Spring 1989

A Classification Approach for Automated Reasoning Systems--A Case Study in Graph Theory

Rong Lin
Old Dominion University

A CLASSIFICATION APPROACH FOR AUTOMATED REASONING SYSTEMS – A CASE STUDY IN GRAPH THEORY

By

Rong Lin

A THESIS
SUBMITTED TO THE FACULTY OF
OLD DOMINION UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

Norfolk, Virginia
April, 1989

Approved by:

Shunichi Toida (Director)

ABSTRACT

Reasoning systems which create classifications of structured objects face the problem of how object descriptions can be used to reflect their components as well as relations among these components. Current reasoning systems on graph theory do not adequately provide models to discover complex relations among mathematical concepts (eg: relations involving subgraphs) mainly due to the inability to solve this problem. This thesis presents an approach to construct a knowledge-based system, GC (Graph Classification), which overcomes this difficulty in performing automated reasoning in graph theory. We describe graph concepts based on an attribute called Linear Recursive Constructivity (LRC) . LRC defines classes by an algebraic formula supported by background knowledge of graph types. We use subsumption checking on decomposed algebraic expressions of graph classes as a major proof method. The search is guided by case-split-based inferencing. Using the approach GC has generated proofs for many theorems such as "any two distinct cycles (closed paths) having a common edge e contain a cycle not traversing e", "if cycle C1 contains edges e1, e2, and cycle C2 contains edges e2, e3, then there exists a cycle that contains e1 and e3" and "the union of a tree and a path is a tree if they have only a single common vertex".

The main contributions of this thesis are:

1. Development of a classification-based knowledge representation and a reasoning approach for graph concepts, thus providing a simple model for structured mathematical objects.

2. Development of an algebraic theory for simplifying and decomposing graph concepts.

3. Development of a proof search and a case-splitting technique with the guidance of graph type knowledge.

4. Development of a proving mechanism that can generate constructive proofs by manipulating only simple linear formalization of theorems.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to Professor Shunichi Toida (my thesis advisor) for his guidance and valuable help during the course of this research. I also wish to thank Professor Stephen Olariu, Professor J. Christian Wild, Eminent Professor Kurt J. Maly (Department Chairman) and Professor Ravi Mukkamala for constructive suggestions, comments and grateful supports. Special thanks are due Mr. Bill Bright for his help in editing the text.

Contents

1 Introduction 1

1.1 The Problems of Theorem Proving 2

1.2 The Problems in Knowledge-Based Reasoning 4

1.3 A Solution: The Approach of The Thesis 7

1.4 The Organization of The Thesis 9

2 Related Previous Works 11

2.1 Perspective on Theorem Proving 12

2.1.1 Recursive Function Theory - BMTP 14

2.1.2 The Resolution Proving Technique: ITP 17

2.1.3 Natural Deduction: IMPLY 22

2.2 Perspective on Knowledge-Based Reasoning 25

2.2.1 Knowledge-Based Automated Equation Solving: DReam 27

2.2.2 Automated Reasoning in Graph Theory: GT 28

2.2.3 Man-Machine Graph Theory Proving: GRAPH 31

3 Algebraic Knowledge Formalization 33

3.1 The Syntax and Semantics of Atomic Expression 33

3.2 The Primitive Algebraic Operations 35

3.3 The Algebraic Properties 36

4 **Procedural Knowledge Formalization** 41

4.1 Recursive Decomposition of Graph Class 41

4.2 Procedural Knowledge of Graph Types 42

4.2.1 Notations and Definitions 43

4.2.2 T-procedures 45

4.3 The Features and The Correctness of T-Procedures 59

4.3.1 A Proof for Cycle-Type T-Procedure 60

4.3.2 A Proof for Tree-Type T-Procedure 61

5 **Proof Process** 63

5.1 Background Knowledge Formalization 63

5.1.1 Conceptual Frames 63

5.1.2 Representation about Constraints 64

5.2 Representation of Theorems 69

5.3 Proof of Existence 71

5.3.1 Subsumption Primitives 71

5.3.2 Goal-Assertion Subsumption Method 72

5.4 Leveled Methods 75

5.4.1 Level 1 75

5.4.2 Level 2 75

5.4.3 Level 3 76

5.4.4 Level 4 76

6 Proof Search: Case-Split-Based Inferencing 77

6.1 Inference Rule and Case Splitting 77

6.2 Inference Rule Set 78

6.3 Soundness of The Rules and Equivalent Substitutions 82

6.4 Induction Formalization 82

7 The Implementation of System GC 84

7.1 Overview of The System's Modules 84

7.1.1 Module 1 : The Symbolic Processor 84

7.1.2 Module 2 : The Algebraic Processor 85

7.1.3 Module 3 : The Candidate Generator 85

7.1.4 Module 4 : The Proof Engine or Local Processor 85

7.1.5 Module 5 : Global Processor to Perform Tree Expansion 85

7.1.6 Module 6 : The Control Unit 86

7.2 Descriptions of The System GC 86

7.2.1 The Syntax of The Graph Language Subset 86

7.2.2 The Diagrams of The Software Modules 87

7.2.3 The File Structures of GC 95

7.2.4 Example Proofs 95

7.3 Discussion and Future Implementation Plan 100

8 Examples and Some Results 101

8.1 Examples of Proving Theorem and Verifying Types 101

8.2 Typical Theorems Proved 107

8.3 Examples of Induction Proving 109

8.4 A Result on The Tree Representation 113

8.4.1 Level Sequence Representation of Trees 113

8.4.2 Canonical Tree Algorithm 114

9 Discussions And Conclusions 117

9.1 Discussions of The Approach 117

9.1.1 Why The Approach Is Useful 117

9.1.2 The Proof Process of The Approach Is Simple 118

9.1.3 The Conceptual Formalization Is General 118

9.2 Conclusions 122

References 125

CHAPTER 1

Introduction

This thesis is concerned with a principle to accomplish automated reasoning on basic concepts of graph theory through using knowledge-based conceptual classification, a powerful artificial intelligence approach. In this dissertation the following topics are addressed:

1. Representing and manipulating knowledge based on conceptual classification in terms of both an algebraic theory and a recursive procedural theory.

2. Mechanisms for determining relations among basic graph concepts using multiple non-resolution theorem proving methods.

3. Developing a model to be used in proving theorems of existence.

4. Simulating human processes on constructive proofs.

5. Simulating human processes on case-split-based reasoning.

6. Modeling completeness and soundness of reasoning on graph concepts.

7. Modeling an automated reasoning system, GC (Graph Classification), based on the classification approach.

The main contributions of this thesis are:

1. Development of a classification-based knowledge representation and a reasoning approach for graph concepts, thus providing a simple model for structured mathematical objects.

2. Development of an algebraic theory for simplifying and decomposing graph concepts.

3. Development of a proof search and a case-splitting technique with the guidance of graph type knowledge.

4. Development of a proving mechanism that can generate constructive proofs by manipulating only simple linear formalization of theorems.

The motivation of this research stems from the following fact: Though some excellent systems are able to conquer certain types of reasoning problems in the fields where first order predicate calculus can be used as the major representation scheme, the current theorem proving techniques do not provide an efficient approach to prove existence. This is because they do not have general mechanisms to adequately support constructive proofs [Bu1] [Ep]. Proving existence, however, plays a very important role in many mathematical fields such as graph theory. This thesis identifies the fundamental needs to produce constructive proofs on existence and develops mechanisms to provide the required capabilities.

The next two sections introduce the problems related to both theorem provers and knowledge-based reasoning systems. And then there follows an overview of the solutions to these problems with emphasis on formalization of knowledge and modeling of reasoning.

1.1. The Problems of Theorem Proving

Automated Theorem Proving (ATP) has been a major research endeavor in AI since the 1950's. There are basically two distinct approaches for an ATP design. One is called the logic (or resolution) approach, the other is referred to as the human-oriented (or nonresolution [Bl]) approach.

