Chapter 7: AddAtom implementation

The AddAtom algorithm evolved over a number of years and eventually led to the notion of a compressed pseudo-lattice. The author was responsible for modifying and maintaining the implementation environment of the Grand algorithm of Oosthuizen (1991). Originally, most of the utility data structures and supporting code for both AddAtom and compressed pseudo-lattice implementation was derived from the Grand algorithm's implementation environment but over time this was completely redeveloped since Grand and AddAtom use completely different strategies. The code was initially developed in C but was later completely re-written in an object-oriented fashion in C++. The most important C++ classes used in the implementation are the set and lattice classes respectively (sections 7.2 and 7.3). In addition to the basic functionality of lattice construction, a number of other functionalities, such as the caching of the lattice structure to disk, were also implemented.

This chapter outlines the class functions; the tradeoffs made when implementing the algorithm; as well as important features of the code such as memory optimisation, caching of results, etc. It ends with the discussion of several other implementation issues that were addressed.

Note that the code used for the wider performance comparisons in chapter 5 is not discussed here. The AddAtom implementation for that particular exercise relied on the pre-existing code base of S. Obiedkov that had been used for prior comparative studies of lattice construction algorithms (Kuznetsov and Obiedkov 2001 and 2002). Also note that the algorithms for the pilot and wide experimental studies in chapter 5 were different. The pilot study implementation used the additional optimisations of appendix A whereas the wide study used the basic algorithm of section 4.6.

An object-oriented notation is used for functions and not the functional notation used in previous chapters.

7.1 EVOLUTION OF CODE

The code from which the most recent implemented version of AddAtom emerged, has evolved over a long period of time, from 1995 to 2001. The coded was used by various researchers in areas of related to lattice construction, machine learning and machine translation. Various additions and extensions were implemented during this period of time. Three major versions of the code were produced:

Version 1: The original Grand algorithm was described in Oostuizen (1991). This algorithm used heuristics to construct a lattice. The algorithm itself does not have much in common with the AddAtom since the Grand algorithm used a number of heuristics and consisted of a procedure that connected an object concept to each of the attribute concepts in its intent, one attribute at a time, whereas AddAtom considers all attributes of the object at once. A function called transform was used to ensure the integrity of the lattice in having unique suprema and infima. Grand was implemented in C. This code was maintained by the author and adapted for use in machine learning and other experiments.

Version 2: The AddAtom algorithm was implemented in C using the Grand data structures, but with all the Grand lattice construction code completely replaced by the

AddAtom algorithm. The primary reasons for this was to modularise and clean up the version 1 code. In addition, the compressed pseudo-lattice data structures and functions such as virtual arcs, CompressLattice and ExpandLattice were also implemented. This all extensions to the code (relative to version 1) were completely written the author.

Version 3: The object-oriented version of the version 2 code. All data structures were completely rewritten and the option of optimising whether on time or space complexity depending on the application. In order to provide the maximum flexibility a significant amount of modularisation and encapsulation was used in the class design. Version 3 was implemented in C++ by the author. A number of different compressed pseudo-lattice operations such as ExpandLattice were also improved. The primary reason for this rewrite was to obtain flexibility to use the code in many research projects.

7.2 SET CLASS

The most important class of the implementation (other than the lattice class itself) is the set class. It contains implementations set operations such as union, intersection, difference and set complement on sets of integers. These are the basic operations in the implementation of AddAtom and other lattice related algorithms.

Each concept in the lattice is identified by a unique integer. A set may contain any number of concepts. In addition, representation of infinite sets is also supported. (An infinite set would be required to represent, for example, the complement of an empty set.)

The set class represents the set as a string of bits and set operations such as union (or) and intersection (and) can therefore be very efficiently calculated using normal bitwise processor operations that operate on 32 or 64 bits at a time.

The most important methods of the set class are (with parameter names implying types):

- Initialise(anInitValue)
- And(aSet) Return aSet
- Or(aSet) Return aSet
- Not() Return aSet
- FirstElement() Return anElement
- NextElement(anElement) Return anElement
- AddElement(anElement)
- RemoveElement(anElement)
- TestIfSetContains(anElement) Return aBoolean

7.3 LATTICE CLASS

In the lattice class, the lattice is represented by a list of nodes. Each node is numbered and this number serves as the index to that node. Each node has a number of attributes such as its name and support (i.e. the number of concepts in its extent). In addition, each node has two sets that contain references to its parent and child nodes. This represents

the arcs associated with the node. In the case of compressed pseudo-lattices each node has two additional sets that contain its virtual parent and child nodes.

