: (s,(d,e))

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The above triple configurations are applicable to linear containers (vector, set and linked list) and to the binary tree container.

The value of these configurations comes in when the elements are being searched for, addressed/accessed and/or being grouped. One would be able to define a triple with a comparison policy that compares only according to a particular type of association (element component grouping) thereby sorting triples in a particular way. To further clarify this point consider the following example. Suppose a query is conducted to find all destination nodes that have the same source node and edge. All the destination nodes would then be grouped according to the edge and source node pairs by using the comparison policy.

Next we identify two insertion policies provided by the toolkit, namely:

- Insert at-head and
- Insert at-tail.

3.1.3 Insert at-head

This insertion policy is used by a particular container when an element has to be added to the start of the list. The result is a linear container that exhibits the Last In First Out (LIFO) characteristic i.e. similar to the insertion operation of a *stack*.

3.1.4 Insert at-tail

This insertion policy is applied to a particular container when an element has to be added to the end of the list. The result is a linear container exhibiting the First In First Out (FIFO) characteristic i.e. similar to the insertion operation of a *queue*.

3.2 Lookup policies

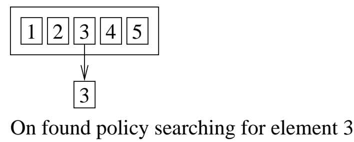
To speed up search requests we apply lookup policies. Theses policies change the contents of the container such that subsequent element search requests are completed faster. Lookup policies have the same requirements as insertion policies. For a list of these requirements refer to (3.1).

Lookup policies examine the underlying container searching for a matching triplet. Once the triplet has been found, the lookup policy is applied to the container for the triplet. The policy (depending on the type) has a beneficial effect on the next search that is conducted for the same element.

3.2.1 On-found lookup

When applied to the container for a particular element, this policy leaves the container and the position of the retrieved element unchanged i.e. the actual element (reference to the element) is returned while maintaining its current position (see figure 3.1).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

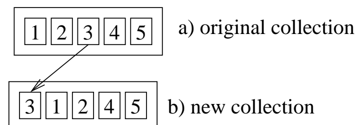


On found policy searching for element 3

Figure 3.1: On found lookup policy

3.2.2 Move to front lookup policy

Applying this policy alters the container by moving the retrieved element from its current position and re-inserting it at the start of the container. Subsequent searches for this element are faster than before regardless of the type of container being used (see figure 3.2).

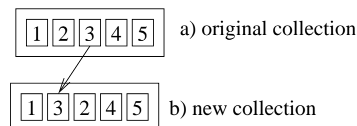


Move to front policy searching for element 3
finds it and moves it to the front.

Figure 3.2: Move to front lookup policy

3.2.3 Migrate forward lookup policy

When this policy is applied, the container is altered gradually by moving the retrieved element out of its slot and advancing it to the front of the container in increments of one. This is a more gradual approach when compared to the move-to-front policy that has an immediate impact. It eventually has the same effect as the move-to-front policy. However, it operates according to the popularity of the particular element look-up. The more that element is required/requested the faster its subsequent retrieval will become (see figure 3.3)



Migrate forward policy searching for element 3
finds it and moves it to the front.

Figure 3.3: Migrate forward lookup policy

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 4

Linear Graph Representations

4.1 Chapter Overview

This chapter examines graphs as a linear representation. The term linear is defined as "involving one dimension only" [Dig94] which is applicable here, since each representation is defined as a single uniform dimension in which items are accessed one after the other (sequentially). The containers that form part of the linear graph representations are:

- Vectors,
- Lists and
- Sets.

Vectors are sequential but also fall under the category of random access (where elements are accessed directly and in any order), but since the elements may also be accessed sequentially the container (vector container) is categorised, used and referred to here as a linear data container.

The containers are described in terms of a the standard structure (as defined by the Standard Template Library, STL) as well as an alternative structure. The aim of the alternative structure is to suggest an approach that may be easier to use (in the context of implementing the graph toolkit) by being more configurable.

Graph characteristics are provided and classified. Characteristics are specified in terms of the container's advantages and disadvantages. In order to make an effective decision regarding the use of a particular type of graph, one needs to take into account the advantages and disadvantages and then weigh up the trade-offs for the particular application. We characterise the container in terms of its benefits and drawbacks because decisions on whether to use the container or not will be based thereon. The classification scheme is used at the end of the chapter to identify the most common graph configurations.

4.2 Standard structure

Our toolkit uses the STL as the cornerstone of the entire solution, since there does not appear to be any good reason for "reinventing the wheel". Most C++ users are already familiar with STL, how it operates and the conventions it uses. We do however propose alternatives to using the STL but this is more an improvement to ease the implementation of our toolkit. The STL has a common input requirement layout (termed as features) and therefore we analyse it in general in the following sub-sections. Thereafter we provide specific alternative approaches for implementing the individual type of containers.

4.2.1 Feature layout

For each of the linear containers (vector, list and set), the STL defines a **type** and an **allocator** as input requirements.

Type

The type denotes an arbitrary type. It is an essential ingredient of each type of container. With particular reference to the toolkit, the type is a particular triple configuration as defined in paragraph 3.1.2. This is a mandatory feature.

Allocator

The allocator is a memory allocation component that allocates memory for the collection to accommodate the given type. Memory allocation occurs in a certain way according to the allocation policy. It is not compulsory to provide a value for this feature. If one is not specified the default memory allocation scheme is used.

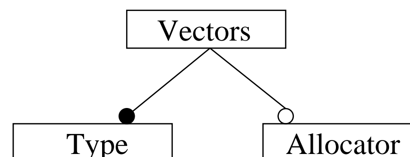


Figure 4.1: Standard feature definition for vectors

We use feature diagrams to describe the configuration options available for a particular container. Feature diagrams are well defined in [CE00]. For the purposes of this dissertation we restrict ourselves to a subset of the feature diagram properties. In particular, we focus on mandatory and optional features (configuration options) drawn as a filled and clear circle respectively.

Here in figure 4.1 we show the standard structure as a feature diagram with two configuration options. As a minimum it requires the Type, which is compulsory (indicated by the filled circle), and the Allocator, which is optional (indicated by the unfilled circle). The disadvantage of this simplification of inputs is that the insertion and lookup policies, which play a vital part in the toolkit, need to be applied at a higher level. In figure 4.2 the graph uses policy components to perform the insertion and retrieval of graph elements. To go

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

into more detail regarding this, there would at exist, at a low level, one or more containers, and these would be vector (*c1*) and set (*c2*). The graph (G) is configured to be the owner of *c1* and to use the insert-at-head policy (x) to insert elements into the container, i.e. (G) is composed of a container (*c1*) and configured to use the insertion policy *insert-at-head* (x). Instead of calling the container's specific insert method directly, G inserts an element into *c1* via x giving x a reference to *c1* and the required element to insert.

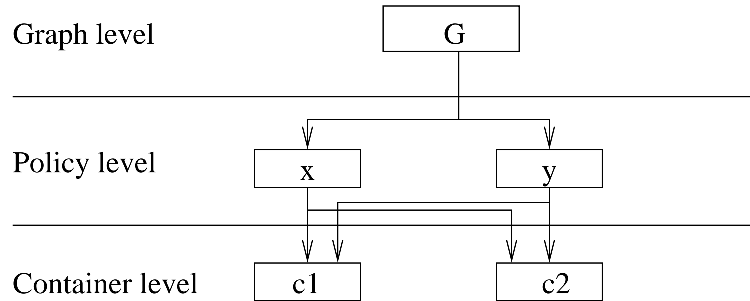


Figure 4.2: High level architecture diagram

4.3 Vector graph representation

Although the STL vector implementation (refer to section 4.2) is simplified and sufficient for most application areas, it can be made more configurable. Allowing the user to choose between a dynamic (growing) and static (constant size) vector has an impact on resource usage and the time it takes to bind values to place-holders in the collection. In addition, as mentioned above, the architecture could be simplified by building the options of insertion and lookup policies directly into the vector container thereby eliminating the need for policy level interaction. Even though we adopted the approach to use the STL as an implementation base, we discuss an alternative structure to see how we could improve the architecture of the graph toolkit in future versions. We define the alternative structure for vectors as requiring a **type**, **size**, **fixed**, **insert policy** and **lookup policy**.

4.3.1 Type

The type denotes an arbitrary type that forms the base of the vector. In the case of the toolkit, the type would typically be a graph element as defined in paragraph 3.1.2.

4.3.2 Fixed

This is an enumerated constant that indicates the vector boundary behaviour. This behaviour is defined in terms of data entry flexibility.

- *0*, normally indicating *false*, results in the vector exhibiting a dynamic behaviour i.e. the vector may grow in size.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- *1*, normally indicating *true*, results in the vector exhibiting a static behaviour i.e. the vector may not grow in size.

4.3.3 Size

The size specifies the vector boundary. If the boundary is fixed this size remains constant. If the boundary is flexible, the size becomes variable allowing the expansion and collapsing of the vector, therefore this parameter is applicable when the *Fixed* flag is *true* (*1*). If however the *Fixed* flag is *false* (*0*) i.e. dynamic, this parameter is ignored and no longer applicable to container implementation and it therefore may be assigned an arbitrary value.

4.3.4 Insert policy

The insertion policy allows the user of the toolkit to customise the entry of data. It controls how data is entered into the collection. The type of policy that may be substituted is one of the policies listed in section 3.1. As mentioned earlier, having an insertion policy at this level allows us to do away with the required additional level.

4.3.5 Lookup policy

The lookup policy allows the user of the toolkit to customise the retrieval of data. The aim behind having a lookup policy is to speed up subsequent search requests through a search heuristic. The search heuristic is the policy being applied. A valid lookup policy is a policy from the list of lookup policies as defined in paragraph 3.2.

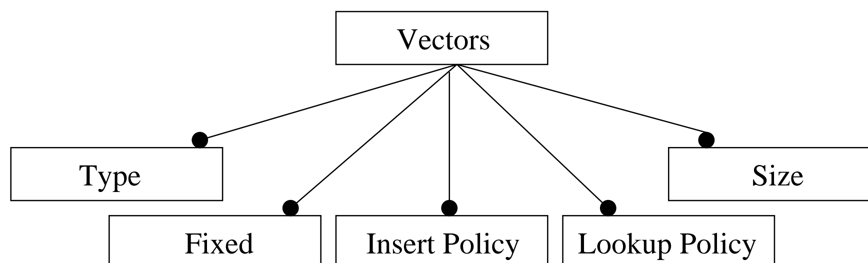


Figure 4.3: Alternative feature definition for vectors

Figure 4.3 depicts the alternative feature diagram.

4.3.6 Vector graph representation advantages

Constant access time

Vectors are beneficial in this regard since every element in the vector has an equal access time. The access to the data is fast, direct and precise.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Easy access

Access to an element only requires a simple calculation. The calculation operates from the base address of the vector combined additionally by offset values and the size of the type of element being stored in the vector.

Continuity is achieved

Vectors maintain a collection of data using a contiguous block of memory. That is to say that they operate within certain boundaries (a lower boundary and an upper boundary). Vectors constantly operate on a fixed size of memory. It may be dynamic or it may be static in which case the size of the vector is bound at runtime and compile time respectively. Most languages do support late binding (dynamic memory allocation mechanisms) such as C, C++, Java, C-Sharp, Pascal, Modula-2 and PL/I. FORTRAN however binds the size to the array at compile time. The advantage of continuity is that access to a given element is almost immediate.

Random access ability

Although a vector is a linear container, it also falls into the category of being random access since any element may be accessed just by issuing its index value (which may end up being a simple calculation).

4.3.7 Vector graph representation disadvantages

High sensitivity

The very ability of a vector providing access to its elements through a computation sequence (such as pointer arithmetic in C/C++) may be quite error prone. This stems from the basis that the vectors have a lower and an upper boundary. Exceeding either of these boundaries leads to undefined program behaviour with the result usually ending up as a memory access violation.

Time consuming element removal

Since a vector operates on a fixed block of memory, removing an element from the vector is usually tedious since it relies on having to resize the block of memory without any further data loss.

To avoid this resizing, it is possible to employ a default value approach to data storage. Each value being stored has a default value for instance. If that value is in the vector, that position may be considered as being available. Any other value therefore denotes that the cell is being occupied.

Another approach is to use a compound structure. The compound structure is made up of a usage indicator and a data indicator. The usage indicator indicates that the cell is indeed occupying data and should not be considered empty if the indicator is set to **true**. If it set to **false** it results in an indication that the cell is empty and may be replaced by new values during insertion. The problem with such an approach however is that the vector becomes sparse. Finding the next empty position to insert an element becomes a sequential search through the container until either the end of the container is reached

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

or a flag indicating an empty position is found. This may be undesirable or acceptable in certain applications.

Fixed size limitation

Having to work with a fixed block of memory, results in the possibility that additional data may need to be stored. If there is no more room in the vector, it has to either be resized or an error generated. In most cases, the vector is resized by doubling the original size. This results in possibly fewer resize requests. However, it may also result in wasted resources and additional time consumption. The problem with resizing lies with the need to copy the original block of memory over to the new block of memory and then having to de-allocate the original block. Usually the time spent doing this is negligible, bearing in mind that the size of the collection needs to be fairly small. It is also important to note that the above process is applicable when the additional block of memory could not be allocated such that collection memory runs consecutively.

4.4 Linked list graph representation

The alternative list structure is defined by **type**, **link type**, **insertion policy**, and **lookup policy**.

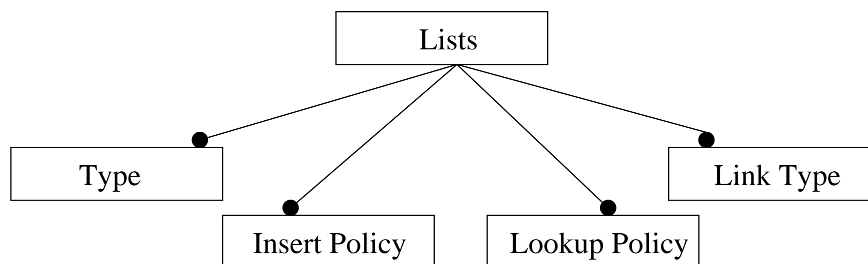


Figure 4.4: Alternative list features

Figure 4.4 shows the alternative feature diagram that dictates the linked list input requirements. From the diagram, it is evident that all features are mandatory unlike the STL representation.

4.4.1 Type

The type denotes an arbitrary type forming the base of the list. In the case of the toolkit, the type would typically be a graph element as defined in paragraph 3.1.2.

4.4.2 Link type

This is an enumeration constant that specifies the type of linked list being implemented. Currently only two options are available indicated by the value 0 and 1.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- A unidirectional linked list is defined by the value 0. This is the common type of linked list where links proceed sequentially from head to tail. In essence this means that successive links have no links to their predecessor.
- A bidirectional linked list is defined by the value 1. In this approach, the linked list node maintains a link to its predecessor if one exists (In the case of the first element in the list, there is not a predecessor to link).

4.4.3 Insert policy

The insertion policy allows the customisation of data entry. It controls how data is entered into the collection. The type of policy that may be substituted is one of the policies listed in section 3.1.

4.4.4 Lookup policy

The lookup policy allows the customisation of data lookup. It attempts to speed up subsequent search requests through a search heuristic. The search heuristic is the policy being applied. The possible lookup policies that may be used is one of the policies listed in section 3.2

4.4.5 Linked list graph representation advantages

Efficiency

Linked lists are a more efficient approach to data storage in terms of resource consumption when compared to vectors. Dynamically allocated units maintain a small amount of data to produce this efficiency i.e. they form a dense collection of items.

Ease of element insertion

Insertion of elements is straightforward. An additional list compartment (list element) is created, populated and linked to the rest of the list. This is considerably easier and faster than having to resize the collection every time as is the case with vectors.

Ease of element removal

Removing elements is equally easy and efficient. Removal maintains a dense collection of items unlike vectors that may result in a sparse data collection (depending on the type of implementation method used).

4.4.6 Linked list graph representation disadvantages

Overhead

Unlike vectors, linked lists need to maintain extra data indicating the predecessor and the successor of the current node in addition to the data compartment that stores the value of the list element.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Access times

Although linked lists offer efficient resource utilisation, they have a drawback in that they are fully sequential unlike vectors that have a random access capability. Elements are removed in a sequential manner. The location of the element needs to be determined by traversing the list from head to tail or from tail to head (depending on the direction of travel). Only once the element is found (if it is found) may it be changed or removed.

4.5 Set graph representation

The alternative structure is defined by **type**, **representation**, **insert policy** and **lookup policy**.

4.5.1 Type

The type denotes an arbitrary type forming the base of the set. In the case of the toolkit, the type would typically be a graph element as defined in paragraph 3.1.2.

4.5.2 Representation

This variable defines the representation used by the set as a container. Since the set is linear and vectors and linked lists are the primary linear alternatives, the set container is implemented as one of the two. The only real difference that the set has to offer is that it disallows duplicate entries to occur. Binary trees could be included as a representation however this would make the set a binary set and thus would end up being a binary graph representation.

