DATA CUBE BASED ON

FORMAL CONCEPT ANALYSIS TECHNIQUE

MR. TOUCHARIT SOMNIMITR

MR. YAHYA HAMAD

MISS. SARANRATCH KOED-IM

A SPECIAL PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF SCIENCE

IN COMPUTER SCIENCE, INTERNATIONAL PROGRAM

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

ACADEMIC YEAR 2009

Title	Data Cube Based on Formal Concept Analysis Technique
Students	Mr. Toucharit Somnimitr
	Mr. Yahya Hamad
	Miss. Saranratch Koed-im
Degree	Bachelor of Science
Program	Computer Science, International Program
Academic Year	2009
Advisor	Mr. Suntana Oudomying

ABSTRACT

We implement the new approach that computes Online Analytical Processing Cube or data cubes by applied the concept of Formal Concept Analysis and Graph to the process of constructing data cube. The result from this approach led us believe that this approach can be used in the real world data; therefore, we also implement a small pivot table application using this approach. The application shows the aggregate data, roll up and drill down the data can be easily accomplished.

ACKNOWLEDGEMENT

The authors would like to take this opportunity to express our gratitude to all those who gave us great advises and fully support us throughout the project.

The authors wish to express our deep gratitude to our project advisor, Mr. Suntana Oudomying, for these valuable suggestions, supports, encourage and be a guidance to us throughout every phase of implementing this project from the beginning of choosing the topic until the end of writing document.

The authors also would like to express our gratitude to our project committee, Dr. Veera Boonjing and Dr. Sarun Intakosum who gave permission and encourage us to go forward with our project with great advices and suggestions.

We are also would like to express our appreciation to all professors for such a valuable knowledge and guidance during our time of study at department of computer science, King Mongkut's Institute of Technology Ladkrabang.

Lastly, we would like to take this opportunity to be grateful to our family and friends, who always support and encourage with love and care throughout our lives.

Mr. Thoucharit Somnimitr

Mr. Yahya Hamad

Miss. Saranratch Koed-im

Table of Contents

Chapter 1

Introduction

1.1 Rational

Online Analytical Processing (OLAP) plays a significant role in the data analysis task nowadays. It is used to support the decisions making at the management level. One of the core operations of OLAP is data cube. Data cube allows data to be viewed at a multi-dimensional. Users can take this advantage to view data in a different perspective, for example; Users may want to see the sales report in the Item dimension; to see which items sell the most. Alternatively user can also see the report at the stores dimension; to see which store sells the most.

However, the result of data cube is extremely large, especially when the number of the CUBE BY attributes and the number of a base relation is large. Consequence from the large result, the other problem of data cube is that it stores data in the array (table) and if the cube table becomes huge (i.e. dense data). It would take a lot of time to query the data if data is at the last row of the cube table.

Formal Concept Analysis has been developed to support data analysis and knowledge processing therefore; we think it would be possible to applied FCA to data cube. With this motivation, we propose the approach that computes data cube by applied the concept of Formal Concept Analysis (FCA) and Graph to the process of constructing data cube. With FCA the data would be represent as object and property, in our data cube case the object would be dimension such as store, location and time and the property would be the sales or the measurement value. Also with this approach, the data cube would be stored in the graph manner rather than table or array and user does not have to have the knowledge of SQL language. Retrieving aggregated data would be performed by the users themselves via a pivot table application that we implemented for this project.

1.2 Objective

To implement the approach that compute data cube by using the concept of Formal Concept Analysis (FCA) and be able to use the approach pivot table. The pivot table application from the new approach must still contain the functional such as roll-up and drill down data for aggregations in particular subtotal and grand total. User can use this application to also search for data without the knowledge of SQL.

1.3 Scope of study

Our project will focus on using formal concept analysis to compute and store data cube in the 3 dimensional data.

- To implement the approach of computing data cube based on the concept of Formal Concept Analysis.
- To provide the approach of query the computed data cube that based on the concept of Formal Concept Analysis.
- To implement the real world application, pivot table.

1.4 Research Methodology

To conduct the approach, we have studied the data cube and related research about data cube, and we have also studied the formal concept analysis and related research about formal concept analysis, such as the algorithms and the application of formal concept analysis. All related literature will provide in details in chapter 2. Chapter 3 will show how the algorithm works in pseudo code accompanied by an example case. Chapter 4 will show the application of the approach, pivot table, and how it works. Chapter 5 will be the conclusion of our project and the suggestion from our future work.

Chapter 2

Related Literatures

In this chapter, we give an overview about the basic knowledge that we used for implementation of our project as followed – Online Analytical Processing, Data Cube, Pivot Table, and Formal Concept Analysis (FCA). The detail of the tool and technology used will be included in the appendix.

2.1 Online Analytical Processing (OLAP)

Decision support has become focused and essentials in many businesses. It helps making decision in the management level, from manager to chief executive officer (CEO) in a company. The decision that needed to be made are varied in each level, for example, manager may only want to make decision based on day-to-day work and CEO need to make decision that will impact the future or the company.

Online Analytical Processing (OLAP) is one of the most essential elements of decision support. The functional and performance of OLAP give the management level people to make a decision easier, and see the overall picture of the company. Online analytical process application is different from online transaction processing application supported by the operational (relational) databases.

Online transaction processing (OLTP) application typically automate clerical data processing tasks such as order entry and banking transaction that are the day-to-day operation of the company. These tasks are structured and repetitive, and consist of short, atomic, isolated transactions [6]. OLTP database needs to be consistency and require up-to-date data; therefore, the key of OLTP is consistency and recoverability. Online analytical Processing, in contrast, is targeted for decision support. It is included in the broader category of business intelligence. Applications of OLAP included business report for sales, marketing, management reporting, business process management (BPM), budgeting and forecasting, financial reporting and etc. The data in OLAP include historical, summarized and consolidated data. Consequently, the data that used in OLAP tends to be enormously larger than the data from the operational database. Queries needed to access millions of records and perform a lot of scan and joins and aggregates therefore, the key performance of OLAP is query throughput and response time.

2.1.1 Type of OLAP

OLAP can be classified in to 3 categories.

MOLAP (Multidimensional OLAP) this type of OLAP sometime called just OLAP. It is the classic type of OLAP so most of the time when we refer to just OLAP, it is MOLAP. MOLAP stores data in multidimensional array, rather than in a relational database. Therefore it requires the pre-computing and storage of information in the data cube. MOLAP can also use data from one or more than 2 relational databases to be queried and store in MOLAP multidimensional array.

ROLAP (Relational-OLAP) as the name said, ROLAP works directly with relational databases. The base data and the dimension tables are stores as relational tables and new tables are created to hold the aggregated information depends on the specialized schema design.

HOLAP (Hybrid-OLAP) HOLAP is the use of both MOLAP and ROLAP. For example, for some vendors, a HOLAP database will use relational tables to hold the larger quantities of detailed data, and use specialized storage for at least some aspects of the smaller quantities of more-aggregate or less-detailed data.

2.2 Data Cube – Multidimensional View

A very famous conceptual model for OLAP is the multidimensional view of the data in the warehouse, or multidimensional database which is defined as a form of database that is designed to make the best use of storing and utilizing data. Multidimensional database can receive data from a variety of relational databases and structure the information into categories and sections that can be accessed in a number of different ways.

In the multidimensional data model, there is a set of numeric measures that are the objects of analysis, for example, sales, budget, revenue, inventory, ROI (return on investment). For each numeric measures depends on a set of dimensions which provide the context for the measure, the example of the dimensions would be product, city, and time. The dimensions together are assumed to be uniquely determining the measure value (e.g. sales, budget and revenue).

The example below shows the multidimensional data or data cube which consists of 3 dimensions, Product, City and Date

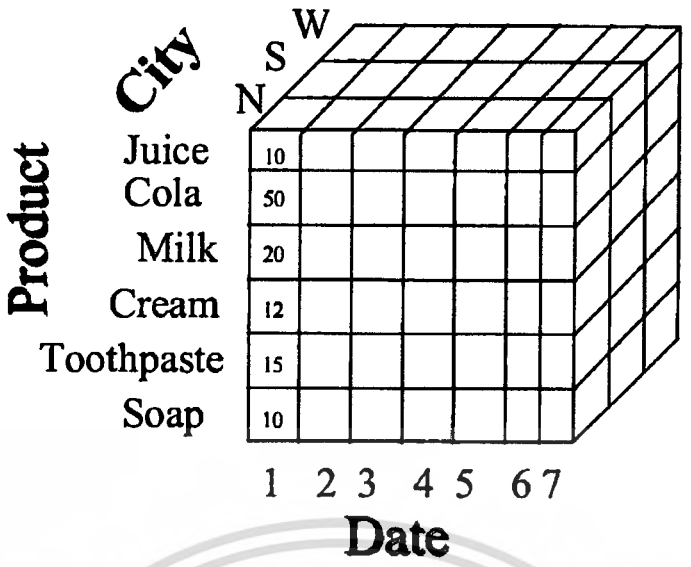


Figure 2.1 The example of data cube.

From the example, each dimension is described by a set of attributes. The "Product" dimensions from the example consist of Juice, Cola, Milk, Cream, Toothpaste and Soap. The "City" dimensions consist of North (N), South (S) and West (W) and the "Date" dimension consist of seven attribute, 1 – 7. For example, the analyst can view data in the product dimension, see which product sell the most? And the product that sold the most is in which city on which date?

2.2.1 Aggregation

The most important mechanism in OLAP is aggregations. Aggregations are built by changing the granularity on specific dimensions and aggregating up data along these dimensions. The number of possible aggregations is determined by every possible combination of dimension granularities. The combination of all possible aggregations and the base data contains the answers to every query which can be answered from the data.

2.2.2 Data Cube Operation

Common operations of data cube include slice and dice, drill down, roll up and pivot.

Slice and dice: Slice and dice is the ability to access a data warehouse through any of its dimensions equally. Slicing and dicing is the process of separating and combining data in seemingly endless combination.

Roll-up or Summarization: Roll-up involves computing all of data relationship for one or more dimension to present higher level of summarization.

Drill down: Drill down is reverse of roll-up. A drill down goes from less detailed data to more detailed data

Pivot: The analyst may want to view or "pivot" data in a different ways. The cube could effectively be re-oriented so that the data displayed now has periods across the page and type of cost down the page. Because this re-orientation involves re-summarizing an enormous data, the new view of the data has to be generated efficiently to avoid wasting the time. (pivot will be mentioned more on a separate topic in this chapter)

2.2.3 Data Cube Calculation – CUBE BY operator

In order to effectively support data cube queries, CUBE BY operator was proposed [2]. It is a multidimensional extension of the relational operator GROUP BY by compute GROUP BY corresponding to all possible combinations of attributes in the CUBE BY clause.

Now it is obvious that the CUBE BY clause is expensive and the result from the CUBE BY operator is extremely large especially when the number of the CUBE BY attributes and the number of tuples in the base relation is large. For example, the result from the CUBE BY with 3 attribute (e.g. A, B, C) is larger than the result from the CUBE BY attribute (A, B) with the same base relation. If the base relation is larger the result is very much larger too.

Given a base relation R with n attributes, the number of tuples in a k-attribute cuboid (GROUP BY), $0 \leq k \leq n$, is the number of tuples in R that have distinct attribute values on the k-attribute. The size of "a" cuboid is possibly close to the size of R. However, the complete cube of R consist of 2^n cuboid, therefore, the size of the union 2^n cuboids is much larger than the size of R.

2.2.4 Example of CUBE BY operation

In this example, we will use relation R(A,B,C,M) with 5 tuples. R has three dimensions attribute: A, B and C, and one measure attribute M. The aggregate function used is SUM. (We will use * notation to denote a special value ALL.)

Given the base table below

TID	A	B	C	M
1	0	1	1	50
2	1	1	1	100
3	2	3	1	60
4	4	5	1	70
5	6	5	2	80

Table 2.1 Base table

The cuboid lattice for 3 dimensional attribute relations is show below. The nodes in the lattice are cuboids. An arrow between two cuboids means that one cuboid can be computed from the other. For example, cuboids on AB, AC, and BC can be computed from cuboid ABC.

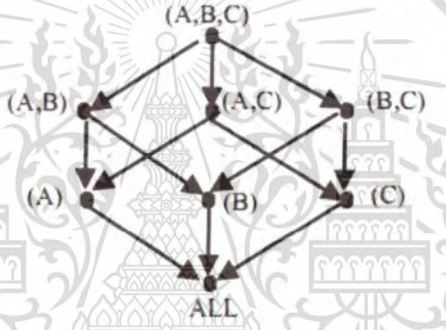


Figure 2.2 The cuboid

For instance, tuple TID = 1 consist of (A0, B1, C1, 50) to compute the base cuboid first it will generate a cuboid with 8 nodes.

A0, B1, C1 (Corresponding to A,B,C) 50

A0,B1,* (Corresponding to A,B) 50

A0,*,C1 (Corresponding to A,C) 50

*,B1,C1 (Corresponding to B,C) 50

A0,*,*(Corresponding to A) 50

, B1,(Corresponding to B) 50

,,C1(Corresponding to C) 50 and

,,* (Corresponding to ALL) 50

Each tuple will produce one cuboid. In order to generate the completed cubed, each cuboid has to union one another. The complete cube is shown below with 30 tuples.

TID	Cuboid	A	B	C	M
1	ALL	*	*	*	360
2	ABC	0	1	1	50
3	A	0	*	*	50
4	AB	0	1	*	50
5	AC	0	*	1	50
6	ABC	1	1	1	100
7	A	1	*	*	100
8	AB	1	1	*	100
9	AC	1	*	1	100
10	ABC	2	3	1	60
11	A	2	*	*	60
12	AB	2	3	*	60
13	AC	2	*	1	60
14	B	*	3	*	60
15	BC	*	3	1	60
16	ABC	4	5	1	70
17	A	4	*	*	70
18	AB	4	5	*	70
19	AC	4	*	1	70
20	ABC	6	5	2	80
21	A	6	*	*	80
22	AB	6	5	*	80
23	AC	6	*	2	80
24	C	*	*	2	80
25	BC	*	5	2	80
26	B	*	1	*	150
27	B	*	5	*	150
28	C	*	*	1	280
29	BC	*	1	1	150
30	BC	*	5	1	70

Table 2.2 Cube by table

As the example shows, the result of cube by is extremely large. For a base relation with 5 tuples, CUBE BY will generate the table with 30 tuples (That's 5 time!).

2.2.5 Other Techniques of Data Cube

2.2.5.1 Condensed Cube

A condensed cube [1] is a fully computed cube that condenses those tuples, aggregated from the same set of base relation tuples, into one physical tuple.

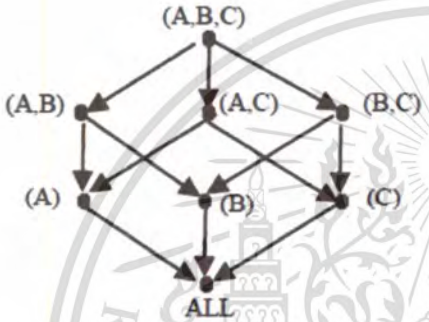
Recall from the Figure 2.2, Figure 2.3 shows the data cube before condensed. After the CUBE BY operation the base table (a) become into cube table (b). Here you can see that number of tuples is increased from 5 tuples to 30 tuples (6 times increased).And in the lattice (c), we will have 8 cuboids which came from 2^n, when n is number of dimension (we have 3 dimensions A, B, C).

Before we will go to the condensing, we will talk about Base Single Tuple first. First let see in Table (a), in TID 1 (first tuple) is base single tuple on dimension A, this because it has 0 value in dimension A, which is the only 0 in any tuples. This tuple is not base single tuple value on B and C, because in tuple TID 2 also has the same value in B and C dimension. TID 2 is also base single tuple on dimension A. Now let see on TID 3, this tuple

is base single tuple on dimension A and B, because it has value 4 in dimension A and 5 in dimension B which they are no duplicate value at all.

TID	A	B	C	M
1	0	1	1	50
2	1	1	1	100
3	2	3	1	60
4	4	5	1	70
5	6	5	2	80

(a)



(c)

TID	Cuboid	A	B	C	M
1	ALL	*	*	*	360
2	ABC	0	1	1	50
3	A	0	*	*	50
4	AB	0	1	*	50
5	AC	0	*	1	50
6	ABC	1	1	1	100
7	A	1	*	*	100
8	AB	1	1	*	100
9	AC	1	*	1	100
10	ABC	2	3	1	60
11	A	2	*	*	60
12	AB	2	3	*	60
13	AC	2	*	1	60
14	B	*	3	*	60
15	BC	*	3	1	60
16	ABC	4	5	1	70
17	A	4	*	*	70
18	AB	4	5	*	70
19	AC	4	*	1	70
20	ABC	6	5	2	80
21	A	6	*	*	80
22	AB	6	5	*	80
23	AC	6	*	2	80
24	C	*	*	2	80
25	BC	*	5	2	80
26	B	*	1	*	150
27	B	*	5	*	150
28	C	*	*	1	280
29	BC	*	1	1	150
30	BC	*	5	1	70

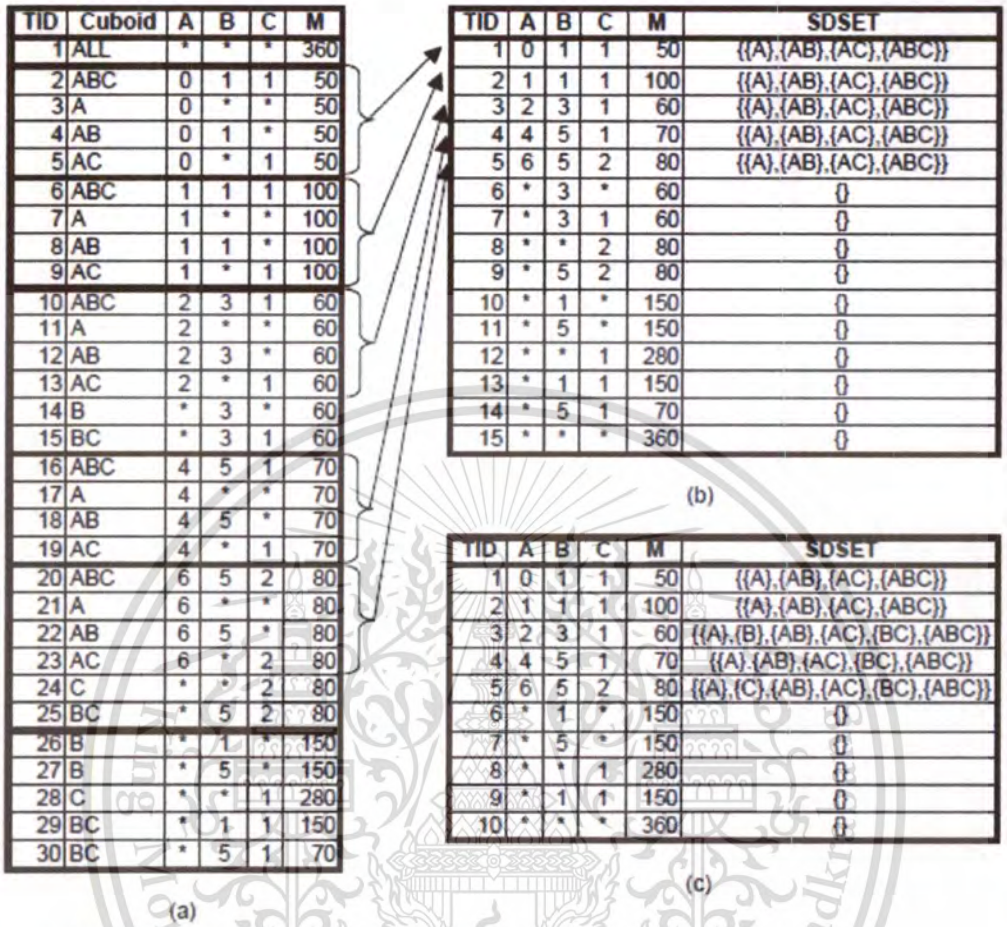
(b)

Figure 2.3 (a) An example relation R, (b) Its complete cube Q(R), and (c) Cuboid lattice

This Figure2.4 shows how data is condensed. Table (a) is the complete cube table (uncondensed), table (b) is a BST-Condensed cube. This BST-Condensed cube is the aggregation from tuples in the complete cube table (a), which choose from only one base single tuple. In this example they choose only from tuple which is base single tuple on dimension A. Then aggregate them together into one tuple and add one schema named SDSET which is the set of cuboids that are aggregated together. The rest of the tuples that are not base single tuple on dimension A will be put in with SDSET of {} (empty). The last table, minimal BST-Condensed cube (c) is similar to the BST-Condensed cube (b), the only difference is that BST-Condensed cube (b) will aggregate based on the tuple which has base single tuple on only one dimension, while minimal BST-Condensed cube (c) will aggregate based on all based single tuple.

If we want to find "ALL" which mean A=*, B=*, C=* we would have to retrieve TID 10 which has the value of 360 without having to search at the SDSET. On the other hand, if we want to find A=4, B=*, C=* we have to search to TID number 4 which A=4 and

continue to search at SDSET, we would find {A} which imply A=4, B=*, C=* and retrieve the value which is 70.