In principle, a lattice can be represented as a set of sets (i.e. a set of nodes where each node is a set of attributes representing the associated concept's intent) without explicitly representing the cover relationship (parents and children). Pragmatically, however, the explicit representation of the cover relationship is irnportant for the efficiency of the AddAtom lattice construction algorithm. Without it, cover relationships have to be rediscovered/inferred each time the lattice is traversed.

Additionally, and also for efficiency reasons, the intent and extent sets of a node are stored explicitly with each node, despite the fact that these sets can be derived from its upward closure. The decision to explicitly store these sets represents a trade-off between space and time efficiency: to save on time, the sets are precomputed and stored, costing space. This developed into a lattice operation cache. The purpose of the cache was to keep the results of lattice closure operations in memory or on disk and avoid. This cache of pre-computed values significantly improved the performance of the lattice construction algorithms as well as the browsing and traversing thereof.

The upward- and downward closure operations are important lattice operations. They return all the nodes above or below a node, respectively, and include the node itself. In addition to these standard closure operations, a number of variations were also implemented. These involve, for example, returning nodes encountered in a given direction (upwards or downwards) when:

- Following a maximum number of successive arcs.
- Following only lattice arcs in a compressed pseudo-lattice.
- Following only virtual arcs in a compressed pseudo-lattice.

The AddAtom function in the lattice class inserts a new object into the lattice. It implements the AddAtom lattice construction algorithm described in chapter 4. It inserts the object as a new node, creating and connecting the new intermediate nodes that are required to retain lattice properties.

Due to the symmetry of lattices, all functions have duals that operate in the opposite direction (e.g. UpwardClosure and DownwardClosure; AddAtom and AddCoAtom). Rather than having separate functions for each of these, an additional parameter (aDirection) was added to each of the methods. This parameter identifies the direction of the operation (i.e. either upwards or downwards).

A number of the functions have been overloaded so that the functions can be called either with a single concept as parameter or with a set of concepts (e.g. instead of Closure returning only the upward closure of a single node, it can return the union of the upward closures of a set of nodes).

The following Lattice class methods were implemented:

- Closure(aNode, aMaximumLevel, aDirection) Returns aSet
- UpwardClosure(aNode) Returns aSet
- DownwardClosure(aNode) Returns aSet
- EIR(anAttrSet, aNode, aDirection) Return aSet
- AIR(anAttrSet, aNode, aDirection) Return aSet

- AddAtom(aAttSet, anObject, RootNode, aDirection) Return aNode
- AddCoatom(aObjSet, anAttr, RootNode, aDirection) Return aNode
- CreateNewNode(aNodeType) Return aNode
- Link(aNode1, aNode2, aDirection)
- DeLink(aNode1, aNode2, aDirection)
- GetNodeIntent(aNode1, aDirection) Return aSet
- GetNodeExtent(aNode1, aDirection) Return aSet
- GetNodeType(aNode1, aDirection) Return aNodeType (e.g. returns all attributes is aDirection = upward)
- GetAllAttributes(aDirection) Return aSet
- GetAllObjects(aDirection) Return aSet
- Join(Attrs, aDirection) Return aNode
- Meet(Attrs, aDirection) Return aNode
- GetMinimalConcepts(aSet, aDirection) Return aSet

The PersistantLattice class was used to implement functionality such as persistency and the caching of concepts and closures. Over the period of time the code was developed a number of different lattice and other classes with various functionality and optimisations were developed. Although this approach was beneficial for experimentation, it did negatively affect the performance of the implementation. This was due to the increased number of function calls which required a significant amount of parameter passing and range checking.

Various additional utility methods were also irnplemented to perform a number of commonly used basic functions. An exhaustive enumeration of these methods is not appropriate here.

7.4 COMPRESSED PSEUDO-LATTICE IMPLEMENTATION

Due to the similarity between compressed pseudo-lattices and EA-lattices, a single class was used for the implementation of both. As was mentioned, this required the modification of methods such as AddAtom, Closure, EIR, Link etc. to be able to cope with a data structure in which both lattice- and virtual arcs can occur. In many of the lattice class methods mentioned above, this required yet another parameter to indicate whether the relevant operation (for example a closure operation) should follow lattice-, virtual- or both types of arcs.