4.5.3 Insert policy

The insertion policy allows the customisation of data entry. It controls how data is entered into the collection. The type of policy that may be substituted is one of the policies listed in section 3.1.

4.5.4 Lookup policy

The lookup policy allows the customisation of data lookup. It attempts to speed up subsequent search requests through a search heuristic. The search heuristic is the policy being applied. The possible lookup policies that may be used is one of the policies listed in section 3.2.

Figure 4.5 depicts the alternative structure. All parameters are indicated as mandatory. From the diagram we have introduced a new aspect of feature diagrams. The unfilled arc linking the vector and linked list representation denotes an **alternative** relationship between the linked items i.e. either a vector or linked list can be used as suitable replacement for the representation. It specifies that the representation should be one of the sub-features and because the vector and linked list features are compulsory (denoted by the filled circle attached to the component) the representation must be one of the listed sub-features.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

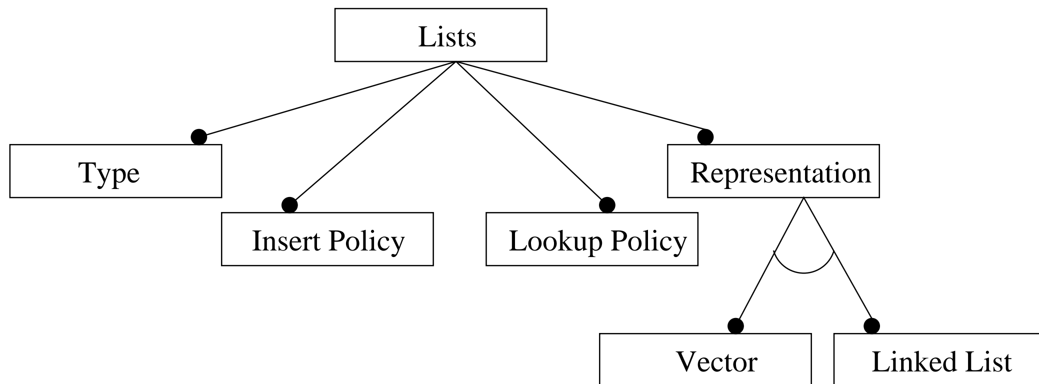


Figure 4.5: Alternative feature definition for sets

4.5.5 Set graph representation advantages and disadvantages

Depending on the type of representation being used, sets exhibit the advantages and disadvantages of that particular representation because it has been implemented based on that representation. An additional aspect that functions both as an advantage and disadvantage to sets is that it implements the policy of duplicate element elimination. In the context of being an advantage, it automatically fits in correctly with graphs, where graphs are defined as a set of nodes i.e. each node is unique. In the context of being a disadvantage, the sets automatically apply this check for duplication whenever an entry is added to the list.

4.6 Classification

In each chapter dealing with the graph toolkit containers, we present a classification system for each type of graph representation. This form of classification system breaks away from the classical hierarchical classification mechanism used to produce a taxonomy. Hierarchical systems lack the ability to classify multiple iterations of morphing easily (see figure 4.6). Using a finite state automaton (basically a graph) allows us to derive a unique name for any graph arising from a selection of options from the available configuration options.

Figure 4.6[1] presents the classification system we use for linear graph representations.

4.6.1 The way that the classification system works

Proceed from top to bottom. Start with *Triple* and create a path to an end-point (*lookup on found*, *lookup migrate forward* or *lookup move to front*) via the

[1]It should be noted that the arrows leading to and from the isomorphisms (transpose, reverse, left-associate and right-associate) should be considered to be bi-directional. It is diagramed in this way to prevent cluttering the diagram and thus this should be considered as a short hand notation

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

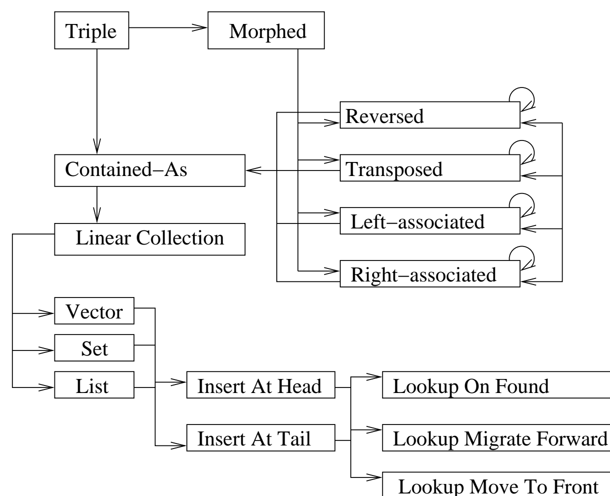


Figure 4.6: Linear graph classification machine

arrows. This will result in a long descriptive name that identifies the type of graph precisely. This name may be hyphen abbreviated as will be seen in the sections that follow.

As an example:

$Triple \rightarrow Morphed \rightarrow Transposed \rightarrow Reversed \rightarrow Transposed \rightarrow Contained-As \rightarrow Linear\ Collection \rightarrow vector \rightarrow Insert\ At-Tail \rightarrow Lookup\ On-Found.$

Several different classifications follow in the paragraphs below. Each classification is specified in terms of its abbreviated form, a full identification and a brief description of the classification. Advantages and disadvantages apply as described in the preceding sections for the respective container type.

4.6.2 T-CA-LC-V-IAH-LOF

Identification

Triple Contained As Linear Container Vector with Insert At Head (insertion policy) and Lookup On Found (lookup policy).

Description

This class of graph is a vector that employs the insert-at-head insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of $Source, Edge, Destination$ without isomorphic operations applied.

4.6.3 T-CA-LC-S-IAH-LOF

Identification

Triple Contained As Linear Collection Set with Insert At Head (insertion policy) and Lookup On Found (lookup policy).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Description

This class of graph is a set that employs the insert-at-head insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of $Source, Edge, Destination$ without isomorphic operations applied.

4.6.4 T-CA-LC-L-IAH-LOF

Identification

Triple Contained As Linear Collection List with Insert At Head (insertion policy) and Lookup On Found (lookup policy).

Description

This class of graph is a linked list that employs the insert-at-head insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of $Source, Edge, Destination$ without isomorphic operations applied.

4.6.5 T-CA-LC-V-IAT-LOF

Identification

Triple Contained As Linear Collection Vector with Insert At Tail (insertion policy) and Lookup On Found (lookup policy).

Description

This class of graph is a vector that employs the insert-at-tail insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of $Source, Edge, Destination$ without isomorphic operations applied.

4.6.6 T-CA-LC-S-IAT-LOF

Identification

Triple Contained As Linear Collection Set with Insert At Tail (insertion policy) and Lookup On Found (lookup policy).

Description

This class of graph is a set that employs the insert-at-tail insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of $Source, Edge, Destination$ without isomorphic operations applied.

4.6.7 T-CA-LC-L-IAT-LOF

Identification

Triple Contained As Linear Collection List with Insert At Tail (insertion policy) and Lookup On Found (lookup policy).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Description

This class of graph is a linked list that employs the insert-at-tail insertion policy as well as a lookup policy of on-found. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

4.7 Summary

In this chapter we looked at the first type of graph representations that make use of linear container technologies such as vectors, sets and linked lists to implement a graph. We provided a high level architectural look at the solution indicating how the container fits in with the graph policies. We examined the aspects of the STL as a solution to our toolkit and also described each container in terms of its features using feature diagrams. For each of the linear containers we discussed the advantages and disadvantages of each container. The chapter came to an end with the classification system for the linear type of graphs. The classification system is a mechanism that can be used to uniquely identify a type of graph that is based on using triples. Incidentally, the graph classification system is described in terms of a graph itself as opposed to using a tree based taxonomy to define the graph family. We found that this mechanism is also useful for producing a unique name identifying the type of graph. This chapter introduced the basic graph representations.

These are fairly easy to implement and the implementation concepts behind them are not very complicated and fairly straightforward. In the following chapter we will extend the graph toolkit by including binary containers to store the underlying graph data.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 5

Binary Graph Representations

5.1 Chapter Overview

In this chapter we examine binary graph representations. We focus on the different containers that can be used to represent binary graphs. A container can be used to represent a binary graph if it can be searched using a binary search algorithm.

When we speak of binary algorithms then immediately, the binary tree comes to mind. Its operation is defined by its dual branch nature. When searching for an element searching for the next insertion point for an element, traversal is based on Boolean (true or false) values. When searching for an element, the value of the current element is compared to the candidate element. If the value of the candidate element is greater than the value of the current element then traversal continues on the right branch of the current element[1]. If however the candidate element value is less than that of the current element, then traversal continues to the left. The above process of comparing and making a decision based on two options as described above forms the basis of the binary algorithm.

Now we have mentioned one container that can be used to represent graphs through binary technology. Although not implemented by the toolkit, we identify and describe two other forms of binary containers: ascending vector and descending vector.

We start by looking at the configuration structure of the binary tree using a feature diagram. Since the STL does not currently provide a binary tree implementation the feature diagram described represents the binary tree that has been implemented in the toolkit.

Thereafter a sorted vector representation is examined. Again using feature diagrams to discuss the structure. We provide this discussion point as an alternative to the binary tree representation since items in the sorted vector may be searched for using a binary search algorithm.

We then classify several binary representations ending the chapter with a

[1] The side of the branch is irrelevant, what is important is that if the value of the candidate element is less than or equal to the value of the current element then traversal must commence in the opposite direction

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

summary highlighting the important points of the chapter as well as comparisons of the mentioned binary graph classifications.

5.2 Binary tree graphs

We define our binary tree graphs according to feature diagram 5.1.

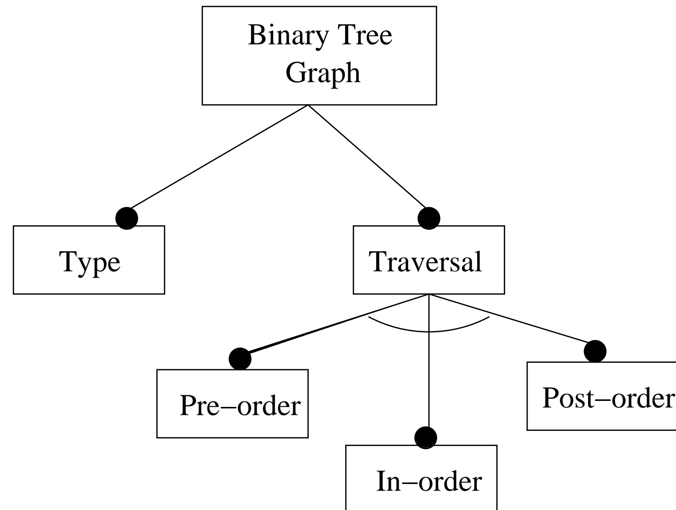


Figure 5.1: Feature diagram for a binary tree graph

5.2.1 Type

This is an arbitrary type. By arbitrary type we mean any kind of type e.g. integer, character, string or even user defined types. In the context of the toolkit and graph implementation, the type is a triple (user defined type) with a specific formation. The entire collection of triples ultimately makes up the graph and therefore the triple is fundamental to the solution (see section 2.2).

5.2.2 Traversal

This parameter specifies the traversal method used. It may be one of three ([Bud94]):

- Pre-order,
- In-order and
- Post-order.

These types of traversals are implemented as policies and affect the way in which the tree is traversed. An alternative approach to implementing the traversal policies is to use the strategy design pattern as defined in [GHJV94] and described in sections 3.0.1, 3.0.2 and 3.0.3.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Pre-order tree traversal

In a pre-order tree traversal the left most leaf nodes are identified then the left leaf node's parent followed by the right most leaf node as illustrated in figure 5.2 (a).

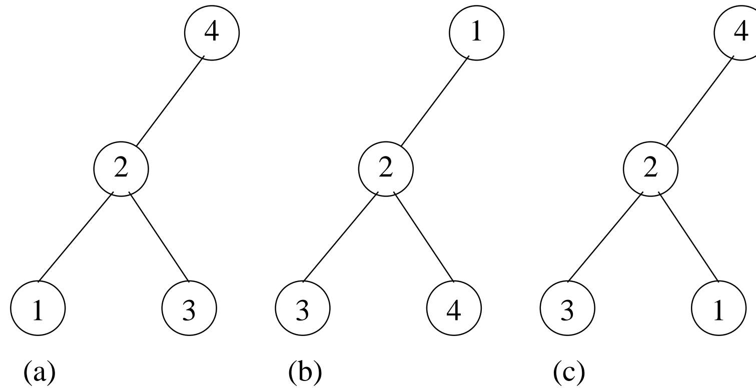


Figure 5.2: Binary tree traversals

In-order tree traversal

In an in-order tree traversal the parent nodes are visited prior to the child nodes as illustrated in figure 5.2 (b).

Post-order tree traversal

In a post-order tree traversal the right most leaf nodes are identified then the right leaf node's parent followed by the left most leaf node as illustrated in figure 5.2 (c).

5.2.3 Binary tree graph advantages

Flexibility

Insertion of elements into this type of data structure is flexible and easy. Insertion is not direct and immediate as with a vector or linked list where the next insertion point may be known. There may be a slight delay in the insertion of elements to the collection since the tree operates according to a particular algorithm. When insertion is complete the inserted value obeys the characteristics of the binary search algorithm.

Simplified searching

Since the items in the collection are sorted, retrieval is trivial. Searching is based on a similar approach used during insertion except that the search ends when the end of the tree is reached or the current element matches the desired search criteria. The search is binary (or more generally termed logarithmic) that is faster than linear searches in both the average and worst case.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5.2.4 Binary tree graph disadvantages

Element removal is expensive

Deletion of elements from a tree is tricky. It is not as trivial as removing elements from collections such as vectors, sets, linked lists or even hash tables. Since each element needs to obey a strict insertion protocol deleting a node in the middle of some sub-branch may result in incorrect searches. The other issue is that when deleting a node, the right branch of that node may already have data attached to it and therefore elementary node reassignment is not possible. Luckily there are algorithms in place that do cater for this situation but nonetheless when compared to other collections, element removal is more involved and can be quite tricky.

Time consuming insertions

Insertion of elements can become costly as the size of the container grows. Every insertion requires a search for the correct location in the tree before an element may be saved. However binary trees still offer better performance over unsorted linear containers.

5.3 Linear sorted graph representations

We define the linear sorted graph representations according to feature diagram 5.3.

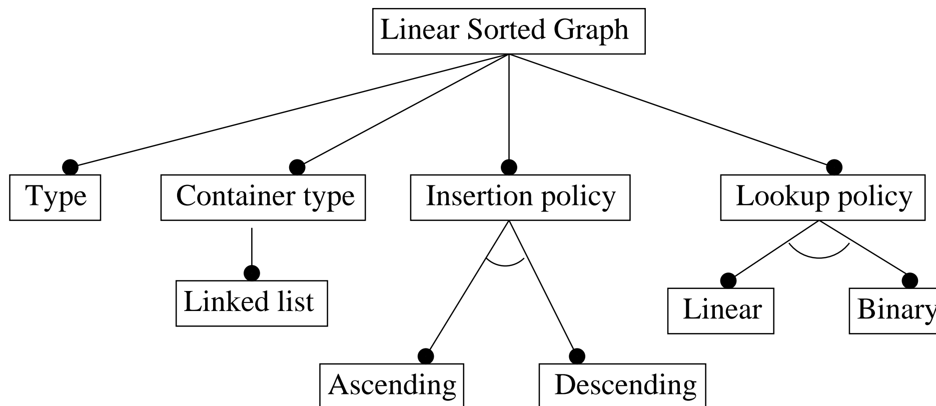


Figure 5.3: Feature diagram for sorted containers

5.3.1 Type

This is an arbitrary type however in the context of the toolkit and graph implementation, the type is a triple with a specific formation.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5.3.2 Container type

This is the type of container being used. Since we are dealing with linear containers, we may only use a vector because the binary search algorithm requires that linear container is a random access based. Elements are searched for within two ranges every time decreasing the range until the item is found as illustrated in 5.4.

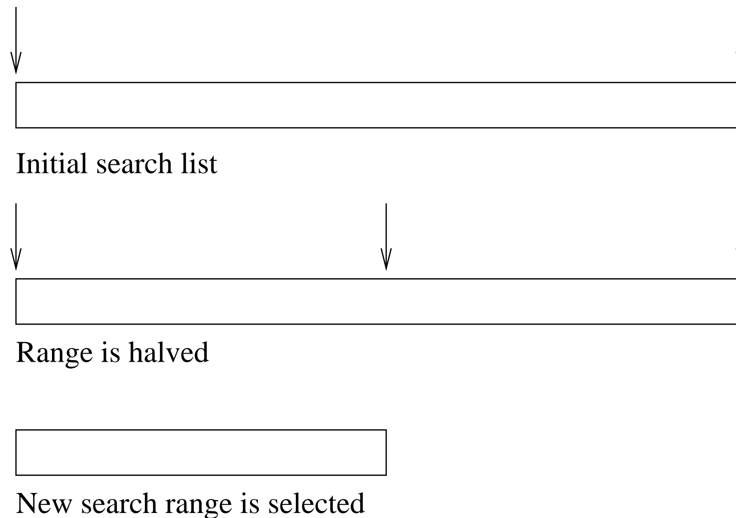


Figure 5.4: Binary search algorithm for linear containers

5.3.3 Insertion policy

There are two possibilities for inserting elements into the sorted container, namely: ascending insertion and descending insertion.