TID	Cuboid	A	B	C	M
1	ALL	*	*	*	360
2	ABC	0	1	1	50
3	A	0	*	*	50
4	AB	0	1	*	50
5	AC	0	*	1	50
6	ABC	1	1	1	100
7	A	1	*	*	100
8	AB	1	1	*	100
9	AC	1	*	1	100
10	ABC	2	3	1	60
11	A	2	*	*	60
12	AB	2	3	*	60
13	AC	2	*	1	60
14	B	*	3	*	60
15	BC	*	3	1	60
16	ABC	4	5	1	70
17	A	4	*	*	70
18	AB	4	5	*	70
19	AC	4	*	1	70
20	ABC	6	5	2	80
21	A	6	*	*	80
22	AB	6	5	*	80
23	AC	6	*	2	80
24	C	*	*	2	80
25	BC	*	5	2	80
26	B	*	1	*	150
27	B	*	5	*	150
28	C	*	*	1	280
29	BC	*	1	1	150
30	BC	*	5	1	70

(a)

TID	A	B	C	M	SDSET
1	0	1	1	50	{{A},{AB},{AC},{ABC}}
2	1	1	1	100	{{A},{AB},{AC},{ABC}}
3	2	3	1	60	{{A},{AB},{AC},{ABC}}
4	4	5	1	70	{{A},{AB},{AC},{ABC}}
5	6	5	2	80	{{A},{AB},{AC},{ABC}}
6	*	3	*	60	{}
7	*	3	1	60	{}
8	*	*	2	80	{}
9	*	5	2	80	{}
10	*	1	*	150	{}
11	*	5	*	150	{}
12	*	*	1	280	{}
13	*	1	1	150	{}
14	*	5	1	70	{}
15	*	*	*	360	{}

(b)

TID	A	B	C	M	SDSET
1	0	1	1	50	{{A},{AB},{AC},{ABC}}
2	1	1	1	100	{{A},{AB},{AC},{ABC}}
3	2	3	1	60	{{A},{B},{AB},{AC},{BC},{ABC}}
4	4	5	1	70	{{A},{AB},{AC},{BC},{ABC}}
5	6	5	2	80	{{A},{C},{AB},{AC},{BC},{ABC}}
6	*	1	*	150	{}
7	*	5	*	150	{}
8	*	*	1	280	{}
9	*	1	1	150	{}
10	*	*	*	360	{}

(c)

Figure 2.4 (a) The complete cube Q, (b) A BST-Condensed cube CQ of R and (c) A minimal BST-Condensed cube of R

2.2.5.2 Dwarf Cube

Dwarf is a highly compressed structure for computing, storing, and querying data cubes. Dwarf identifies prefix and suffix structural redundancies and factors them out by coalescing their store. Prefix redundancy is high on dense areas of cubes but suffix redundancy is significantly higher for sparse areas. Putting the two together fuses the exponential sizes of high dimensional full cubes into a dramatically condensed data structure. The elimination of redundant suffixes is avoided. This effect is multiplied in the presence of correlation amongst attributes in the cube. A petabyte 25-dimensional cube was shrunk this way to a 2.3GB Dwarf Cube, in less than 20 minutes, a 1:400000 storage reduction ratio. Still, Dwarf provides 100% precision on cube queries and is a self-sufficient structure which requires no access to the fact table [4].

The example of dwarf cube

Given the base relation below

Store	Customer	Product	Price
S1	C2	P2	$70
S1	C3	P1	$40
S2	C1	P1	$90
S2	C1	P2	$50

Table 2.3 Base relation table

The complete dwarf cube is shown in figure 2.5

If we want to find S1, C2, P2 we would navigate to node number 3 at the first column which has the value of 70. If we want to find S*, C=1, P=2 we would navigate to node (7) at the second column which has the value of 50.

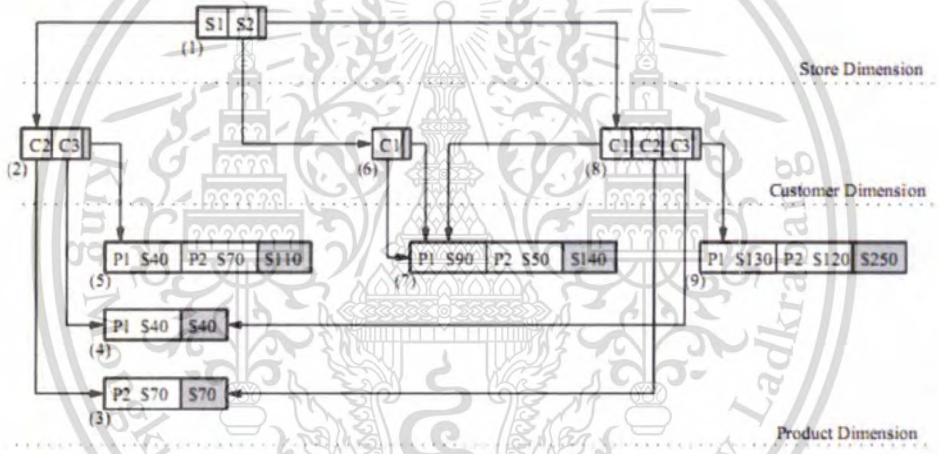


Figure 2.5 Dwarf cube

2.2.5.3 Star-Cubing

Star-Cubing [5] uses a hyper-tree structure, called Star-Tree, which is an extension of H-tree[13], to facilitate cube computation. Each level in the tree represents a dimension in the base cuboid. The algorithm takes the advantages of the above two models. On the global computation order, it uses the simultaneous aggregation similar to MultiWay. However, it has a sub-layer underneath based on the bottom-up model by exploring the notion of shared dimension. This integration allows the algorithm to aggregate on multiple dimensions, while it can still partition parent group-by's and use Apriori-based pruning on child group-by's.

Star-Cubing also introduced a special value "Star" for attributes whose count is less than min_sup (this is different from "*" in this project, which is a wildcard) which has

been proved useful for skewed data. It performs well on dense, skewed and not-so-sparse data. However, in every sparse data sets, it has to handle the additional traversal cost introduced by the tree structure.

The example of star cube

Given the base relation below

A	B	C	D	Count
a1	b1	c1	d1	1
a1	b1	c3	d3	1
a1	b2	c2	d2	1
a2	b3	c3	d4	1
a2	b4	c3	d4	1

A	B	C	D	Count
a1	b1	*	*	2
a1	*	*	*	1
a2	*	c3	d4	2

Table 2.4 , 2.5 : Base Relation Table, Compressed Base Table after star reduction with min_sub =2

Figure 2.6 shows the complete Star-cubing. If we want to find a1, b2, c*, d* first we will navigate from root node, then we check root's children to find a1. After we found a1 we check a1's children in order to find b2. In this case we found b* node with count=1. As we can see in Star Table that b2 is star node so the result of a1, b2, c*, d* is 1.

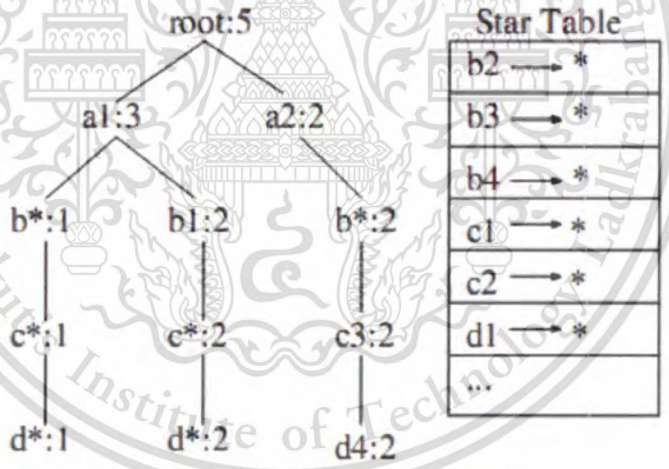


Figure 2.6 Star Tree and Star Table

2.3 Pivot Table

Pivot table plays a crucial role in terms of visualization data, especially the multidimensional data like data cube. A pivot table is a program tool that allows analyst to reorganize and summarize selected columns and rows of data in a spreadsheet or database table to obtain a desired report. To pivot is to view data from different perspectives. Among other functions, pivot table can automatically sort, count, and total the date store in one table or spreadsheet.

In this section, we will give the example of pivot table in the MS EXCEL spreadsheet.

Given the table in the spreadsheet

A	B	C	M
A0	B1	C1	50
A1	B1	C1	100
A2	B3	C1	60
A4	B5	C1	70
A6	B5	C2	80

Table 2.6 Base relation Table

If we would like to view A, B in the row and C in Column and M as a value

Sum of M	Column Labels		
Row Labels	C1	C2	Grand Total
A0	50		50
B1	50		50
A1	100		100
B1	100		100
A2	60		60
B3	60		60
A4	70		70
B5	70		70
A6		80	80
B5		80	80
Grand Total	280	80	360

Table 2.7 Pivot Table

This is the simple pivot table in MS EXCEL spreadsheet. If we would like to know the grand total of A0 which implies A=0, B=*, C=*, the result will be 50.

2.4 Formal Concept Analysis (FCA)

In this section, we will only mention Formal Concept Analysis (FCA) in fundamental way. The more detailed of formal concepts and concept lattices should read the papers of WILLE (82) and WOLFF (88).

Formal Concept Analysis (FCA) is a data analysis technique based on ordered lattice theory. Formal Concept Analysis has been developed to support data analysis and knowledge

processing. It provides graph-based visualizations of tabular data and has successfully been applied to a number of fields including Text Data Mining, Psychology, Social Science, Sociology, Anthropology, Medicine, Biology, Linguistics, Computer Science, Mathematics, Industrial Engineering and Software Engineering.

2.4.1 Formal Contexts

FCA is the process of describing the world in terms of a number of objects and a number of attributes which may be possessed by those objects. A formal context is represented by a cross-table with objects and attributes and a cross indicating that a certain object has a certain attribute for example,

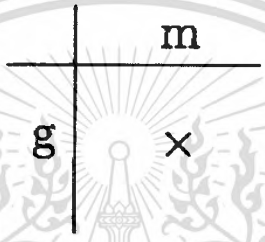


Figure 2.7 Cross Table

Definition 1: Let G, M be sets, and $I \subseteq G \times M$. The triple (G, M, I) is called a formal context.

G is the set of elements called objects,

M is the set of elements called attributes,

I is called the incidence relation ($\subseteq G \times M$).

The $(g, m) \in I$ read as "object g has attribute m".

For example, the table below describes for some animals which of the mentioned attributes they have. This is indicated by crosses. An empty cell means that the corresponding animal does not have the corresponding attribute.

	Predator	Can fly	Bird	Mammal
Lion	×			×
Canary		×	×	
Eagle	×	×	×	
Rabbit				×
Emu			×	

Table 2.8 Cross Table

2.4.2 Concepts and Concept Lattice

Definition 2: Let $K := (G, M, I)$ be a formal context. A pair (A, B) with $A \subseteq G$, $B \subseteq M$ is called a formal concept of the formal context if A is the set of all objects, having all attributes in B and B is the set of all attributes that all objects in A have. A is called the extent and B is called the intent of the concept (A, B). The set of all concepts of K is called the concept lattice of K and is written B (K).

$$(\beta(K), \leq) \text{ is a complete lattice}$$

To explain the notion of a formal concept of a context we look at the attribute of Canary and ask for all those animals (in this context) which have all the attributes of the Canary. Therefore, we get the set A consisting of Canary and Eagle. This set A of objects is closely connected to the set B consisting of the attributes flying and bird: A is the set of all objects having all the attribute of B, and B is the set of all attributes which are valid for the object A. Each pair (A, B) is called a formal concept for a given context. The set A is called the extent, the set B the intent of the concept (A, B). The extent of the concept determines the intent and the intent determines the extent.

Definition 3: For $A \subseteq G$, we define $A' = A^{\uparrow} := \{m \in M \mid \forall g \in A : gIm\}$

And for $B \subseteq M$, we define $B = B^{\downarrow} := \{g \in G \mid \forall m \in B : gIm\}$

$$A \rightarrow (A'', A') = (A^{\uparrow\downarrow}, A^{\uparrow}) \in \beta(K).$$

$$B \rightarrow (B', B'') = (B^{\downarrow}, B^{\downarrow\uparrow}) \in \beta(K).$$

For example, given A = Lion A' would be Predator and Mammal and A'' would be Predator. On the other hand, given B = Can Fly, B' would be Canary and Eagle and B'' would be can fly and bird.

Definition 4: Let $g \in G$. We write g' instead of {g}' for the object intent of g and we write m' instead of {m}' for the attribute extent of g. We write γg for the object concept (g'', g') and μm for the attribute concept (m', m'')

Construction of all concepts from the subsets A of G:

Create A' ($\subseteq$ M)

Create A'' ($\subseteq$ G)

The figure below shows the mappings from A to A' to A''.

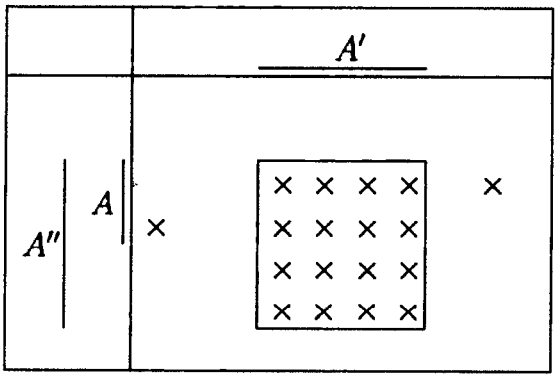


Figure 2.8 Cross Table Mapping from A to A' to A''

Example of using the context in Figure above

Let A = {Canary}

A' = {CanFly, Bird}

A'' = {Canary, Eagle}

This can subsequently lead to the proposition inference

{Canary} $\longrightarrow$ ({Canary, Eagle}, {CanFly, Bird}) $\in \beta$ (K)

For the entire example we can create all the inferences drawn from the formal context K:

{ } $\longrightarrow$ ({}, {Predator, CanFly, Bird, Mammal})

{Lion} $\longrightarrow$ ({Lion}, {Predator, Mammal})

{Canary} $\longrightarrow$ ({Canary, Eagle}, {CanFly, Bird})

{Eagle} $\longrightarrow$ ({Eagle}, {Predator, CanFly, Bird})

{Rabbit} $\longrightarrow$ ({Lion, Rabbit}, {Mammal})

{Emu} $\longrightarrow$ ({Canary, Eagle, Emu}, {Bird})

{Lion, Canary} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Eagle} $\longrightarrow$ ({Lion, Eagle}, {Predator})

{Lion, Rabbit} $\longrightarrow$ ({Lion, Rabbit}, {Mammal})

{Lion, Emu} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Canary, Eagle} $\longrightarrow$ ({Canary, Eagle}, {CanFly, Bird})

{Canary, Rabbit} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Canary, Emu} $\longrightarrow$ ({Canary, Eagle, Emu}, {Bird})

{Eagle, Rabbit} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Eagle, Emu} $\longrightarrow$ ({Canary, Eagle, Emu}, {Bird})

{Rabbit, Emu} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Eagle} $\longrightarrow$ ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Rabbit } → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Eagle, Rabbit} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Eagle, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Canary, Eagle, Rabbit} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Canary, Eagle, Emu} → ({Canary, Eagle, Emu}, {Bird})

{Canary, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Eagle, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Eagle, Rabbit} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Eagle, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Eagle, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Canary, Eagle, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

{Lion, Canary, Eagle, Rabbit, Emu} → ({Lion, Canary, Eagle, Rabbit, Emu}, {})

The formal concepts in K therefore are:

({}, {Predator, CanFly, Bird, Mammal})

({Lion}, {Predator, Mammal})

({Canary, Eagle}, {CanFly, Bird})

({Eagle}, {Predator, CanFly, Bird})

({Lion, Rabbit}, {Mammal})

({Canary, Eagle, Emu}, {Bird})

({Lion, Eagle}, {Predator})

({Lion, Canary, Eagle, Rabbit, Emu}, {})

2.4.3 Line-diagram of the concept lattice

Definition 5: Let K:= (G, M, I) be a context and let β (K) be the concept lattice of K. For $(A_1, B_1), (A_2, B_2) \in \beta$ (K), let $(A_1, B_1) \leq (A_2, B_2) :\Leftrightarrow A_1 \subseteq A_2$ $A_1 \subseteq A_2 \Leftrightarrow B_1 \supseteq B_2$

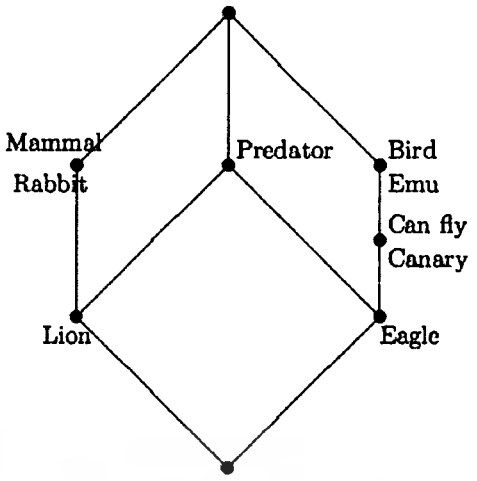


Figure 2.9 The complete line diagram

The figure 2.8 shows the line diagram from the formal context that has been described earlier. From the figure, we can see that Lion has the Mammal and Predator as its attributes as shown earlier in the Table 2.6 and the animal which has the attribute Predator are Lion and Eagle.

2.4.4 Read Diagram

The line diagram consists of circles, lines and the names of all objects and all attributes of the given context. The circles represent the concepts and the information of the context can be read from the line diagram by following the simple reading rule:
"An object g has an attribute m if and only if there is an upwards leading path from the circle named by 'g' to the circle named by 'm' ".

For example, Canary has the attribute Can Fly and Bird. As a consequence of the reading rule we can easily read from the line diagram the extent and the intent of each concept by collecting all objects below respectively all attributes above the circle of the given concept. Hence the object concept "Canary" has the extent Canary and Eagle and the intent is Can Fly and Bird.

The extent of the top concept is always the set of all objects while the intent of it does not contain any attribute (in this case). But in other contexts it may occur that the intent of the top concept is not empty.

Chapter 3

Implementing Data Cube Based on Formal Concept Analysis Technique

In this chapter we will show the algorithm for constructing data cube graph based on the definition of formal concept analysis and also show the algorithm step by step with the real data set. Finally, we will show the algorithm to retrieve data from data cube graph in the end of the chapter.

3.1 Constructing Data Cube Graph

From the basis of formal concept analysis, that is if two or more attribute have the same object. That same object will create into new object nodes. This is the basic idea to construct the data cube graph. The algorithm goes as follow.

- In data cube graph, the object is the dimension (e.g. Shirt (in Item dimension), Bangkok (Location dimension), or 2007(Year dimension)) and the attribute is the measurement value (e.g. Sales, Return on Investment, Measurement)
- A row in data set except the measurement value means the set of object in data cube graph (e.g. if a row of data set is (A0, B1, C1, 50) the object set is (A0, B1, C1))
- If two or more object sets share the same object, the "same" object will become the new nodes. (This is for roll up and drill down the data). For example, (A0, B1, C1) and (A1, B1, C1) share the same (B1, C1) so (B1, C1) will become the new node.
- The data cube graph does not contain the same node. For example, the node (A0, B1, C1) will appear only once in the graph.

3.2 Term used in the algorithm

Superset: A is super set of B when A contains all elements of B

Subset: A is subset set of B when B contains all elements of A

3.3 Graph Constructing Algorithm

In this section, we give all the algorithms that is used to construct the graph.

3.3.1 Process Main(fileLocation)

```
BEGIN
```

```
Input: textfile Output: graph
Create "ROOT" node.
While(row(textfile)is not null) do
        If(row == firstrow)
                Then get(column name)
        EndIf
        Else
                Get(object)
                Get(measurement)
                Cube(root, object, measurement)
        EndIf
EndWhile
Create "END" node
Create edge (LEAF, END)
Pre-compute(graph) ; Output: all possible combination of object (hiddenObj)
END
```

Figure 3.1 Shows the algorithm of Process Main method

This algorithm requires text data file's location as an input. After that it will create ROOT node and then read each row in the text data file. The first row which contain column name will be stored as column name. Other row will be read row by row, get(object) will get object from data and get(measurement) will get measurement data. Then root node, object and measurement will be sent into Cube algorithm. After finished reading data from the text file, it will create END node and create connection edge from every leaf node to END node. At last graph which Cube algorithm created will be sent to Pre-compute algorithm. This algorithm will find out all possible combination of object, which we called it "hiddenObj".

3.3.2 Process Cube(parent, object, measure)

```
BEGIN
        Input: root node, object, measurement
        CreateNode(object, measurement) return node
        CreateEdge(parent, object) return shareQueue
        While (shareQueue is not empty) do
                List[result] = GetSharedObject(shareQueue1, shareQueue2)
```

```
                While (result is not empty)
                        For all result r
                                AddNodeProcess(r, shareObj1, shareObj2)
                        EndFor
                EndWhile
        EndWhile
END
```

Figure 3.2 Shows the algorithm of Cube method

This algorithm requires parent node, object and measurement. The object and measurement will be sent to CreateNode algorithm to create new object node. Then it will sent parent node and object to CreateEdge algorithm in order to add connection from parent node to object node. CreateEdge will return shareQueue which if not empty, this algorithm will get each data in sharedQueue and find the shared object by sending shareQueue data (shareQueue1 and shareQueue2) into GetSharedObject algorithm then store the shared object in a list named result. If the list in not empty, each data in the list and shareQueue data which shared the same object will be sent to AddNodeProcess algorithm.