The following compressed pseudo-lattice related methods were added to the lattice class:

- Closure(aNode, aMaximumLevel, FollowVirtualArcs, aDirection) Returns aSet
- CompressLattice(aNode, aDirection) Return aSuccessIndicator
- ExpandLattice(aNode, aDirection) Return aSuccessIndicator
- InsertVirtualObject(anAttrSet, anObject, aDirection)

- Link(aNode1, aNode2, CreateVirtualArc, aDirection)
- DeLink(aNode1, aNode2, CreateVirtualArc, aDirection)

The extension of the AddAtom algorithm to operate on compressed pseudo-lattices greatly increased its complexity since a large number of exceptions that would not occur in EA-lattices had to be tested and catered for (e.g. the removal of virtual arcs when adding concepts at the top or bottom of the embedded lattice and maintenance of these arcs).

As was mentioned earlier, the ExpandLattice algorithm can be used as a non-incremental lattice construction algorithm that is an alternative to the AddAtom algorithm for constructing lattices. The basic approach would be to start with a fully compressed pseudo-lattice such as the bipartite graph in figure 6.1 and using successive ExpandLattice calls, construct the complete lattice. However, this strategy proved to be very inefficient and was ruled out as a useful lattice construction algorithm. Nevertheless, ExpandLattice is a useful operation in manipulating and forming compressed pseudo-lattices.

7.5 IMPLEMENTATION ISSUES

The most important hurdles encountered during the implementation and testing of the code stem from the exponential nature of a lattice as a data structure. This creates problems both in terms of the time and memory size needed to build and represent the lattice. A number of trade-offs thus presented themselves and had to be dealt with. Less efficient implementations used too many resources or took a long time to execute and test.

7.5.1 Time

To build a data structure that has an exponential number of elements, obviously and unavoidably takes an exponential amount time to complete. However, even in this context, inefficient coding can lead to even worse time inefficiencies than is necessary.

A number of different implementation options were evaluated by implementing each as a different class and comparing the classes using test data sets. These options relate to the use of different internal data structures such as hash tables combined with lists instead of unordered sets. Different optimisations of the AddAtom algorithm as well as the calculation of meets and generator concepts were also considered. A number of ways to prevent the unnecessary consideration of concepts were also investigated. The algorithm in appendix A documents the algorithm with the best time performance. The strategy of caching closures of nodes was also implemented and did also improve the performance of the algorithms.

7.5.2 Space / memory

The amount of memory available for data structures is an important limitation and will always remain potentially problematic, since the worst-case number of lattice nodes is exponential in the number of attributes. The problem is thus no less acute now than when the first implementation of the AddAtom algorithm was developed (i.e. in 1996) even though the average PC of today could probably handle data sets were problematic at the time. To ameliorate the inherent problem of exponential space requirements, inefficient memory use should therefore be avoided wherever possible.

Since the algorithm is generic, any number of objects or attributes can, in principle, be added at any time. Thus, the maximum size of arrays and sets needed for a particular lattice cannot be reliably calculated beforehand. The maximum size of many of the sets is determined by the number of intermediate nodes, which is in turn determined by the data. It is a catch-22 situation in that to determine the limits, the lattice must first be built.

Two alternative approaches were taken to address this problem. The first approach declared a fixed amount of memory for each node and set. This strategy is very memory-inefficient since the maximum size required is only needed in a limited number of instances. The second approach was to use variable length sets and lists. The first approach had the benefit of not requiring the reorganisation and additional testing required for variable length sets and lists used in the second approach and was therefore very time efficient. Using this approach with a data structure that is already exponential in size resulted in reaching the practical limit of memory very quickly on the smaller PC based platforms.

The second approach is beneficial in terms of memory usage, but the increase in computing overheads to manage the data structure slowed the performance down.

In the light of this time-space trade-off it was opted to retain both of the approaches as alternative lattice classes. The appropriate class were then chosen based on the availability of memory and computing time and power in the particular application. Typically the fastest approach possible was used when building lattices (usually on UNIX servers with large memories), whilst browsing lattices on smaller systems used a more memory efficient approach.

Since many of the larger lattices built for testing could not be used in PCs running under Windows a persistent data structure and cache was implemented. This has the capability of keeping only a number of nodes in a memory cache and swapping them from disk as needed. This slowed down the performance considerably, but had the benefit that very large lattices could be traversed in a machine with a relatively small amount of memory. This cache was further extended to cache the upward and downward closures of concepts as they were calculated. This (pre-computed) closure cache improved the time efficiency of the AddAtom algorithm.