1. **Ascending** insertion maintains a strict ordering of elements ranging from smallest to largest. When this policy is applied elements are inserted while maintaining the sort order.

2. **Descending** insertion maintains an ordered collection of elements that range from largest to smallest.

5.3.4 Lookup policy

There are at least two possible lookup policies that are applicable to linear sorted containers, namely: linear search and binary search.

1. A **linear** policy applies a search that commences from the start of the collection to the end of the collection or until the desired element is found. Movement through the collection is strictly linear.

2. A **binary** policy applies the popular binary search algorithm to the search space.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5.3.5 Sorted linear graph representation advantages

Linear sorted graph representations combine the benefits of linear unsorted graph representations (commonly known as linear graph representations, see chapter 4) with the advantages that the binary search algorithm brings. To recapitulate, these benefits are listed as follows:

1. Insertion of elements is fast and easily implemented.
2. Searching for elements can be achieved through a binary search algorithm.
3. Access to elements is fast and easily implemented.

5.3.6 Sorted linear graph disadvantages

1. Element removal becomes costly when marinating a sorted policy.
2. Resizing of the vector becomes expensive in terms of resource utilisation. This is applicable when dealing with dynamically resizing vector as in the STL.

5.4 Classification

Figure 5.5 presents a classification system for binary graph representations:

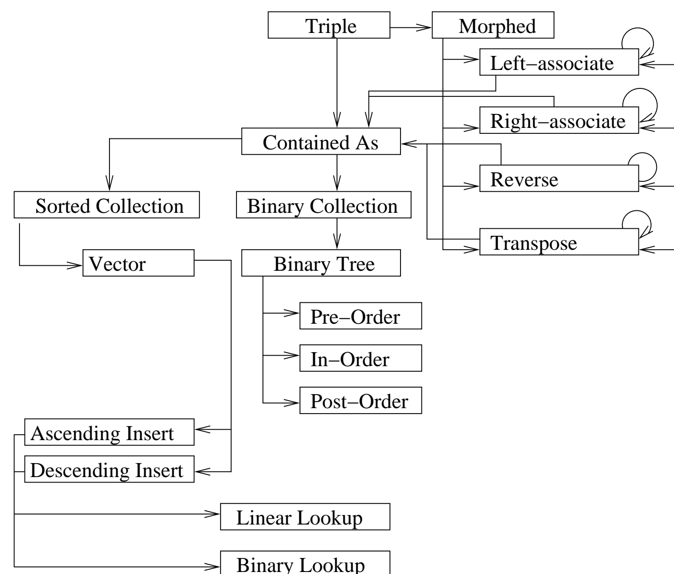


Figure 5.5: Binary graph classification machine

We shall identify several types of binary graph representations. The different classifications follow in the paragraphs below. Each classification is specified in terms of its abbreviated form, a full identification and a brief description of the classification.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5.4.1 T-CA-BC-BT-PRT

Identification

Triple Contained As Binary Collection Binary Tree with Pre-Order Traversal.

Description

This class of graph is a binary tree that uses a pre-order traversal. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.4.2 T-CA-BC-BT-POT

Identification

Triple Contained As Binary Collection Binary Tree with Post-Order Traversal.

Description

This class of graph is a binary tree that uses a post-order traversal. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.4.3 T-CA-BC-BT-IOT

Identification

Triple Contained As Binary Collection Binary Tree with In-Order Traversal.

Description

This class of graph is a binary tree that uses the in-order traversal. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.4.4 T-CA-SC-V-AI-LL

Identification

Triple Contained As Sorted Collection Vector with Ascending Insertion with Linear Lookup.

Description

This class of graph is a sorted vector container, where elements are sorted in ascending order, the elements are searched using a linear approach. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5.4.5 T-CA-SC-V-DI-LL

Identification

Triple Contained As Sorted Collection Vector with Descending Insertion with Linear Lookup.

Description

This class of graph is a sorted vector container, where elements are sorted in descending order, the elements are searched using a linear approach. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.4.6 T-CA-SC-V-AI-BL

Identification

Triple Contained As Sorted Collection Vector with Ascending Insertion with Binary Lookup.

Description

This class of graph is a sorted vector container, where elements are sorted in ascending order, the elements are searched using a binary search algorithm. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.4.7 T-CA-SC-V-DI-BL

Identification

Triple Contained As Sorted Collection Vector with Descending Insertion with Binary Lookup.

Description

This class of graph is a sorted vector container, where elements are sorted in descending order, the elements are searched using a binary search algorithm. The graph formation is a standard triple in the form of *Source*, *Edge*, *Destination* without isomorphic operations applied.

5.5 Summary

This chapter dealt with graphs that use binary type containers such as binary trees and sorted linear collections. We first started with a broad description of binary trees and the concept behind them. We then provided the feature diagram for binary tree graph representations and described their advantages and disadvantages. We then examined sorted linear containers as alternative binary representation to binary trees. We identified the difference between the two approaches as being structural and indicated that the commonality between the approaches lies in the algorithm. Where both algorithms used for searching

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

for an element is binary in the sense that each algorithm makes a decision based on two options. In the case of a binary tree, a decision is made to traverse the left branch or the right branch based on the current node. Similarly, in a linearly sorted container, the search algorithm halves the search space and decides which half should be searched for based on the value of the search key. We also defined the feature diagram of the sorted linear container and briefly described the advantages and disadvantages surrounding it. We then created a classification mechanism to be in line with our standard classification mechanism, while being similar to yet extending the classification system produced in the previous chapter.

Now that we have identified binary graph representations we will extend our toolkit even further in the next chapter introducing maps and see how they can be used to implement graphs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 6

Mapped Graph Representations

6.1 Chapter Overview

A map is an associative collection of keys and values (see figure 6.1), where keys have an arbitrary type and are linked to values also of an arbitrary type. The key type may differ from the value type. However this is not a compulsory requirement but more a coincidence. In the case of our toolkit, the elements stored in the mapping container reflect triples using a mapping approach e.g. source nodes may be mapped to edge and destination node pairs. This container does away with the single triple type and rather focuses on associations.

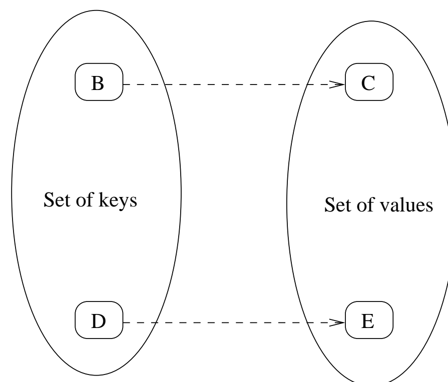


Figure 6.1: Single mapping illustration

Figure 6.1 depicts a simplistic illustration of a mapping relation. An element from one set is mapped to an element residing in another set. This figure illustrates a type of graph mapping where a source node for instance may be mapped to a corresponding edge and destination node pair.

As an extension to these singular pair-maps (referred to hereafter as single maps), we examine a further mapping concept namely: maps-of-maps (also

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

known as two maps). In this type of collection there is a mapping from a set of key elements to another collection of key elements that is then mapped to a value (see figure 6.2).

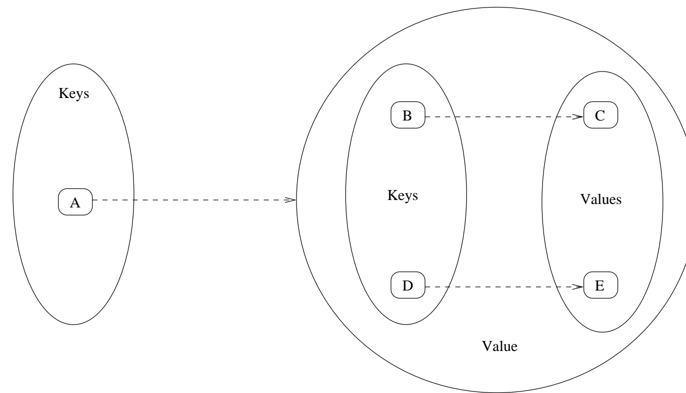


Figure 6.2: Map-Of-Map illustration

In the first section we present single mapped graph representations, followed by map-of-map graph representations. Each section is broken up into sub-categories that present the features of each type of graph using a feature diagram. Thereafter we identify the characteristics of the particular container using its advantages and disadvantages. After having identified the characteristics, we present the graph classification system used to name the graphs in this category (mapping category). We conclude the chapter with a summary of the chapter.

6.2 Single map graph representations

In chapter 4 we examined graphs as linear representations containing different triple configurations (the fundamental constituent in directed graphs). Among these were left-associated and right-associated triples that are in essence linearly associated graph representations i.e. linear mapped graph representations.

Figure 6.3 describes a feature diagram illustrating the singly mapped collection of keys and their associated values. This is the approach provided by the STL and is reused by our toolkit.

6.2.1 Key type

This is the key field type defining the type of element that is used when elements are being indexed and searched for. It forms the basis of the association and is a primary search component. In the context of our toolkit the key type is a component of a triple (unary), either source, edge or destination or it may be a paired combination (binary) such as source-edge, destination-edge, edge-destination or edge-source combination. The key type is a mandatory feature. This feature is mandatory because the user needs to specify exactly what the key type should be that maps to the value type.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

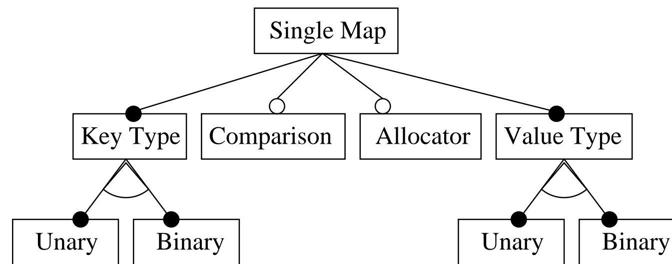


Figure 6.3: Standard single map features

6.2.2 Value type

This is the field type defining the types of values that are associated with the key. These values are either a single triple type (unary) i.e. source node, edge or destination node or a paired type (binary) holding paired triple components i.e. source-edge, source-destination, destination-source, destination-edge, edge-source or edge-destination. This feature is also compulsory. This feature is mandatory because the user needs to indicate what exactly should be the value type/types associated with the key.

6.2.3 Comparison

This variable specifies the manner in which values of the key type are sorted in the container. Using for instance a less-than operator causes values to be stored in an ascending order whereas a greater-than operator causes values to be sorted in descending order. This feature is optional as it is not a critical (critical in the sense that if a value is not provided a default value can be substituted to continue operation) for the container to function correctly.

6.2.4 Allocator

As with the other containers already mentioned (list, vector, etc.), facilitates with the allocation of memory for the various key and value elements occurring in the association. Again this feature is optional because a default memory allocation can be selected if one is not specifically appointed.

6.2.5 Advantages of using single mapped graph representations

Efficient

Map containers may be compared to dictionaries and are similar, in principle, to hash tables[1] in that they also operate on a key for data basis. The advantage that maps have over hash tables is that the underlying container used to represent the mapped collection is not restricted to random access containers

[1]See chapter 7.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

but may also make use of binary search trees (as discussed in chapter 5). Implementing maps as a binary search tree provides one with a reasonably good performance and excellent resource usage when compared to hash tables.

Faster retrieval of a collection of values

Consider the following scenario where the key is defined as the source node. We map the source node to a set of edge and destination pairs. Thus the mapping is from a source node to edge and destination combination contained in a set (i.e., only on of each combination). Adding out edges for a specific source node is very easy as it is merely a single search invocation for the key and an insertion into the set container. The advantage comes in when we want to find all the out edges for a given node. We simply need to find the key i.e. the source node. Once the source node is found the set of edges and destination nodes is returned in one easy step. This could be quite powerful when building a parser or regular expression component since we can very quickly determine what the next possible input symbols are allowed to be as shown in figure 6.4.

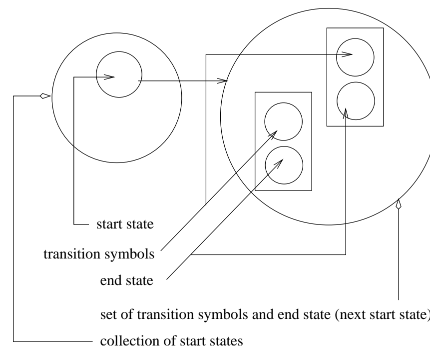


Figure 6.4: Example of mapping to a collection of values

6.2.6 Disadvantages of using single mapped graph representations

Limited number of values

A disadvantage of single mapped graph representations is that the key is mapped only to one value. In principle, this can be overcome by making the value component of the mapping relation part of another container type such as linked list, set, vector etc. This approach, however, possesses its own limitations in that it requires an additional search procedure to be employed. From the previous chapters we realise that each container has its limitations in terms of speed and resource usage. To obtain fast search results, this type of graph representation is best suited for applications where the key is mapped to exactly one value. That way when the key is found, the correct value associated with that key is immediately in hand - no additional searches are necessary. This is something to bear in mind when using mapped graph representations. On the flip side of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

that coin lies the advantage discussed above of having a set of values linked to a key.

6.3 Map-of-map graph representations

We subdivide map-of-maps into two association dimensions, namely: left and right. The feature diagram 6.5 specifies the features of the map-of-map container.

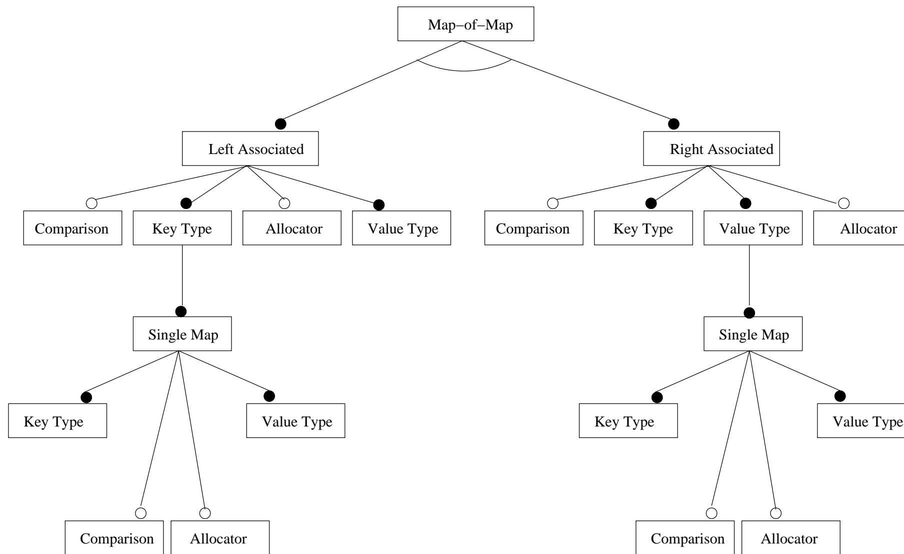


Figure 6.5: Standard map-of-map structure

6.3.1 Key type

The key type feature is either a single type or a single map. If we use a single type as the key type then it means that the key type maps to a single map (the sub-map), the association is then to the right i.e. right-associated because the single map contains the remaining types. If the key type is a single map then it means the key type maps to the final type as in left-association. Figure 6.6 illustrates this concept. If the key type is a single type then figure 6.6 (a) is applicable with the value type being a single map where one type maps to another type. If the key type is a single map then the figure 6.6 (b) is applicable with the value type being the single type.

6.3.2 Value type

This is a mandatory feature as it defines the type that is associated with the key type. See explanation above.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

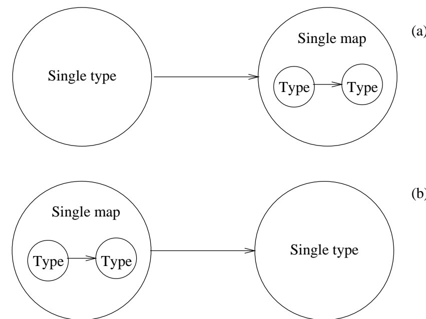


Figure 6.6: Map-of-map depicting right and left associations respectively

6.3.3 Comparison

This feature specifies the manner in which values of the key type are sorted in the container. Using *less-than* comparison causes values to be stored in an ascending order whereas a *greater-than* comparison causes values to be sorted in descending order. As this is not deemed critical, this feature is optional.