One of the main features of the resolution approach is that the control heuristics are suggested based solely on theorem syntax. Therefore it has a relatively simple structure. However, the proving process of the approach can be extremely time consuming due to the combinational explosion involved in choosing clauses for applying the resolution rule. By the early 70's most of ATP researchers have believed that a resolution type system could not go beyond proving simple theorems without extensive change in philosophy. Therefore the major interests in ATP research were turned to the human-oriented approach. The techniques emphasized in this new approach include the followings: (for detail see chapter 2)

- Knowledge base
- Natural deduction
- Reduction
- Typing
- Procedure
- Examples
- Models and counterexamples
- Analogy
- Learning
- Algebraic simplification
- Man-machine interaction

This human-oriented approach has turned out to be more efficient in many aspects. The main reason seems to be that it simulates various kinds of human proving techniques for clever use of knowledge, thus avoiding many unnecessary searches. In the design of a powerful system, a combination of these two approaches is often used [Lol] [BL1] [Bu1]. Generally, a human-oriented

ATP uses more than one inference rule and the search heuristics employed are not solely syntax directed. As a consequence, the structure of the method is more complex.

Though progress has been made since non-resolution approaches were used [Ne1] [Ne2] [Ne3] [BM1] [BT1], the main goal (the ability to compete with human provers) is still far from being reached. It has been recognized recently that an intelligent reasoning system should not only stress general purpose heuristic search techniques (that are central concerns of a non-resolution theorem prover), but also stress the need for the availability of expert knowledge in the system along with associated knowledge handling facilities [MJJ]. Thus developing an approach which combines theorem proving methodology with knowledge based reasoning techniques is the central theme of this dissertation.

1.2. The Problems in Knowledge-Based Reasoning

During the late 1970's and early 80's, knowledge-based automated reasoning was identified, invented and evolved. Automated reasoning systems are used in many research areas such as mathematics and circuit design. The main emphasis of knowledge-based reasoning depends more on intelligently using knowledge (or expert knowledge) than on using various deductive inference methods. Variety of programs are available and some significant systems are named below:

- EMYCIN is used for chemical synthesis[Va],
- PROSPECTOR for geology[Re],
- MYCIN for medical diagnosis [Sh],
- AURA for mathematical research, circuit design and circuit validation [Wo].

Even with much progress that has been made, many problems still exist. Wos [Wo] has said that: it is difficult to determine what problem need to be solved what available means are appreciable to a particular problem. Many questions on the representation of knowledge remain unanswered. The principal problem is how to properly deal with the interactions between representation, inference rule and strategy [Wo]. Researches focusing on the problem using conceptual classification have caused increasing interests in mathematics and various application fields.

One successful program in elementary mathematical discovery is Lenat's AM [Le], which builds up complex classes of concepts from more primitive concepts in step by step fashion [CF2]. AM defines (in terms of frame) concepts of classes, investigates examples of classes, conjectures and discovers the relations among classes, and refines concepts about classes.

Many other works related to conceptual classification in problem solving (see Clancey [Cl]), knowledge organization (see Hanson [HB]), and machine learning (see Michalsky [MS]) have shown that it is a very important approach to reduce search space, maximize inferencing ability and increase the chance of discovering and proving.

A recently published work applying this approach on discovering mathematical theorems in graph theory is Epstein's GT [Ep]. GT semantically encodes the domain specific knowledge into conceptual definitions of classes. GT uses an algebraic formula (called p-generator) to model many benchmark graph properties. GT's discovery and proof are based on the ability of examining relations among mathematical concepts.

The main shortcoming of GT's approach is that no components for a class can be directly represented. This greatly restricts its proving ability, for instance, the relations involving subgraphs can not be explored by GT; but clearly, reasoning on subgraphs plays an important role in proving graph theory theorems. Another limitation of GT is that its formulation of a concept has only a very loose connection to the previously built up concepts (described in a slot of the conceptual frame but not in algebraic formulas). To develop formulation that is able to be manipulated on such connections, however, is one of the main features of human mathematical research. For example, formulation of the concept of cycle should directly relate to the concept of path.

In order to overcome the limitations and shortcomings discussed in the above two sections and to provide efficient reasoning models for mathematical objects such as graph theory, certain kind of changes in the philosophy of representing knowledge and manipulating concepts are necessary. The following outlines such an endeavor done and a solution obtained by the thesis research.

1.3. A Solution: The Approach of The Thesis

It has been recognized [SM] that for the creation of classifications of structured objects the description of such objects must involve not only attributes of objects as a whole but also attributes of object components and relationships among these components. In order to obtain a knowledge representation scheme equipped with such a feature, the research has analyzed widely ranging tools from current techniques available [AE] and has resulted in a combination of algebraic and procedural representation methods for the knowledge in the domain of graph theory. An important graph attribute called Linear Recursive Constructivity (LRC) is identified, which can be stated as: graphs (the discussion is restricted to simple and undirected graphs) with a given vertex set and a property can always be constructed by applying a specified recursive procedure on all sequences of these vertices (e.g. first generate all graphs for the vertex set, and then through property checking remove those of the generated graphs which do not have the property).

The value of LRC is determined by giving an algebraic formula implicitly associated with the background knowledge of graph types. The formula is called a class, and has the form of $< S, T >$. It consists of three components: (1) S (called v-list), a list of vertices and unknowns (each unknown is a fixed but unknown list of vertices); (2) T (a type), which is associated with a recursive procedure called T-procedure; (3) constraints, which may be embedded in the formula to provide further descriptions for certain vertices, unknowns, and the entile class. Graphs can be generated by running a T-procedure on input S. If a generated graph satisfies the corresponding constraints, it is said to have the value of attribute LRC given by the formula, thus the formula defines a graph class. One of the important tasks in building GC is to provide sufficient number of such procedures that construct the desired classes but do not generate too many unsatisfied candidates.

In this research the features of such conceptual classification have been fully studied and utilized to develop an effective mechanism to discover the relations between graph concepts. Relating to the class formula, an algebra called C-algebra has been developed, which is able to manipulate a set of graphs thus can be used for simplifying and normalizing class expressions. The knowledge representation scheme includes both key mathematical concepts (types) descriptions and their subset descriptions (classes). A Complex graph concept (a class) can be transformed by runing the corresponding T-procedures into a C-algebraic expression composed of more primitive concepts (classes having simpler types).

Based on the scheme an automated reasoning system called Graph Classification (GC) has been constructed (partially implemented). GC organizes its proof as an and/or proof tree search using only linear operations of formulation. The system not only can manipulate widely ranging mathematical concepts (as factual input) but also can apply various effective methods in proving processes such as decomposition, subsumption, decision procedure and mathematical induction formulation.

Expansion the and/or proof tree rooted with an original theorem generate subtheorems which have the same goal as that of the original theorem but each of which has a simpler assertion, thus generally simplifying a constructive proof. Furthermore, an unproved subtheorem can always be regarded as a newly generated lemma and might be proved by some other system. A conversion between the class representation and the predicate logic representation has been formalized in the research, this provides the classification approach with the properties of theoretical completeness and soundness.

Normally, the proving processes in each node of the and/or tree are leveled like those used in Boyer-Moore theorem prover [BM].

The system uses an abstract representation (as a component of a class) for an unknown part of a graph concept. Its semantic meaning is provided by the background knowledge of graph type. The use of such an abstraction not only greatly reduces the search space in deduction process, but also provides well controlled case-splitting based reasoning through using various kinds of inference rules indexed by the types and unknowns during proof tree expansion. Choosing rules can be either exhaustive (if only a few candidates exist) or with the guidance of effective heuristics including domain specific and non-domain specific knowledge.

GC is a knowledge-intensive system. This kind of systems may involve some problem. One such problem is the inability to derive a proof for lack of relevant knowledge. Another problem is excessive computational complexity[MKK]. Thus sufficient and well organized background knowledge is tremendously important. GC utilizes the knowledge of graph theory for algorithms in its proving process if applicable (for example, finding all cycles in a graph containing no unknown). More general knowledge is initially built into the attributes corresponding to frame slots of each key graph concept (type). The major slots of a graph type frame are shown below (the correctness of the knowledge is proved mathematically):

- A T-procedure (Type procedure) associated with a precedence for class recursive decompositions of the type in a predefined order
- Subsumption relation with other types
- Indexes of inference rules to rewrite formulas with the type
- Indexes of axioms available to the type.