3.3.3 Process CreateNode (node, measure)

```
BEGIN
        Input: node, measurement
        If(node is existed)
                Then update measurement (measure)
        EndIf
        Else
                Then add node (node)
                Add measure(measurement)
        EndIf
END
```

Figure 3.3 Shows the algorithm of CreateNode methods

This algorithm requires node and measurement. First it will check whether the node is already existed or not. If the node is existed the measurement of that node will be updated. But if it does not exist yet, it will create new node and add measurement to it.

3.3.4 Process CreateEdge(parent, child)

```
BEGIN
        Input: parent node, child node
        //Check if edge is existed
        If(edge(parent, child) is existed)
                Return; (do not thing because already have edge)
        EndIf
        If (parent = child)
                Return error; (parent cannot be child)
        EndIf
        Else
                Add edge from parent to child
        EndIf
        Check parent's children
        If(parent's children >1)
                Add parent's children to shareQueue1 and add child to shareQueue2
        EndIf
END
```

Figure 3.4 Shows the algorithm of CreateEdge method

This algorithm requires two parameters parent node and child node. First it will check wheater edge which connect parent node with child node is already existed or not. If the connection is already existed, it will not do anything. If not, it will continue and check if parent node and child node is the same node or not. Parent and child node cannot be the same node, so if parent and child is the same node, it will not do anything and return error. If parent node and child node is not the same node it will add connection from parent node to child node. After that it will check parent node's children. If parent node have more than one children, the parent node's children and child node will be add into shareQueue1 and shareQueue2 in order.

3.3.5 Process GetShareObject(queue, child)

```
BEGIN
        Input: queue, child node
        For all queue q
                FindShareObject (q, child); Add result into List result
```

```
        EndFor
        If(result not empty)
                For all result r
                        find superset of the result
                        remove subset
                EndFor
        EndIf
        Return result
END
```

Figure 3.5 Shows the algorithm of GetShareObject method

Queue (shareQueue1) and child node (shareQueue2) are required in this algorithm. It will find share object in queue and child node and add shared object (result) into a list called "result". After that if result is not empty, all data in result list will be checked for subset and superset. If there is any subset superset, the subset will be removed from list. Finally the result list is returned.

3.3.6 Process AddNode (addNodeObj, shareObj1, shareObj2)

```
BEGIN
        Input: addNodeObj, shareObj1, shareObj2
        If (addNodeObj is already existed)
                CreateEdge(shareObj1, addNodeObj)
        EndIf
        ElseIf(addNodeObj is subset of existing node)
                CreateNode (addNodeObj,0)
                CreateEdge(shareObj1, addNodeObj)
                CreateEdge(existing node, addNodeObj)
        EndIf
        ElseIf (addNodeObj is superset of existing node)
                CreateNode(addNodeObj,0)
                RemoveEdge(shareObj2, existing node)
                CreateEdge(shareObj2, addNodeObj)
                CreateEdge(shareObj1, addNodeObj)
                CreateEdge(addNodeObj, existing node)
```

```
        EndIf
        Else
                CreateNode(addNodeObj,0)
                CreateEdge(shareObj1, addNod.eObj)
                CreateEdge(shareObj2, addNodeObj)
        EndIf
END
```

Figure 3.6 Shows the algorithm of AddNode method

This algorithm requires addNodeObj (r in result list), shareObj1 and shareObj2. First this algorithm will check if addNodeObj is already exist, send shareObj1 and addObj2 to CreateEdge algorithm. Then check if addNodeObj is subset of existing node (every node in graph which is not parent level), send addNodeObj into CreateNode algorithm (create new node addNodeObj). After that, AddNode send shareObj1 and addNodeObj into CreateEdge algorithm (create connection from shareObj1 to addNodeObj). Finally, it sends existing node which is the superset of addNodeObj and addNodeObj into CreateEdge algorithm (create connection from existing node to addNodeObj). In case that the addNodeObj is the superset of existing node (every node in graph which is not parent level), send addNodeObj into CreateNode algorithm (create new node addNodeObj). After that, it removes edge between shareObj2(parent which is superset of addNodeObj and existing node) and existing node. Then send shareObj2, addNodeObj to CreateEdge algorithm, send shareObj2, addNodeObj to CreateEdge algorithm and send addNodeObj, existing node to CreateEdge algorithm. Finally if addNodeObj does not match any condition, it just adds node addNodeObj by send addNodeObj into CreateNode algorithm, add connection from shareObj1 to addNodeObj and from shareObj2 to addNodeObj by sending shareObj1, addNodeObj and shareObj2, addNodeObj into CreateEdge algorithm.

3.4 Example of the algorithm

Given the base table, input as the text file

TID	A	B	C	M
1	0	1	1	50
2	1	1	1	100
3	2	3	1	60
4	4	5	1	70
5	6	5	2	80

Table 3.1 Shows the base table example

The text file must be in the format

Column1, Column2, Column3, Column4 (Column Names)

Row1

Row2

.

.

.

Row n

From the example table, the text file will be

A,B,C,M (Column Names)

A0,B1,C1,50

A1,B1,C1,100

.

.

.

A6,B5,C2,80

1. Create "ROOT" node called ROOT without any edges

Figure 3.7 Shows the result after the first step.

This step is the same as in the Main method that create ROOT node.

2. Read 1st Row and get object which is A0, B1, C1 and measurement which is 50 then, send the object and measurement along with ROOT node to Cube method which will call CreateNode to create a node called A0, B1, C1 (Keep the vertex in hash map with key = vertex, value = measurement)

Figure 3.8 Shows the result after the second step.

3. Then Cube method will call CreateEdge and send the parameter of parent (which means the parent of the node) and object (which means the object of the parent). This method will add edges from ROOT to node A0, B1, C1

   The CreateEdge method will return shareQueue which is the queue if the parent's node (which is ROOT) has more than one children. However, in this case node ROOT only has one child; therefore, continue to step 4

Figure 3.9 Shows the result after the third step

4. Read another row which is A1, B1, C1. Repeat 2nd and 3rd step. As well as the 2nd and 3rd step, first it will create node A1, B1, C1 and then create edge from node ROOT to node A1, B1, C1. Figure 3.10 shown the result after create node and edge.

Figure 3.10 Shows the result after adding node A1, B1, C1 and edge from ROOT to A1, B1, C1

However the parent of node "A1, B1, C1" which is node ROOT has more than one children, the shareQueue from CreateEdge method will return the children of the parent's node (in this case, the parent node is ROOT and the children is node A0, B1, C1 and node A1, B1, C1). Then call method getShareObject (for all elements under ROOT with the current row). Node A0, B1, C1 and node A1, B1, C1 have same shareObject B1, C1. The result from method getShareObject is B1, C1. B1, C1 will be sent into method addNode which will check if B1, C1 is already existed under both node (A1, B1, C1 and A1, B1, C1) or not, if not check if B1, C1 is superset of node A1, B1, C1 and node A0, B1, C1's children or not, if not check if add vertex B1, C1 is subset of node A1, B1, C1 and node A0, B1, C1's children or not, if not add edges from node A0, B1, C1 and node A1, B1, C1 to node B1, C1 as show in figure 3.11

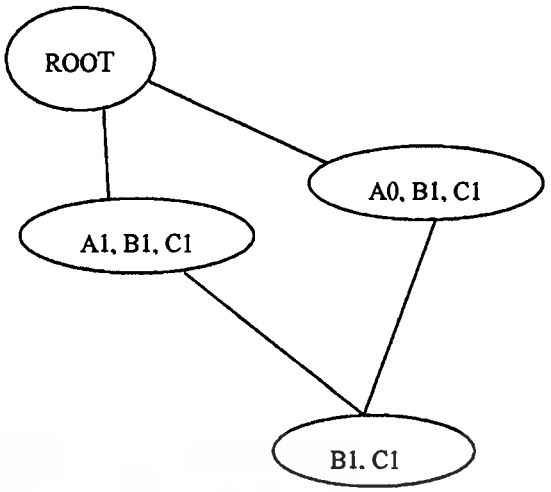


Figure 3.11 Shows the result after the fourth step

After that, check both B1, C1 parent's node. If parent node has more than one child node, getShareObject will be call to find shareObject under the parent node and continue the process which will send shareObject in to addNode method. In this case both B1, C1's parents node have only one child, then continue to next step.

5. Read more row which is A2, B3, C1. Node A2, B3, C1's parent which is ROOT has more than 1 vertex so we call getShareObject (for all element under ROOT with the current row). Node A2, B3, C1 and A0, B1, C1 have same shareObject **C1**, node A2, B3, C1 and A1, B1, C1 have same shareObject **C1**. The result which returned from getShareObject method is C1. C1 will be sent into addNode method and it will be checked if there is node C1 in the graph or not (in this case, not). Then check if C1 is superset of A1, B1, C1's or A0, B1, C1's children or not (in this case not). Then check if C1 is subset of A1, B1, C1's or A0, B1, C1's children or not (in this case, C1 is subset of B1, C1). Therefore, call createNode to add node C1 and then call createEdge to add Edge from A2, B3, C1 and from B1, C1 to C1. While adding edge createEdge method will check both C1 parents whether they have more than one child or not (in this case both of them have only one child, so continue to the 6^{th} step).
6. Reads more row A4, B5, C1 and check if the node is existed or not, which it is not (in this case). Then call createNode to add node A4, B5, C1 and call createEdge to add edges from ROOT to A4, B5, C1 then check A4, B5, C1's parent (which is ROOT), node ROOT has more than one child then method getShareObject is called. Node A4, B5, C1 and node A0, B1, C1 have same shareObject **C1**, node A4, B5, C1 and node A1, B1, C1 have same shareObject **C1**, node A4, B5, C1 and node A2, B3, C1 have same shareObject **C1**. The returned result from shareObject method will be sent into addNode

method. There C1 will be check if it is already exist or not, in this case node C1 is already existed, so we just call method createEdge to add edge from node A4,B5,C1 to node C1. Then we check if node A4, B5, C1 has more than one child or not, in this case A4, B5, C1 has only one child. Continue to next step.

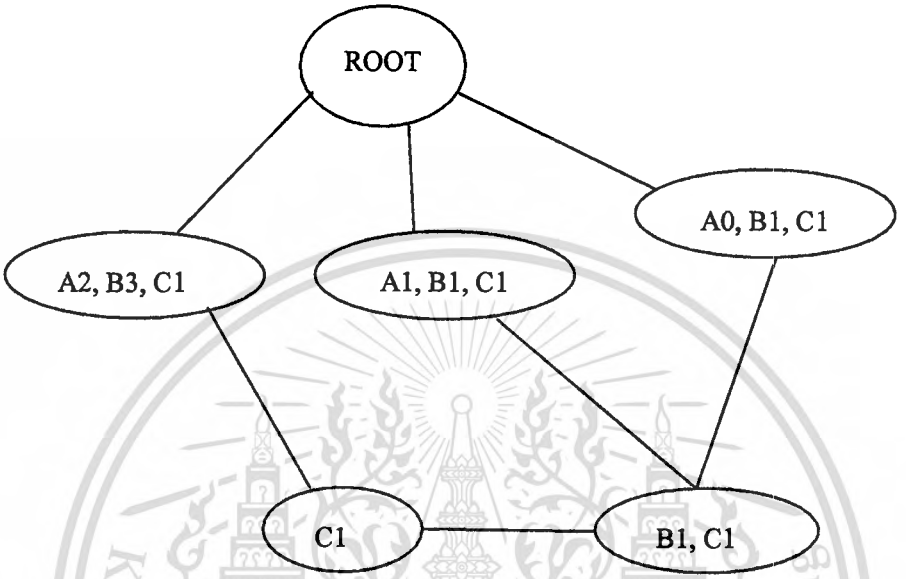


Figure 3.12 Shows the result after the fifth step

7. Read new row A6, B5, C2 and check if the node is already exist or not (in this case it is not exist). Add node A4, B5, C1 and add edges from ROOT to A6, B5, C2 then we check if A6, B5, C2's parent has more than one child or not, in this case ROOT has more than one child. Then call method getShareObject to find shareObject. Node A6, B5, C2 and node A4,B5,C1 have same shareObject **B5.** B5 will be sent into addNode method and check if node B5 is already exist or not, in this case it is not exist yet. Then we check if node B5 is superset of node A6, B5, C2 and node A4, B5, C1 or not, in not check if node B5 is subset of node A6, B5, C2 and node A4, B5, C1 or not(in this case not). Then call method createNode to add node B5 and method createEdge to add Edge from node node A6, B5, C2 to node B5. The process will then check both node A6, B5, C2 and node A4, B5, C1's child (node B5 parents) wheater it have more than one child or not, in this case Node A6, B5, C2 has only one child but node A4, B5, C1 has two children(node B5 and node C1). Then node B5 and node C1 will be sent into getShareObject to find shareObject, in this case it cannot find shareObject from node B5 and node C1, so continue to next step.

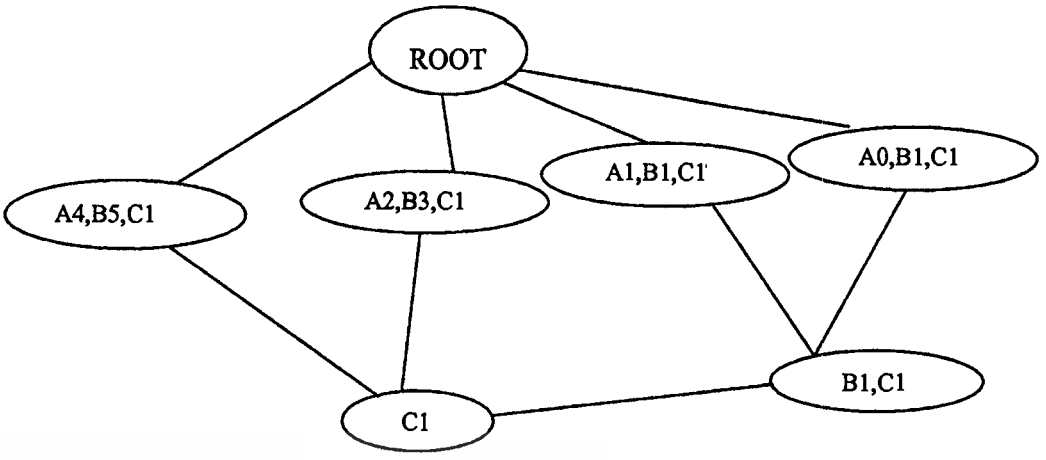


Figure 3.13 Shows the result after the sixth step.

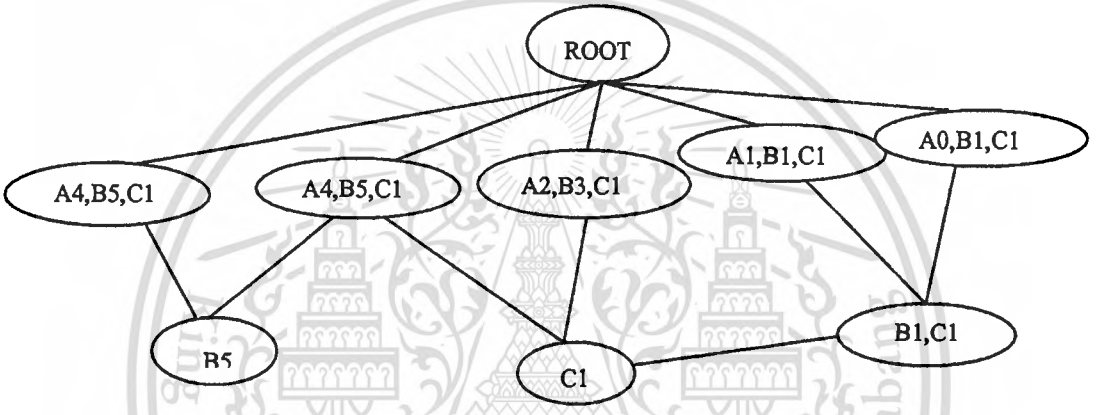


Figure 3.14 Shows the result after the seventh step

8. Now there is no more row to read. The END node will be created using method createNode to add END node. And then add edge from every exist node which have no child (in this example is node B5 and node C1) to END node. Therefore, the graph below shows the overall graph.

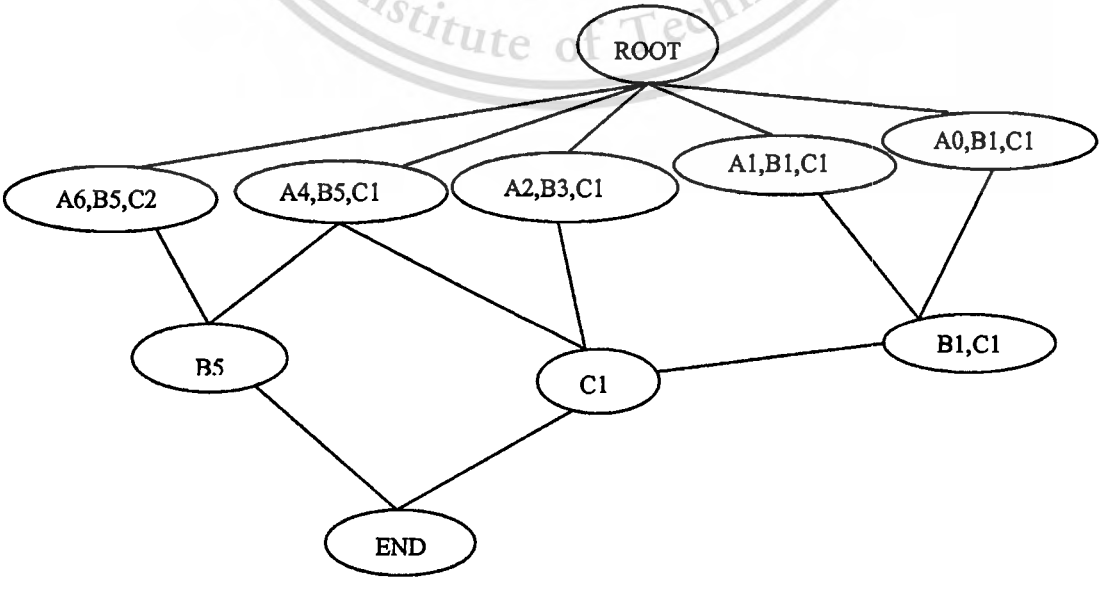


Figure 3.15 Shows the completed graph

3.5 Traverse Graph

After finish constructing graph, we can see that the result of the graph use less storage than the result of the original cube by operation (Table 2.2). The result from the graph use only 8 nodes to store data but the result from the cube by operation use 30 tuples to store data. From our construction idea of the graph, the object that does not share with another will not be created as a new node. Figure 3.16 shows that one node in the FCA graph should represent 8 nodes in a lattice cuboid.

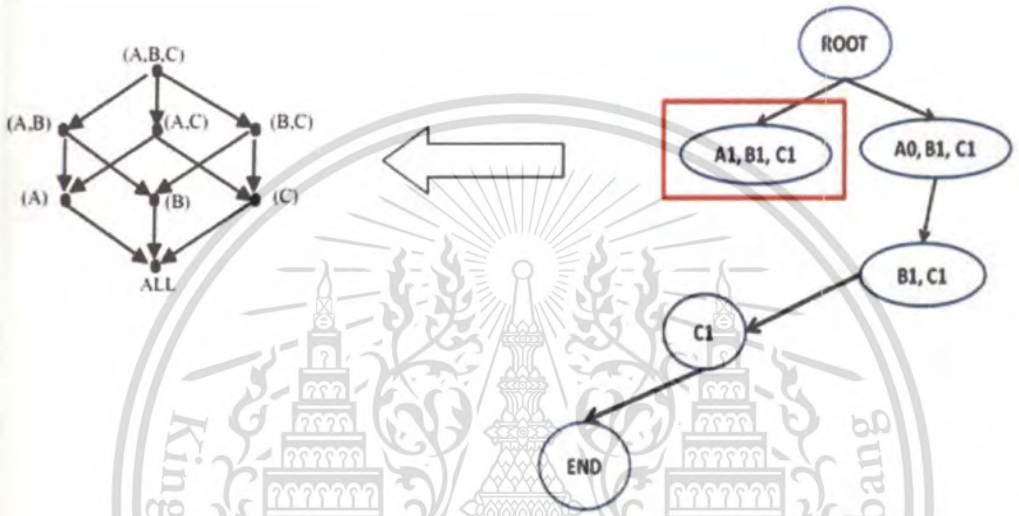


Figure 3.16 Shows that one node in the graph(right) should consist up to 8 nodes in the cube lattice(left)

For example, A1, B1, C1 should have (A1, B1) (A1, C1) (B1, C1) (A1) (B1) (C1) (ALL) However, (B1, C1) (C1) is already existed in the graph, therefore we have to bottom-up traverse graph after graph construction is finished and get all this hidden object. The hidden object will not redundant with any nodes in the FCA graph. After bottom up traverse graph the result will be shown in Table 3.2.

Key	Hidden Object
A0, B1, C1	(A0, B0), (A0, C1), (A0)
A1, B1, C1	(A1, B1), (A1, C1), (A1)
A2, B3, C1	(A2, B3), (A2, C1), (B3, C1), (A2), (B3)
A4, B5, C1	(A4, B5), (A4, C1), (B5, C1), (A4), (B5)
A6, B5, C2	(A6, B5), (A6, C2), (B5, C2), (A6), (B5), (C2)
B1, C1	(B1)
C1	()
B5	()

Table 3.2 Shows the hidden object result.

From the result table 3.2, the key column of the result is the node where the hidden object is store. The result and hidden node are store in a HashMap. The nodes are key in the HashMap and the Hidden Objects are values of each key.

3.6 Retrieve Data from Graph

In this section, we provide the approach to retrieve the data from the graph that we had constructed before. The approach to retrieve the data cube from graph will be separated into cases.