6.3.4 Allocator

As with the other containers (list, vector, set, etc.) already mentioned, this feature facilitates with the allocation of memory for the various key and value elements occurring in the association. This feature is optional as a default feature can be selected if one is not provided.

6.3.5 Map-of-map graph representation advantages

One of the benefits of map-of-map graph representations is due to the second mapping capability. We are now able to link source and edge to either a value or as mentioned in 6.2.5. This greatly enhances performance when we know for instance what the source node and the transition symbol is. These two elements form the key to the destination node i.e. the next start node. If we look at another example where we need to know what the next possible states of a program can be for a particular type of condition, we can very easily follow the double mapping to the set of states linked to the source state and the edge (condition).

6.3.6 Map-of-map graph representation disadvantages

The downside of the above advantage is that it is geared at a specific application architecture/solution.

6.4 Classification

The following diagram presents the classification system for mapped graph representations:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

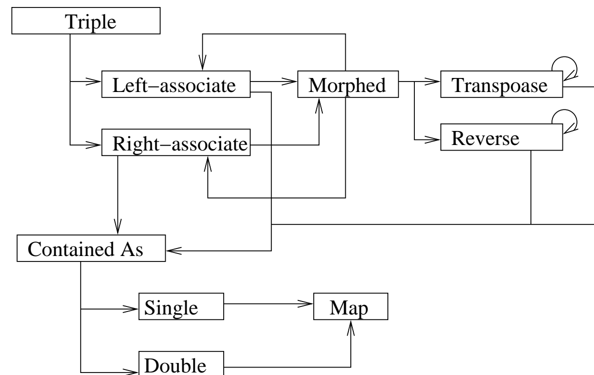


Figure 6.7: Mapped graph classification machine

Each section hereafter identifies and describes a mapped graph representation classification.

6.4.1 T-RA-CA-S-M

Identification

Triple is Right Associated and Contained As Single Map.

Description

This class of graph is a single mapped container. The container is a single mapping from a singular type to a compound paired type. An association is compulsory for this type of graph and therefore uses the right-associate formation for this representation.

6.4.2 T-LA-CA-S-M

Identification

Triple is Left Associated and Contained As Single Map.

Description

This class of graph is a single mapped container. The container is a single mapping from a compound paired type to a single type. An association is compulsory for this type of graph and therefore uses the left-associate formation for this representation.

6.4.3 T-RA-CA-D-M

Identification

Formation is Right Associated Contained As Double Map (Map-of-Map).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Description

This class of graph is a map-of-map container. The container is a single mapping from a single type to another map. The sub-map is a single mapping from a single type to another singular type. It is right associated since the sub-map forms an association to the right of the singular type. An association is compulsory for this type of graph and therefore uses the right-associate formation for this representation.

6.4.4 T-LA-CA-D-M

Identification

Triple is Left Associated and Contained As Double Map (Map-of-Map).

Description

This class of graph is a map-of-map container. The container is a sub-map to a single type. The sub-map is a mapping from a single type to another type. It is left associated since the sub-map forms an association to the left of the singular type. An association is compulsory for this type of graph and therefore uses the left-associate formation for this representation.

6.5 Summary

In this chapter we started out by broadly defining maps as being an association of one element with another. We also saw that we could nest maps by associating an element with another map element. Thereafter we incorporated this information into our toolkit by associating a source node with a paired edge and destination node. Later we included this concept with map-of-maps by associating an element such as the source node with another association. The key of this sub-association was then mapped to the value. This approach is useful when you have a source node that has a unique edge and destination combination.

It is possible that a mapping approach is required but multiple edge-destination pairs are required for the same source node. In this case a single map representation is used with the source node being associated with a collection of edge-destination pairs using an appropriate container such as a linked list, vector, set, binary tree, hash table or even a list of single maps.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 7

Hashed Graph Representations

7.1 Chapter Overview

Hash tables are yet another way to represent graphs. They allow efficient and fast access to data. The data boils down to being a triple formation as defined in sections 3.1.2, 3.1.2 and 3.1.2.

Hash tables provide access to elements through a hashing function. The hashing function has the specific purpose of calculating the index for a particular element. This hashing function is a component that should be as lean as possible to provide near instantaneous access to the data. Different forms of hash tables exist. We shall focus on the following types of hash tables:

- Function hash tables and
- Direct hash tables.

Although there is a third method (rehashing), it shall not be considered in this dissertation.

By function hash tables we mean that the hash table is implemented as a vector with buckets for collision resolution. These linear buckets could either be vectors, linked lists, sets or binary trees. This type of hash table also makes use of a hashing function component that calculates a base index that is in fact the index to the bucket that stores the specific element.

Direct hash tables are similar to function hash tables except that the direct hash table does not require a hashing function to determine the index for a particular element. Instead it reuses the element itself. In the case of the toolkit, it makes use of the first element type from the fundamental graph unit, the triple.

The chapter commences by looking at the structure of the different types of hash tables as was done in previous chapters. Following each structure is a brief list describing the characteristics in terms of the benefits and drawbacks of the particular type of data collection. After both types of hash table graphs have been discussed, we shall proceed to identifying different representations according to the applicable classification machine pertinent to hash table graphs. We will conclude the chapter with a brief review.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.2 Hash function graph representations

7.2.1 Type

The type defines the underlying type being the primary constituent in the hash table. It is also the primary type used by the bucket type. In terms of the toolkit, the type is a triple configuration as defined in sections 3.1.2, 3.1.2 and 3.1.2.

7.2.2 Hash function

The hash function is the primary driving force behind the hash table concept. It produces the appropriate index that classifies an element in the data structure by categorising it into a bucket.

7.2.3 Bucket type

The bucket type is the collision resolution mechanism employed by the hash table. Unless the hashing function produces a 100 percent unique hash index for an element there exists the possibility of collisions occurring. A collision is classified as an event arising when two or more elements are passed through the hashing function thereby producing for each of the elements the same hashing index. Clearly, they cannot occupy the same cell in the table. For this reason a bucket for each cell exists so that when such an event occurs the elements are inserted into the bucket of that cell. The idea is to have each bucket holding only a few items, that way searching the buckets for the correct element is very fast.

7.2.4 Hash function graph representation advantages

Faster lookups

Hash tables are a significant improvement when compared to linear and binary data structures. They are implemented using a random access container[1] in combination with buckets for conflict resolution. Whereas linear containers process elements consecutively one element at a time, the hash table implements a dictionary-like mechanism that allows one to define a key and link the key to the data being stored. It is in the nature of the random access container that elements are accessed directly - i.e. by means of their address or position identifier[2]. If, for instance, the key into the hash table is a string of characters i.e. a word or phrase, it first needs to be converted into an integer based value corresponding to the exact (if buckets are not used) or approximate (as in the case where buckets are used) location of the data associated with it (the key). This conversion mechanism is the hash function that takes as input a string of characters and produces the index into the table where the element can be found.

[1] Also known as an array and classified in this thesis as a linear data structure because it can still be searched using a linear method.

[2] We define the position identifier as the integer based index value used to access an element in the random access container directly

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The advantage is this: had we not used a hash function, we would have to search through the entire table for the key that we are looking for. Since the key is now a word, our search process is extended even further as we need to match individual characters until a complete match is found (in which case we have found the correct location) or until a mismatch is found (in which case we need to proceed to the next element). Using a hash function we are able to determine immediately that:

1. The element exists
2. Where it is in the container and
3. The data associated with the key.

7.2.5 Hash function graph representation disadvantages

Inefficient resource utilisation

As with vectors/arrays/random access containers, discussed in section 4.3, hash tables also suffer from the draw back of inefficient resource management. Usually, either too much or too little space is allocated for elements resulting in a sparse container being created or an over densely populated container. Under very controlled circumstances it is possible to use all the allocated memory more efficiently. However, this is subject to a very good hashing function, also known as a perfect hashing function[3]. In addition to that, the hashing function needs to produce indices such that the largest index represents the total number of elements and all elements then have an index less than or equal to the largest index. If these conditions are met, it may be possible to make optimum use of the resource consumption. However, this is a trade-off (inefficient resource utilisation) that may be negligible in certain applications where speed has priority over the amount of resources.

Costly expansion

As already mentioned, the hash table is based on the random access container and as such it is subject to the drawback that when the container becomes full, a three step expansion procedure is usually employed.

1. More memory needs to be allocated (usually double the size of the current container size).
2. The previous data is copied over to the new larger location.
3. The previous block of memory deleted.

Not only is this expansion costly in terms of time spent but also in terms of the amount of resources that is allocated.

[3]A perfect hashing function is one where the function produces a unique index for a key

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.3 Direct hashed graph representations

7.3.1 Direct hashed graph representation advantages

Simplicity

A great benefit of direct hashed table graph representations is that they are very simple to implement. They do not require a hash function. Instead, the index is known beforehand without calculation. Implementing the direct hashed graph representation requires little in terms of implementation by being implemented as a vector as the primary container and a continuing with the trend of having buckets to resolve conflicts. The buckets can be one of the types mentioned in this dissertation. If the index to a particular element is known, it can be accessed directly i.e. without calculation.

Performance

Due to the nature of this type of graph representation that lacks the use of hash functions, it has the ability to access elements in constant time[4]. For this to be the case the index needs to be known beforehand. In our case where we are dealing with triples that represent the linked nodes, the keys need to be integer based. For example, we could configure a graph that uses integers for the source and destination nodes and a string of characters to represent some information. The direct hashed graph representation would take the first parameter of the triple as its indexing type, which would in this case be an integer. Each element in the hash table contains a triple. When a lookup is to take place, the first parameter of the input that corresponds to the index type is used to access the element in the table (see figure 7.1).

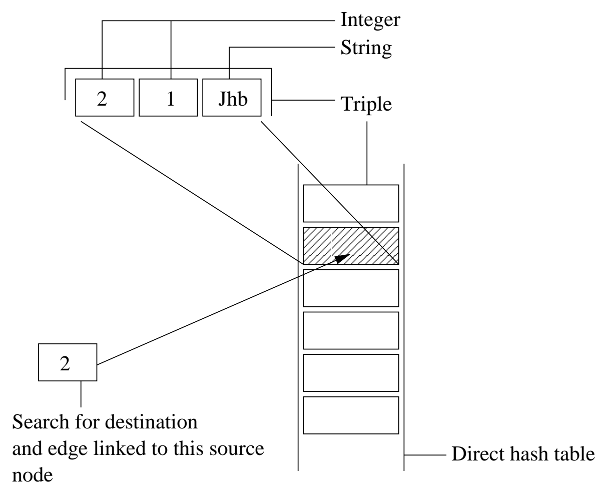


Figure 7.1: Direct hashing example

[4]If there are no collisions and no buckets being used. If buckets were being used collisions are catered for but immediately slows down the access time required to find an element as it requires conducting multiple searches.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.3.2 Direct hashed graph representation disadvantages

Restrictive

Although the direct hashed graph representation offers quick access to the contents it is restricted in terms of the types that can be used to represent the source and destination nodes and possibly the edge type too. C++ however allows overloading operators, one of which is the integer operator. By overloading this operator, the author of the C++ class allows an object to compute its own index. This could be construed as being a hashing function, which indeed it is but also masks the instance as an integer and can therefore be used in places where an integer is expected. To get around this calculation process (hash function) one could define the unique identifier as an auto increment integer by making it a class variable in which every instance that is created results in an increment of the identification number. The newly created identification number can then be assigned to each instance thereby allowing it to identify itself uniquely without the need for any calculations.

7.4 Classification

The following diagram presents the classification system for hashed graph representations:

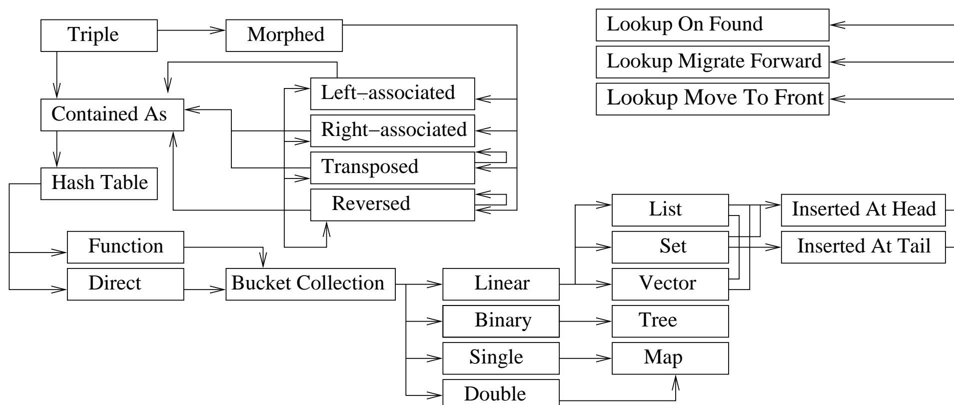


Figure 7.2: Hashed graph classification system

In each of the sections that follow we identify and describe a particular type of hashed graph representation.

7.4.1 T-CA-HT-F-BC-L-IAH-LOF

Identification

Triple Contained As Hash Table with a hash Function with Bucket Collection as Linear Vector with Insert-At-Head (insertion policy) and Lookup On Found (lookup policy).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Description

This class of graph is a hash table that uses a hashing function to compute the index where the element is inserted. The hash table uses several containers to address element index collisions. The bucket type for this type of graph is a vector with an insert-at-head insertion policy and an on-found lookup policy. This type of graph does not make use of isomorphisms.

7.4.2 T-CA-HT-F-BC-B-T

Identification

Triple Contained As Hash Table with a hash Function with Bucket Collection as Binary Tree.

Description

This class of graph is a hash table that uses a hashing function to compute the index where the element is inserted. The hash table uses several containers to address element index collisions. The bucket type for this type of graph is a binary tree. This type of graph does not make use of isomorphisms.

7.4.3 T-CA-HT-F-BC-S-M

Identification

Triple Contained As Hash Table with a hash Function with Bucket Collection as Single Map.

Description

This class of graph is a hash table that uses a hashing function to compute the index where the element is inserted. The hash table uses several containers to address element index collisions. The bucket type for this type of graph is single map graph (for more information regarding this refer to chapter 6). This type of graph does not make use of isomorphisms.

7.4.4 T-CA-HT-D-BC-L-IAH-LOF

Identification

Triple Contained As Hash Table with a Direct hashing function with Bucket Collection as Linear Vector with Insert-At-Head (insertion policy) and Lookup On Found (lookup policy).

Description

This class of graph is a hash table that uses a key value as the index where the element is inserted. The direct hashing function relies on the first element type as the index. This means that the key itself is used as the index into the hash table. The hash table uses several containers to address element index

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

collisions. The bucket type for this type of graph is a vector with an insert-at-head insertion policy and an on-found lookup policy. This type of graph does not make use of isomorphisms.

7.4.5 T-CA-HT-D-BC-B-T

Identification

Triple Contained As Hash Table with a Direct hash function with Bucket Collection as Binary Tree.

Description

This class of graph is a hash table that uses a key value as the index where the element is inserted. The direct hashing function relies on the first element type as the index. This means that the key itself is used as the index into the hash table. The hash table uses several containers to address element index collisions. The bucket type for this type of graph is a binary tree. This type of graph does not make use of isomorphisms.

7.4.6 T-CA-HT-D-BC-S-M

Identification

Triple Contained As Hash Table with a Direct hash function with Bucket Collection as Single Map.

Description

This class of graph is a hash table that uses a key value as the index where the element is inserted. The hash table uses several containers to address element index collisions. The bucket type for this type of graph is single map graph (for more information regarding this refer to chapter 6). This type of graph does not make use of isomorphisms.

Additional information

It is valuable to note that this representation uses the triple configured as left-associate or right associate. However, when the triple is configured as right-associated, there is minimal effort required to implement and configure it. If however the representation is configured as a left-associated triple a typecast to integer (for the index) has to be implemented for the association structure. This typecast implementation needs to then either return the first element type or some function of the two elements forming the key. Either way the representation requires some form of computation that will determine the index of the key. By implementing this function/typecast the representation really should fall under the Hash Table Function category.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.5 Summary

We started the chapter by defining function hash tables and direct hash tables and thereby outlining their differences. A function hash table was defined as being a table that uses a hashing function to determine the entry point for an element and a direct hash table was one that uses the first element as the index into the hash table. This second approach improves the searching capability as no computation for the index is required. The second part of the chapter described the different types of buckets that can be used by hash tables, namely: vectors, sets, lists, binary trees etc. Before selecting a particular bucket, the limitations of that bucket need to be considered to evaluate if it is best suited for the application.

Chapter 8

Meta Programming

8.1 Chapter Overview

This chapter deals with the different meta-programming techniques at our disposal. We examine in an incremental fashion, the utilities that are used by the generative toolkit. In this chapter we focus on:

- Partial specialisation. Here we define what it is and how to implement it with examples that serve as a reference to users.
- Selectors. Here we define what a selector is and what the primary selectors are. These are the selectors available to the toolkit.
- Traits. We define what traits are and the different methods of implementation. A set of advantages and disadvantages are provided.
- The section on compile-time assertions contains a discussion on the different types of assertions and how runtime assertions differ from compile time assertions. Since compile time assertions are necessary, it is shown how such compile time assertions may be implemented.
- Lastly we summarise the chapter by highlighting the important points discussed in the chapter.