GC has shown its proving ability by having proved many theorems in text books of graph theory [Ha] [Bo] [ST]. GC also has shown the ability of manipulating variety of graph types.

1.4. The Organization of The Thesis

Chapter 2 provides an overview of the related significant researches. A general survey of various approaches to the problem of effective proving and reasoning is given and specific solutions on the domain of graph theory that have been suggested by other researchers are described.

Chapter 3 and 4 present an effective classification-based knowledge representation scheme. The definitions, operations, postulations and theorems of the 'graph class algebra' are given in chapter 3.

The expert knowledge about these graph types is formed in terms of a set of pattern-match-based recursive procedures, (called T-procedures) which are described in chapter 4.

With the support of background knowledge the reasoning system is able to employ various simple and powerful proof methods. The modeling of these proof methods to be suitable for the class representation scheme is discussed in chapter 5.

The case-splitting based global reasoning process is discussed in chapter 6 where variety of inference rules are presented.

The example theorems for which GC has generated proofs are demonstrated in chapter 7. A special representation of tree which brings efficiency and flexibility for certain kinds of problem solving has been studied during the research. Several sections of the chapter are devoted to this subject. The implementation of GC and software tools used to support the approach are described in chapter 8. The discussion of the approach as well as the conclusion and the future research plans are given in the final chapter.

CHAPTER 2

Related Previous Works

Automated reasoning is concerned with the study of using the computer to assist in that part of problem solving requiring reasoning [Wo]. In this thesis the reasoning generally refers to mathematical reasoning. There are basically two types of automated reasoning systems. One of these emphasizes heuristic search techniques, variety of rules, and their usage [Ni]. The other, on the other hand, stresses the knowledge (or expert knowledge) needed and the techniques to process the associated knowledge. The traditional theorem provers belong to the first type. The second type refers to the so called knowledge-based reasoning systems.

The early study in automated reasoning was dominated by 'stand-alone' type, in which a reasoning system worked independently of users and completely draws results on its own. The system IMPLY [CF2] and Wu's geometry machine [Cs1] belong to this kind. Since the mid 60's more and more researches have shifted to 'interactive' style, in which a system is used to find parts of proofs and to provide information to the user who then can reform a new lemma or create a new search guide in terms of both the output and his own intuitions. The BMTP (Boyer-Moore Theorem Prover) and ITP (Interactive Theorem Prover) belong to this kind.

Automated reasoning systems like BMTP and ITP are referred to as general purpose systems, and others like the geometry machine [Cs1] are referred to as special purpose systems. A general purpose reasoning system provides a research environment and a variety of techniques of reasoning capable of obtaining proofs in many different fields [Bl2]. A special purpose reasoning system mainly works on a special domain of an area which includes proving properties about certain kind of programs, answering questions over a specified database, proving theorems in a subfield of a branch of mathematics, and even designing some electronic circuits.

Based on the different reasoning strategies used, at least three major distinct functions are provided in reasoning systems. The first function is proof checking (to check a proof for correctness) as in the system AUTOMATH [DeB]. The second one is theorem proving as provided by various well-known provers [Lol]. The third one is reasoning by decision procedures as Bledsoe's inequality prover. Some systems offer all these functions, Edinburgh LCE [La1] as an example.

The research discussed in this thesis adopts an approach which combines the techniques provided from systems with both theorem proving type and knowledge-based reasoning type for different reasoning processes. The general review of the techniques of automated reasoning systems is given in subsequent sections (section 2.1 and 2.2) where many technique details are directly cited from the original publications.

2.1 Perspective on Theorem Proving

Automated Theorem Proving (ATP) can be viewed as the heart of many artificial intelligence endeavors. For example, in a question answering system of a database, in a planning system of a robot control, or in an inference system of an expert system, the required automated reasoning can be regarded as some side effects of traversing correct proofs. The recent success of programming language PROLOG even shows that a theorem proving system can form the basis for a general purpose AI language.

The early research efforts for ATP were focused on propositional logic. In late 50's the major research shifted to first-order logic. The most important impact on the logic approach occurred in the mid 60's, when Robinson described his famous resolution procedure in first-order predicate calculus [Ro1].

The resolution method is a method of proof which repeatedly uses an inference rule, called resolution rule, during a proof process. The process starts with the union of a negated theorem and a set of axioms, and ends with an empty formula form. The cancellations take place in the process due to the applications of the resolution rule, which can be briefly shown as: $(p \vee q) \wedge (\neg q \vee r) \rightarrow p \vee r$. Here $\neg q$ represents 'not q', q and $\neg q$ are canceled from clauses $(p \vee q) \wedge (\neg q \vee r)$, thus simplifying the clauses. One of the main features of the resolution approach is that the control heuristics are suggested based on theorem syntax only. Therefore, it has a relatively simple structure. Robinson has shown that the resolution is complete and sound for first-order logic.

The main drawback of this approach is that the proving process may suffer from the so called combinatorial explosion problem, i.e. the choices to apply the resolution rule increase rapidly during the reasoning process. It is widely believed that a resolution-type system requires an extensive change in philosophy in order to prove real hard problems. After the mid 70's the human-oriented approach became the major interest in ATP research.

The human-oriented approach emphasizes the techniques:

- Knowledge base: a data base (library) of facts,
- Natural deduction: logic form $A \rightarrow B$ rather than $\neg A \vee B$,
- Reduction: rewrite rules instantiating variables to fit applications,
- Typing: a mechanism to utilize different domains of objects,
- Procedure: a mechanism for integrating search control with inference.
- Models and counterexamples: a mechanism used to provide correct and reject wrong search paths of pursuit,
- Analogy: the use of a similar reasoning pattern in a related situation to guide search,
- Learning: the improvement of performance by automated retention of prior knowledge or processing thereof,
- Algebraic simplification: reduction of algebraic expressions to facilitate further processing,
- Man-machine interaction: the close involvement of the user in the proof search process [B11].

Generally, a human-oriented ATP uses multiple inference rules, and the search heuristics are not only directed by syntax alone, but also created by simulating human proving. The structure of this method is more complex than that of the logic method. Much progress has been made since the approach was used to construct an automated theorem prover [Ne1] [Ne2] [Ne3] [BM1] [BT1] [Lo1] [BL1].

The major problem which prevents an ATP from being powerful is that the ATP program often generates thousands of consequences of the axioms before a correct proof can be found. To avoid getting into a big search space, an ATP should have a large enough amount of 'intelligent' guidance. Therefore, one major research activity is to devise methods (strategies) for guiding the program to make automated search of proofs.

In the following subsections the techniques which reflect the state of the art in automated reasoning provided by several theorem provers are reviewed. These techniques are very helpful for developing a graph theory reasoning systems such as the one presented in this thesis.

2.1.1 Recursive Function Theory - BMTP

The BMTP is one of the most powerful theorem provers currently available. It embodies an extensible mathematical theory (recursive function theory) in which theorems can be stated and automatically proved [BM1]. It also can be seen as an alternate to the predicate calculus.

2.1.1.1 Overview of The Function Theory

The axioms and theorems in BMTP are represented by functions. Theorems have the value T(TRUE) or F(FALSE). The proof of a theorem is a process of showing that the value of the theorem is TRUE. A general form of function definition is

$(f\, x_1\, x_2 \cdots x_n) =< functionbody >$

The $< functionbody >$ is an expression of the theory. The proof is done by using measure functions and well-founded relations. The system must exhaustively search through all lexicographic orders of all well-founded relations to find possible candidates which may reduce the expression to true. A well-founded relation is a function r of two arguments, which admits no infinite sequence $x_1, x_2, x_3, \cdots x_n$

with the property that (r x_{i+1}, x_i) $\neq$ F for all integers i greater than 0 [BM1].