1. Retrieve all cube data from graph, which means all possible combination from data cube will be retrieved.
2. Retrieve the specific data from graph, which means that user might want to look at the specific data instead of all data.

3.6.1 Retrieve All Cube Data

This example shows the approach to retrieve all data cube from graph. The graph in this example is the same as the complete graph in figure 3.15

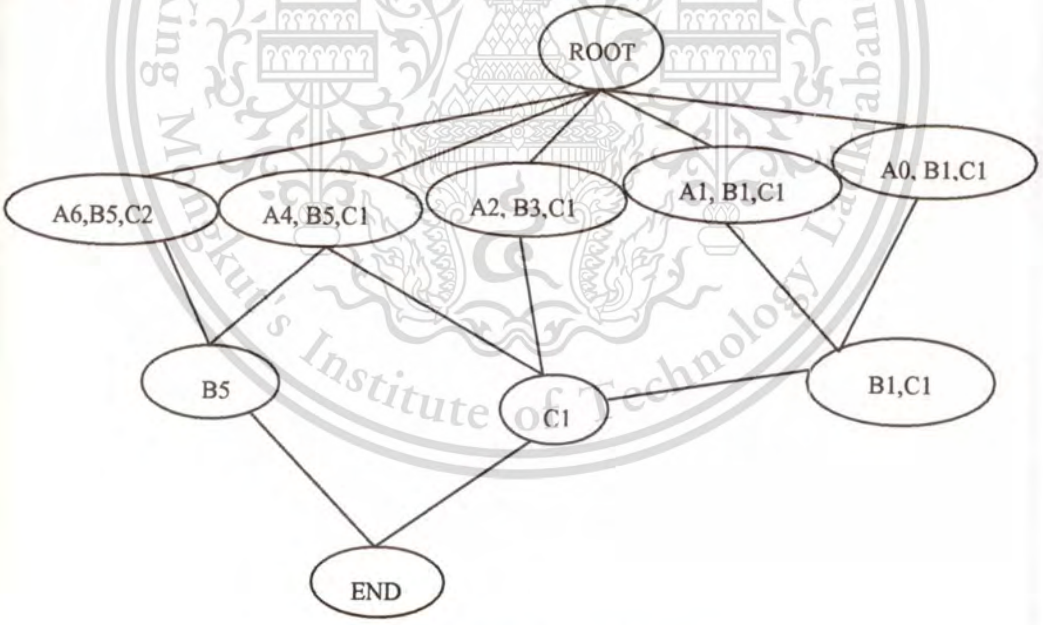


Figure 3.17 The complete graph

First, Visit node A0, B1, C1 and get the measurement value and the hidden object with the same measurement value as node A0, B1, C1 (Figure 3.18).

Second, If A0, B1, C1 has children, visit the children node (node B1, C1), then get the hidden object and the measurement value of node B1, C1 by bottom up traverse (Figure 3.19).

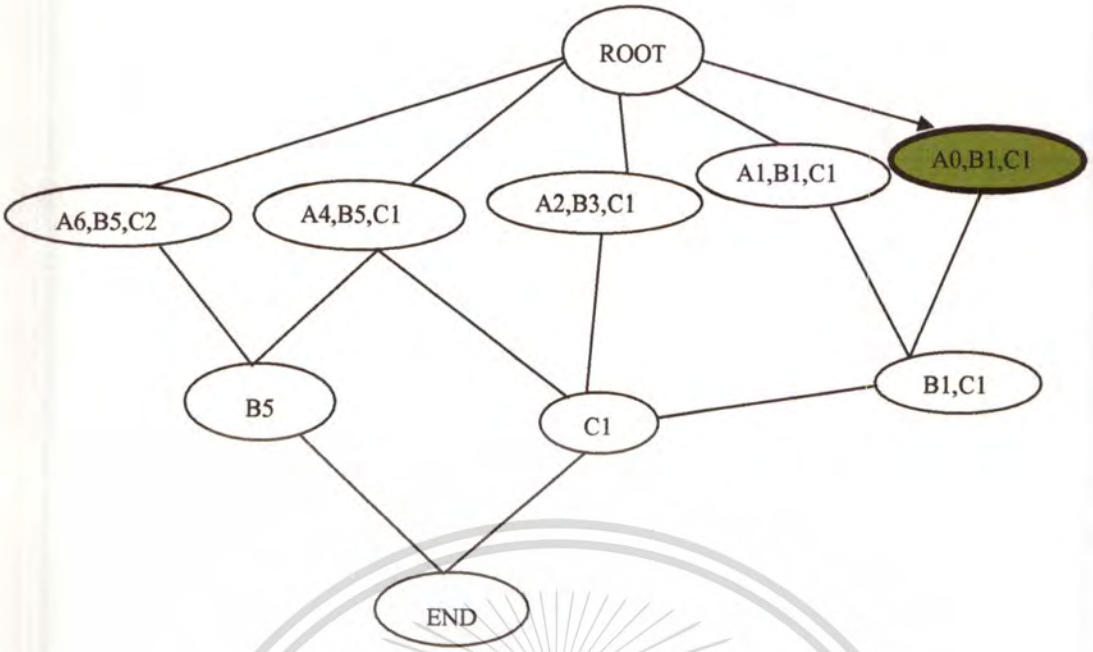


Figure 3.18 Shows the first step of retrieving data

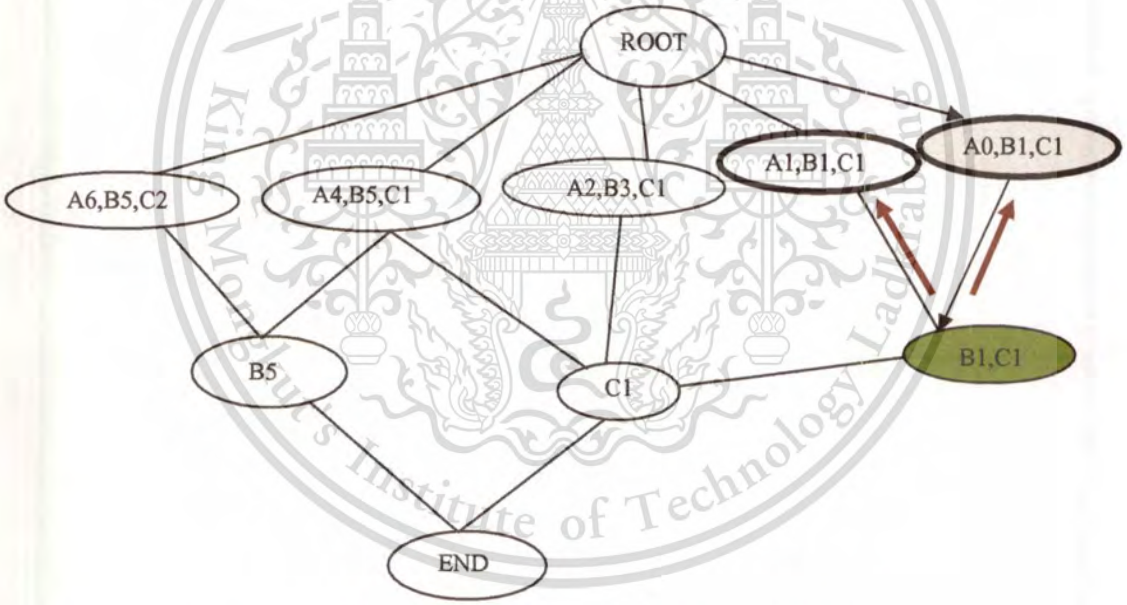


Figure 3.19 Shows the second step of retrieving data

Third, If node B1, C1 has children (which is node C1), visit node C1 then get the hidden object and the measurement value of node C1 by bottom up traverse (Figure 3.20).

Fourth, Node C1's child is END node, then visits the new path (When it reaches the END node it will visit the new path) (Figure 3.21).

Fifth, After the first path (start from node A0, B1, C1) reach the END node, continue to the next path which is node A1, B1, C1 then get measurement value and the hidden object with the same measurement value as node A1, B1, C1 (Figure 3.22).

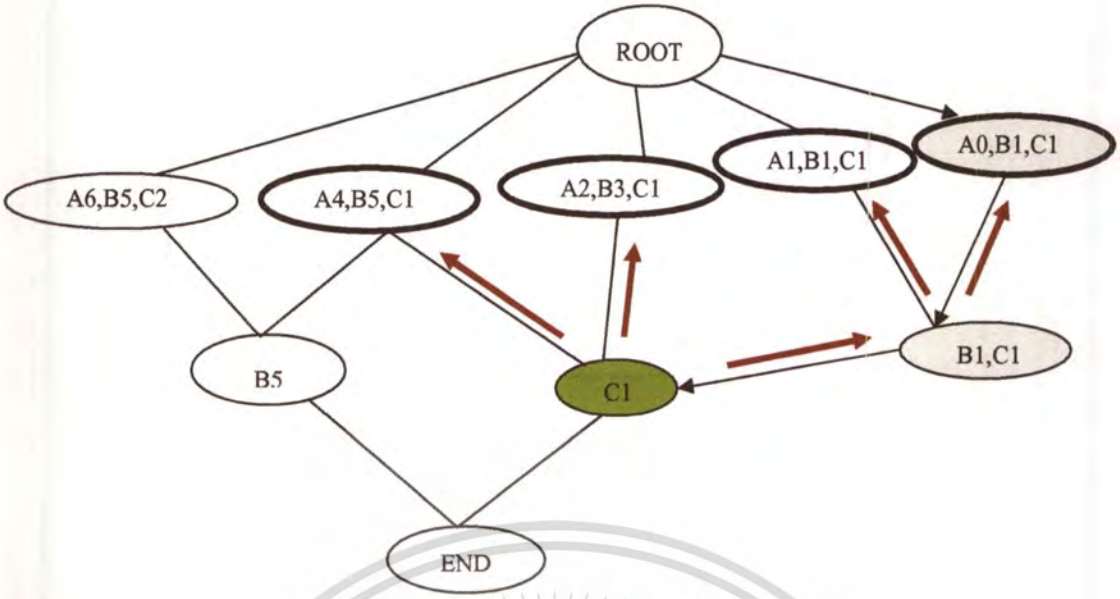


Figure 3.20 Shows the third step of retrieving data

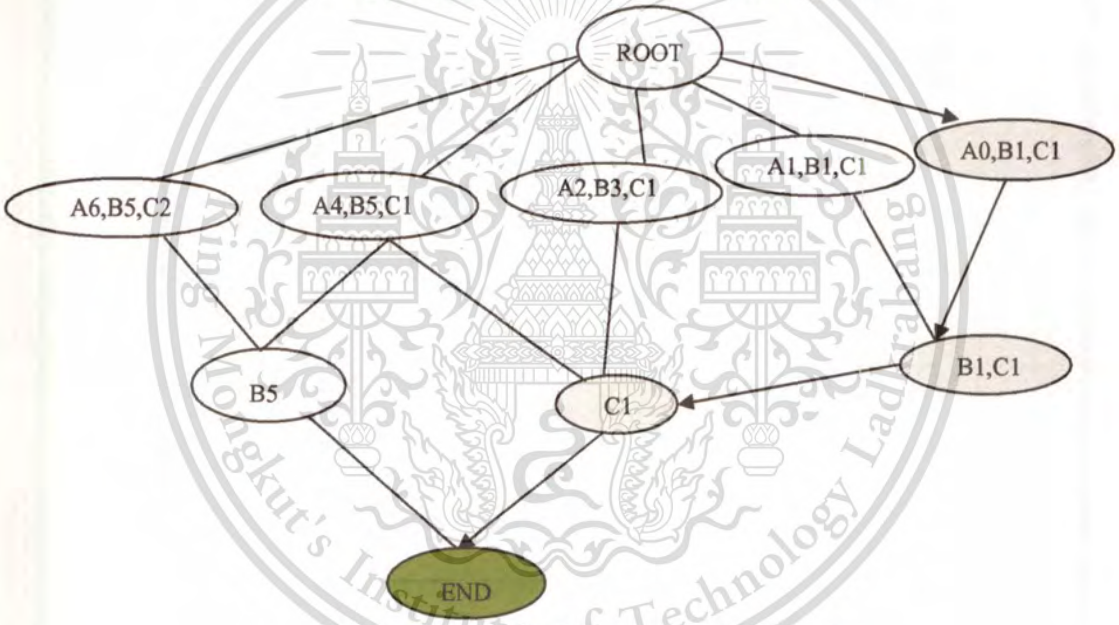


Figure 3.21 Shows the fourth step of retrieving data

Sixth, If node A1, B1, C1 has children, visit the children node. In this case node A1, B1, C1 children is node B1, C1 but node B1, C1 is already visited so it will visit the new path (Figure 3.23).

Seventh, Visit node A2, B3, C1 then get measurement value and hidden object with the same measurement value as node A2, B3, C1 (Figure 3.24).

Eighth, If node A2, B3, C1 has children, visit the children node. In this case node A1, B1, C1 children is node C1 but node C1 is already visited (Figure 3.25).

Ninth, Visit node A4, B5, C1 then get measurement value and the hidden object with the same measurement value as node A4, B5, C1 (Figure 3.26).

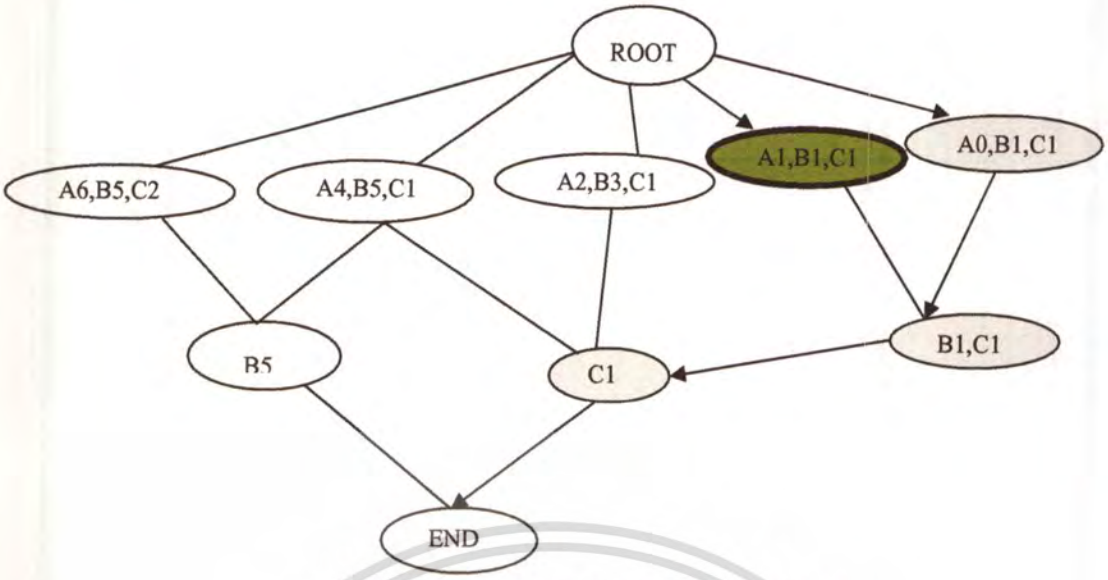


Figure 3.22 Shows the fifth step of retrieving data

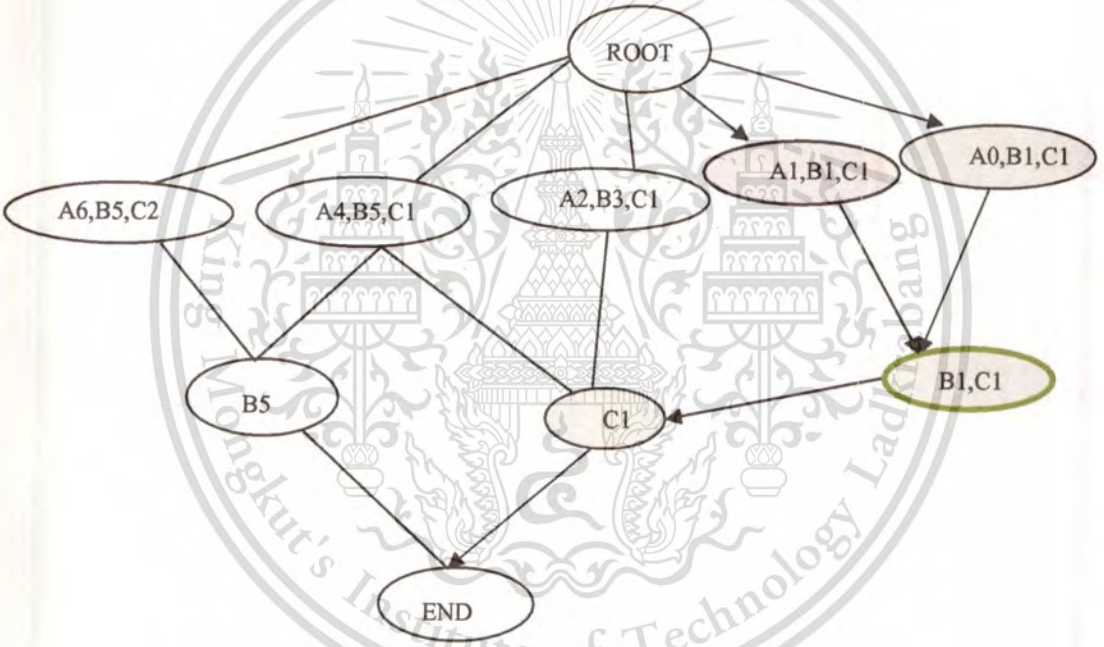


Figure 3.23 Shows the sixth step of retrieving data

Tenth, If node A4, B5, C1 has children, visit the children node. In this case node C1 and node B5 is children of node A4, B5, C1 which node C1 is already visited (Figure 3.27).

Eleventh, Visit another child of node A4, B5, C1 which is node B5. Then get the hidden object and the measurement value of node B5 by bottom up traverse (Figure 3.28).

Twelfth, Node B5 child is END node, then visits the new path (Reach END node change to visit new path) (Figure 3.29).

Thirteenth, Visit node A6, B5, C2 then get measurement value and the hidden object with same measurement value as node A6, B5, C2 (Figure 3.30).