8.2 Partial Specialisation

Partial specialisation is the process of redefining a template class such that the compiler selects the correct implementation based on specific values provided at compile time. An example will make this clearer. Consider a conversion system that accepts as input a floating-point value and outputs some converted value. A list of supported conversion methods are specified as converting from nautical miles to meters, meters to feet, feet to meters and meters to nautical miles. This problem may be solved in a number of ways. However, we restrict ourselves to a partial specialisation solution.

```
enum
{
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
  Nautical_Miles_To_Meters,
  Meters_To_Nautical_Miles
};
```

First the different conversion methods are associated with numerical constants:

- 0: Nautical Miles To Meters
- 1: Meters to Nautical Miles

Thereafter the various conversion classes are defined using an integer to identify the type of conversion. Notice that each time we specialise only the identification constant. This is the key to the specialisation process.

```
// default case shall not convert value
template< int id >
struct converter
{ double convert( double input ) { return input; }
};

// specialisation converts input (nautical miles) to meters
template<>
struct converter< Nautical_Miles_To_Meters >
{ double convert( double input ) { return input * 1852; }
};

// specialisation converts input (meters) to nautical miles
template<>
struct converter< Meters_To_Nautical_Miles >
{ double convert( double input ) { return input / 1852; }
};
```

In the end, we have code where several converters exist: a converter for nautical miles to meters (as above), one for meters to feet, one for feet to meters and one for meters to nautical miles implemented in a similar way as the first (nautical miles to meters). This specialisation approach is a key component in generative programming. None of these components have been selected for compilation yet and remain as possibilities to the compiler. Only the selected converter shall appear as final code. The others are discarded.

To be more clear and to see the advantage of this consider the following illustration. Traditionally one would create a *converter* class that has a conversion function for each conversion possibility (nautical miles to meters, meters to nautical miles, etc.), which in the above case is four. This *converter* class would typically contain the conversion factors for each conversion possibility. When used in an application one would typically only use one of these conversion functions at a time. Creating the *converter* object wastes resources.

By using the specialisation approach when we create the *converter* object, memory is allocated for what we are actually going to use.

In the "main" function that is described below, a specific type of converter is selected as the conversion method. What can be concluded from this is that the resulting software is smaller and precisely geared to solve a particular set of requirements, in this case to convert meters to nautical miles. (We are, of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

course, assuming that we do not need to also convert back from nautical miles to meters).

```
int main( void )
{
 converter< Nautical_Miles_To_Meters > x;
 std::cout << x.convert( 5 ) << " meters" << std::endl; // output = 9260
 return 0;
}
```

This approach provides a way to generate specific code based on a specific requirement with the result being a specific solution to a specific problem i.e. to convert nautical miles to meters in essence leading to less memory consumption when creating objects.

8.3 Traits

Traits are a group of settings that describe and define a particular type of class/object i.e. the properties of the class. They act as settings that influence the behaviour of a particular class. Three methods exist in which to specify and customise these traits, namely:

- Trait specialisation,
- Trait parameterisation and
- Trait lists (type list)

8.3.1 Specialisation

Through specialisation, a common base class for a specific type is created that contains a list of variables. These variables exist as types, constants and/or methods/operations. The methods may be pure virtual methods (thereby making the trait class an interface).

It should be noted that virtual methods may come at an extra cost. Through virtual function tables the appropriate function is found through a sequence of at least five instructions which is incurred for every virtual function call. Even though the instructions may be executed in parallel by the latest processors, it leaves no room for the compiler to optimise the code since the targets are not known at compile time. (For more information regarding this refer to [DH96]).

Once these variables have been specified, specialisation may take place as discussed in the previous section specialising only the necessary variables and leaving the remaining variables untouched.

It should be noted that this approach requires that all the variables present in the original set of traits should be included (copied) in the new set of traits (specialised traits). If these variables are omitted the specialised set of traits shall exist without them and shall form part of the specialisation. When one uses the specialised form, expecting the original values to be present and they are not, then the compilation will fail.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

8.3.2 Parameterisation

Through parameterisation, a list of the customisable features is specified as template parameters. The advantages of such an approach are:

1. Customisation occurs as needed at the point where it is needed.
2. The need for having predefined cases is extinguished.
3. The customisation results in a generic solution for defining traits of a specific type.

This approach has a notable drawback: the number of customisation points may be quite large. If there are many customisation points the solution becomes quite unfriendly unless there is another mechanism in place that eases the customisation e.g. supplying default values for various types and/or constants.

8.3.3 Trait lists

From the disadvantage presented by parameterisation, one other way creating traits is through a trait list (also known as type lists). Trait lists operate in a similar fashion to linked lists except that these now operate on types and constants. Although this is more complex and introduces more overhead than the previous two approaches, it provides an efficient way of specifying requirements (properties) for a class generator. The class generator becomes more complex because it needs to examine the trait list and see exactly what has been provided and what not.

To ease this approach, an intermediate approach can be introduced that extracts all the properties from the trait list and forms an intermediate class that contains the specified traits. This intermediate class can then be passed to the class generator that produces the final class. This approach is explained in depth in [Ale01].

8.4 Selectors

Selectors are also a crucial part to generative programming as they provide the compile time selection of one class over another based on a certain condition. Class generators make use of this technology extensively when deciding what to include and not to include in the final class. Two important selectors are examined here:

- **If** and
- **Switch**

The **if** selector makes a selection based on the condition being either true or false. It makes a selection based on two options only. The result of the selection is a type or a constant depending on the type of selector being used. A constant selector allows selection between constants based on some condition. A type selector allows the selection between types based on some condition. Both methods make use of partial specialisation to accomplish its goals.

The partial specialisation is based on a Boolean condition. If the condition turns out to be true, the first type/constant is selected. If the condition turns

out to be the specialised form (false) the second type/constant is selected instead of the first.

This selection approach is particularly useful when only precise types need to exist based on a certain condition. An example of this is shown below:

```
template<bool cond, class T1, class T2> struct IF {typedef T1 RET;};
template<class T1, class T2> struct IF<false,T1,T2> {typedef T2 RET;};

int main( void )
{
 IF<1 < 0, int, std::string>::RET value1;
 IF<1 > 0, int, std::string>::RET value2;

 // value1 = 1; error
 value1 = "1"; // okay

 // value2 = "1"; error
 value2 = 1; // okay
 return 0;
}
```

The **switch** statement has the same functionality as the **if** except it allows multiple types or constants to be tested based on a single condition thus resembling the runtime equivalent notion in terms of behaviour. It is implemented using a combination of partial specialisation and template recursion to achieve type selection from a list of types (type lists).

Another form of conditional code generation mechanism exists in C++. This mechanism is conditional pre-processor directives i.e. macros. The directives rely on previously defined values. These values may be defined as an additional compiler option or they may be specified in a certain definition file that contains definitions that together influence program construction resulting in specific types of programs being created. This approach is particularly helpful when programming in C that does not support meta-programming.

8.5 Assertions

Runtime assertions are fairly straightforward and well known. All that is needed is to include *assert.h*, after which assertions may be included at will. These assertions may be turned off during runtime to optimise code. The assertions are merely used to check that certain assumptions about the executing function/procedure are correct. These assertions however only become active when the program is executing and the certain condition is violated.

Compile Time Assertions in C++

Compile time assertions differ in this regard. They detect errors that may occur as a result of the erroneous configuration combinations e.g. a certain graph toolkit allowing the user to create a specific type of graph by providing loosely coupled bits of data. Here the user of the library/toolkit may be unaware that certain options only operate under certain conditions. In essence, errors are detected when the source code is built rather than when it is executing.

In this situation, the onus of checking could be shifted on to the user of the toolkit to ensure that the correct bits of data are provided resulting in the correct combination being generated. Assertions may help the user of the toolkit before execution of the application commences by generating a compilation error so as to indicate that an incorrect configuration was specified. This compilation error should be easy to understand by indicating which part of the configuration is incorrect.

An easy way to generate a compilation error is to deliberately create an instance of a class that has a private constructor. The error message however has to be conveyed somewhere in the name of the class being instantiated to indicate to the user of the library the nature of the problem. The following source code illustrates this:

```
#include <iostream>

template<bool _c, class _t1, class _t2> struct IF { typedef _t1 _rt; };
template<class _t1, class _t2> struct IF<false, _t1, _t2> { typedef _t2
_rt; };

class combination_invalid { combination_invalid(){} };
struct combination_valid { combination_valid(){} };

template<bool cond, class _t1, class _t2> struct assertion
{
  typedef typename IF<cond, _t1, _t2>::_rt _rt ;

};

enum { OPTION_1, OPTION_2, OPTION_3 };
template< int _a, int _b >
struct configurator
{
  typename assertion< _a == OPTION_1, combination_valid,
combination_invalid >::_rt _check_1;
  typename assertion< _b == OPTION_3, combination_valid,
combination_invalid >::_rt _check_2;
};

int main( void )
{
  configurator< OPTION_2, OPTION_3 > cfg1; // compilation error
  return 0;
}
```

Selectors serve this problem well since combinations may be checked and if a combination is found to be invalid (evaluates to false) an error type class may be created and instantiated. When the instantiation occurs and has a private constructor, the compiler will generate an error message to the effect of:

```
assertions.cpp:
In constructor ‘configurator<1, 2>::configurator()’:
assertions.cpp:6: error:
‘combination_invalid::combination_invalid()’ is
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
   private
assertions.cpp:25: error: within this context
```

From the excerpt it is easy to notice that a configuration object is being created with an invalid combination. In addition, the user could be directed to consult the user manual for further help with this error.

Compile Time Assertions In C

Compile time assertions may also be implemented in C using conditional compilation directives as explained above. If certain conditions are invalid, the **#error** directive may be used to generate a compilation error. The formatting of this error message may be more informative than the C++ compile time assertion technique.

The C++ compile time assertion approach is preferred as it deals with object orientation and it results in neater and more readable source code and fits in well with generic programming.

8.6 Summary

In this chapter we looked at the building blocks used in toolkit development in C++. We looked at generic aspects available to C++ in its different forms. Incidentally, it should be noted that the ability to be generic is present in other languages too such as Ada95, ML/2, $C\sharp$, Java and Modula 3. We discussed issues dealing with partial specialisation, which, in short is the ability to have a library of possibilities with specific possibilities being used at any given moment. This partial specialisation forms the basis of generative programming. It was further explained through two particularly important primitives: **if** and **switch**. We discussed traits and different approaches for implementing the traits. We outlined the strengths and weaknesses of these approaches and conclude that parameterisation is a good option if the number of customisation points is kept to a minimum. However, having a library of possibilities is also useful (as is the case with specialisation). We ended with a discussion on assertions. We conclude that assertions are useful tools to ensure that certain pre-conditions are maintained before the execution of code. We also found it helpful in the context of static meta-programming where checks may be made to verify that certain options are not in conflict with each other.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 9

Toolkit

9.1 Chapter Overview

The objective of this chapter is to provide an understanding of how the toolkit works by explaining how to use it. We also explain why it was implemented the way it was and what the benefits and drawbacks of the techniques are. This will give a practical approach to understanding how meta-programming is employed in a real application. By reading and understanding this chapter, one should be able to understand the pitfalls of generative and generic programming. The chapter also puts the theory of past chapters into perspective.

In the following sections we discuss the toolkit in terms of its features, describe its architecture and state the design decisions that were taken. We continue by describing different implementation routes that could have been followed.

Thereafter we define and explain the naming convention used for files, structures and types and the code itself in general used by the toolkit. These naming conventions could be useful in the work environment and the explanation thereof is provided to aid in reading and understanding the toolkit source code.

We then discuss the construction and syntax of the toolkit. This is where we explain how to manually create a graph instance in C++ as apposed to using a Domain Specific Language (DSL). For this, it is necessary to be familiar with the naming conventions used by the toolkit. To assist in using the toolkit, the construction process discusses constructing a neural network using a graph representation. We provide a walk through all the steps required to construct a graph. We conclude the chapter with a summary, indicating the highlights of the chapter and the lessons learned.

9.2 Features

In this section the toolkit is described in terms of the features that it provides to its user. We define a feature as being an aspect that the toolkit has to offer. Bearing in mind that the toolkit is a discovery on what generative programming has to offer we were able to produce the following benefits:

1. A compile time configurable toolkit.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2. An efficient class construction process.
3. A common interface that supports different representations.
4. Different ways to query the graph for specific data that could simplify graph algorithm implementation.

9.2.1 A configurable toolkit

The graph toolkit that is the focus of the research was designed to be highly configurable. The user of the toolkit is able to customise the operational aspects of the toolkit. These aspects, once customised, change the behaviour of the toolkit by changing the underlying representation and/or the specific operation implementation that manipulates the representation. For example, the user of the toolkit could create a particular representation as presented in this dissertation by stating the graph classification information in terms of template parameters. The generator takes the classification information and with the help of the compiler is able to produce a graph class matching the classification.

9.2.2 Efficiency

The implemented graph toolkit provides efficiency by integrating specific operations from a larger set of operations (the algorithm selection space). These selected operations collectively form the operational backbone of the graph toolkit. Although there are many representations, the result is a single compact interface supporting the requested representation.

What is the difference between this and inheritance? Inheritance requires that a new graph class be created for every type of graph classification. Common features are inherited from a common base class, particularly the common interface that each graph class should implement. Each new class must however specify its own graph element container (linked list, set, vector etc.) as each representation's algorithms operate differently depending on the type of container being used. When a user of the toolkit wants to use a particular configuration, he/she needs to include the correct file and provide the correct name of the graph (graph classification).

9.2.3 A common interface

As already mentioned, the toolkit is able to limit itself to a generic graph interface through the use of generative programming techniques. These generative programming techniques take advantage of the properties of meta-programming by allowing the compiler to extract from the solution space, components and algorithms that satisfy particular requirements. The solution is compiled according to configuration rules which are designed and specified beforehand by the author of the solution generator.

9.2.4 Different queries

Several queries are provided as part of the graph class. They provide access to the data stored in the graph from several perspectives. In some cases certain

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

query operations are more appropriate than others. However in all cases, all query operations are available.

9.3 Software architecture

The primary objective of this research was to develop a C++ meta-programming solution. We aimed at making the toolkit both generic and generative and finding ways in which these concepts cooperate best. Here we indicate at a high-level what is needed and how the different components in the toolkit are combined to co-operate with each other. We also show how STL is integrated into the toolkit. In subsequent sections we explain different implementation strategies that can be adopted to solve this problem.

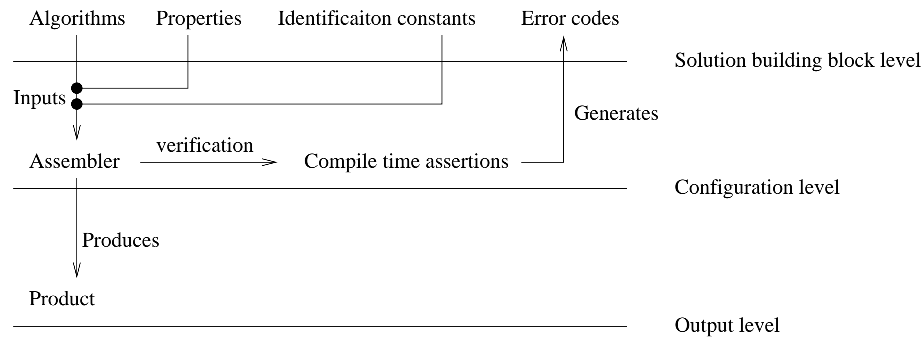


Figure 9.1: General generative solution architecture

In figure 9.1 we illustrate a general architecture for implementing solutions similar to the one in this dissertation. We have broken it up into three layers/levels (solution building block level, configuration level and output level). Starting with the solution building block level we find the primary components of the solution. Algorithms together with the properties and identification constants form the solution space. The final product is constructed from this solution space by selecting one or more elements from it.

9.3.1 Algorithms

The algorithms are loosely coupled components implementing a particular operation i.e. each component implements a particular algorithm.

Further benefits that are derived from using this architecture is seen in the example from chapter 8 in which we had an algorithm for converting nautical miles to meters and meters back to nautical miles. Each algorithm was embedded in its own class/structure and each was separate from each other. These two algorithms form part of the solution space.