The BMTP uses four axioms to define the function EQUAL and IF, which form the core of BMTP:

1. X=Y $\Rightarrow$ (EQUAL X Y)=T
2. X$\neq$Y $\Rightarrow$ (EQUAL X Y)=F
3. X=F $\Rightarrow$ (IF X Y Z)=Z
4. X$\neq$F $\Rightarrow$ (IF X Y Z)=Y

The common logical connectives are defined with IF as followings:

1. (NOT P) = (IF P F T)
2. (AND P Q) = (IF P (IF Q T F) F)
3. (OR P Q) = (IF P T (IF Q T F))
4. (IMPLIES P Q) = (IF P (IF Q T F) T)

Arbitrary data types can be created. Typically, they are defined inductively by the so called shell mechanism. The new definition is added to the system's axiom sets, and the consistency of the system is guaranteed by the shell mechanism.

An induction principle is presented within the theory. It describes a base case, k induction steps (each allows several hypotheses), and a function (relation) that is well-founded on a measure set of variables over all substitutions required to instantiate the k+1 case.

2.1.1.2 Techniques in BMTP

The proving process in BMTP is simply to continually rewrite the formula until it is reduced to T without backtracking. The rewriting guarantees that equivalence rules are applied first. For example, algebraic simplification is an equivalence rule, and induction rewrite rules are applied last.

Thus induction is applied on the simplest and most general form of a formula. Many of the rewrite rules have been designed to produce formulas that are more amenable to inductive arguments. In the BMTP the rewriting rules are divided into many levels. Only those rules in lower levels fail to be applicable, while rules in one higher level will be tried. The rewriting is processed recursively starting from the lowest level.

For example, function APPEND is defined as:

(APPEND x y) = (COND x (CONS (CAR x)(APPEND(CDR x) y)) y)

The following lemma

(APPEND (APPEND x y) z)= (APPEND x (APPEND y z))

can be derived by applying rewrite rules repeatedly.

Inductions are formulated by using information collected at the time when the recursive function is defined and the time when the actual induction is needed. The system exhaustively searches through all lexicographic orders of all well-founded functions. The following illustrates the creation of induction templates at recursive function definition time:

In the definition of function APPEND, two function arguments are x and y, but only x satisfies induction termination checking, i.e. a well-founded function LESSP that decreases when applied to the measured set of x (CDR x). Then BMTP automatically tries to prove the theorem as follows:

(IMPLIES (LISTP x) (LESSP (CDR x) x))

The system proves the theorem by matching an axiom added by the shell mechanism during the definition of LISTP. The only induction template is produced as:

```
(AND (IMPLIES (NOT(LISTP x) (p x))
 (IMPLIES (AND (LISTP x)
    (p (CDR x)))
    (p x))))
```

This states that to prove formula (p x y), where p is a conjecture which involves the APPEND function, it is sufficient to prove that:

1. If x is not a list (the base case), then (p x y) can be proved.
2. If x is a list and (p (CDR x)) is assumed to be true (the induction hypothesis), then (p x) can be proved.

The applications of rewrite rules in BMTP are organized in reverse chronological order, i.e. simplify the conjecture by applying axioms, lemmas, function definitions, equalities, and logical proof rules and so on, from complex results to simpler ones. The induction is applied only to the simplest and most generally stated propositions possible. This strategy is based on the fact that it is difficult to invent the right induction until the simplest strongest conjecture is available, and induction increases the size and complexity of the conjecture.

2.1.1.3 Performance and Usage

The BMTP has been applied to a considerable amount of theorem proving tasks, including some theorems which are very difficult by human standards. Many interesting facts and theorems can be represented and proved in reasonable time if a proper interaction is provided. Many theorem provers suffer from a so called referencing problem, which is a phenomenon of performance degradation due to increased knowledge. This is because the search efforts increase when the possible solution space increases. Though the BMTP has not entirely overcome this problem, its heuristic control has been proved effective to operate the system within an environment containing more than 400 theorems.

The ability of BMTP to automatically produce an induction proof is very attractive for the design of a graph theory theorem prover. This is not only because of the position of the induction method in graph theory theorem proving, but also because the heuristics used for the automation in BMTP are so useful for improving a proving system's power.

2.1.2. The Resolution Proving Technique: ITP

The system ITP was developed at Argonne National Laboratory [LO1] [LO2]. It is an inference-based system built from the LMA (Logic Machine Architecture) package of tools. The main function of the ITP is interactively conducting clause-based reasoning, which is generally able to support most of major techniques currently used in an automated deduction research project including various resolution rules.

2.1.2.1 Overview of The System

The basic item in ITP is a clause which is a disjunction of a set of literals. A literal is an assertion or a denial of facts. The ITP derives new clauses from existing clauses using one or more inference rules. There are four lists of clauses, each of them plays a specific role in the operations of the reasoning system:

1. The axiom list.
2. The set of support list.
3. The have-been-given list.
4. The demodulator list.

The major operations in the ITP are:

1. Choose a clause from the set of support list, and this clause is called 'the given clause'.
2. Infer a set of clauses that have the given clause as one parent, and have other parent clauses selected from the axiom list, the have-been-given list, and the demodulator list.
3. Process generated clauses by simplification and subsumption etc..
4. Move a given clause from a support list to the have-been-given list.

Inference rules are processes for producing new clauses from existing clauses. LMA supports many inference rules. The followings illustrate the major inference rules through examples:

- Hyper-Resolution

  Example:

  $\neg p$ | q

  p

  ---

  q

  Hyper-Resolution naturally corresponds to a mode of human reasoning. It is one of the most commonly used inference rules.

- UR-Resolution (Unit-Resulting Resolution)

  This is a resolution in which all but one of the clauses that participate in the deduction must be unit clauses, although they can be either positive or negative.

  Example:

  $\neg p$ | $\neg q$ | r

  p

  $\neg r$

  $\neg q$

  A unit clause can be regarded as a statement of fact, whereas multi-literal clause represents conditional statements.

- Binary Resolution

  Exactly two clauses may participate in the clash (note no positive or unit restriction on these two clauses).

Example:

$\neg p$ | q | $\neg r$

$\neg q$ | s

$\neg p$ | s | $\neg r$

- Unit Resolution

  One of the two clauses involved in the clash should be a unit.

  Example:

  $\neg p$ | $\neg q$ | r | s

  $\neg r$

  $\neg p$ | $\neg q$ | s

- Factoring

  Factoring is an inference rule that derives new clauses from a single clause rather than from a pair of clauses. The new clause is said to be a factor of the original one.

  Example:

  p(a x) | p(y b)

  ---

  p(a b)

- Paramodulation

  This is a rule based on the substitution properties of the equality relation.

Example:

p(a)

equal(a b)

---

p(b)

The result p(b) is called a paramodulant which is added to the clause set, and p(a) is not deleted. A more general example is:

Example:

p(f(a x)) | q(x)

equal(y f(y y))

---

p(a) | q(a)

- Demodulation

  Demodulation is a paramodulation with the existing clause deleted.

  Example:

  p(f(a) b)

  equal (f(a) c)

  ---

  p(c b)

Here, p(f(a) b) is deleted, and p(c b) is added to the clause set. To demodulate a clause, special function evaluations may be involved. For example, $sum(n1 n2) would be rewritten as n1 + n2, if n1 and n2 are self-defined numeric values. There are many system defined functions in the ITP. For example:

$sum(n1 n2), $comp(n1 n2), $gt(t1 t2), $neg(n1),

$comp(n1 n2), $and(x1 x2), $not(x), etc..

2.1.2.2. Interfaces

The current ITP includes facilities to allow the Prolog component to access status and proof information from the inference mechanisms. Logic programming using language Prolog can be effectively utilized in the construction of automated reasoning systems. Prolog plays a significant role in most of the automated reasoning systems [LO3]. The Prolog component of ITP may be run independently or coordinated with a normal theorem prover invoked by some specified commands. The interface facilities allow the user to create a Prolog routine to process information passed from other systems. ITP also provides a LISP interface which works under Berkeley UNIX and interfaces the layer 2 routines to Franz LISP.

Though the ITP is a resolution-based theorem prover, its rich subroutines and facilities, which support many general proving techniques, could be a very valuable resource for implementation of general reasoning systems.

2.1.3. Natural Deduction: IMPLY

The system IMPLY was developed at the University of Texas by Bledsoe and Tyson. It is a goal-directed natural deduction proving system where the goal and the antecedents are distinct. And the inference rules used there simulate the reasoning of human theorem provers.