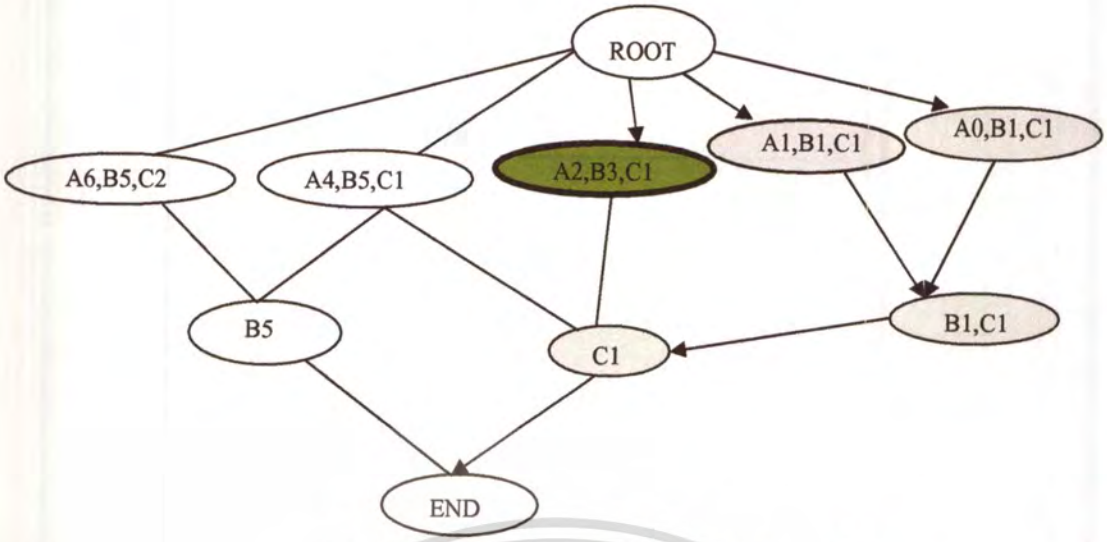


Figure 3.24 Shows the seventh step of retrieving data

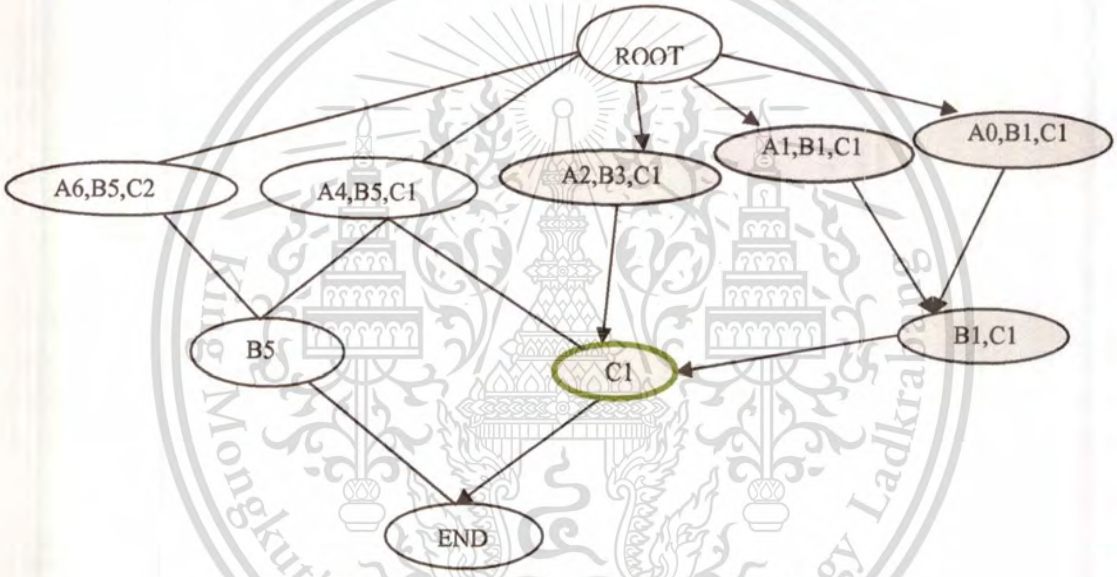


Figure 3.25 Shows the eighth step of retrieving data

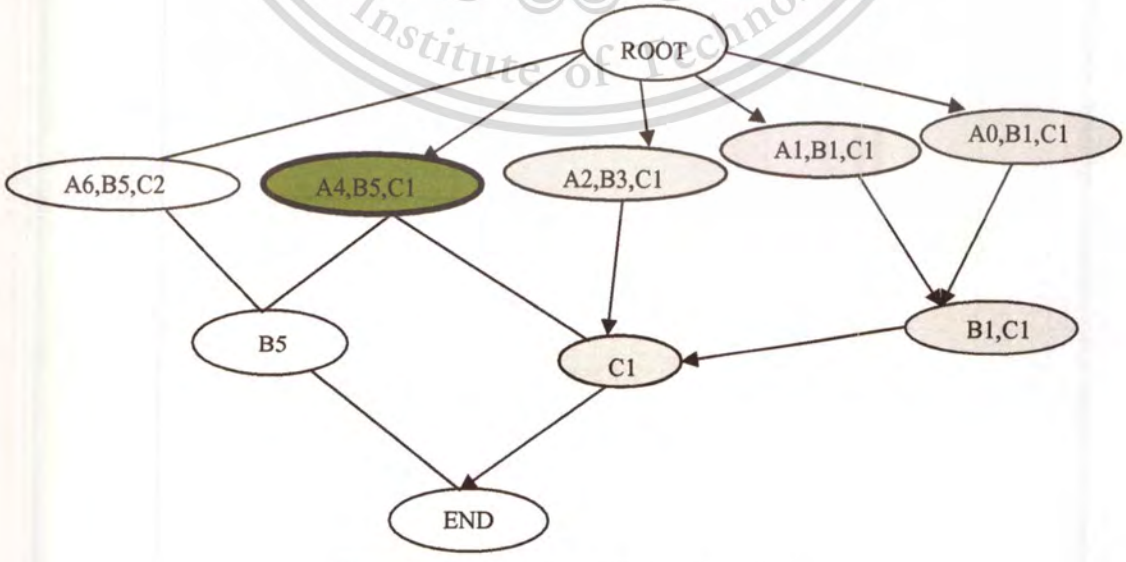


Figure 3.26 Shows the ninth step of retrieving data

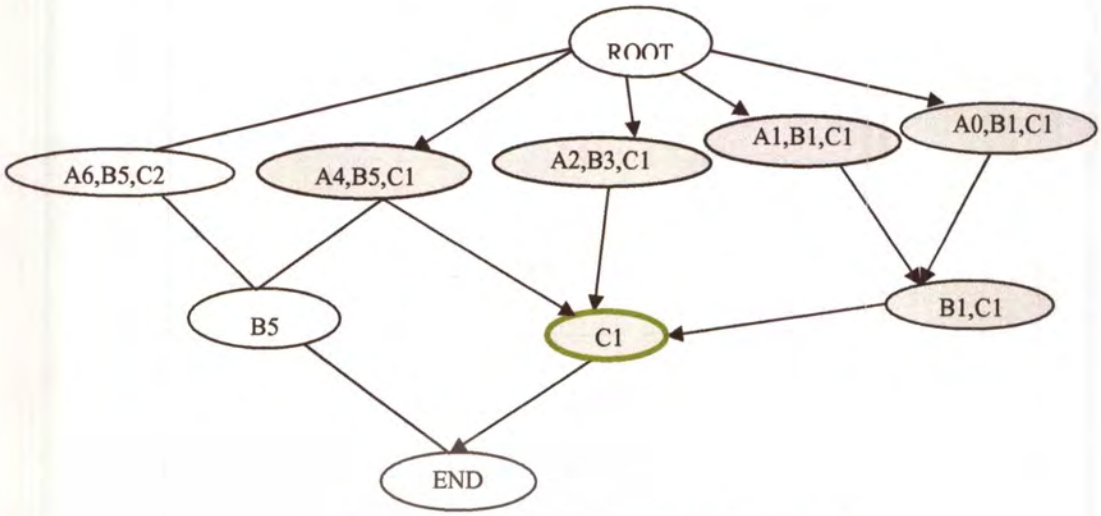


Figure 3.27 Shows the tenth step of retrieving data

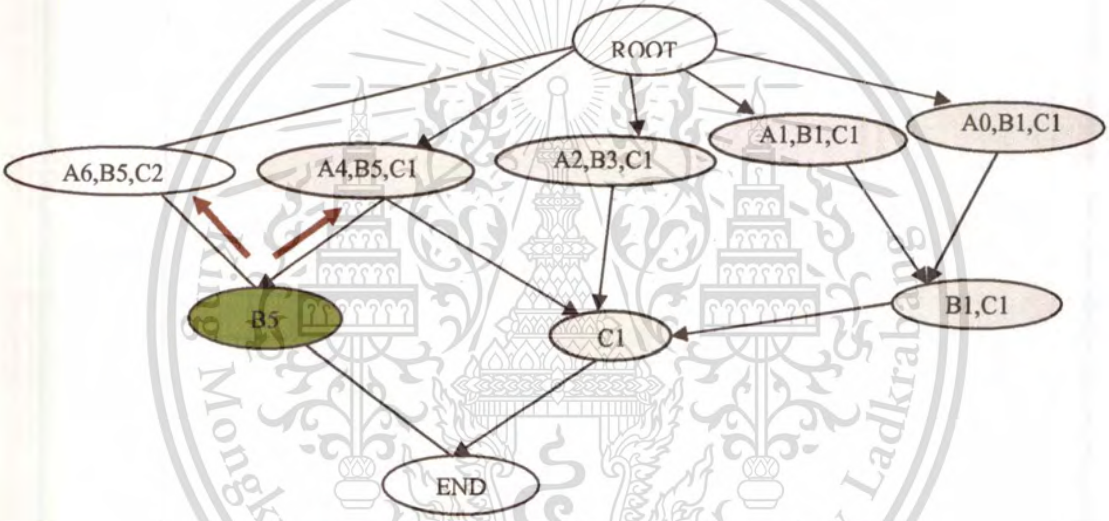


Figure 3.28 Shows the eleventh step of retrieving data

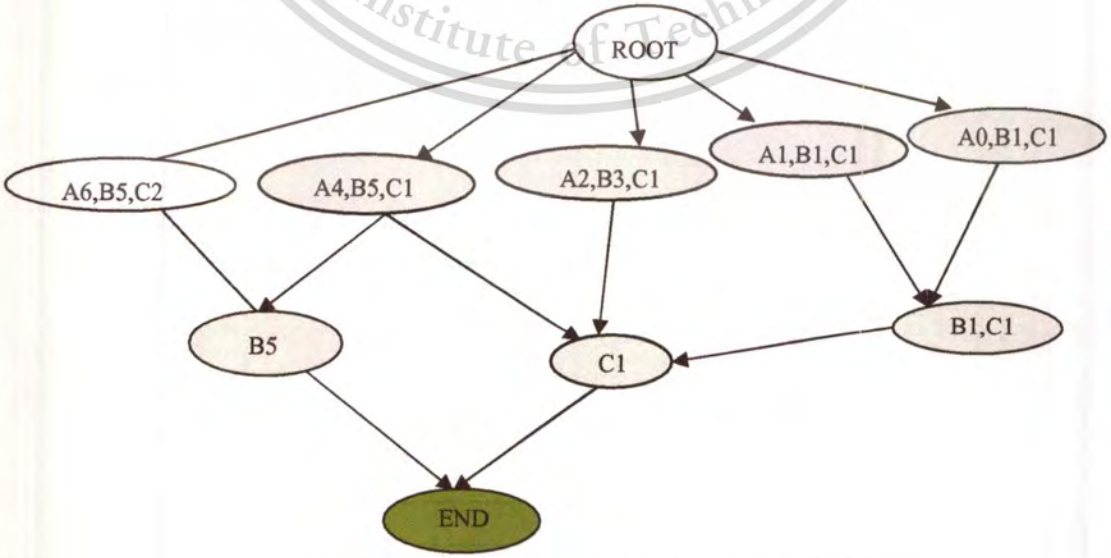


Figure 3.29 Shows the twelveth step of retrieving data

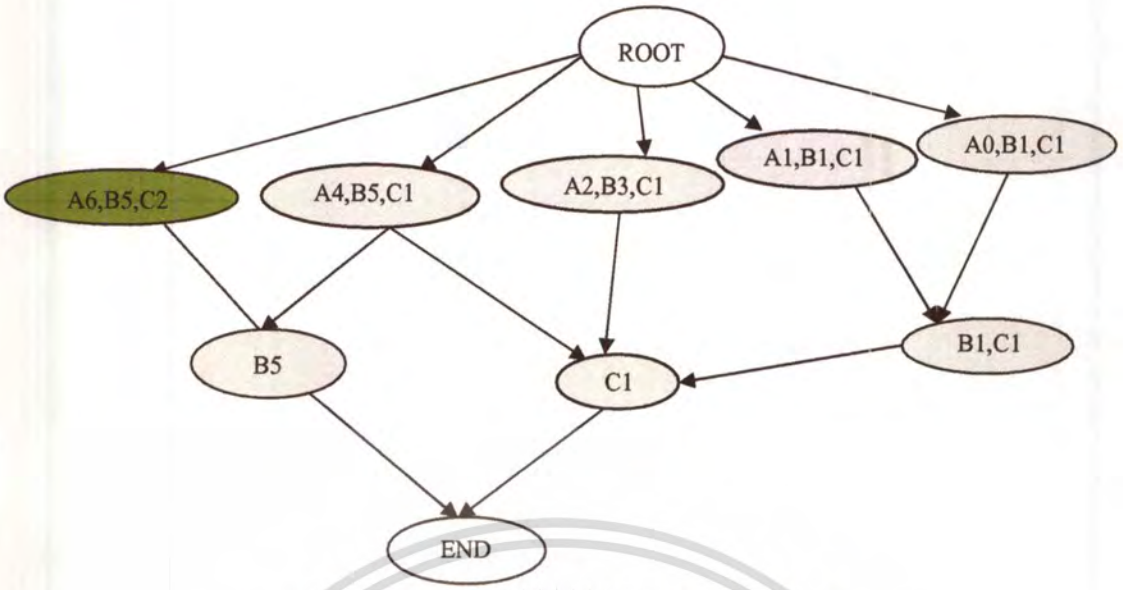


Figure 3.30 Shows the thirteenth step of retrieving data

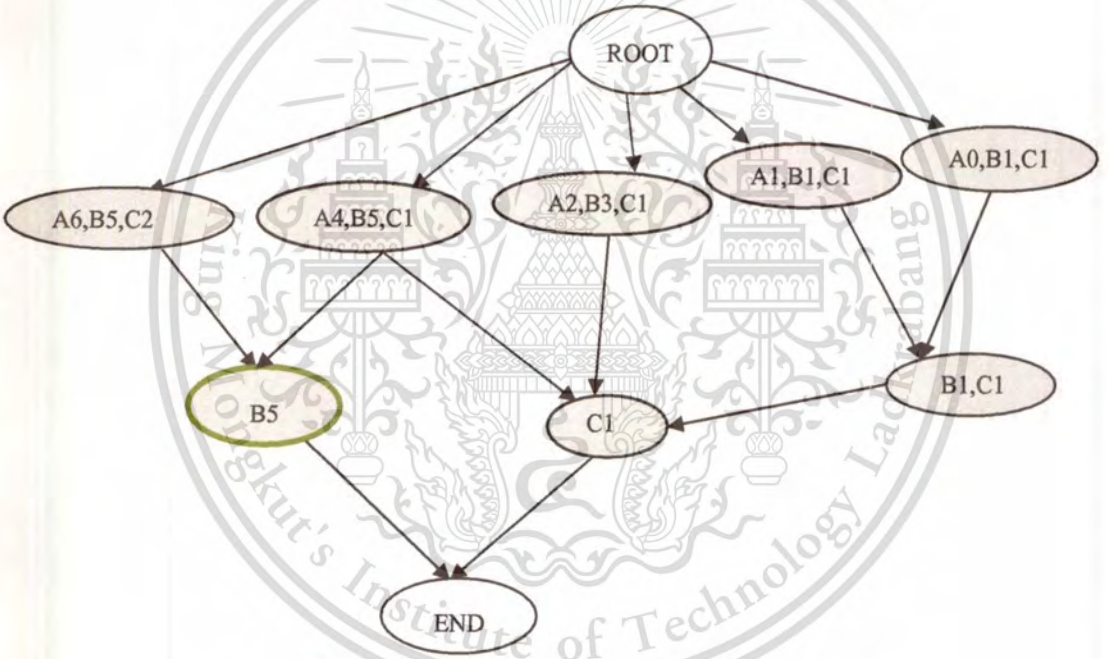


Figure 3.31 Shows the fourteenth step of retrieving data

Fourteenth, If node A6, B5, C2 has children, visit the children node. In this case node B5 is child node of node A6, B5, C2 which it is already visited (Figure 3.31).

Lastly, No more node to visit. The process is done.

At the end, the result will be the same as CUBE-BY operation

- [All]	: 360	- [a4, b5, c1]	: 70
- [a1, b1, c1]	: 100	- [a4, b5]	: 70
- [a1, b1]	: 100	- [a4, c1]	: 70
- [a1, c1]	: 100	- [b5, c1]	: 70

- [a1]	: 100	- [a4]	: 70
- [a0, b1, c1]	: 50	- [a6, b5, c2]	: 80
- [a0, b1]	: 50	- [a6, b5]	: 80
- [a0, c1]	: 50	- [a6, c2]	: 80
- [a0]	: 50	- [b5, c2]	: 80
- [a2, b3, c1]	: 60	- [a6]	: 80
- [a2, b3]	: 60	- [c2]	: 80
- [a2, c1]	: 60	- [b1, c1]	: 150
- [b3, c1]	: 60	- [b1]	: 150
- [a2]	: 60	- [b5]	: 150
- [b3]	: 60	- [c1]	: 280

3.6.2 Retrieve Specific Data

In this section, we show the approach that user may want to see at the specific data instead of all data. There are 2 examples in this section which will be different on how to find the result.

Example 1 – If user wants to find measurement of B1, C1

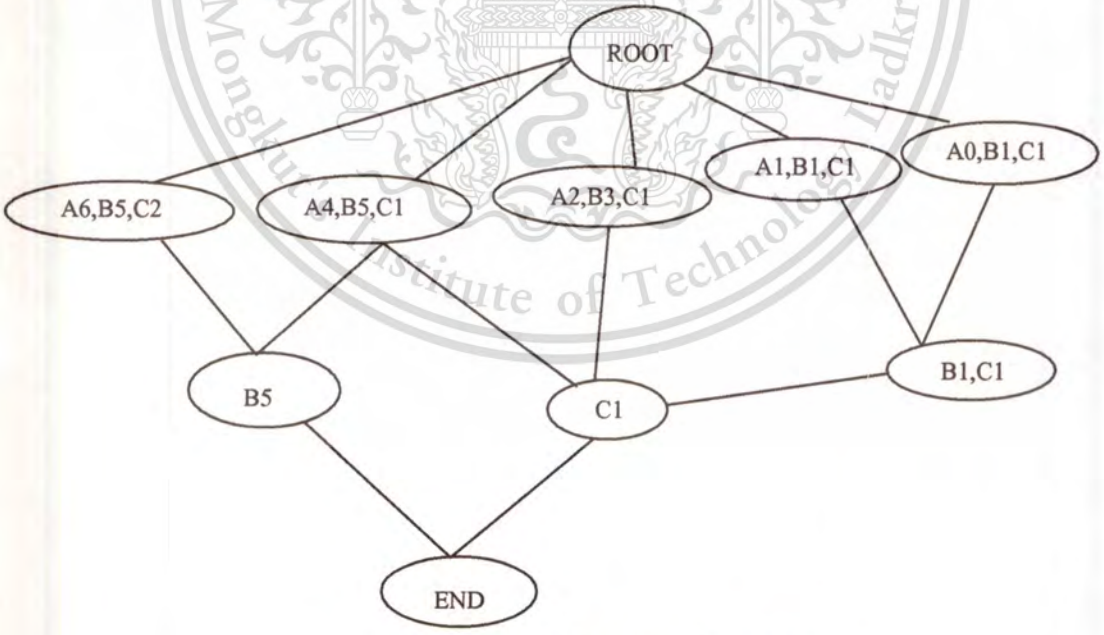


Figure 3.32 Shows the complete graph

First, Visit node A0, B1, C1. Value that we want is node B1, C1 which is subset of node A0, B1, C1 (Figure 3.33).

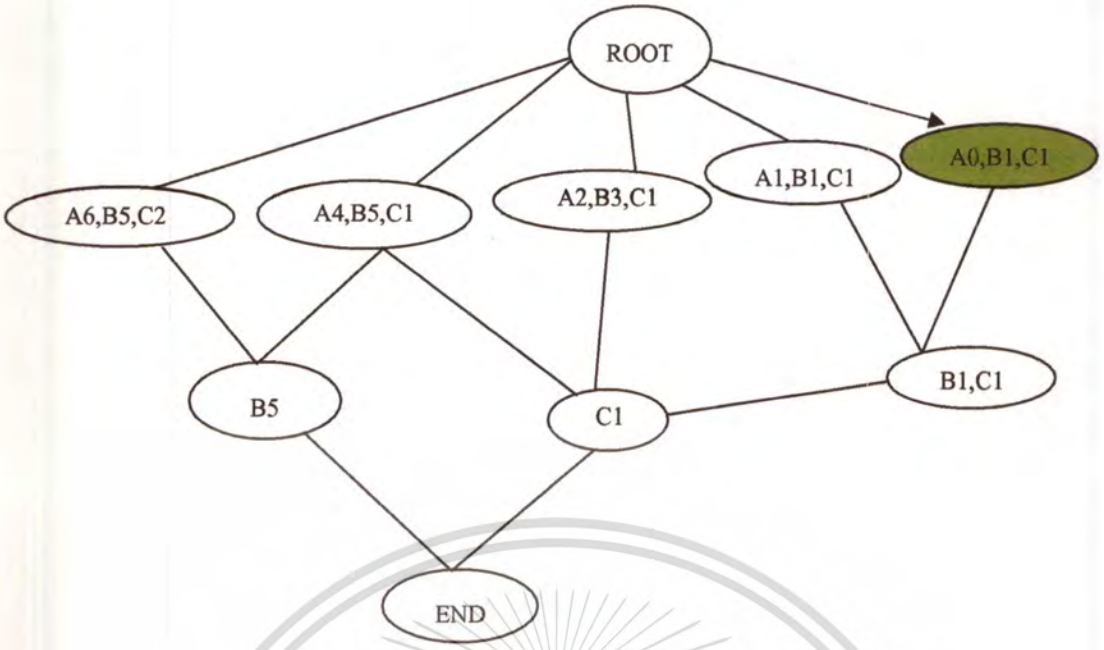


Figure 3.33 Shows the first step

Second, Visit all A0, B1, C1 children node. In this case node A0, B1, C1 has only one child which is node B1, C1.

Finally, Now node B1, C1 is found. Get measurement value by bottom up traverse (Figure 3.34).

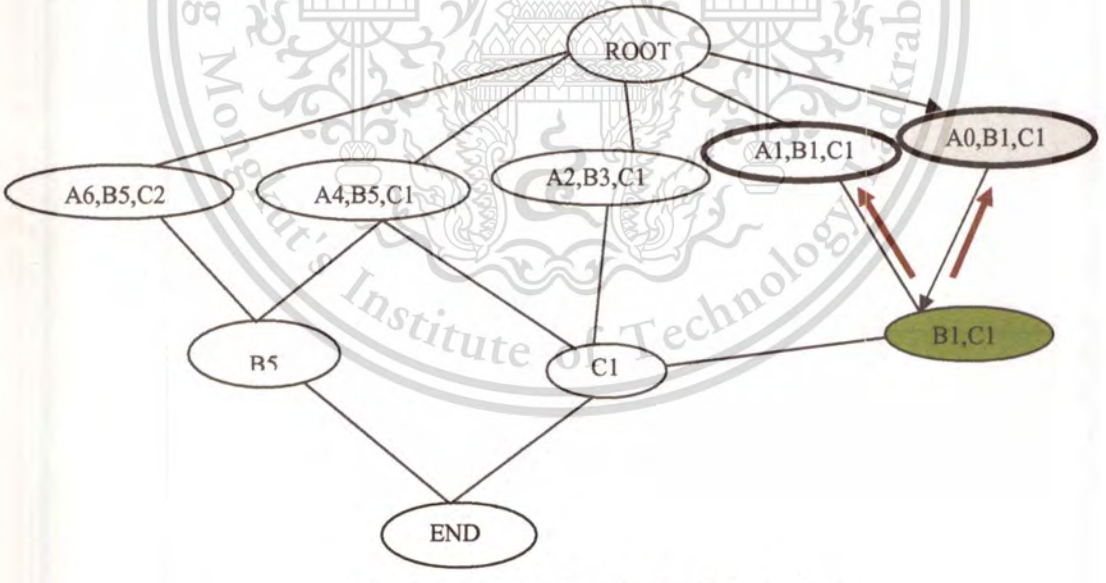


Figure 3.34 Shows the second and the final step

Example 2 – If user wants to find measurement of B3, C1

First, Visit node A0, B1, C1. But node A0, B1, C1 does not contain node B3, C1. If the first node does not contain the data that we are searching for, it will not contain in that path also; therefore, we can move on to another path.