9.3.2 Properties

Properties are loosely coupled definitions that provide information and additional types applicable to the final product.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

To be able to construct the final product, one may need various additional program types that are applicable to the product. Consider the example listing below that depicts different container types that can be used to represent a graph. Each container type is componentised in a similar manner to algorithm components i.e. the container types have a loose coupling.

```
template< class TRIPLE >
struct vector_based_graph
{ typedef std::vector< TRIPLE > container;
};

template< class TRIPLE >
struct list_based_graph
{ typedef std::list< TRIPLE > container;
};

template< class CONTAINER >
struct graph_iterator
{
  typedef typename CONTAINER::iterator         icontainer;
  typedef typename CONTAINER::const_iterator   cicontainer;
};
```

For instance, if one wanted specifically to create a linked list based graph one would define the type **container** from **list_based_graph** as the type to be used and not the one from **vector_based_graph** (see below).

```
template< class TRIPLE >
struct list_graph
: public list_based_graph< TRIPLE >,
  public graph_iterator< typename list_based_graph<TRIPLE>::container >
{
};
```

9.3.3 Identification constants

Identification constants are identifiers used by the generator/assembler and the user of the product to identify and specify requirements for a particular solution. As in the example expressed in 8 we defined constants that identify the different types of algorithms. This is one use of the constants but another is to use them to identify configuration errors. By passing the error constant into the configuration checker, if an invalid configuration is discovered, the corresponding error code can be generated. The user can then look up the error code and map it to a descriptive message and solution.

9.3.4 Error codes, assembler, compile time assertions and the product

Error codes are also part of the solution space. However, these codes are only used by the lower level components to indicate to the user of the product that something is wrong by providing a comment next to the error code describing

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

what causes that error, e.g. a user may attempt to configure a solution but finds himself/herself mixing the incorrect combination of items thereby resulting in a particular error code being generated. This is the job of the compile time assertions. The assembler takes a set of inputs originating from the solution building block level and verifies the configuration inputs through compile time assertions. If an invalid configuration has been provided an assertion will fail with a compile time error as the result. The compile time error would typically indicate the error code and the user would then look the error code up in some form of table or help file to determine what went wrong and possible how to fix the configuration. If everything succeeds however, the toolkit generates the final product as output from the input requirements.

9.4 Design decisions

In the previous section we described a general architecture to solve a generative problem. In the following two sections we provide two alternative solutions and provide the solution we used in the implementation of the toolkit. We also compare the different solution alternatives and state the advantages and disadvantages of each approach.

9.4.1 First consideration

As an alternative architecture, we may attempt using a pipeline to effectively build the properties that we would like to use instead of a once off specification. In this approach we feed the properties of the graph to the next filter via a specific input specification. The input specification is implemented by means of template parameters. Therefore the product created by the previous filter is the input to the next filter. Construction of the product is achieved by following an assembly line approach.

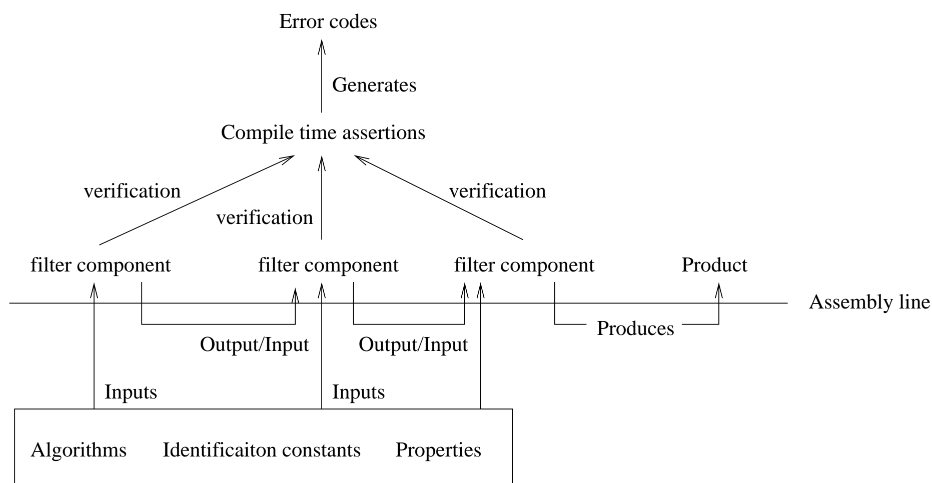


Figure 9.2: Generative pipeline solution architecture

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

It has the advantage that it allows the user of the toolkit to easily build a final product through specifying exactly what is needed. The downside however is that the construction procedure is very precise as it requires knowledge regarding which filter may follow another. This effectively places unnecessary requirements on the user. The user of the toolkit should be able provide his/her requirements to the toolkit in a standard manner as it enables the user to become familiar with the toolkit faster.

The pipeline architecture is implemented by means of inheritance where the next filter inherits the result of the previous filter until the final filter is reached. Once the final filter is reached the end product is constructed.

The pipeline architecture attempts to solve a larger problem (to produce a product) by letting individual components focus on their own subtask. Each component (filter) is expected to produce an output that is used as an input to the next component. Each filter generates part of the full product by contributing additional properties specific to that filter. The final filter therefore produces the end product. The listing below illustrates this.

```
#include <string>
template< class WHEEL_TYPE >
struct wheel_type
{
  typedef WHEEL_TYPE wheel_brand;
  wheel_brand wheel_id;
};
template< class WHEEL_TYPE, int WIDTH, int HEIGHT >
struct car_floor :public WHEEL_TYPE
{ enum { width = WIDTH, height = HEIGHT };
};
template< class CAR_FLOOR, int BODY_COLOUR, int NUM_DOORS >
struct body : public CAR_FLOOR
{ enum { colour = BODY_COLOUR, number_of_doors = NUM_DOORS };
};
template< class BODY >
struct car : public BODY
{ std::string name;
};
enum { black, green };
int main( int argc, char** argv )
{
  car<body<car_floor<wheel_type<std::string>,1,7>,black,4> > merc;
  car<body<car_floor<wheel_type<std::string>,1,5>,green,5> > golf;
  merc.name     = "SLK 500";
  merc.wheel_id = "Dunlop SP500 205/60";
  golf.name     = "GOLF 4 GTI";
  golf.wheel_id = "Continental 205/65";
  return 0;
}
```

9.4.2 Second consideration

Another design combines object orientation techniques with meta programming techniques. Techniques such as abstraction, encapsulation, inheritance and com-

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

position are used to manipulate types. Data members are implemented as **typedefs** and constants implemented as **enums**. Operations are implemented as **structs** and/or **classes**. **Structs** are useful as they represent their contents as **public** by default. In addition, they provide an implementation space where the manipulation may occur. In some cases, it is useful to have everything as public. However, if the implementation becomes lengthy it would be best to hide (or at least deny access to) the by-products. Here declaring a class is more appropriate. Although structs may also be implemented using compartments (public, private and protected sections pertaining to object definitions) it is more correct to implement them as classes since classes implement the members as private by default. The principle is this: if data is to be abstracted from the user, a class is more appropriate. If abstraction is not necessary then a struct is more appropriate.

In terms of implementation, one would typically implement one of the above architectures only in a more object oriented approach. It is certainly a very interesting approach that can simplify certain tasks but can easily complicate matters.

First we define the triplet inputs. These inputs are the bare bones of a **triple** and in effect the basis of the graph. These inputs are also provided as inputs to the mapping containers i.e. the triple compound type constituents. Hashed, linear and binary containers depend on the triple compound type as input. Once these containers are defined, they are provided as inputs to a particular manipulator. The manipulator may be considered as an assembly helper. What this means is the following:

1. Each manipulator is specific to the type of container.
2. Each manipulator has common functionality.
3. The manipulator provides specific query functionality and interface component functionality.
4. Isomorphisms are implemented by a manipulator.

A final generic graph interface is produced through the graph assembler. Instead of receiving the product of an isomorphism, the product is generated by the specific manipulator. Thus the manipulator is responsible for producing graph properties. Once the graph properties have been defined and given to the graph assembler the assembler has the same task as in the previous designs, to generate a generic graph interface. Some of this is illustrated in the listing below.

```
#include <vector>
#include <string>
template< class T1, class T2, class T3 >
struct triple
{ typedef T1 t1;
  typedef T2 t2;
  typedef T3 t3;
  t1 a;
  t2 b;
  t3 c;
```

```
  triple( t1 _1, t2 _2, t3 _3 ): a(_1), b(_2), c(_3) {}
};
template< class C >
struct basic_elements
{ typedef C ctr;
  ctr container;
};
template< class T1, class T2, class T3 >
struct vector_manip
{
  struct get_source { typedef T1 result; };
  struct get_edge { typedef T2 result; };
  struct get_destination { typedef T3 result; };
  // create a container type
  struct create_container_type
  { typedef typename
     basic_elements<std::vector<triple<T1,T2,T3> > >::ctr result;
  };
  // (isomorphism) transpose the triple and create a new basic element
  // type and leave the answer in result
  struct transpose
  { typedef vector_manip<T2,T1,T3> result;
  };
  // (isomorphism) reverse the triple and create a new basic element
  // type and leave the answer in result
  struct reverse
  { typedef vector_manip<T3,T2,T1> result;
  };
  struct ins
  { void operator()( T1 a, T2 b, T3 c,
                     typename create_container_type::result& C )
    { C.insert( C.end(), triple<T1,T2,T3>(a,b,c) );
    }
  };
};
template< class MANIP >
class graph
{
  typename MANIP::create_container_type::result ctr;
  public:
    typedef typename MANIP::get_source::result S;
    typedef typename MANIP::get_edge::result E;
    typedef typename MANIP::get_destination::result D;
    void insert( S s, E e, D d )
    { typename MANIP::ins i;
      i( s, e, d, ctr );
    }
};
int main( int argc, char** argv )
{
  graph< vector_manip<float,char,std::string> > g1;
  g1.insert( 1.0, '2', "3" );
  graph< vector_manip<float,char,std::string>::transpose::result > g2;
  //g2.insert( 1.0, '2', "3" ); //produces an error
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
  g2.insert( '2', 1, "3" ); // confirming transpose took place
  graph< vector_manip<float,char,std::string>::reverse::result > g3;
  //g3.insert( 1.0, '2', "3" ); //produces an error
  g3.insert( "3", '2', 1.0 ); // confirming reverse took place
  return 0;
}
```

The advantage of this approach is that it groups the functionality of each type of representation into a common location. Supporting new types of containers - i.e. new representations is a matter of implementing functionality of a manipulator that supports the requirements of the graph assembler and is based on the container. Drawbacks to using this approach is that it is hard to maintain and results in duplication of code and is very susceptible to introducing errors.

9.5 Graph construction

In this section we focus on construction of graphs. We look at what is required and how exactly to specify what type of graph we want. Generating a graph is fairly simple in the sense that it really only requires one or two steps. One step is required if all the steps are combined, or two if each step is specified individually.

9.5.1 Defining the problem

As an example, we consider a neural network application. We wish to construct a feed forward back propagation neural network. The neural network has one input, one hidden and one output layer. We wish to teach the network to recognise the binary **or** operation. We require two inputs and a single output. We require one additional input as a bias unit totaling to three inputs. Our hidden layer has a size of two neurons with a bias neuron totaling three. To summarise, we need a network with a $3x3x1$ topology. Our network is depicted in figure 9.3.

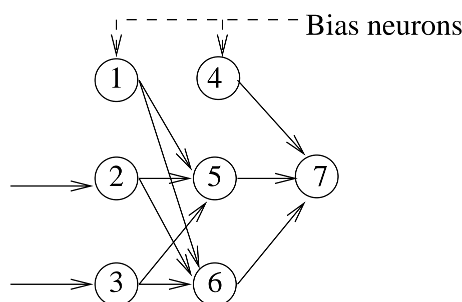


Figure 9.3: Neural network for OR operation

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

9.5.2 Define the objective

First we need to decide what it is that we wish to achieve. Here we need to decide between acceptable tradeoffs. Specifically, is one concerned with how long it takes to insert a million nodes or how much memory it would take to represent a million nodes. Refer to the advantages and disadvantages of each type of representation as discussed in chapters 4, 5, 6 and 7. These chapters identify the representation and provide information that assists in deciding what tradeoffs are acceptable for a particular application. Since our graph is small, we decide that the time to insert nodes is negligible and since we are going to perform numerous lookups, searching should be as fast as possible. We are not concerned with memory consumption for this example. We notice that each node is unique and decide that a hash table with linear buckets would be suitable for our application.

9.5.3 Creating the graph

After having decided what our objective is we construct the graph. What we need to do is the following:

1. Create graph properties through the graph traits specification unit (step 1),
2. Feed the specification to the assembler (step 2),
3. Declare the graph instance
4. Insert the nodes
5. Execute queries to solve the problem

Creating the graph properties

1. Creating the graph properties is like filling out an application form. Here we indicate what we want and what we do not want. First we start by indicating what operation is being applied to the traits (used by isomorphisms to transform the graph, the operation constant tracks what transformation operation was applied last to the graph properties).
2. Next we state our graph element types. The graph consists of nodes and edges. Each node is represented by an integer. The edges are real numbers that represent the weights of the connections. We define our basic representation as

$$R_1 = \{N_s, E_e, N_d\}$$

3. Next we state how we want our triple represented. We have decided to use a straight triple (standard triple form) as opposed to left associated and right associated triples. We have chosen this simply because it makes more sense to keep the triple standard and not associate the edge with a particular type of node. Also if need be, we could always execute the appropriate query to get the correct associations.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Graph element	Type
s	int
e	double
d	int

Table 9.1: Graph element types

4. Next we specify the bucket type. We currently have the option between four bucket types namely: linked list, vector, set and binary tree. We have chosen a linear bucket type and because of the access advantages of vectors we have chosen our bucket type to be a vector.
5. Next we specify the primary container. The primary container we have decided to use is a direct hash table. We specify this as one of the predefined constants available to use. We define our secondary container as none.
6. Since we are dealing with a linear container as our bucket, we define the following policies:
    - Insertion policy: insert-at-tail
    - Lookup policy: lookup-on-found
    - Hashing policy: direct hash

Assembling the graph interface

What is left now is to assemble the graph interface. We accomplish this by passing the traits that have been created to the graph assembler. The graph assembler is really just an assembly unit that selects the correct types required for the interface from the traits that have been provided. It selects:

- The container type,
- An iterator for the container,
- A constant iterator for the container,
- Insertion policies,
- Lookup policies,
- An insertion component,
- A lookup component,
- A deletion component,
- An in-domain query component,
- Various helper query components and
- Size determination component.

The graph assembler takes all of these selections into consideration and constructs a graph class based on its inputs. What we receive after assembly is a generic graph class that is constructed according to our input specification.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Declaring the graph instance

Declaring the graph instance is quite trivial, as the graph assembler provides us with the fully defined graph type based on the initial specifications. We use this resulting type and declare an instance in the usual way. The resulting type is the final definition of the graph class. Declaring an instance of it allows us to use the graph.

Specifying the graph contents

Finally what is left now is to specify the contents of the graph. Here we basically create the nodes and the connections between them. It is accomplished through the **insert** operation that requires only three parameters. In the order specified in the specification, we specify the source node, edge and destination values. Typically this would end up looking as follows:

```
graph_assembler< my_graph_type >::graph_type g1;
g1.insert( 1, (rand()%1000)/10000, 5 );
g1.insert( 1, (rand()%1000)/10000, 6 );
g1.insert( 2, (rand()%1000)/10000, 5 );
g1.insert( 2, (rand()%1000)/10000, 6 );
g1.insert( 3, (rand()%1000)/10000, 5 );
g1.insert( 3, (rand()%1000)/10000, 6 );
g1.insert( 4, (rand()%1000)/10000, 7 );
g1.insert( 5, (rand()%1000)/10000, 7 );
g1.insert( 6, (rand()%1000)/10000, 7 );
```

9.6 Summary

This chapter described a practical approach to graph construction. A list of features were presented indicating the usefulness of the toolkit. We focused on different architectures that may be employed to build a graph toolkit. We provided several design decisions for the architecture that was chosen. We discussed naming conventions, proposed our own set of naming conventions and why it is suitable for the implementation of toolkits. Finally we put everything in perspective by examining a real world problem of constructing a neural network using the toolkit. Here we indicated what was necessary to create the neural network (graph) and stated the steps necessary when doing so.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 10

Toolkit Comparisons

10.1 Chapter Overview

After having provided some background into graphs and examining aspects of our graph toolkit as a solution, the time has come to evaluate other available graph toolkits. In this chapter we focus on two alternative solutions to implementing graphs. The solutions that we look at are well known toolkits available on the market, namely LEDA and the Boost Graph Library (BGL). We look at the benefits and limitations of each type of implementation and its method of representation. Our toolkit is compared to these alternative solutions. The chapter aims to provide and evaluate the concrete ideas presented by each toolkit. In doing this we are able to provide an indication of which toolkit could be best suited for a particular application. After examining the advantages and limitations of each toolkit we will compare it to our toolkit. The chapter concludes by summarizing the important points in the chapter. In addition we also indicate how and where the Graph Template Toolkit can be improved.