2.1.3.1. Inference Rules

The followings are the typical inference rules in ITP:

- MATCH: [$H \rightarrow G$]

  if $H(s)=G(s)$

  then (s).

If the substitution s can be found between the goal G and hyperthesis H, then the substitution 's' is returned.

- AND-SPLIT: [$H \rightarrow A \wedge B$]

  if [$H \rightarrow A$] is (s)

  and [$H \rightarrow B(s)$] is (p)

  then (s)(p).

  To prove that H implies A and B, the clause ($H \rightarrow A$) is matched for some substitution s, and the substitution is applied for B. Then the clause ($H \rightarrow B$) is matched for second substitution p. The composition of s and p – (s)(p) is returned.

- CASES: [$H1 \vee H2 \rightarrow G$]

  if [$H1 \rightarrow G$] is (s)

  and [$H2(s) \rightarrow G$] is (p)

  then (s)(p).

  To prove that either H1 or H2 implies G, the system first proves [$H1 \rightarrow G$] for some substitution. After applying the substitution to the clause [$H2 \rightarrow G$] the system attempts to find the second substitution for it. The composition of these two substitutions is the result of the inference rule.

- OR-FORK: [$A \wedge B \rightarrow C$]

  if [$A \rightarrow C$] is (s)

  then (s);

  else [$B \rightarrow C$]

  To show that A and B imply C, the system tries to prove that [$A \rightarrow C$] or that [$B \rightarrow C$].

- PROMOTE: [$H \rightarrow (A \rightarrow B)$]

$[H \wedge A \rightarrow B]$

In order to prove a goal which is an implication $A \rightarrow B$ the system simply takes A as an additional hypothesis.

- BACH-CHAIN: $[H \wedge (A \rightarrow B) \rightarrow C]$

  if $[B \rightarrow C]$ is (s)

  and $[H \rightarrow A(s)]$ is (p)

  then (s)(p).

  This rule says if the term that implies the goal has an antecedent the system must try to prove the antecedent first. For example, to prove Q in $[P(y) \wedge (P(a) \rightarrow Q(x)) \rightarrow Q(b)]$, the system tries

  $Q(x) \rightarrow Q(b)$, which returns substitution b/x and

  $P(y) \rightarrow P(a)$, which returns a/y

  then

  $(b/x)(a/y)$ is finally returned.

2.1.3.2. Heuristics of Proof Search

Two methods of proof search heuristics are used: reduction and forward chaining.

Reduction refers to the replacement of a term by a simpler term. The expression

$L \rightarrow R$

stands for a reducer. For example, in set theory IMPLY uses following reducer:

$t \subset (A \wedge B) \rightarrow t \subset A \wedge t \subset B$

$t \subset (A \vee B) \rightarrow t \subset A \vee t \subset B$

The examples of reducers from algebra include:

$x + 0 \rightarrow x$

$x * 1 \rightarrow x$

$x + (-x) \rightarrow 0$

$-(x + y) \rightarrow (-x) + (-y)$

Each newly created expression must be tried by using a list of reducers which is maintained by the system. The resulting expression is called the irreducible form of the original expression relative to the list of reducers.

Forward chaining is shown by the rule:

$[(A \wedge (A' \rightarrow B)) \rightarrow C]$

if A is ground (i.e. has no variables) and $A' = A(s)$,

then $[B(s) \wedge A \wedge (A' \rightarrow B) \rightarrow C]$.

This rule does not produce smaller subgoals, but it is used to infer auxiliary terms (B(s)). The method has the following advantages:

- It is easy for humans to understand.
- It includes domain-specific heuristics to speed up a proof.

2.2. Perspective on Knowledge-Based Reasoning

The importance of knowledge-based reasoning stems from the fact that the current design paradigm for 'intelligent' systems stresses the expert knowledge availability and the knowledge processing facilities. This is in sharp contrast to the classical general purpose theorem prover design paradigm, where heuristic search techniques dominate. There are many different forms of knowledge-based reasoning: formal reasoning; procedural reasoning; generalization; abstraction; and metalevel reasoning.

Formal reasoning involves the syntactic manipulation of data structure to deduce new structures based on prespecified inference rules. Such reasoning is widely used in logic-based and production rule-based representation systems.

Procedural reasoning involves specialized routines or procedures for producing conclusions and solving problems. It is most naturally and most commonly used in frame-based and semantic network-based systems.

Generalization and abstraction are also natural processes for human beings. The two areas are the core of most human learning. They involve induction rules and instantiation rules in reasoning.

Metalevel reasoning involves using knowledge, particularly the knowledge about the usage of knowledge. Recent research indicates that metalevel reasoning plays a key role in human cognition [Fl].

One important property of knowledge-based reasoning is clear separation of the knowledge base from the inference processing strategy. The advantage of such separation is that as the operation of the system gains in sophistication it can improve itself in relatively noninterfering ways. This can be done either by adding knowledge or by reusing the same knowledge with more sophisticated inference engines [Sc]. Knowledge-based reasoning consists of not only deductions but also the automation of reasoning processes [Bu1]. The following explains several such processes for reasoning in mathematics:

1. Formalizing problems. In order to implement automatical reasoning, reasoning systems need to automatically translate informal problem statements into mathematical formulae to which deduction may be applied.

2. Learning. Newly proved theorems must be assimilated into the theorem proving system so that they may be used effectively in the future. The required assimilation technique will depend particularly on the search control technique being used, e.g. the way used to label new theorems so that they can be accessed when needed. Similarly, the learning of new proof methods, the defining of new concepts, the conjecturing of results etc. may involve a proof analysis technique to analyze new proofs and to generalize them to extract control information.

3. Using analogy. Analogical matching techniques are used to find and apply the relationship

between the target and the source of the analogy in order to suggest conjectures and definitions.

In the following subsections three reasoning systems are reviewed. Techniques provided by these systems are used or considered in this thesis research.

2.2.1 Knowledge-Based Automated Equation Solving: DReam

A knowledge-based reasoning system called DReam (Discovery and Reasoning in Mathematics) has been developed at the Department of Artificial Intelligence of Edinburgh University by Bundy's research group. This system attempts to realize the above processes and many other mathematical reasoning techniques. There is a family of programs working on the common domain of symbolic equation solving. Followings are explanations of the relationships among various programs in this group.

The PRESS program uses the deduction technique of rewrite rules to generate equation solutions. This deduction technique is guided by the search control technique of metalevel inference [BW1].

The LP (Learning PRESS) program uses the proof analysis technique of precondition analysis to extract and conjecture new equation solving methods which are then used by PRESS.

The IMPRESS (Inferring Meta-Knowledge about PRESS) program [SB1] is a theorem proving program for proving logic program properties. It is used to prove properties of the Prolog code of PRESS using a modified version of recursion guidance.

The GRASS is a rewrite rule system for Grassman Geometry modeled on PRESS, it uses symmetry to control the application of the rules [FS1].

The ECO and ASA are 'intelligent' front-ends to ecological modeling and statistics packages respectively.

The RUT is a rational reconstruction of Bledsoe's natural deduction theorem prover [Bl1], and VOYEUR extends RUT with interaction technique.

The MT program is the core of system DReam. It consists of two parts:

- the object-language: a logic for expressing problems,
- the meta-language: a logic for expressing proof plans.

MT uses a process of meta-level inference to analyze a conjecture, choose an appropriate proof plan and then uses it to guide the search for a proof. The short term objectives of Bundy's group are to extend their existing reasoning techniques and invent new techniques applicable to a variety of domains. It also investigates the interaction of deduction, search control, proof analysis and inductive inference within a single reasoning system. The object-level deduction technique is based on Bibel's Connection Calculus [Bil]. In Bundy's group there is a plan to design and implement several proof plans on the MT system. Heuristics developed from natural deduction proofs can be readily translated into the Connection Calculus. Another plan which aims to improve the system's theorem proving ability is to add a learning component using the meta-level concepts already embodied in the MT proof plans and to use precondition analysis and other techniques to modify the existing plans and build new plans.

2.2.2. Automated Reasoning in Graph Theory: GT

GT is a system which performs mathematical reasoning in graph theory [Ep]. It is a knowledge intensive domain specific system which uses algorithmic class description to prove relations among mathematical concepts.