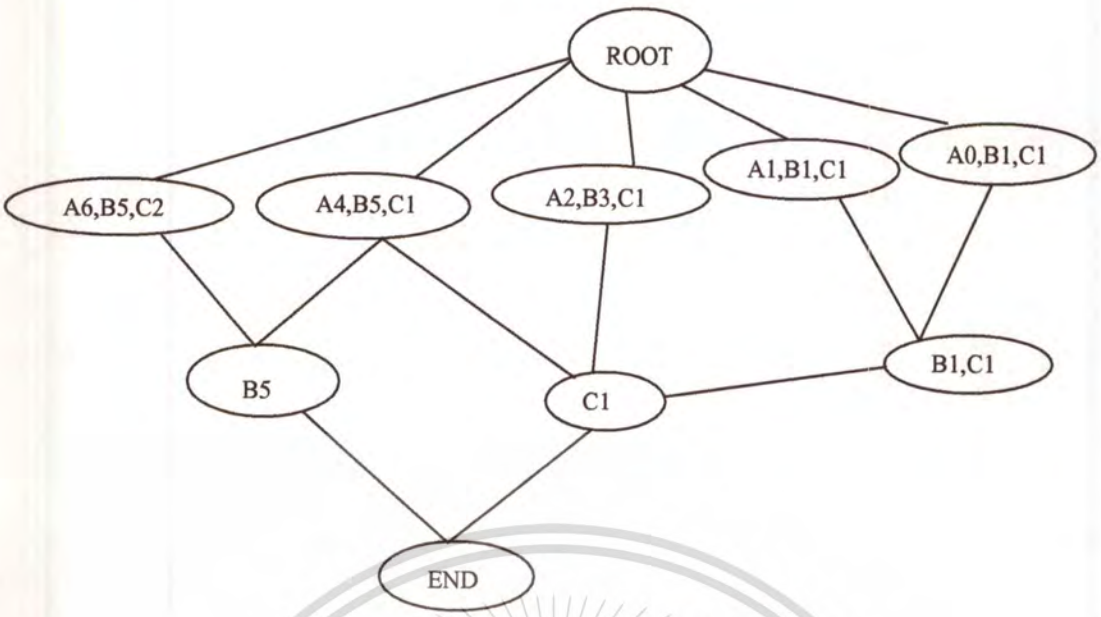


Figure 3.35 Shows the complete graph

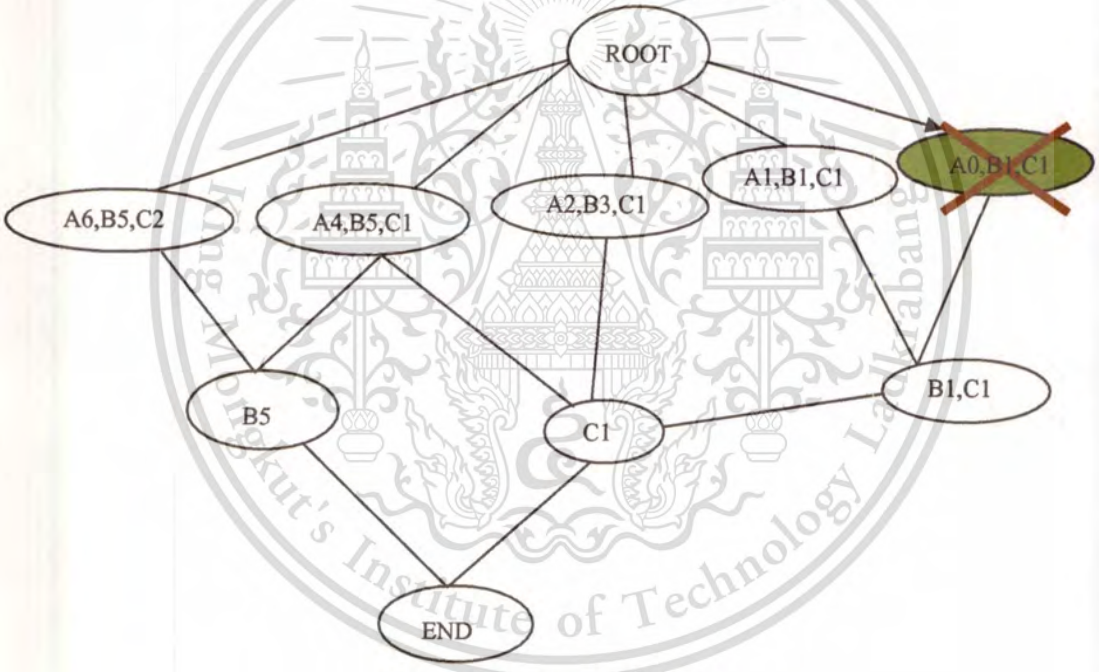


Figure 3.36 Shows the first step

Second, from the first step, we know that the first path we have visited does not have B3, C1 in that path. Therefore, in this step we will go on to the new path which we will visit node A1, B1, C1. But again A1, B1, C1 does not contain B3, C1 (Figure 3.37).

Third, Change path to visit node A2, B3, C1. Now node A2, B3, C1 contain node B3, C1 (Figure3.38).

Fourth, Visit all A2, B3, C1 children node. In this case node A2, B3, C1 has only one child which is node C1. But node C1 is subset of node B3, C1 (We need node B3, C1 not only node C1) (Figure 3.39).

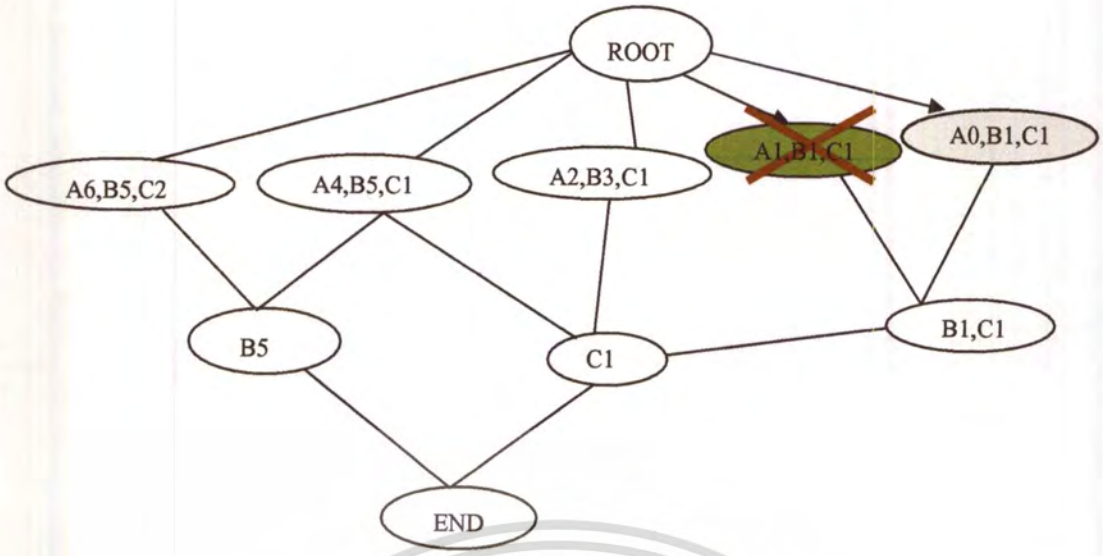


Figure 3.37 Shows the second step

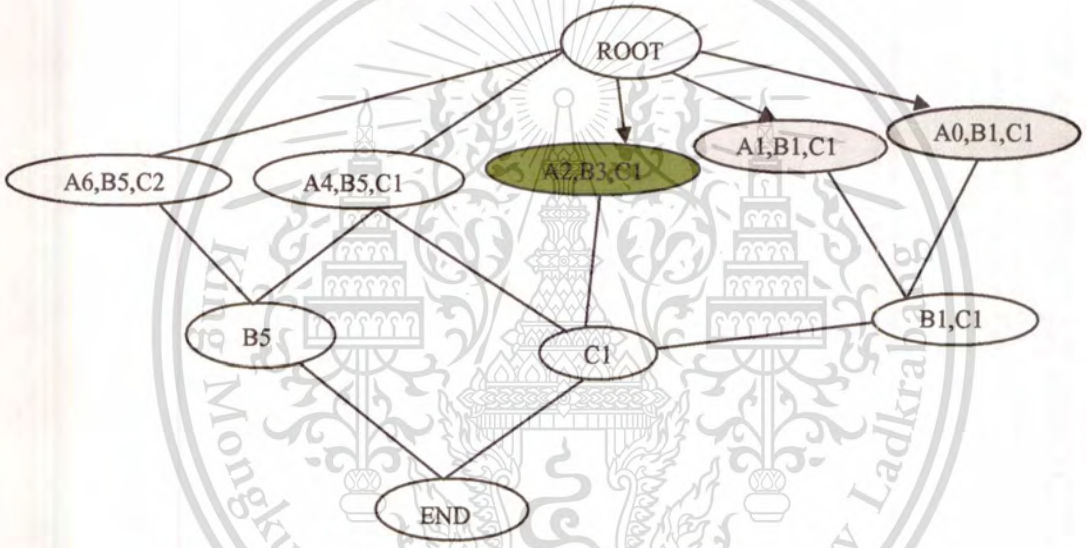


Figure 3.38 Shows the third step

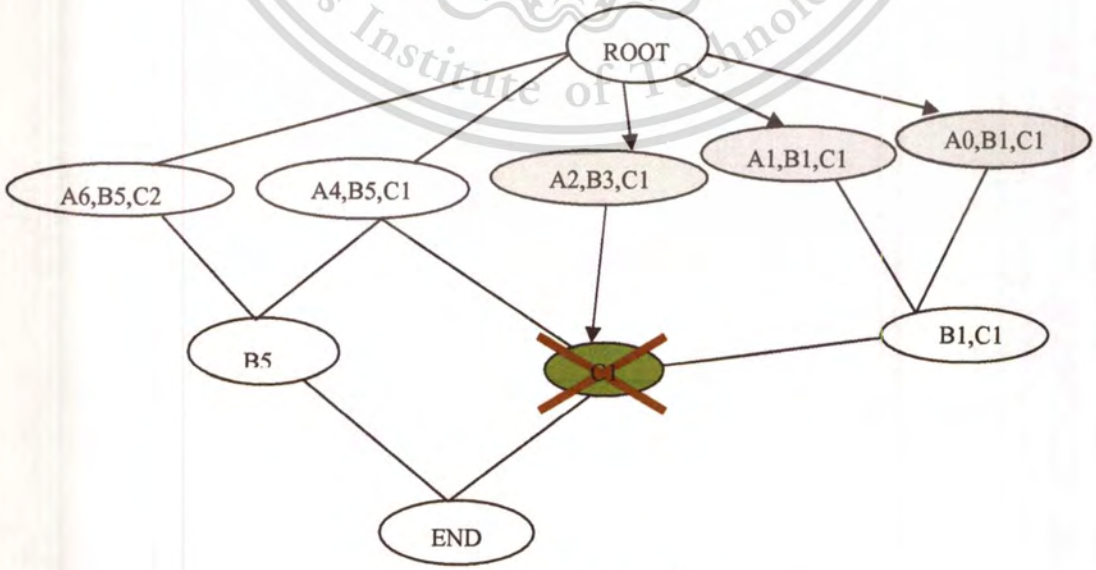


Figure 3.39 Shows the fourth step

Finally, Visit previous node which is node A2, B3, C1 and get measurement value. The measurement of B3, C1 is the same as measurement of A2, B3, C1 (Figure 3.40).

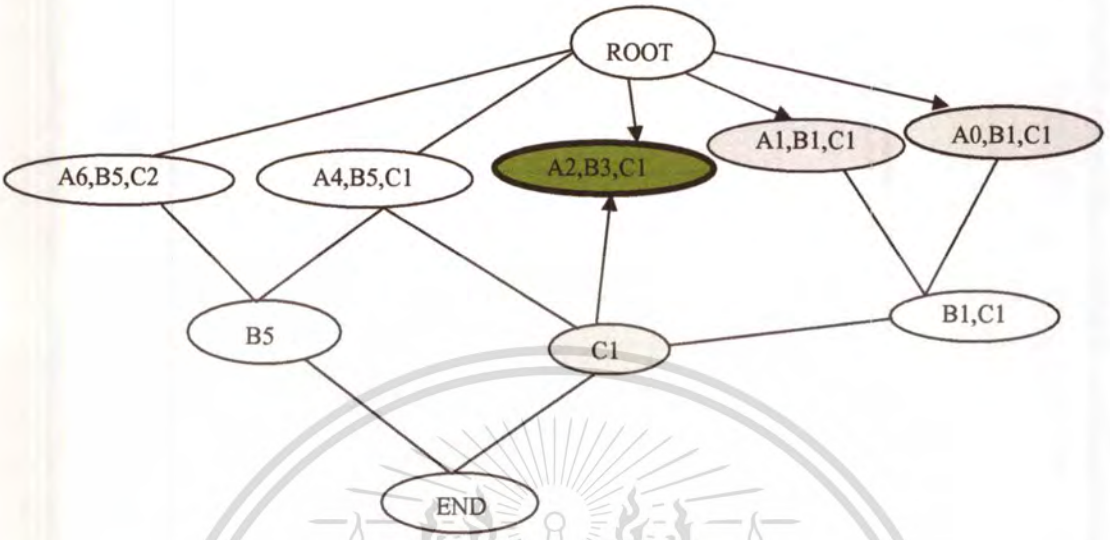


Figure 3.40 Shows the final step

3.7 Classes

In this session we will provide information about the main classes and methods in the class which is used for FCA CUBE program.

3.7.1 Class Main

This class will control all the system in the program, it will tell system what to do first, how to handle the input, how to create graph, etc. This class provides only one method called "controller". Method controller's algorithm has already described in main process algorithm which describe in session 3.3.

3.7.2 Class CreateXML

This class is a class that retrieve each node from graph and generate in into the XML file. This XML file will be use for prefuse to generate the output for the graph.

3.7.3 Class RadialGraphView

This class is the class which prefuce has provided. For further information please check in Apandix.

3.7.4 Class SimpleGraph

This class provides the following methods.

- Method controller. This method will receive input row from main class. And send it to method createNode, createEdge, findShareObject, findSuperSet and removeSubset.
- Method createNode. This method is used for add new node in the graph.
- Method removeNode. This method is used for delete node from the graph.
- Method createEdge. This method is used for add new edge between two nodes in the graph. And also it will check if parent node has more than one child or not, by sending the child node into method getEdgeCount. And if it has more than one child, It will add all child node in the queue using addQueue method.
- Method getEdgeCount. This method will check hoe many children does the input node has.
- Method addQueue. This method used for adding the input node into queue.
- Method removeEdge. This method is used for delete the edge between two nodes from the graph.
- Method getShareObject. This method is used for manage input node (ie. Send it to method findShareObject then send it to method findSuperSet and method removeSubset).
- Method findShareObject. This method is used for find the same shared object from two input nodes, and put it in the list name "result".
- Method findSuperSet. This method will find the superset in the shareObject and send it into method remove subset.
- Method removeSubset. This method will delete all subset from shareQueue.
- Method addNode. This method will check if the input node is already existed or not. If not it will check if the now node is superset of existing node or not. If not it will check if the input node is subset of any node or not. This method also check the new node's parent wheater it has more than on child or not. It it has more than one child, all the children will be add into queue.
- Method nochildToEnd. This method is used to find all node which has no child node, and send it to method createEdge for add edge from those no-child node to END node.

- Method createHidden. This method is used to create hidden node which described in the session 3.5.

The algorithm of this class is described in process algorithm Cube in session 3.3.

3.7.5 Class SimpleGraph

This class provide the following methods.

- Method searchAll. This method used when user want to see result for all data. This will simply SUM all the first level nodes's measurement value.
- Method searchSpecific. This method used when user want to see result for specific data value (ie. A0, B1, C*).
- Method searchPivot. This method is used for retrieve all possible data in the graph and store it, so that when user want to view the pivot table, it can retrieve data from the precomputed data (faster).

3.7.6 Class SearchNode

This class is used for output the result of computed value from method searchSpecific in SearchGraph class. The output will be in form of table using jTable.

3.7.7 Class Pivot Class

This class is used for output results which already computed by method searchPivot in SearchGraph class, and show it in form of pivot table which looks similar to what MSExcel dose by using jTable.

Chapter 4

Application: Pivot Table

Our Application is pivot table created by using the data cube based on formal concept analysis algorithm.

First look of our application (Figure 4.1).

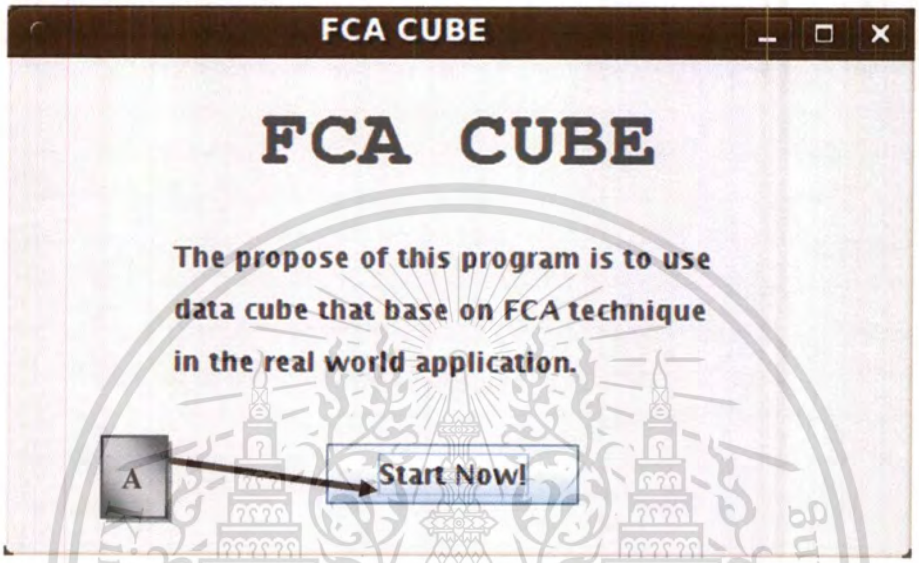


Figure 4.1 FCA CUBE program

A: Button "Start Now!" – Allow user to choose data file.

Choose data file to be compute (data file should be in .txt extension) (Figure 4.2).

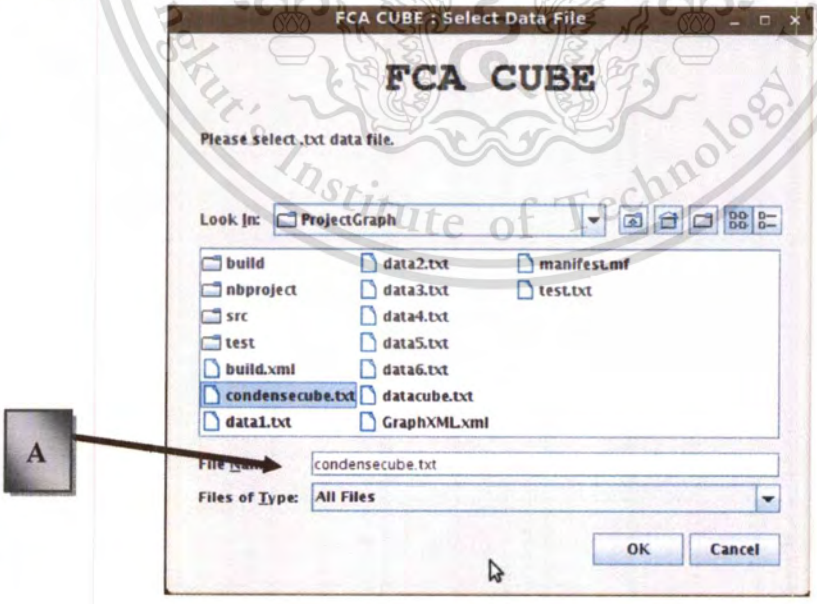


Figure 4.2 Choose data .txt file

A: Data text file named "condensecube.txt".

Choose to view pivot table, view graph or search for result (Figure 4.3).

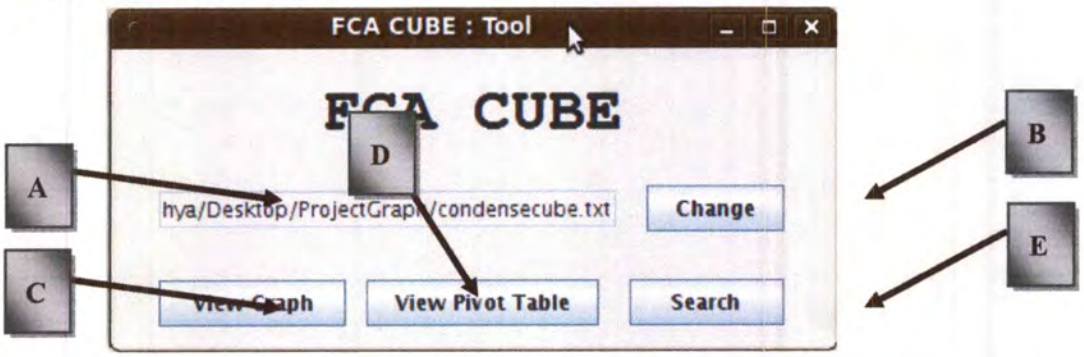


Figure 4.3 FCA CUBE options

A: Show data file location.