10.2 LEDA

In order to solve combinatorial computing problems, one usually needs to make use of complex data types. Complex data types stand in contrast to the fundamental data types available to all languages. We define fundamental types as basic types provided by most languages. These are the types that are used to form complex data types. Generally, most languages provide a mechanism for representing integers, characters, real numbers, strings and Boolean types. These types that are being represented, are used to form new data types by combining them. Furthermore a need arises to group numerous instances of a type or complex types, usually achieved by adding the instances into a container or collection of instances. These collections are usually offered as additional packages and extensions to the programming language. Typically interpreter-based languages such as Python and Perl are examples of languages that provide collection types as part of the language, i.e. built into the language. Compiled languages such as C++ provide these collection types as extensions to the language by allowing them to be incorporated into the program being compiled.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The program being compiled would typically include the extension library into the program before it can be used.

Here is where LEDA comes in. LEDA is a commercial toolkit similar to the Standard Template Library for C++, with the exception that it does not make use of templates. It is aimed at being a high performance toolkit that supports many data structures and algorithms for those data structures. As of 2001, Algorithmic Solutions Software GmbH became the sole distributor of LEDA. LEDA provides several data types and distinguishes between the following categories:

- Simple data types,
- Dictionaries and priority queues,
- Graphs,
- Data types for two-dimensional geometry,
- Number types,
- Basic data types,
- Lossless compression,
- Data types for three-dimensional geometry,
- Graphics and
- Geometry window types.

Each category defines a list of classes that are applicable to that category. Number types, basic data types, lossless compression, data types for two-dimensional geometry, data types for three-dimensional geometry, graphics and geometry window types are not be covered in this thesis due to the lack of direct applicability to the research direction. Instead we highlight interesting aspects from the simple data types, dictionaries and priority queues and graph categories.

10.2.1 Simple data types

In the simple data types category, one finds definitions for the following types:

- Strings,
- Streams including file input and output streams as well as string input and output streams,
- Random sources,
- Random variates,
- Dynamic random variates,
- Memory management,
- Memory allocator,

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Error handling,
- Files and directories,
- Some Useful Functions,
- Timer,
- Two tuples,
- Three tuples,
- Four tuples and
- A date interface.

As can be noticed from this list, LEDA provides many classes that are also found in the Standard Template Library for C++ including a few additional classes. These classes are object oriented classes that usually do not use templates. Interesting aspects that stand out from the list are the way that memory management takes place, errors are handled and the type of tuples that are supported.

Memory management

LEDA memory management is handled by a garbage collector. Speed is achieved by de-allocating memory once the destructor for the memory manager is invoked. Memory may however be released at any time through a specific function call that returns the allocated memory back to the system.

Error handling

If an assertion fails, i.e. an error occurs the toolkit error handler is invoked. Before error handling can commence, a call to change the error handler may be required as the default error handler writes the error message to the standard error stream (*cerr*). LEDA also provides a way to handle errors as C++ exceptions.

Tuples

LEDA provides an implementation for several types of tuples. A tuple is a group of types. As a standard, STL uses *pair* to describe a two element tuple. As you may recall, our toolkit uses a tuple with three elements (a triple). LEDA also uses templates to represent the types of the tuple. It provides tuples that can house two, three and four elements. Each tuple is generic allowing the tuple to be composed of arbitrary types.

Alternatively, it is possible to represent generic tuples in **C** by using the generic pointer *void*. The disadvantage of this is that it lacks type safety - i.e. it is not possible to know what the pointer is actually pointing to unless each object has a unique identifier that can identify the object. The user of the tuple is required to know what is being stored in the pointer. Another big problem is that errors are only detected at runtime whereas template meta-programming allows the compiler to detect some of the errors at compile time. What is useful

about the LEDA tuples is that it provides a three element tuple that is similar to the triple type used by our toolkit. Remember the triple is the fundamental graph component in our toolkit.

10.2.2 Dictionaries and priority queues

We have chosen to highlight this category as it contains classes used by our toolkit. Classes such as hashing arrays, maps and two-dimensional maps are interesting because they are also used by our toolkit. It is interesting because our toolkit could in future be developed with LEDA as the backbone.

Hashing arrays

Hashing arrays are synonymous with hash tables and also employ generic implementation to accomplish their objectives. The hash table class is simplified greatly by requiring only two inputs: a hash function type and the arbitrary type being stored.

Maps

LEDA provides the STL equivalent of maps by operating in a similar fashion. It differs in the way in which it stores the data. LEDA stores the data using a vector relying on a direct indexing mechanism. This approach is similar to the approach adopted by our toolkit where it is presented as the direct-hashing graph.

Two-dimensional maps

Implementing two-dimensional maps in STL is easily achieved by reusing the single mapping class. It is not as intuitive as the LEDA approach but is more flexible. LEDA has specifically created a new class to handle two dimensional mapping. Another difference between LEDA's approach and the approach adopted by the STL is that LEDA requires that the indexing of an element is integer based whereas the STL allows indexing on any type. For instance, in the STL a two-dimensional map would allow the following:

```
Example 1:

.
.
.
std::map< int, std::map<std::string, std::string> > mapping;
mapping[ 0 ][ "hello" ] = "world";
.
.
.
```

This two-dimensional mapping maps the *0* and the string *hello* to the string *world*. LEDA on the other hand only allows the indexing parameter to be a pointer, item, or handle type or the integer type (**int**) which, in essence is a numerical indexing mechanism. This is accomplished as follows:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
Example 2:

.
.
.
map2< int, int, std::string > mapping;
mapping[ 0 ][ 0 ] = "world";
.
.
.
```

10.2.3 Graphs

LEDA defines graphs using the standard approach, using a list of nodes and a list of edges that are used to describe the graph. The list of nodes is implemented using a doubly linked list data structure. A graph is formed by using pairs of nodes from the list of available nodes and associating them with edges. This collection of paired nodes associated with edges then forms the graph.

The graph class is among the largest classes comprising of approximately 40 access methods. These are, in essence, methods used to query the graph in several ways. These access methods include among others, methods such as finding cyclic successors and predecessors of a given edge, adjacent successors and predecessors of a given edge, successor and predecessor nodes when given a specific node. The access methods also provide ways of retrieving a list of all the edges and nodes as well as adjacent edges (edges adjacent to a given node), adjacent nodes (nodes adjacent to a given edge) and outgoing and incoming edges for a given node.

In the same graph class, operations are provided to add, update and remove nodes. Interesting features available from the graph interface is the ability to hide graph nodes temporarily and restore them later. Additionally one is able to sort the nodes, sort the edges, move edges around, reverse edges and to convert the graph back and forth from being directed to being undirected graph. In addition to this, the LEDA graph system has support for face and planar maps as well as for writing the graph to an output stream and reading it back from an input stream.

Iteration through the graph is achieved through their *forall* macros. These macros iterate over nodes, edges, adjacent nodes, adjacent edges and incoming and outgoing edges.

Two other types of graph class interfaces are also provided by LEDA. They fall under the headings of parameterised graphs and static graphs. The parameterised version of the graph interface allows the incorporation of user defined information. The requirements imposed on the user of the toolkit are minimal in terms of what the user needs to supply to create a graph. The user is only expected to supply the node type and edge type. This approach limits the user to a homogenous node set which is practical most of the time. However, more flexibility is achieved by individualising the node types into a source node type and a destination node type. The disadvantage of this is that it requires more resources for the storage of the additional node type.

For example: if two node types were used such as an *integer* and *string* (to represent a reference point and a destination name respectively), more resources

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

could be required to represent the *string* (because its size is variable) when compared to replacing the *string* type with an *integer* type.

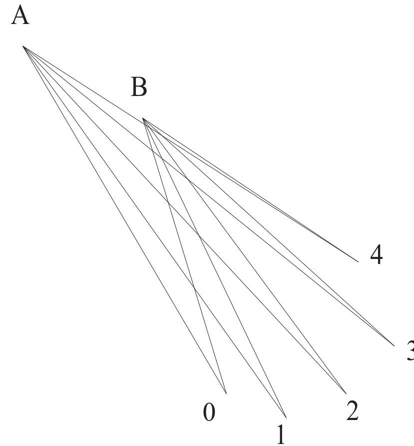


Figure 10.1: Common Point heterogeneous Graph

A practical use of having heterogeneous node types could be illustrated in the following example. If we wanted to find the distance between two cities and the route that we need to use in order to get there. One way could be to use a homogenous node set with the node types being of type *string* thereby allowing for the identification of the cities by node type and capturing the edge as a structure, housing information about the distance and the route to the destination node. This approach is perfect if one expects the city of origin or destination to change continuously. If only one changes continuously i.e. the source node (the originating city), then the destination node may be removed safely and replaced by one of the pieces of information housed in the edge structure i.e. the distance to the destination from the source, thereby becoming a heterogeneous graph. Now that the graph has been arranged with a common point, it is easy to query the graph for the set of all destination nodes (distance) and edges (route) for a given source node (originating city). Once we have this set of destination nodes and edges, it is easy to sort the set in ascending order to find the shortest distances and the route to travel from the given originating city to the known constant destination city.

Static graphs originated from the realisation that most graph algorithms work on a constant graph and that certain algorithms work on certain types of graphs. The static graph is based on a fixed sequence of nodes and edges. LEDA distinguishes between five classes of graphs, namely:

1. Directional,
2. Bi-directional,
3. Opposite,
4. Bi-directed and

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5. Undirected graphs.

At the time of this writing, only *directional*, *bi-directional* and the *opposite* graph categories were being supported. It is important to note that static in this context does not refer to the meta-programming approach that focuses on static programming but rather the emphasis is placed on the contents of the graph. The contents of the graph is therefore said to remain static i.e. constant. When using a meta-programming approach, the content of the graph remains constant at compile time whereas the approach adopted by LEDA, the graph contents remains constant at runtime.

Algorithms

Algorithms implemented by LEDA are generic and rely on parameterised graph instances created by users as input to the graph algorithm. At the time of this writing, the following graph algorithms were provided:

1. Basic,
2. Shortest path,
3. Maximum flow,
4. Minimum cost flow,
5. Maximum cardinality matching in bipartite graphs,
6. Bipartite weighted matching in bipartite graphs,
7. Maximum cardinality matchins in general graphs,
8. General weighted matching,
9. Stable matching,
10. Minimum spanning trees,
11. Euler tours,
12. Planar graph algorithms and
13. Algorithms for graph drawing.

10.3 The Boost Graph Library

Boost is open source software and therefore consists of software developers and reviewers operating with a peer-review approach to accepting specifically free C++ libraries from contributors. A team of reviewers led by a review manager assesses a particular library before finally accepting it as part of the Boost library collection. Boost therefore maintains a collection of freely contributed C++ libraries that have been reviewed and approved to be portable and work well with the C++ standard library.

The origin of the Boost Graph Library (BGL) stems from the Generic Graph Component Library (GGCL). The GGCL was part of a software project at the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Lab for Scientific Computing (LSC). GCCL came about due to a demand for reusable graph algorithms that were not available at the time from toolkits such as LEDA, GTL and Stanford GraphBase (SGB). The other reason is that these toolkits did not use generic programming (i.e. C++ templates) and did not fulfill the flexibility and high-performance requirements of the LSC. On 4 September 2000 BGL had a successful formal review and was released officially on 27th September 2000.

10.3.1 Types of graphs

In the BGL a distinction is made between two different types of graph classes, namely:

1. The adjacency list class and
2. the adjacency matrix class.

The BGL also offers several adaptors that change an existing graph to another form of graph, namely:

1. SGB Graph Pointer,
2. LEDA Graph,
3. Edge list,
4. Reverse graph,
5. Filtered graph and
6. A vector of Edge-List.

In addition to all of this, the BGL offers several traversal concepts, modification concepts and visitor concepts. The BGL defines a concept as being an interface - i.e. a concept is a way in which a Boost Graph may be examined and manipulated. The idea behind *concepts* is to encapsulate interface requirements for algorithms. This is done to improve algorithm re-usability by only including operations in the interface that are required for correct algorithm execution. The following is a list of the different types of concepts provided by the BGL.

- Traversal concepts: Traversal concepts concentrate on providing the requirements for traversing a graph using different approaches. These approaches are:
    1. Undirected Graphs,
    2. Graph (this concept is as a base concept defining types common to all graph concepts),
    3. Incidence Graph,
    4. Bidirectional Graph,
    5. Adjacency Graph,
    6. Vertex List Graph,
    7. Edge List Graph and

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

8. Adjacency Matrix.

- Modification concepts: Modification concepts provide the requirements that allow the contents of a graph to be changed - i.e. adding and removing vertices and edges as well as changing the properties of vertices and edges in the graph.

  1. Vertex Mutable Graph,
  2. Edge Mutable Graph,
  3. Mutable Incidence Graph,
  4. Mutable Bidirectional Graph,
  5. Mutable Edge List Graph,
  6. Property Graph,
  7. Vertex Mutable Property Graph and
  8. Edge Mutable Property Graph.

  [SLL02] may be consulted for further explanation and information regarding the above concepts.

- Visitor concepts: The BGL provides visitors for customising incidents of an algorithm i.e. it allows users of the toolkit to customise what happens at different points in an algorithm. The visitors are provided to achieve the same objective as functors (function objects which are really just call-back functions implemented as objects) in the STL.

  Visitors are described in [GHJV94] as hierarchies that define operations that operate on classes in a separate hierarchy. Simply put, an abstract visitor exists defining the operations that may be executed on specific types of objects (visitable elements). These operations (the ones defined by the visitor) are executed indirectly by the objects (visitable elements) supported by the visitor when the object *accepts* the visitor. In this way, a sort of call-back functionality is achieved by providing the visitor method access to the calling object (visitable element).

  The visitor design pattern has several benefits over function objects. Some of these benefits are listed below:

  – Visitors allow new operations to be easily added.
  – Probably more important about the visitor design pattern, it allows related operations to be grouped and separated from unrelated operations. For instance it is possible to have a hierarchy of different types of fruit, where each type of fruit is able to draw itself in a window. What the visitor pattern does is group the display functionality into a drawing visitor class that implements all the draw methods for the different kinds of fruit.

  If we quickly compare this to function objects we notice the following advantages of function objects:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

– Functors allow you to create call-backs in the form of objects. These type of call-backs are often more powerful than ordinary member function pointers (C++ specifically) that are used to implement call-backs to a method in an object.
– Functors allow easy and natural assignment and invocation techniques thereby improving the readability and understanding ability of the source code when compared to the complexity of the visitor equivalent solution.

The BGL provides several ways of traversing/iterating through the graph. This is done using different algorithms making use of the visitor design pattern.

1. BFS Visitor (Breadth First Search Visitor),
2. DFS Visitor (Depth First Search Visitor),
3. Dijkstra Visitor (Dijkstra shortest paths and related algorithms) and
4. Bellman Ford Visitor (Bellman Ford shortest paths and related algorithms)

10.3.2 Construction and usage

Constructing a BGL graph involves a sequence of steps - i.e. it follows an implicitly defined decision process. Firstly a decision needs to be made on whether an adjacency list graph or an adjacency matrix graph should be used. Adjacency lists are particularly useful when representing sparse graphs whereas the adjacency matrix would be more suited for dense graphs. A sparse graph is one where there maybe a large gap between nodes compared to a dense graph where entries are more tightly packed.

In a sparse graph the nodes are consecutively numbered, for instance from one to a million. However, only a handful of the actual nodes are used such as node one, ten, fifteen, thirty, seventy, etc. Thus the actual graph elements are quite scattered, making the adjacency list a good choice for implementation since there is no unnecessary resources spent on representing nodes that are not used or will never be used.

Once the type of graph class has been decided on, it is necessary to consider which type of container should be used for vertices and which type of container should be used for the out-edges. To make an informed decision on this, knowledge of each type of available container is necessary. (This is provided in this dissertation). It should be noted however that only STL compatible containers are allowed, therefore this includes: vectors, sets, lists, maps and hash tables.

The BGL then provides two ways of inserting nodes. In the first method a combination of calls to the `add_vertex`, `add_edge` and `tie` methods are issued. A call to `add_vertex` creates a vertex descriptor for a graph. Next a call to `add_edge` creates an edge based on two vertex descriptors which are then bound to an edge descriptor through the `tie` method.

In the second method creating the graph contents is more convenient. However, it requires that a data file containing the graph contents exists. A file input stream iterator is passed to the constructor. The input stream iterator is based on pair of integers that represent the vertices.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Once a graph is constructed with valid data, the BGL offers several ways of accessing and processing the data. This is done through several of their algorithms. At the time of writing, the following algorithms were in existence:

1. Breadth first search and breadth first visit,
2. Depth first search and depth first visit,
3. Topological sort,
4. Dijkstra shortest paths,
5. Bellman Ford shortest paths,
6. Johnson all pairs shortest paths,
7. Kruskal minimum spanning tree,
8. Prim minimum spanning tree,
9. Connected components,
10. Strong components,
11. Initialise incremental components,
12. Incremental components,
13. Same component,
14. Component index,
15. Edmund Karp max flow and
16. Push re-label max flow.

Now that we have examined two widely used toolkits, we focus our attention on evaluating how our toolkit (the GraTe-Tk) measures up against LEDA and the BOOST Graph Library. Specifically we evaluate the architectures and designs of the toolkits by outlining the advantages and disadvantages of each as well as where improvements can be made to our toolkit in the future.

10.4 The GraTe-Tk vs. LEDA

Evaluating the GraTe-Tk with LEDA we compare the architectures, representations and usability of the two toolkits. We outline similarities, differences and suggest possible improvements.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.4.1 Architecture

As discussed in chapter 9 we identified the GraTe-Tk as having a layered architecture. The advantage to this approach is that it allows us to categorise components of the solution into precise areas. By doing this we enhance the ease of future maintenance of the toolkit. According to [BMR+88] layering has more benefits: components in one or more layers are easily interchanged and can also be replaced with new components with a similar interface.

For example, the algorithm components shown in figure 9.1 could be replaced with further enhanced algorithms without affecting the identification constants, properties or even the assembler (provided the interface remains the same).

Furthermore, re-usability of components is enforced by the graph assembler (generator), which selects a combination of components from its preceding level as the inputs required to produce the end product.

LEDA too is implemented using a layered architecture. It has at least three layers with the building block layer consisting of its simple data types and support operations as well as the number types and linear algebra components. On the next layer are the basic data types (one and two dimensional arrays, dynamic integer sets etc.) that can make use of the components residing in the preceding layer. The next layer houses at least dictionary types, graphs and priority queues using components from the preceding layer.

As with the GraTe-Tk, LEDA lends itself to similar benefits when compared to the GraTe-Tk as it also has some of the benefits pertaining to a layered architecture. LEDA achieves re-usability by making use of components defined in previous layers. By making each of the preceding components independent from the succeeding components, the dependencies are localised.

According to [BMR+88], cascades of changing behaviour are experienced when using a layered architecture. Although this could be true for most systems, the GraTe-Tk doesn't have this problem as far as algorithm implementation is concerned. However when looking at the interface, there exists a strong dependency on input parameters and them having a specific interface. Changing the interface on a low level component certainly has implications on the functioning of subsequent levels thereby making the GraTe-Tk highly sensitive to changes on an interface level. However this type of failure is less critical as it would be detected early during software compilation.

10.4.2 Representations

In terms of representations, the GraTe-Tk allows a user to create numerous types of graphs (graph representations) as described in preceding chapters. As already mentioned, we consider a representation to be the memory configuration (memory layout) of the graph as well as the combination of features comprising the graph. By following the graph identification scheme presented in the previous chapters, we are able to precisely identify the type of graph being used, what its advantages and disadvantages are and what features it has.

LEDA on the other hand provides a limited number of implementations ranging from parameterised graphs to bounded node priority queues. In terms of the type of representations, LEDA provides node, edge and face arrays and maps, node matrices, two dimensional node maps, edge sets. It also provides node sets, lists, partitions and priority queues. It lacks configurability in terms

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

of the way in which the algorithms operate i.e. you are not able to configure the behaviour of an insertion operation as we are able to do in the GraTe-Tk. In terms of the operational interface, LEDA has an extensive graph instruction set. Unfortunately, this leads to an information overload as well as to the need to unnecessarily include methods that are rarely used. The real advantage from this research is that we have seen that we can dramatically decrease the Application Program Interface (API) size through meta-programming. The LEDA toolkit would have significantly benefited by using a generator approach as in the GraTe-Tk and by using graph traits (properties), allowing the user of their toolkit to specify exactly which methods he/she intends using. These traits could then be passed on to the graph generator that would build the required graph interface, exposing only the methods that have been requested. A disadvantage of this approach however is that in do so the user has to know beforehand what the names of the operations are that have been requested. Failure to do so would be like working in the dark, unless, of course, the client was relying on an Integrated Development Environment (IDE) that supports code insight technology [1].

10.4.3 Usability

The GraTe-Tk presents a user friendly API that is consistent for all types of representations. The interface is not bloated with too many unnecessary operations and functions. It also provides basic functionality present in all containers for inserting, removing and retrieval. A possible change that could be made is to make provision for further specification so that the user may indicate in the interface the type of retrieval methods that are to be included. This will make the interface more compact and precise and easier to use.

As mentioned before, LEDA has an exhaustive list of possible operations that can be performed on a graph. Most of the operations may never be used. Cluttering the interface with operations that are rarely used, increases the complexity unnecessarily. It makes searching for the correct operation tedious and laborious. In terms of usage, LEDA has undoubtedly impressive flexibility in terms of what can be done with the graph.

10.5 The GraTe-Tk vs. the BOOST Graph Library

10.5.1 Architecture

The BOOST Graph Library (BGL) also makes use of a layered architecture where the graph types are manipulated by graph algorithms or graph concepts. As the algorithms are generic in nature they accept inputs from the graph type layer quite easily. The BGL also uses visitors for traversal purposes. It however

[1] Code insight is a technology included with most modern IDE's that assists developers in looking up programming elements within the current programming context. A developer would for instance want to invoke a method on an object but does not know what it is called. When indexing the instance a window appears listing the possible operations, events, attributes for that object.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

is not implemented precisely as outlined according to the pattern defined in [GHJV94].

According to [GHJV94], using the visitor pattern entails that one typically creates an abstract visitor class with concrete visitors implementing the interface of the abstract class. Next one would have a base class that implements an acceptance operation. The acceptance operation typically accepts any visitor object that implements the defined abstract visitor interface.

In a generic environment such as the BGL, they very cleverly make use of an algorithm visitor class that provides multiple call-back member functions[2]. Doing away with the acceptance method, the BGL includes it as a parameter to the call-back function. Algorithms that make use of the visitor to manipulate the contents of the graph basically state that they require a visitor and implement the algorithm using an implicitly defined visitor interface as their algorithm base. Users of this algorithm would then provide the graph that should be manipulated as well as the concrete visitor that operates on that graph. Agreeably, alternative solutions to this approach in a generic environment would not be as effective and efficient.

In short the architecture used in the BGL easily allows for additional graph classes and user defined classes to be defined provided that they follow the precedent set by the original set of graph classes. This is necessary as the algorithms are dependent on the interface provided by the graph classes. This architecture differs dramatically from the GraTe-Tk that has a single graph with a specific interface. Although graph algorithms are not provided at this time, catering for them would either be done by extending the interface or following the trend set by the BGL and STL with generic algorithms operating on a class. By extending the interface we stand the risk of bloating the interface similar to LEDA. Exporting the algorithms and making them generic in terms of what type of graph they operate on allows us to add algorithms incrementally without impacting on the interface of the graph. This is an ideal approach. However a third approach can be examined in which we allow the user to specify what algorithms he/she requires. The graph generator would then include the appropriate methods into the graph interface. This approach results in an efficient implementation scheme - i.e. the interface is not bloated. It also makes more sense from an object oriented point of view because the algorithms are actually linked to the graph and operate on the graph container. Therefore it makes more sense to keep the operations embedded in the primary graph class. When we eventually decide to provide algorithms for the GraTe-Tk, the last approach would be the approach that would receive more favourable consideration.

10.5.2 Representations

In terms of the number of representations that the BGL provides, we find that it is limited to adjacency matrices, adjacency lists, edge lists and vertex lists. These types of graphs can be represented as directed, undirected and bidirec-

[2]In STL the call-back function mechanism is called a functor i.e. a function object. Practically the function object is implemented as the parenthesis operator allowing instances of the object to be invoked as normal functions/methods. This method is typically used to implement policies as described in chapter 3. The BGL has found this to be a limitation when trying to have multiple call-back functions within the same object and therefore have implemented the call-back functions as descriptive generic member functions.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

tional graphs. Although the BGL is currently limited to these types of graphs, it does however allow for the different implementation containers such as a linked list, set, vector etcetera. In this respect it is similar to the GraTe-Tk. It however does not support isomorphisms that allow the graph configuration or graph triple configuration to be automatically changed. This is due to the type of implementation scheme adopted by the BGL that implements graphs as either an adjacency matrix or an adjacency list etc. By representing the graph as a triple we are also able to cover more graph configurations through isomorphisms thereby affecting the total number of representations that are supported.

10.5.3 Usability

The typical way one would create a BOOST graph is by simply stating the type of graph to use - e.g. adjacency list - and create an instance of it. However this is in its simplest form. One could specialise the implementation of the adjacency list to use a specific container to represent the list of edges and vertices e.g. the edge list could be represented as a linked list and the list of vertices could be represented using a binary tree or a vector. Thereafter, nodes can be added to the graph by invoking an operation that adds an edge. One provides the source and destination node values to the operation as well as the graph object to be affected. This is a very short setup process when compared to the GraTe-Tk that is a three step process, requiring that you first specify how the graph should be configured followed by the graph interface generation process. Only then are we ready to insert nodes and edges to the graph. Once the graph has been configured and an instance created the GraTe-Tk is very straightforward to use as it has a standard way of manipulating the graph in an object oriented manner. The GraTe-Tk graph is still generic as it also supports arbitrary types of nodes.

10.6 Summary

In this chapter we examined the features of the commonly used toolkits such as LEDA and BOOST and compared them to the GraTe-Tk. We noticed that the LEDA, BOOST and GraTe-Tk graph toolkits make use of a layered software architecture allowing components in one or more layers to be easily interchanged or replaced by components with similar interfaces. We then compared the GraTe-Tk toolkit in terms of the number of representations that can be created and found that the GraTe-Tk outweighs the LEDA and BOOST graph toolkits in terms of the number of representations that may be created. LEDA has a limited number of implementations, such as: node, edge and face arrays and maps, node matrices, two dimensional node maps, edge sets. It also provides node sets, lists, partitions and priority queues. We also found that the GraTe-Tk is more configurable than the LEDA toolkit. When comparing the number of representations to the BOOST graph library, we found it to be limited to adjacency matrices, adjacency lists, edge lists and vertex lists which can be represented as directed, undirected and bidirectional graphs. Although the BOOST graph library is currently limited to these types of graphs, it does however allow for different implementation containers such as linked lists, sets, vectors etcetera. Comparing the GraTe-Tk to LEDA and BOOST in terms of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

its usability it is found that the GraTe-Tk has a clean non-bloated application programming interface with a consistent set of operations that apply to any type of representation i.e. it has one interface for any representation, whereas the LEDA has an exhaustive set of operations that can be performed on a graph. BOOST on the other hand allows you to easily create a simple graph in a non-complicated manner. Although it is not very complex to use, it is not object oriented and requires generic functions to manipulate the graph type. Both the BOOST graph library and the GraTe-Tk are efficient in this respect except that the BOOST graph library loses its flexibility when needing to extend the functionality and has to revert back to a structured programming paradigm to achieve this, whereas the GraTe-Tk can be easily extended through inheritance as it is object oriented while still remaining generic. In this chapter we noticed that each toolkit has its limitations and its areas that it excels in and that there are lessons to be learned from each toolkit implementation.

Chapter 11

Conclusion

In this chapter, we will be reflecting on what our initial objectives were and then we will discuss how we achieved these objectives through this dissertation.

11.1 Retrospection

When we started out with this research we had several objectives in mind. We split our objectives into primary and secondary objectives and set out to achieve the following primary objectives:

- To discover a different approach to implementing the graph data structure.
- The focus would be on directed graphs and how to implement them in C++ using static meta-programming techniques.
- We would concentrate on providing a generic and generative toolkit that provides access to directed graph representations.
- We would find a method of naming the graph representations, such that it is easy to identify the graph representation as well as being able to configure the exact representation by looking at the name.

We also had the following secondary objectives:

- We would provide a set of querying functions. A querying function would typically provide a set of nodes, edges or a combination thereof while requiring a given node, edge or a combination of the two.
- The toolkit would be designed to be easy to use in terms of construction.
- The user of the toolkit would provide a set of requirements necessary to construct a graph with a specific representation.

11.2 Evaluation of Objectives

In this section we will evaluate the above mentioned objectives and discuss how we achieved these. If these objectives were not met we will provide at least one possible solution that could be used to work around the problem.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

11.2.1 A different approach

We aimed to find a different approach to implementing the graph data structure. In chapter 2 started by defining what constitutes a directed graph. We defined a graph as being a set of triples having a source node, an edge and a destination node. We then defined the concept of isomorphisms that can be used to change the representation of our directed graph and defined four isomorphisms, namely: transpose, reverse, left associate and right associate. We then defined the operations that our graph toolkit would provide, namely: insertion, removal, querying, determining the size of the graph and detecting if an element exists in the graph. At the end of this chapter we had discovered and defined a new approach to implementing directed graphs and laid the foundation for the next objective.

11.2.2 Implementation techniques

Having accomplished our first primary objective, we shifted our focus to determining mechanisms that would be best implemented in C++ using meta-programming techniques. In chapter 3 we found a useful mechanism to achieve the configuration aspect of the toolkit, namely, policies. We decided to use policies through parameterisation i.e. template parameters. Here we also defined what types of policies would be implemented and how they would operate. We also defined the requirements of container classes in terms of what operations should be available as a minimum to achieve a consistent implementation interface. In chapter 8 we expanded on meta-programming concepts and techniques that we used to implement the toolkit.

11.2.3 Different representations

After defining the platform and techniques that would be used to implement the graph toolkit, we defined the different representations that would be implemented in chapters 4, 5, 6 and 7. We looked at these representations from a feature oriented perspective, evaluating the advantages and disadvantages of each. We also analysed the Standard Template Library implementation of these underlying containers and compared them with ideal corresponding containers.

11.2.4 Naming conventions

Our next objective was to create naming conventions that could be used to uniquely identify each representation. The naming convention uses a graph to assign a name to a particular representation. Instead of being a hierarchical structure as traditional taxonomies, this form of naming convention allows the graph representation to be identifiable from the abbreviated form as well as the expanded form. The reason for this approach stems from the isomorphic property of our toolkit which caters for repeated isomorphic transformations on the original representation. At the end of chapters 4, 5, 6 and 7 we provided the naming convention graph and provided sample representations as well as the type of graph such a representation would be.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

11.2.5 Secondary objectives

As already mentioned, we defined a set of querying functions in chapter 2 that would be provided by our graph toolkit. We aimed to create a toolkit that would have an easy construction mechanism. During our implementation cycles, we found that the complexity of the static meta-programming mechanism provided by C++ can be quite development intensive, sensitive to change and difficult to maintain. Despite the complexities surrounding static meta-programming, it does offer excellent efficiency in source code reuse, memory management and toolkit construction (the end product would be more easy to use and understand, whereas the internal structure might not).

These complexities can be overcome by the user becoming more familiar with the domain as well as with meta-programming techniques in C++. Construction of the toolkit could further be simplified by the introduction of a Domain Specific Language (DSL) that acts as a domain language compiler. The DSL would understand for instance a set of instructions relevant for constructing, populating or manipulating and retrieving data from a graph toolkit. The language used by the DSL would be more user friendly than the underlying C++ source code. The DSL would then take the domain specific program and translate it into correct C++ code, thereby reducing the complexity and maintenance of the actual C++ code as the user of the toolkit would only interact with the DSL program.

11.3 Summary

In conclusion, after reflecting and evaluating the objectives, both primary and secondary, we were able to achieve all of our primary objectives and the majority of our secondary objectives. We have also provided a possible solution to the complexities arising from using our approach and feel that these complexities are limited and could be overcome with time and further experience.

Bibliography

[Ale01] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Professional, 2001.

[Amm95] L. Ammeraal. *C++ For Programmers*. Wiley, 1995.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques and tools*. Addison-Wesley Professional, 1988.

[BMR+88] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1988.

[BS03] Gabor Barla-Szabo. A taxonomy of graph representations. Master's thesis, University of Pretoria, 2003.

[BSWK04] Gabor Barla-Szabo, Bruce W. Watson, and Derrick G. Kourie. A taxonomy of directed graph representations. *IEE Proceedings - Software*, 151(6):257–264, November 2004.

[Bud94] T. Budd. *Classic data structures in C++*. Addison-Wesley Professional, 1994.

[CE00] K. Czarnecki and U. Eisenecker. *Generitve programming: methods, tools, and applications*. Addison-Wesley Professional, 2000.

[CL86] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman and Hall/CRC, 1986.

[DH96] K. Driesen and U. Holzle. *The Direct Cost of Virtual Function Calls in C++*. 1996.

[Dig94] Readers Digest. *Oxford Complete Word Finder*. Reders Digest, 1994.

[ESD96] R. Eberhart, P. Simpson, and R. Dobbins. *Computational Intelligence PC Tools*. Morgan Kaufmann, 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elemens of reusable object-oriented software*. Addison-Wesley Professional, 1994.

[Mor98] C. Morgan. *Programming from specifications*. Prentice Hall, 1998.

[SLL02] J. Siek, L. Lee, and A. Lumsdaine. *The Boost graph library: user guide and reference manual*. Addison-Wesley Professional, 2002.