2.2.2.1. The Knowledge Representation

In GT a graph concept is a frame, which represents a graph property and knowledge associated with it. The main slots of the frame are related to property descriptions. A property description is a triple $< f, S, \sigma >$ where S is the seed set, a set of one or more minimal graphs (seeds), each of which has the property. An operator f describes how a graph with the property may be transformed to construct another graph with the same property. In GT the primitive operations are:

1. A_x: add the vertex x,
2. A_{xy}: add the edge between x and y,
3. D_x: delete the vertex x,
4. D_{xy}: delete the edge between x and y.

These primitives may be concatenated into terms such as $A_{yx}A_x$ to denote sequential operations from right to left. Terms may be summed (as in $A_x + A_{yz}A_z$) to represent alternative actions. The selector σ in the definition gives the restrictions for binding the variable appearing in the operator f to the vertices and edges in a graph. GT has the following selector descriptions for a graph:

1. whether or not a vertex is in the graph,
2. whether or not a vertex is distinct from another vertex,
3. the degree of a vertex,
4. whether or not a degree of a vertex is maximum,
5. whether or not an edge is in the graph,
6. whether or not the endpoints of an edge is distinct.

The semantic interpretation of a property definition is a single uniform algorithm, called a p-generator, which generates a class of graphs with the same property. The p-generator may be thought of as an automation which starts with any graph in the seed set S. The algorithm for generating the class P of the graphs is:

```
Accept $G \in S$
Output G
Until $\sigma$ fail do
    $G \leftarrow f\sigma(G)$
```

```
        Output G
    Halt
```

2.2.2.2 The Proving Mechanism

The theorems considered by GT are restricted to the following types:

- TYPE 1: If a graph has property p, then it has property q
- TYPE 2: A graph has property p, if and only if it has property q
- TYPE 3: If a graph has property p and property q, then it has property r
- TYPE 4: It is not possible for a graph to have both property p and property q

Note: it is clear that this approach dose not handle the theorems involving subgraphs.

The proving process in GT relies on procedural manipulations of graph properties. There are two procedures used on proving process. The first procedure tests for subsumption: property p for class P subsums property q for class Q if and only if Q is a subset of P. The second procedure constructs mergers: the merger of a property p for class P with a property q for class Q results in a new property represented by P $\cap$ Q, the set of graphs with both properties. The method used by subsumption procedure can be described as follows:

Given property $p_1 = < f_1, S_1, \sigma_1 >$, property $p_2 = < f_2, S_2, \sigma_2 >$, and a conjecture that p_1 subsumes p_2, GT attempts to show that:

1. f_2 is subsumed by f_1, i.e. f_2 is a special case of f_1
2. every graph in S_2 has property p_1
3. σ_2 is subsumed by σ_1, i.e. σ_2, is more restrictive than σ_1.

The method used by merge procedure is similar:

Given property $p_1 =< f_1, S_1, \sigma_1 >$, property $p_2 =< f_2, S_2, \sigma_2 >$, GT attempts to construct the merger $p =< f, S, \sigma >$ of p_1 and p_2. The algorithms are:

1. If p_1 subsumes p_2 the merger is simply p_2.
2. When f_1 subsumes f_2 and every seed in S_2 has property p_1, the merger is $< f_2, S_2, \sigma >$ where σ is σ_1 and σ_2 with any references to variables not in f_2 eliminated
3. When f_1 subsumes f_2, σ_1 subsumes σ_2, and S is non-empty the merger is $< f_2, S, \sigma_2 >$ where

$$S = G|G \in S_2 \cap P_1 \cup G|G \in S_1 \cap P_2$$

GT's proving process is driven both by examples (specific graphs) and by definitional forms (algebraic forms) with support from the background knowledge of graph properties (frames).

2.2.3. Man-Machine Graph Theory Proving: GRAPH

The system consists of three knowledge based interactive expert system components used to prove or to help proving graph theorems: (1) THEOR: a resolution type theorem prover, (2) ALGOR: a knowledge system on graph algorithm, (3) BIBLI: a knowledge system on a bibliography of graph theory [PC]. The theorem prover THEOR formalizes graph theory by a special first-order predicate calculus, called AGT. The sysntax of AGT is as:

X, Y, Z point variables,

U, V, W line (or edge) variables,

K, L, M, N integer variables,

G, H graph names,

O constants,

F function names,

A operations over graphs,

P, Q, R, S, T predicates.

The graph itself is not a variable in such formalization, but it can be added as a suffix of a predicate. For example, S1G(X, Y, U) has the meaning 'vertices X and Y are joined by an edge U in graph G. The suffixes B, BE, E, BCE, and BC can be added to the predicate names, where

B and C denote the name of a particular graph and E is a string denoting a graph operation. For example, Q5H7A1(Z) has the meaning that Z is an isolated point in the complement of the graph H7. The theorem prover provides several standard theorem proving techniques interactively including splitting procedures described by Bledsoe. The techniques are:

1. forward chaining: subgoal A is replaced by P
2. case analysis
3. reduction and absurdum
4. simplification and extension of the subgoal
5. modification of the subgoal using the equivalence transformations
6. sending the subgoal to the resolution-based prover
7. telling the system that the current subgoal is true
8. moving the root of the tree up, down to the left, to the right, to a specific point
9. Omittint a subtree of the generated tree

GRAPH is a system which provides human with variety kinds of knowledge supports in the research related to graph theory. Its main reasoning process belongs to interactive resolution refutation theorem proving. The main shortcoming of the system is that it basically has no stand-alone proving ability so it can not reason on problems requiring complex domain specific knowledge. For example, it is hard to be used for reasoning relations relating to subgraph.

The advantage of using the system is that it is able to cope with some powerful existing resolution type theorem provers without doing much representation transformation.

CHAPTER 3

Algebraic Knowledge Formalization

The graphs considered in this thesis are assumed to be simple (i.e. no parallel edges and self-loops are allowed). In general, a graph is defined as G=(V, E), where V represents an arbitrary finite set of vertices, and E, an arbitrary subset of the unordered (thus only undirected graphs are considered) Cartesian product V*V (edges). Let U be the universe of all graphs, then a graph type T is a subset of U and a graph class is a subset of the type T. The research has formalized several dozen types algorithmically defined in graph theory. For example, vertices (or type-0), single-edge (or type-e), path (or type-1), star, clique, cycle, Hamiltonian, acyclic, binary tree, tree, connected, disconnected, bipartite, simple graph, etc.. The formalization is done in terms of an algebraic language, called Graph Class Language (GCL), applied to mathematical knowledge about graph types. The following two sections discuss the algebra and then chapter 4 presents the type procedures.

3.1 The Syntax and Semantics of Atomic Expression

The components of the language GCL consists of v-list, graph type, constraint symbol, constraint argument and class set off by parentheses, brackets and commas in a manner to be illustrated in the syntax descriptions. A class is regarded as a set of all graphs corresponding to a value of LRC. A constraint is a predicate, which is used to represent a relation in the domains involved in graph theory (e.g. number theory, set theory, etc.). The syntax and semantics of the language are described below:

- **Vertex** is a string of characters starting with a letter x followed by zero or more letters and/or digits representing a vertex in graph theory. Example, x1, x, xn1 are vertices.
- **Unknown** is a string of characters starting with a letter y followed by letters and/or digits representing a sequence of unknown vertices. Example, y, y1, yn0 are unknowns.

- **Unknown followed by a prime** means that the unknown can be an empty sequence, otherwise the unknown contains at least one vertex.

- **V-list** is a list of vertices and unknowns (both are called elements of the v-list), where all vertices including those contained (implicitly) in unknowns are distinct. Example: x1yx2 is a v-list and x1, y and x2 are elements of the v-list, also x1, x2 and all vertices contained in y are distinct.

- **Constraint** is a form: [P], or [P arg1..argn]. It must follow a predecessor: an element of S (v-list) or a class. Here P is a predefined predicate symbol that takes its predecessor and arg1 .. argn as arguments. If the predecessor is an element of a v-list then the constraint is called an element constraint, otherwise (the predecessor is a class) the constraint is called a class constraint. Example: y[not-in y1] shows that the element constraint (predicate) symbol 'not-in' has two arguments: y, the predecessor, and y1. y[not-in y1] means that every vertex contained in y is not in y1.