B: Button "Change" – Allow user to change data file.

C: Button "View Graph" – Show graph.

D: Button "View Pivot Table" – Allow user to choose input for row and column for pivot table.

E: Button "Search" – Allow user to search for specified value.

If users choose to view graph (Figure 4.4).

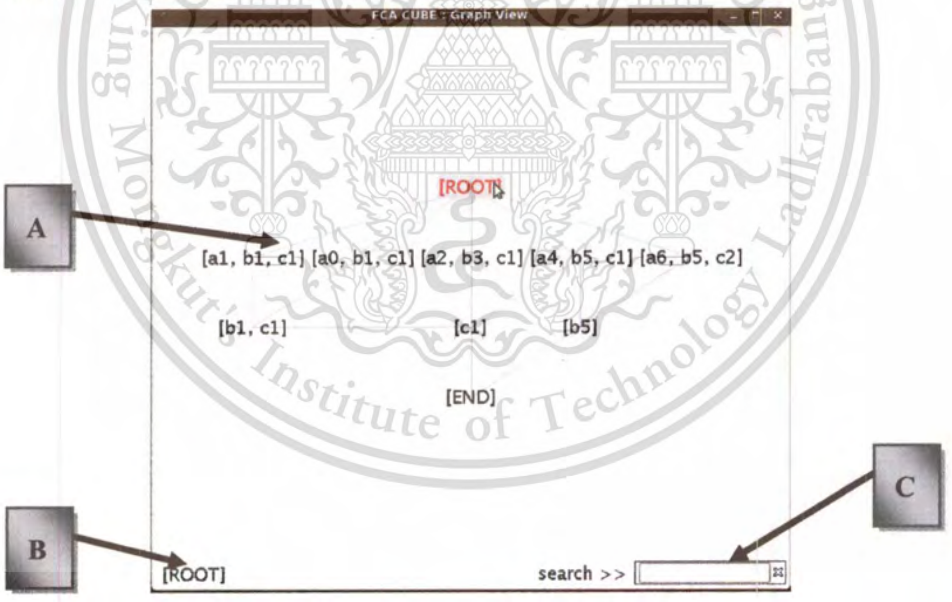


Figure 4.4 Graph view

A: Show graph.

B: Show present top parent node.

C: Search input area – allow user to search for node.

If users choose to view pivot, user can choose the input row and input column (Figure 4.5).

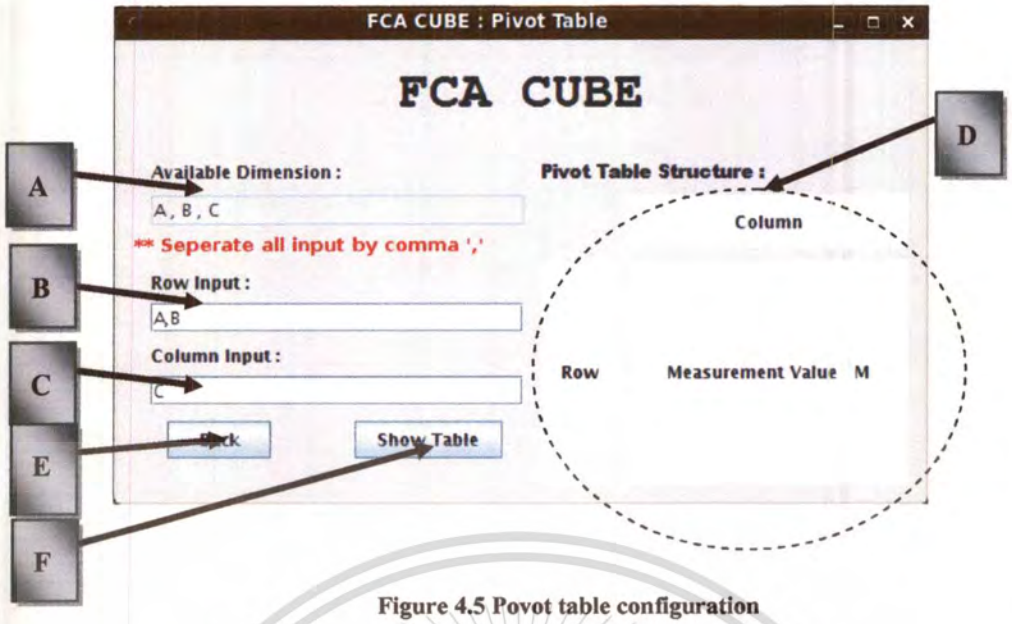


Figure 4.5 Povot table configuration

A: Available Dimension – Show dimensions that available for user to use.

B: Row input – Allow user to input dimensions for row in pivot table.

C: Column input – Allow user to input dimensions for column in pivot table.

D: Pivot Table Structure – Show structure and field's name of pivot table.

E: Button "Back" – Back to previous page (Page after choose data file).

F: Button "Show Table" – Show pivot table according to user input.

Pivot table (Figure 4.6).

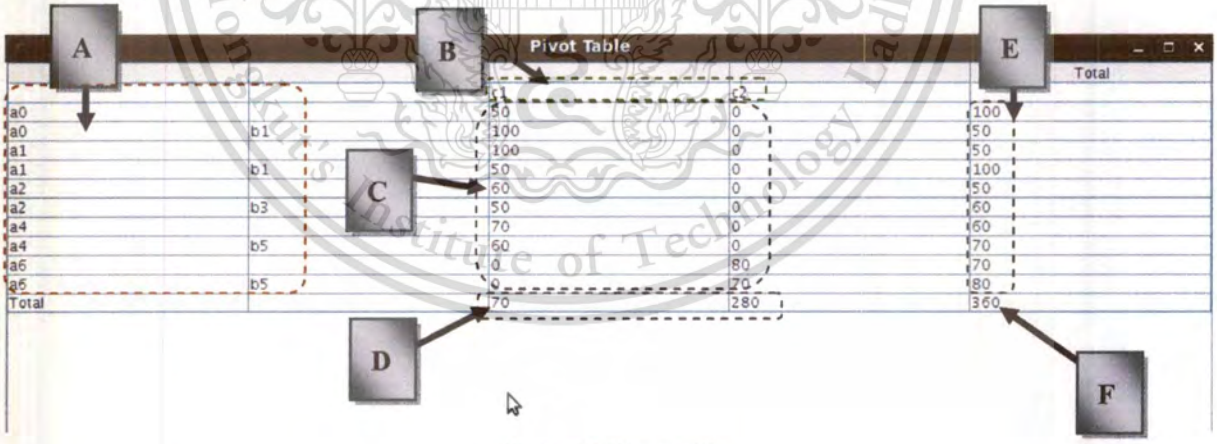


Figure 4.6 Pivot table

A: Row field – show every possible value for every dimension which user specified before in row Input field.

B: Column field - Show every possible value for every dimension which user specified before in column Input field.

C: Show measurement value.

D: Show total measurement value for column dimension values.

E: Show total measurement value for row dimension values.

F: Show total measurement value for ALL values.

Search specific value (Figure 4.7).

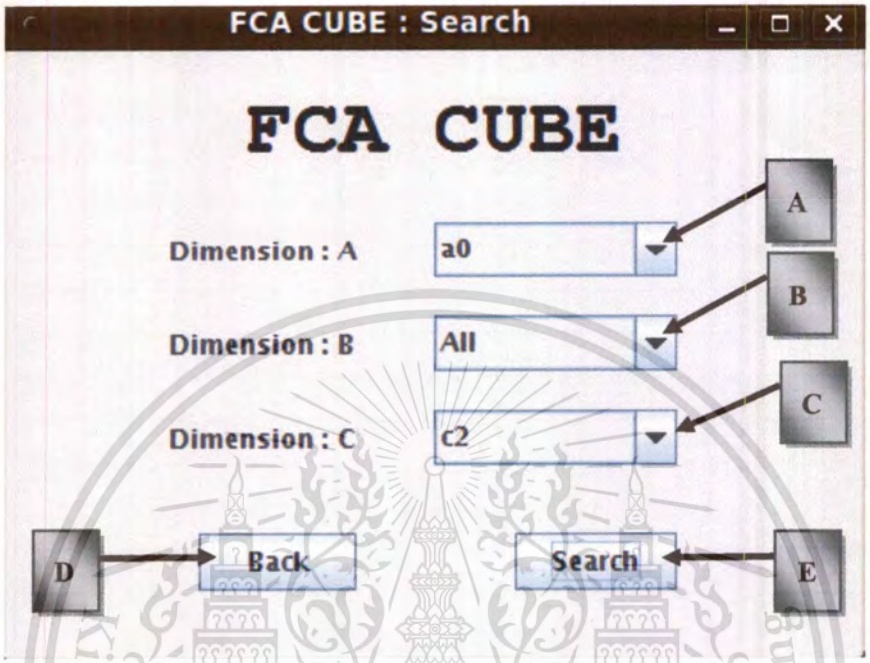


Figure 4.7 Search option

A: Drop down menu allow user to choose value from A dimension.

B: Drop down menu allow user to choose value from B dimension.

C: Drop down menu allow user to choose value from C dimension.

D: Button "Back" – Bring user to previous page (Page after choose data file).

E: Button "Search" – Search value which user choose and show result.

Search Result (Figure 4.8).

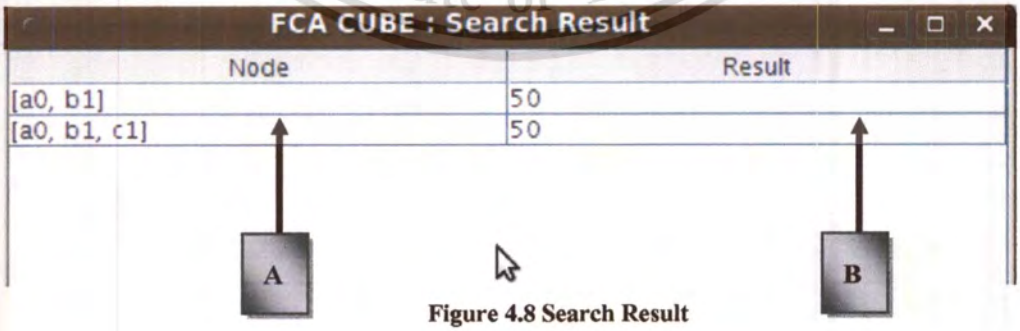


Figure 4.8 Search Result

A: Show all node that contain user's specified value (in this case a0, b1).

B: Show measurement value for each node.

Chapter 5

Conclusion and Recommendation

5.1 Conclusion

The objective of our project is to apply Formal Concept Analysis (FCA) in the field of data cube in order to compute data cube and produce the same result with the CUBE-BY operation and also implement the application of pivot table to prove that the result from the algorithm can be used in real world application.

We have implement the approach that constructs the data cube based on Formal Concept Analysis technique and also proposed the approach to retrieve the data from the data cube graph. The result shows that Formal Concept Analysis can be applied to use with data cube and can produce the same result as the original data cube. We also implemented a pivot table application to allow users to retrieve aggregate data the way he want without having to have any knowledge of SQL.

5.2 Recommendation

There are many areas where this project can be improved.

- This approach of computing data cube based on FCA is not fully pre-computed with the measurement value. As the project shows that we have to traverse in order to get the measurement value. The measurement value, therefore, should be computed during the construction of the graph and store in every node.
- Update graph algorithm for reducing computation time if there is any change.
- The pivot application should integrate the search feature.
- This algorithm has not been fully verified for stability when the data cube is more than 3 dimensional. (i.e. the scalability of the approach has not been proven.)
- The pivot application should be able to use with any database server.

References

[1] W.Wang, J. Feng, H. Lu, and J Xu Yu. Condense Cube: An effective approach to reducing data cube size.

[2] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotal.1996.

[3] D. Xin, J.Han, X. Li, and B. Wah. Star-cubing: Computing Iceberg Cubes by top-down and bottom-up integration.

[4] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis. Dwarf : Shrinking the PetaCube.

[5] Z. Shao, J. Han, D. Xin. MM-Cubing : Computing Iceberg Cubes by Factorizing the Lattice Space.

[6] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technology.

[7] L V.S. Lakshmanan, J.Pei, Y.Zhao. Efficious Data Cube Exploration by Semantic Summerization and Compression.

[8] T.Tilley. Formal Concept Analysis and Formal Methods. 2000.

[9] K. E. Wolff. A First Course in Formal Concept Analysis. Advances in Statistical Software 4, 429-438

[10] Y. Sismanis, A. Deligiannakis, Y. Kotidis, N. Roussopoulos. Hierarchical Dwarfs for the Rollup Cube.

[11] J. Ham, J. Pei, G. Dong, and K. Wang. Efficient Computation of Iceberg Cubes with Complex measures.

[12] D. Xin, J. Han, X. Li, B. W. Wah. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration.

[13] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures.

[14] http://en.wikipedia.org/wiki/Formal_concept_analysis

[15] http://prefuse.org/

[16] http://www.infovis-wiki.net/index.php/Prefuse

[17] http://www.cs.mun.ca/~hoeber/teaching/cs4767/notes/04-prefuse/

[18] http://java.sun.com/docs/books/tutorial/uiswing/TOC.html

Appendix

In this section, we give the overview about tools and technology used in this project include, PREFUSE, Netbean IDE.

1. PREFUSE

In this project, PREFUSE is used as the visualization of the graph presentation. In this section we give the overview of PREFUSE, the features of PREFUSE, and PREFUSE structure.

1.1 PREFUSE Overview

PREFUSE is an extensible software framework for helping software developers create interactive information visualization applications using the Java programming language. It can be used to build standalone applications, visual components embedded in larger applications, and web applets.

1.2 PREFUSE Features

- Table, Graph and Tree data structures supporting arbitrary data attributes, data indexing, and selection queries, all with an efficient memory footprint.
- Components for layout, color, size, and shape encodings, distortion techniques, animation, and more.
- A library of interaction controls for common interactive, direct-manipulation operations.
- Animation support through a general activity scheduling mechanism.
- View transformations supporting panning and zooming, including both geometric and semantic zooming.
- Dynamic queries for interactive filtering of data.
- Integrated text search using a number of available search engines.
- A physical force simulation engine for dynamic layout and animation.
- Flexibility for multiple views, including "overview+detail" and "small multiples" displays.
- A built in, SQL-like expression language for writing queries to PREFUSE data structures and creating derived data fields.
- Support for issuing queries to SQL databases and mapping query results into PREFUSE data structures.

- Simple, developer-friendly APIs for creating custom processing, interaction, and rendering components. PREFUSE's API can be found at http://prefuse.org/doc/api/

1.3 PREFUSE Structure

PREFUSE is designed based upon the information visualization reference model, a software architecture pattern that breaks up the visualization process into a series of discrete steps. The process is illustrated in the figure below.

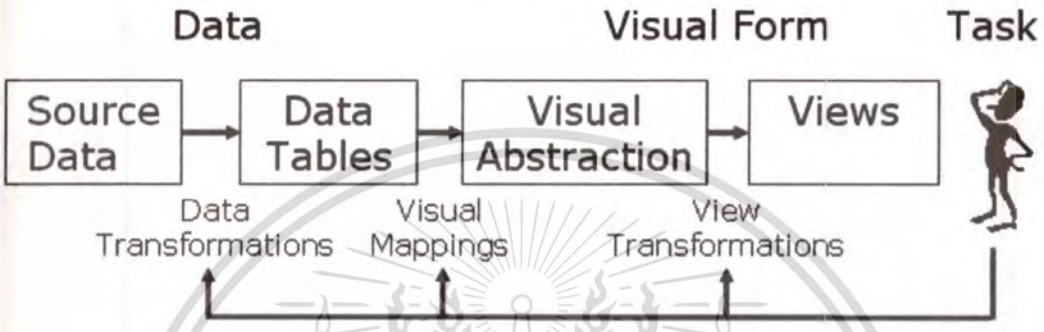


Figure A.1 PREFUSE process.

- The first step is the collection of the source data to visualize; it could be table, social network, file or any data set.
- Next, this source data is then used to construct data tables, internal representations of the data as it is to be visualized.
- After that, the data tables are then subject to visual mapping to create a visual abstraction, a data model that includes visual features such as spatial layout, color, size and shape.
- The actual rendering of the data in the visual abstraction is done through a process of view transformations. The rendering components draw the contents of the visual abstraction into any number of interactive views.
- User interacts with the visualization can feedback into this process, causing changes or updates at any stage of the visualization pipeline.

The process above is similarly to the model-view-controller design pattern

- A model containing backing data values
- One of more views displaying the contents of this model
- Controllers for processing user input and appropriately updating the model and view in response.

The figure below illustrates how the different packages and classes of the PREFUSE implement the information visualization reference model, providing support for each state of the visualization pipeline.

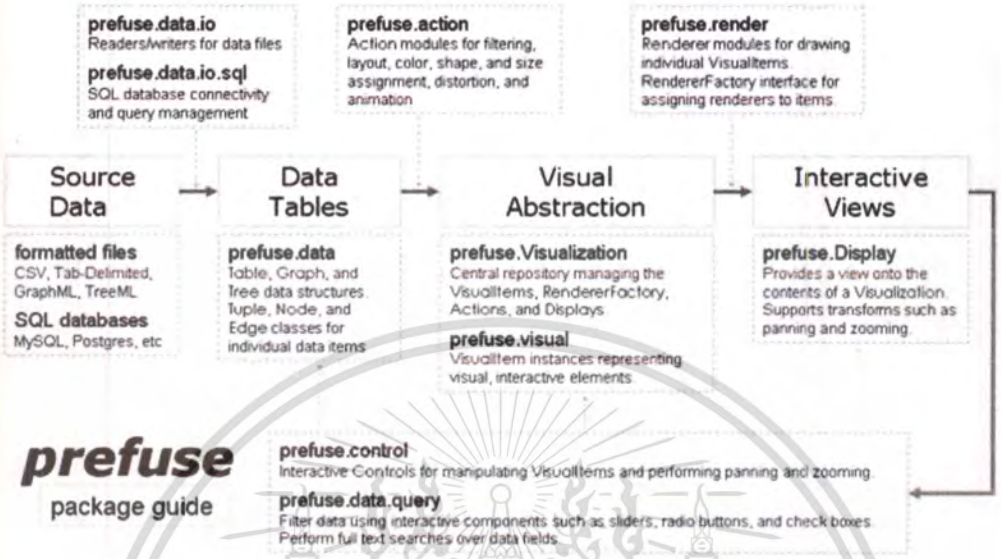


Figure A.2 PREFUSE package guide.

1.4 Example Application

In this section, we show step-by-step of how to construct the visualization by using PREFUSE.

1.4.1 Download and Install PREFUSE

PREFUSE can be downloaded at http://prefuse.org/. The file structure after download is shown below.

- build: complied classes and jars (once generated)
- data: demo data
- demos: demo applications
- doc: javaDoc files (one generated)
- lib: third-party libraries
- src: source code for PREFUSE
- test: JUnit tests

PREFUSE is a Eclipse project but can be import into Netbean project also by import Eclipse project into Netbean by.

Navigate to the Eclipse Project Importer under File→ Import Project →Eclipse Project

Figure A.3 Import Eclipse project into Netbean

1.4.2 Import into Netbeans

Select File → Import Project → Eclipse Project

Choose the Import Project Ignoring the Dependencies, input the project to import (Eclipse project) and the destination to be put in (for example, under Netbean folder) then click finish.

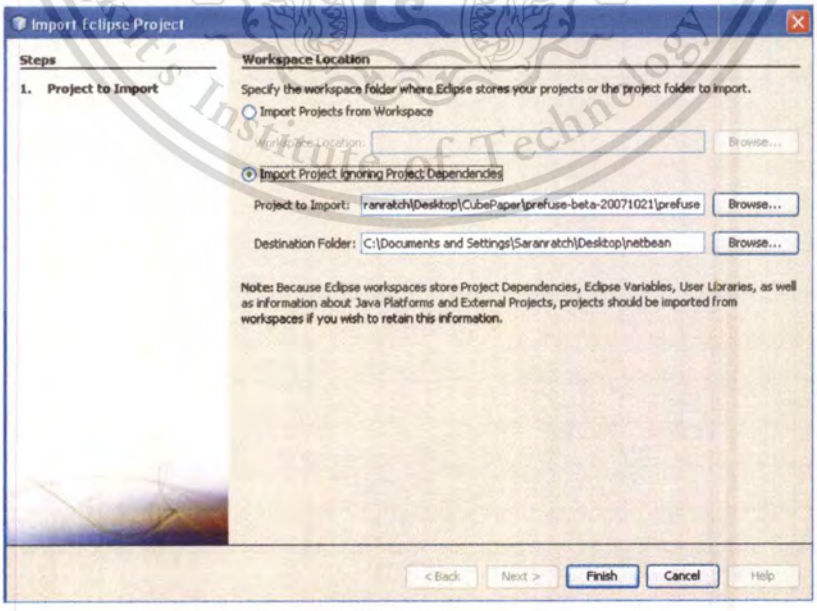


Figure A.4 Select PREFUSE file to be imported into Netbean

Or if you already have project, you can include PREFUSE project by including the jar file as the library. Under your project navigate through Libraries → Add Jar Folder.