7.5.3 Object-oriented implementation

A significant decrease in performance was noted when changing form C to object-oriented and more modular C++ code (estimated at more than 30%). This was due to the increased modularisation and looser coupling of different data structures and objects as well as to the significantly increased number of function calls and parameter range checking. This lead to an increase in the number of function calls to access variables and data structures via formalised interfaces. It did have the benefit of making the experimentation with different memory, caching and optimisation strategies much easier since only small parts of the code needed to be changed. This was however at the expense of efficiency (also refer to section 5.3).

7.5.4 UNIX and Windows

Testing and implementation was conducted on both Windows-based operating systems as well as various flavours of UNIX. In order to minimise the problems of porting code between operating systems, the basic program used a text-based interface. A small number of operating system and hardware specific issues were dealt with using conditional defines in the code. This enabled the code to be compiled under both

operating systems with no modification and allowed development under Windows whilst large test runs were performed under UNIX.

As output, the program created lattice files (with a *.lat extension). These files were created in an operating system agnostic fashion which enabled the transferral between operating systems.

For the purpose of constructing GUI-based interfaces, a Windows DLL library was created that exposed all the lattice object interfaces to any Windows-based application.

7.5.5 Testing

The AddAtom algorithm in its present form was not discovered immediately and came from an iterative process during which a number of refinements and optimisations have been made. During this process an intensive testing procedure was used to determine whether each resulting data structure was indeed a lattice, and to test the various compressed lattice properties.

Some of the tests used were:

- Validating the lattice property by brute force (i.e. testing that any two nodes have only one supremum and infimum).
- Concepts labelled as attributes have no parents and object concepts must have no children whilst intermediate concepts must have both (the unit and empty concepts are not stored as part of the data structure).
- Objects are connected to the correct attributes after the building of the lattice and have not gained or lost any attributes in their intent.
- All node support values are correct.
- No intermediate node has only one parent/child node (EA-lattice property).
- Lattices built on the same context, but with objects in a different order resulted in the equivalent lattices (i.e. the two resulting lattices were isomorphic).
- No redundant links to nodes exist (i.e. there is not an direct and indirect arc between two nodes).
- All intermediate concepts have at least two parents and two children (EA-lattice property).
- The attributes and objects of a concept is the same as all the attributes in its upward and downward closures.
- Any concept c was connected only to concepts in EIR(L, Intent(c), c) and EER(L, Intent(c), c).

Using this intensive testing the correctness of the AddAtom algorithm and its variants were proved empirically whilst problems were also identified early.

In addition, the pseudo code of the (inefficient) AddAtom algorithm as described in section 4.6 was re-implemented from scratch to verify that no mistakes had been made in its formulation.

7.5.6 User interface

Due to the requirement of operating under both Windows, MS DOS and UNIX operating systems, a text based user interface was developed that provided basic functionality.

Some of the user interface functions are:

- Building lattices from a given incidence relation provided as a text-based file.
- Saving and restoring lattices to binary *.lat files.
- Performing single lattice operations such as EIR, AIR, AddAtom, CompressLattice and ExpandLattice.
- Performing specific validity and consistency tests such as the comparison of different lattices.
- Obtaining performance metrics such as the time taken for an operation, the number of certain basic lattice operations such as closures, concept references and set operations performed to construct a lattice.
- Interactively interrogating the lattice structure for debugging and other purposes.
- Producing text based files suitable for human reading that describe the lattice concepts and cover relationships (this was used for debugging purposes).

In addition the user interface allowed text-based script files to be executed that automate the execution of a number of the above functions. This was used for testing, debugging and performance measurement purposes.

7.5.7 Continued advances in hardware

As indicated, the majority of the code was written during 1995-1996 at a time when the average server and personal computer hardware available was significantly slower and had significantly less memory than at present, especially on Windows/Intel based platforms. This factor was therefore a primary driver in the development of more advanced code such as persistency, caching, variable length data structures etc. to cope with the restrictions.

7.6 COMPARISON WITH OTHER LATTICE CONSTRUCTION ALGORITHMS

For comparative purposes, both Godin et al. (1995b) and Carpineto and Romano (1993) algorithms were implemented using the object oriented version 3 data structures, utility functions and virtual classes. However, to ensure unbiased comparison with the newly derived algorithm, certain changes in the data structures were necessary. This was because both of these algorithms made extensive use of a node's intent which was not, at the time, explicitly stored as an attribute of a node. The data structures and utility operations were changed to be fair to both the algorithms before any time complexity comparisons were made. The results of the comparisons mentioned in chapter 5 are therefore on an "apples-with-apples" basis.

As was indicated in chapter 5 the results of the pilot study were confirmed by the wider study which used completely different and separate implementations.