- **Constrained v-list** is a v-list with constraints in it. E.g. x1[not-in y1]yx2 is a constrained v-list, its corresponding v-list is x1yx2, but both may be called a v-list if no confusion results.

- **Type Symbol** defines a subset of the universal graph set and associates with a recursive procedure capable of generating any graph in the subset. It is a predefined character string used as a component of a class. Example: 0, e, 1, tree.

- **Class** defines a subset of graphs within a type and has the form: < S, T >, or < S, T >[P_1 arg_{11}..arg_{1n}]..[P_k arg_{k1} arg_{km}], where S is a constrained v-list, T is a type name. Any graph in the class can be generated by applying the type procedure associated with T on the input S. Also any graph that can be generated in such a way belongs to the class. The symbol '< S, T >' may represent either constrained or unconstrained case if no confusion results. Example: < x1x2y′, cycle >[with x] is a class, where [with x] is a class constraint.

- Edge is a special single-edge type (or e type) class with form < S, e > or simply < S >, where S is a v-list of two vertices. Example: < x1x2, e >, < x1x2 > and < x2x1 > are edges, and they are recognized as the same edge in the pattern match process.

3.2 The Primitive Algebraic Operations

The followings are primitive algebraic operations on the set of graphs:

- AND($g_1..g_n$). defines the set of all graphs, such that each graph is constructed by the union of n graphs, one chosen from each set g_i for i=1..n. We also use $g_1.g_2$ to represent AND(g_1, g_2).
- OR($g_1..g_n$). defines the set of all graphs, such that each graph is chosen from set g_i for some i in (1, n). We also use $g_1 + g_2$ to represent OR(g_1, g_2).
- RV(g_1, g_2) defines a set of all graphs, such that each graph is constructed by removing the vertices in the vertex set of a graph in g_2 from a graph in g_1.
- RE(g_1, g_2) defines a set of all graphs, such that each graph is constructed by removing the edges in the edge set of a graph in g_2 from a graph in g_1.
- IV(g_1, g_2) defines a set of all graphs, such that each graph is constructed by removing the vertices not in vertex set of a graph in g_2 from a graph in g_1 (IV means vertex induced).
- IE(g_1, g_2) defines a set of all graphs, such that each graph is denoted by < V, E > where E is the intersection of the edge sets of a graph in g_2 and a graph in g_1, V is the vertex set of E (IE means edge induced).
- CMPL(g) defines a set of all graphs, such that each graph is obtained by complement of a graph in g.

3.3 The Algebraic Properties

Postulates

1. P1 (Existence of Φ):

   Let Φ represent the empty set of graphs.

   (a) g.Φ = g;

   (b) if g is a class then RV(g, g) = Φ

   (c) g+Φ = g + Φ = g;

2. P2 (Commutativity):

   (a) $g_1 + g_2 = g_2 + g_1$;

   (b) $g_1 . g_2 = g_2 . g_1$

3. P3 (associativity):

   (a) $(g_1 + g_2) + g_3 = g_1 + (g_2 + g_3)$;

   (b) $(g_1 . g_2) . g_3 = g_1 . (g_2 . g_3)$

   Note that the proofs for the above postulates P1, P2 and P3 follow directly from the definitions of the primitive operations and are omitted.

4. P4 (distributivity of AND, RV, RE, IV, IE, CMPL over OR):

   (a) $AND(OR(g_1, g_2), g_3) = OR(AND(g_1, g_3), AND(g_2, g_3))$

   Proof: From the definitions of the primitive operations, it is easy to see the following: any graph G in the set on the left side of the equation is constructed by $G_0 \cup G_3$, where G_0 is either a graph in g_1 or a graph in g_2 and G_3 is in g_3. This means that G is constructed either by $G_1 \cup G_3$ or by $G_2 \cup G_3$, where G_1 is in g_1, G_2 in g_2, and G_3 in g_3. Thus G is a graph in the set on the right side of the equation. Similarly, any graph in the set on the right side of the equation can be proved to be in the set on the left side of the equation.

QED

Note that the proofs for the following postulates of P4 are similar to the above one and are omitted.

(b) $AND(g_1, OR(g_2, g_3)) = OR(AND(g_1, g_2), AND(g_1, g_3))$

(c) $RV(OR(g_1, g_2), g_3) = OR(RV(g_1, g_3), RV(g_2, g_3))$

(d) $RV(g_1, OR(g_2, g_3)) = OR(RV(g_1, g_2), RV(g_1, g_3))$

(e) $RE(OR(g_1, g_2), g_3) = OR(RE(g_1 . g_3), RE(g_2, g_3))$

(f) $RE(g_1, OR(g_2, g_3)) = OR(RE(g_1, g_2), RE(g_1, g_3))$

(g) $IV(OR(g_1, g_2), g_3) = OR(IV(g_1 . g_3), IV(g_2, g_3))$

(h) $IV(g_1, OR(g_2, g_3)) = OR(IV(g_1, g_2), IV(g_1, g_3))$

(i) $IE(OR(g_1, g_2), g_3) = OR(IE(g_1, g_3), IE(g_1, g_3))$

(j) $IE(g_1, OR(g_2, g_3)) = OR(IE(g_1, g_2), IE(g_1, g_3))$

(k) $CMPL(OR(g_1, g_2)) = OR(CMPL(g_1), CMPL(g_2))$

5. P5 (distributivity of AND over RV; RE; IV; IE):

(a) $RV(g_1 . g_2, g_3) = RV(g_1, g_3) . RV(g_2, g_3)$

proof: assume that a graph in the left-side set is

It is constructed by (V1, E1) from g_1, (V2, E2) from g_2 and (V3, E3) from g_3. the graph can be represented as: (((V1 ∪ V2)-V3), (E1 ∪ E2) - $\bar{E}_{1,2;3}$) where $\bar{E}_{1,2;3}$ contains all edges in (E1 ∪ E2) with an end vertex in V3. and the corresponding graph constructed by the same three graphs in right-side set is: (((V1 -V3) ∪ (V2 -V3)), (E1 - $\bar{E}_{1,3}$) ∪ (E2 - $\bar{E}_{2,3}$)) where $\bar{E}_{1,3}$ contains all edges in E1 with an end vertex in V3, $\bar{E}_{2,3}$ contains all edges in E2 with an end vertex in V3. By set operations they are the same graph.

QED

(b) $RE(g_1.g_2,g_3) = RE(g_1,g_3).RE(g_2,g_3)$

proof: assume that a graph in the left-side set is (((V1 ∪ V2)), (E1 ∪ E2) - E3), and it is constructed by (V1, E1) from g_1, (V2, E2) from g_2 and (V3, E3) from g_3 and the corresponding graph constructed by the same three graphs in the right-side set is: ((V1 ∪ V2), ((E1 - E3) ∪ (E2 - E3))) By set operations they are the same graph.

QED

Note that the proofs for the following postulates of P5 and P6 are similar as the above ones and are omitted.

(c) $IV(g_1.g_2,g_3) = IV(g_1.g_3).IV(g_2,g_3)$

(d) $IE(g_1,g_2.g_3) = IE(g_1,g_2).IE(g_1,g_3)$

6. P6 (others):

(a) $RV(g_1,g_2.g_3) = RV(RV(g_1,g_2),g_3)$

(b) $RE(g_1,g_2.g_3) = RE(RV(g_1,g_2),g_3)$

Theorems Note that all the proofs for the following theorems follow directly from definitions of AND and OR operations, and are omitted.

1. T1 (idempotency):

(a) $g + g = g$

(b) if g is a singleton then $g.g = g$

2. T2 (Absorption):

(a) $OR(OR(g_1,g_2),g_3) = OR(g_1,g_2,g_3)$

(b) $OR(g_1,OR(g_2,g_3)) = OR(g_1,g_2,g_3)$

(c) $AND(AND(g_1, g_2), g_3) = AND(g_1, g_2, g_3)$

(d) $AND(g_1, AND(g_2, g_3)) = AND(g_1, g_2, g_3)$

Normalization The above properties are used to simplify a class expression. Particularly, property P4 and T2 provide a base to construct a procedure which converts any class expression on sets of graphs g_1 .. g_n into a normal form:

Definition (normal form) Normal form of a class expression is as:

OR (q_1, .., q_i, .., q_m),

where each q_i contains no OR operator.