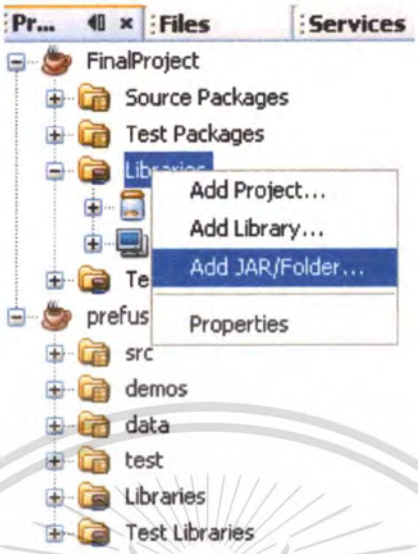


Figure A.5 Add PREFUSE as java library

Add prefuse.jar under dist folder in PREFUSE project (If dist folder do not exist, build the PREFUSE project once).

From this, we will use PREFUSE as the library in the project called "FinalProject".

1.4.3 Step-by-step constructing graph

Loading the data

The first step we need to download data into a PREFUSE graph. In our project we will download the data from the file called GraphXML.xml, the format of this file is an GraphML format. The GraphXML.xml data will be shown at the 2nd section of the appendix.

The following code below shows how to load the GraphML formatted GraphXML.xml file into a PREFUSE Graph data structure using GraphML Reader class.

```
Graph g = null;
try {
        g = new GraphMLReader().readGraph(datafile);
} catch ( Exception e ) {
        e.printStackTrace();
        System.exit(1); }
```

The visualization

Next we need to create a visual abstraction of the graph. First, create a new Visualization instance and add the graph data to it. The code below shows how to create visualization of the graph.

```
Visualization m_vis = new Visualization();
 m_vis.add("tree", g);
```

When Tree instance are added to the visualization, two other subgroups are automatically registered, onw for nedes, and one for edges.

The renderers and renderer factory

The next step is to set up the Renderers used to draw the VisualItems now contained in the Visualization. The code below shows how to set up the Renderers.

```
private LabelRenderer m_nodeRenderer;
 m_nodeRenderer = new LabelRenderer(m_label);
 //set renderer factory
 DefaultRendererFactory rf = new DefaultRendererFactory(m_nodeRenderer);
 m_vis.setRendererFactory(rf);
```

The processing actions

Next, we set up the visual encodings that are provided by creating Action modules that process the VisualItems in the Visualization. The code below shows how to create Action and ActionList instance that groups all the actions into a single executable unit.

```
// colors
 ItemAction nodeColor = new NodeColorAction(treeNodes);
 ItemAction textColor = new TextColorAction(treeNodes);
 m_vis.putAction("textColor", textColor);
 ItemAction edgeColor = new ColorAction(treeEdges,
 VisualItem.STROKECOLOR, ColorLib.rgb(200,200,200));
 FontAction fonts = new FontAction(treeNodes,
 FontLib.getFont("Tahoma", 20));
 fonts.add("ingroup('_focus_')", FontLib.getFont("Tahoma", 20));
// recolor
 ActionList recolor = new ActionList();
```

```
recolor.add(nodeColor);
recolor.add(textColor);
```

Then we add color into the visualization.

```
m_vis.putAction("recolor", recolor);
```

The display and interactive controls

Now, we need to create the display for the graph/tree visualization.

```
// initialize the display
setSize(600,600);
setItemSorter(new TreeDepthItemSorter());
addControlListener(new DragControl()); //drag item around
addControlListener(new ZoomToFitControl()); // zoom item
addControlListener(new ZoomControl()); // zoom in zoom out
addControlListener(new PanControl()); // pan with background
addControlListener(new FocusControl(1, "filter"));
addControlListener(new HoverActionControl("repaint"));
```

Launching the application

Finally, last step we need to actually display what we have built. We use JFrame to display out application. The final code is below.

```
JFrame frame = new JFrame("p r e f u s e | r a d i a l g r a p h v i e w");
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
frame.setContentPane(demo(infile, label));
frame.pack();
frame.setVisible(true);
```

Now we have finished constructing the graph using PREFUSE and the result is shown below.

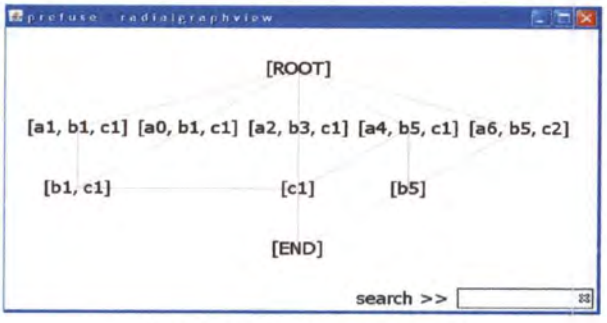


Figure A.6 The complete graph.

1.5 Codes

In this section we will give all the code that is used to display the graph.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.MouseEvent;
import java.util.Iterator;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import prefuse.Constants;
import prefuse.Display;
import prefuse.Visualization;
import prefuse.action.ActionList;
import prefuse.action.GroupAction;
import prefuse.action.ItemAction;
import prefuse.action.RepaintAction;
import prefuse.action.animate.ColorAnimator;
import prefuse.action.animate.PolarLocationAnimator;
import prefuse.action.animate.QualityControlAnimator;
import prefuse.action.animate.VisibilityAnimator;
import prefuse.action.assignment.ColorAction;
import prefuse.action.assignment.FontAction;
import prefuse.action.layout.CollapsedSubtreeLayout;
import prefuse.action.layout.graph.NodeLinkTreeLayout;
import prefuse.activity.SlowInSlowOutPacer;
import prefuse.controls.ControlAdapter;
```

```
import prefuse.controls.DragControl;
import prefuse.controls.FocusControl;
import prefuse.controls.HoverActionControl;
import prefuse.controls.PanControl;
import prefuse.controls.ZoomControl;
import prefuse.controls.ZoomToFitControl;
import prefuse.data.Graph;
import prefuse.data.Node;
import prefuse.data.Table;
import prefuse.data.Tuple;
import prefuse.data.event.TupleSetListener;
import prefuse.data.io.GraphMLReader;
import prefuse.data.query.SearchQueryBinding;
import prefuse.data.search.PrefixSearchTupleSet;
import prefuse.data.search.SearchTupleSet;
import prefuse.data.tuple.DefaultTupleSet;
import prefuse.data.tuple.TupleSet;
import prefuse.demos.TreeMap.NodeRenderer;
import prefuse.render.AbstractShapeRenderer;
import prefuse.render.DefaultRendererFactory;
import prefuse.render.EdgeRenderer;
import prefuse.render.LabelRenderer;
import prefuse.util.ColorLib;
import prefuse.util.FontLib;
import prefuse.util.ui.JFastLabel;
import prefuse.util.ui.JSearchPanel;
import prefuse.util.ui.UILib;
import prefuse.visual.VisualItem;
import prefuse.visual.expression.InGroupPredicate;
import prefuse.visual.sort.TreeDepthItemSorter;
```

```
/**Demonstration of a node-link tree viewer
public class RadialGraphView extends Display {
    public RadialGraphView(){
    }
    public static final String DATA_FILE = "GraphXML.xml";
    private static final String tree = "tree";
    private static final String treeNodes = "tree.nodes";
    private static final String treeEdges = "tree.edges";
    private static final String linear = "linear";
    private LabelRenderer m_nodeRenderer;
    private EdgeRenderer m_edgeRenderer;
    private NodeRenderer node;
     private String m_label = "label";
     private int m_orientation = Constants.ORIENT_TOP_BOTTOM;
     public RadialGraphView(Graph g, String label) {
        super(new Visualization());
        m_label = label;
        // -- set up visualization --
        m_vis.add(tree, g);
        m_vis.setInteractive(treeEdges, null, false);
        // -- set up renderers --
        m_nodeRenderer = new LabelRenderer(m_label);
        m_nodeRenderer.setRenderType(
                                AbstractShapeRenderer.
RENDER_TYPE_DRAW_AND_FILL);
        m_nodeRenderer.setHorizontalAlignment(Constants.CENTER);
        m_nodeRenderer.setRoundedCorner(10,10);
        m_edgeRenderer = new EdgeRenderer();
        DefaultRendererFactory rf = new
DefaultRendererFactory(m_nodeRenderer);
        rf.add(new InGroupPredicate(treeEdges), m_edgeRenderer);
```

```
m_vis.setRendererFactory(rf);
// -- set up processing actions --
   // colors
ItemAction nodeColor = new NodeColorAction(treeNodes);
ItemAction textColor = new TextColorAction(treeNodes);
m_vis.putAction("textColor", textColor);
ItemAction edgeColor = new ColorAction(treeEdges,
VisualItem.STROKECOLOR, ColorLib.rgb(200,200,200));
FontAction fonts = new FontAction(treeNodes,
FontLib.getFont("Tahoma", 20));
fonts.add("ingroup('_focus_')", FontLib.getFont("Tahoma", 20));
   // recolor
ActionList recolor = new ActionList();
recolor.add(nodeColor);
recolor.add(textColor);
m_vis.putAction("recolor", recolor);
   // repaint
ActionList repaint = new ActionList();
repaint.add(recolor);
repaint.add(new RepaintAction());
m_vis.putAction("repaint", repaint);
   // animate paint change
ActionList animatePaint = new ActionList(400);
animatePaint.add(new ColorAnimator(treeNodes));
animatePaint.add(new RepaintAction());
m_vis.putAction("animatePaint", animatePaint);
   // create the tree layout action
NodeLinkTreeLayout treeLayout = new  NodeLinkTreeLayout(tree);
   //treeLayout.setAngularBounds(-Math.PI/2, Math.PI);
m_vis.putAction("treeLayout", treeLayout);
CollapsedSubtreeLayout subLayout = new
```

```
CollapsedSubtreeLayout(tree);
        m_vis.putAction("subLayout", subLayout);
           // create the filtering and layout
        ActionList filter = new ActionList();
        filter.add(new TreeRootAction(tree));
        filter.add(fonts);
        filter.add(treeLayout);
        filter.add(subLayout);
        filter.add(textColor);
        filter.add(nodeColor);
        filter.add(edgeColor);
        m_vis.putAction("filter", filter);
           // animated transition
        ActionList animate = new ActionList(1250);
        animate.setPacingFunction(new SlowInSlowOutPacer());
        animate.add(new QualityControlAnimator());
        animate.add(new VisibilityAnimator(tree));
        animate.add(new PolarLocationAnimator(treeNodes, linear));
        animate.add(new ColorAnimator(treeNodes));
        animate.add(new RepaintAction());
        m_vis.putAction("animate", animate);
        m_vis.alwaysRunAfter("filter", "animate");
        // ------------------------------------------------
        // initialize the display
        setSize(600,600);
        setItemSorter(new TreeDepthItemSorter());
        addControlListener(new DragControl());
        addControlListener(new ZoomToFitControl());
        addControlListener(new ZoomControl());
        addControlListener(new PanControl());
        addControlListener(new FocusControl(1, "filter"));
```

```
        addControlListener(new HoverActionControl("repaint"));
        // --------------------------------------------------
        // filter graph and perform layout
        m_vis.run("filter");
        // maintain a set of items that should be interpolated linearly
        // this isn't absolutely necessary, but makes the animations nicer
        // the PolarLocationAnimator should read this set and act accordingly
        m_vis.addFocusGroup(linear, new DefaultTupleSet());

m_vis.getGroup(Visualization.FOCUS_ITEMS).addTupleSetListener(
          new TupleSetListener() {
            public void tupleSetChanged(TupleSet t, Tuple[] add, Tuple[]
rem) {
              TupleSet linearInterp = m_vis.getGroup(linear);
              if ( add.length < 1 ) return; linearInterp.clear();
              for ( Node n = (Node)add[0]; n!=null; n=n.getParent() )
                linearInterp.addTuple(n);
            }
          }
        );
        SearchTupleSet search = new PrefixSearchTupleSet();
        m_vis.addFocusGroup(Visualization.SEARCH_ITEMS, search);
        search.addTupleSetListener(new TupleSetListener() {
          public void tupleSetChanged(TupleSet t, Tuple[] add, Tuple[] rem)
{
            m_vis.cancel("animatePaint");
            m_vis.run("recolor");
            m_vis.run("animatePaint");
          }
        });
      }
```

```
//--------------------------------------------------------------------
public void setOrientation(int orientation) {
    NodeLinkTreeLayout rtl
        = (NodeLinkTreeLayout)m_vis.getAction("treeLayout");
    CollapsedSubtreeLayout stl
        = (CollapsedSubtreeLayout)m_vis.getAction("subLayout");
    switch ( orientation ) {
        case Constants.ORIENT_LEFT_RIGHT:
            m_nodeRenderer.setHorizontalAlignment(Constants.LEFT);
            m_edgeRenderer.setHorizontalAlignment1(Constants.RIGHT);
            m_edgeRenderer.setHorizontalAlignment2(Constants.LEFT);
            m_edgeRenderer.setVerticalAlignment1(Constants.CENTER);
            m_edgeRenderer.setVerticalAlignment2(Constants.CENTER);
            break;
        case Constants.ORIENT_RIGHT_LEFT:
            m_nodeRenderer.setHorizontalAlignment(Constants.RIGHT);
            m_edgeRenderer.setHorizontalAlignment1(Constants.LEFT);
            m_edgeRenderer.setHorizontalAlignment2(Constants.RIGHT);
            m_edgeRenderer.setVerticalAlignment1(Constants.CENTER);
            m_edgeRenderer.setVerticalAlignment2(Constants.CENTER);
            break;
        case Constants.ORIENT_TOP_BOTTOM:
            m_nodeRenderer.setHorizontalAlignment(Constants.CENTER);
            m_edgeRenderer.setHorizontalAlignment1(Constants.CENTER);
            m_edgeRenderer.setHorizontalAlignment2(Constants.CENTER);
            m_edgeRenderer.setVerticalAlignment1(Constants.BOTTOM);
            m_edgeRenderer.setVerticalAlignment2(Constants.TOP);
            break;
        case Constants.ORIENT_BOTTOM_TOP:
            m_nodeRenderer.setHorizontalAlignment(Constants.CENTER);
            m_edgeRenderer.setHorizontalAlignment1(Constants.CENTER);
```

```
            m_edgeRenderer.setHorizontalAlignment2(Constants.CENTER);
            m_edgeRenderer.setVerticalAlignment1(Constants.TOP);
            m_edgeRenderer.setVerticalAlignment2(Constants.BOTTOM);
            break;
        default:
            throw new IllegalArgumentException(
              "Unrecognized orientation value: "+orientation);
    }
    m_orientation = orientation;
    rtl.setOrientation(orientation);
    stl.setOrientation(orientation);
  }
  public int getOrientation() {
    return m_orientation;
  }
  // ------------------------------------------------------------------------
  public static void main(String argv[]) {
    String infile = DATA_FILE;
    String label = "name";
    if ( argv.length > 1 ) {
      infile = argv[0];
      label = argv[1];
    }
    UILib.setPlatformLookAndFeel();
    JFrame frame = new JFrame("p r e f u s e | r a d i a l g r a p h v i e
w");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setContentPane(demo(infile, label));
    frame.pack();
    frame.setVisible(true);
  }
```

```
public static JPanel demo() {
   return demo(DATA_FILE, "name");
}
public static JPanel demo(String datafile, final String label) {
    Graph g = null;
   public static JPanel demo(String datafile, final String label) {
      Graph g = null;
      try {
         g = new GraphMLReader().readGraph(datafile);
      } catch ( Exception e ) {
         e.printStackTrace();
         System.exit(1);
      }
      System.out.println(g);
      return demo(g, label);
   }
   public static JPanel demo(Graph g, final String label) {
      // create a new radial tree view
      final RadialGraphView gview = new RadialGraphView(g, label);
      Visualization vis = gview.getVisualization();
      // create a search panel for the tree map
      SearchQueryBinding sq = new SearchQueryBinding(
         (Table)vis.getGroup(treeNodes), label,
        (SearchTupleSet)vis.getGroup(Visualization.SEARCH_ITEMS));
      JSearchPanel search = sq.createSearchPanel();
      search.setShowResultCount(true);
      search.setBorder(BorderFactory.createEmptyBorder(5,5,4,0));
      search.setFont(FontLib.getFont("Tahoma", Font.PLAIN, 20));
      final JFastLabel title = new JFastLabel("                ");
      title.setPreferredSize(new Dimension(350, 20));
      title.setVerticalAlignment(SwingConstants.BOTTOM);
```

```
        title.setBorder(BorderFactory.createEmptyBorder(3,0,0,0));
        title.setFont(FontLib.getFont("Tahoma", Font.PLAIN, 20));
        gview.addControlListener(new ControlAdapter() {
            public void itemEntered(VisualItem item, MouseEvent e) {
                if ( item.canGetString(label) )
                    title.setText(item.getString(label));
            }
            public void itemExited(VisualItem item, MouseEvent e) {
                title.setText(null);
            }
        });
        Box box = new Box(BoxLayout.X_AXIS);
        box.add(Box.createHorizontalStrut(10));
        box.add(title);
        box.add(Box.createHorizontalGlue());
        box.add(search);
        box.add(Box.createHorizontalStrut(3));

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(gview, BorderLayout.CENTER);
        panel.add(box, BorderLayout.SOUTH);
        Color BACKGROUND = Color.WHITE;
        Color FOREGROUND = Color.DARK_GRAY;
        UILib.setColor(panel, BACKGROUND, FOREGROUND);
        return panel;
    }
    // ------------------------------------------------------------------------
    /**
     * Switch the root of the tree by requesting a new spanning tree
     * at the desired root
    */
```

```
    public static class TreeRootAction extends GroupAction {
        public TreeRootAction(String graphGroup) {
            super(graphGroup);
        }
        public void run(double frac) {
            TupleSet focus =
m_vis.getGroup(Visualization.FOCUS_ITEMS);
            if ( focus==null || focus.getTupleCount() == 0 ) return;
            Graph g = (Graph)m_vis.getGroup(m_group);
            Node f = null;
            Iterator tuples = focus.tuples();
            while (tuples.hasNext() &&
!g.containsTuple(f=(Node)tuples.next()))
            {   f = null;   }
            if ( f == null ) return;
            g.getSpanningTree(f);
        }
    }
    /**
     * Set node fill colors
    */
    public static class NodeColorAction extends ColorAction {
        public NodeColorAction(String group) {
            super(group, VisualItem.FILLCOLOR, ColorLib.rgba(255, 255,
0,0));
            add("_hover", ColorLib.gray(220,230));
            add("ingroup('_search_')", ColorLib.rgb(255,190,190));
            add("ingroup('_focus_')", ColorLib.rgb(198,229,229));
        }
    }
    /**
```

```
        * Set node text colors
        */
        public static class TextColorAction extends ColorAction {
            public TextColorAction(String group) {
                super(group, VisualItem.TEXTCOLOR, ColorLib.gray(0));
                add("_hover", ColorLib.rgb(255,0,0));
            }
        }
    } // end of inner class JPanel demo
} // end of class RadialGraphView
```

2. GraphXML.xml

In this section, we shows the data in the GraphML format in the file GraphXML.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <graph edgedefault="undirected">
    <key attr.name="name" attr.type="String" for="node" id="name" />
    <key attr.name="measure" attr.type="String" for="node" id="measure" />
    <node id="[ROOT]">
      <data key="name">[ROOT]</data>
      <data key="measure">0</data>
    </node>
    <node id="[c1]">
      <data key="name">[c1]</data>
      <data key="measure">0</data>
    </node>
    <node id="[a1, b1, c1]">
      <data key="name">[a1, b1, c1]</data>
      <data key="measure">100</data>
    </node>
    <node id="[a0, b1, c1]">
      <data key="name">[a0, b1, c1]</data>
```

```
        <data key="measure">50</data>
      </node>
      <node id="[a2, b3, c1]">
        <data key="name">[a2, b3, c1]</data>
        <data key="measure">60</data>
      </node>
      <node id="[b5]">
        <data key="name">[b5]</data>
        <data key="measure">0</data>
      </node>
      <node id="[a4, b5, c1]">
        <data key="name">[a4, b5, c1]</data>
        <data key="measure">70</data>
      </node>
      <node id="[a6, b5, c2]">
        <data key="name">[a6, b5, c2]</data>
        <data key="measure">80</data>
      </node>
      <node id="[b1, c1]">
        <data key="name">[b1, c1]</data>
        <data key="measure">0</data>
      </node>
      <node id="[END]">
        <data key="name">[END]</data>
        <data key="measure">0</data>
      </node>
  <edge source="[c1]" target="[END]" />
  <edge source="[a1, b1, c1]" target="[b1, c1]" />
  <edge source="[ROOT]" target="[a1, b1, c1]" />
  <edge source="[ROOT]" target="[a0, b1, c1]" />
  <edge source="[ROOT]" target="[a2, b3, c1]" />
```

```
<edge source="[ROOT]" target="[a4, b5, c1]" />
<edge source="[ROOT]" target="[a6, b5, c2]" />
<edge source="[a0, b1, c1]" target="[b1, c1]" />
<edge source="[a2, b3, c1]" target="[c1]" />
<edge source="[b5]" target="[END]" />
<edge source="[a4, b5, c1]" target="[c1]" />
<edge source="[a4, b5, c1]" target="[b5]" />
<edge source="[a6, b5, c2]" target="[b5]" />
<edge source="[b1, c1]" target="[c1]" />
</graph> </graphml>
```