It is easy to see that any class expression can be equivalently converted to a normal form (may not be unique) by applying these properties in finite steps.

If each q_i contains only ANDs, then the expression is said to be in OR/AND normal form.

Example 2.1

The expression:

AND(CMPL(OR(g_1, g_2)), RV(g_3, g_4))

is converted to:

AND(OR(CMPL(g_1), CMPL(g_2)), RV(g_3, g_4))

and has the normal form:

OR(AND(CMPL(g_1), RV(g_3, g_4)), AND(CMPL(g_2), RV(g_3, g_4)))

Example 2.2

The expression:

AND(OR(g_1, g_2), AND(g_3, g_4))

has the normal form:

OR(AND(g_1, g_3, g_4), AND(g_2, g_3, g_4)).

An expression of graph classes can be manipulated not only by the algebraic properties for simplification and normalization, but also by corresponding T-procedures for conceptual decomposition, which is shown in chapter 4.

CHAPTER 4

Procedural Knowledge Formalization

The procedural knowledge of graph concepts is formalized by a set of recursive procedures called T-procedures. A T-procedure is used to recursively define constructions of graph classes, (i.e. subsets of a graph type). The specification of a T-procedure describes its output graph set for an input S (i.e. v-list).

4.1 Recursive Decomposition of Graph Class

A T-procedure is associated with a type T. It decomposes a class < S, T > accepting any v-list S. Its output is called a c-decomposition of < S, T >. c-decompositions are also called an OR-AND normal form of the class and they have the same vertex set as that of S. The outputs are characterized by one of the following cases:

1. Φ_g an empty set (if S is empty).
2. Each graph in the set has an empty edge set ϕ_e (or denoted by 'none').
3. The set contains a single graph with one edge < x1x2 >.
4. The set is generally represented as:

   OR(AND(<S_{1_1}, T_{1_1}>, ..<S_{1_j}, T_{1_j}>), .., AND(<S_{i_1}, T_{i_1}>, ..<S_{i_k}, T_{i_k}>))

   where each operand of the operator OR is said to be a sub c-decomposition of < S, T > which represents a subset of type T (by the definition of OR). Each operand of an operator AND is a class called an output class. Each output class represents subgraphs of the class (by the definition of AND).

A T-procedure outputs constraints of v-list elements, and these constraints are following their predecessors. A T-procedure also outputs constraints of classes, and they are following their sub c-decompositions. Each output class has type precedence not greater than the type precedence of the decomposed class or has its v-list shorter than the v-list of the decomposed class.

4.2 Procedural Knowledge of Graph Types

One of important tasks in this research is to provide modelings for sufficient amount of background knowledge to manipulate an arbitrary abstract graph. The most flexible and useful modeling is the type procedures formalized for variety of key graph concepts (e.g. path, cycle, star, clique, rooted tree etc.). Each T-procedure represents the knowledge about the type, the relations among the subsets of the type, and the relations among subgraphs.

We (the system and/or users) use a graph type to define a set of graphs (as a subset of the universal graph set) which possess a graph property. Verifying whether or not a given graph has the property can be done by an existing (or predefined) graph theory algorithm.

We now use the specification of a type procedure to specify the set of graphs that belong to the type and are constructed for a given v-list in certain method. A correct specification should satisfy the condition: the graphs generated by the specified method on the v-list and on all permutations of the v-list is exactly the subset of the type, which includes all graphs with the vertex set of the v-list.

We also use T-procedure to define a set of graphs in decomposed forms. A correct T-procedure should satisfy the condition: the c-decomposition is exactly the same subset of the type as defined by the specification.

The procedural formalization is intended to provide better (but may not be optimal) recursive procedures utilizing the attribute LRC for constructions of graphs with certain properties (types). In order to accomplish this, the formalization provides procedures which construct the desired classes without removing too many non-member candidates. This section presents all T-procedures defined in this research. The discussion and example proofs about the correctness are given in next section.

4.2.1 Notations and Definitions

To simplify the illustrations several bold letters followed by digits are used to represent patterns in v-lists.

z: a pattern of either a vertex or an unknown;

w: a segment of the v-list (i.e. a sub v-list);

x: a vertex;

y: an unknown.

Also some notations are defined to simplify the code:

Definition 1 α_a(S) is a set, which consists of all the **x** type elements in v-list S, here subscript 'a' denotes an arbitrary member of the set.

Example α_a(x1y1x2y2) is the set (x1, x2) and 'a' has two possible values: x1 and x2.

Definition 2 $\beta_{w1,w2}$(S) is a set, which consists of all ordered pairs of **w** type v-lists denoted by (w1, w2), where the following two conditions hold:

(1) w1 is obtained by removing zero or more elements from S in such a way as to obtain one of possible patterns **xw**;

(2) w2 is obtained by removing all elements of w from S.

For example, $\beta_{w1,w2}$(y1x1y2y3) represents the set:

{ (x1y2, y1x1y3), (x1y3, y1x1y2), (x1y2y3, y1x1) }

Definition 3 $\gamma_{w1,w2}(S)$ is a set, the members of which are unordered pairs of **w** type v-lists (denoted by w1, w2) where the following conditions hold:

(1) w1 is obtained by removing one or more elements from S and has pattern **w**;

(2) w2 is obtained by removing all elements of w1 from S.

For example, $\gamma_{w1,w2}$(y1xy2) is the set

{ {y1, xy2}, {x, y1y2}, {y2, y1x} }

Definition 4 $\delta_{a_1,a_2,\cdots a_k}(S)$ is a set, which consists of all **w** type v-list containing k vertices of S. Each of these v-lists is obtained by removing the first vertex of S and additional vertices. S must contain no unknowns and must have more than k vertices, here subscript 'a' has denotes an arbitrary member of the set.

For example, δ_{a_1,a_2} ($xx_1x_2x_3$) is the set

$\{x_1x_2, x_1x_3, x_2x_3\}$

here a_1, a_2 has three possible values: x_1x_2, x_1x_3, x_2x_3

Definition 5 Let f(e) be a class expression having argument e, a v-list segment or a set of v-list segments. Let Θ be one of sets: α, β, γ, and δ, and OR!Θf(e) is defined as:

OR(f(e_1), f(e_2), $\cdots$, f(e_n))

where e_i for i in (1, n) are all distinct members of the set Θ.

Example 1

Let S=x1y1x2y2,

OR!α_x(S)AND(<w, T>, < xw, e >) is equalent to

OR(AND(< w, T >, < x1w, e >), AND(< w, T >, < x2w, e >))

Example 2

Let S=y1xy2, OR!$\beta_{w1,w2}$(S)AND(< $w1$, T1 >, < $w2$, T2>) is equivalent to

OR(AND(< y1, T1 >, < xy2, T2 >),

AND(< x, T1 >, < y1y2, T2 >),

AND(< y2, T1 >, < y1x, T2 >))

Example 3

Let S=y1xy2, OR!$\gamma_{w1,w2}$(S)AND(< $w1$, T1 >, < $w2$, T2>) is equivalent to

OR(AND(< y1, T1 >, < xy2, T2 >), AND(< y1x, T1 >, < y2, T2 >))

Example 4

Let S=y1x1y2y3, OR!$\beta_{w1,w2}$(S)AND(< $w1$, tree >, < $w2$, tree >) is equivalent to

OR(AND(< x1y2, tree >, < y1x1y3, tree >),

AND(< x1y3, tree >, < y1x1y2, tree >),

```
AND(< x1y2y3, tree >, < y1x1, tree >))
```

4.2.2 T-procedures

```
PROCEDURE type-0 (input S: v-list)
    {precedence=0}
    {specification: generates a graph consisting of all vertices in S and no edges}
    BEGIN
    output none
    END.
    PROCEDURE type-e (input S: v-list)
    {precedence=1}
    {specification: generates a set of graphs, such that a graph belongs to the set if and only if
    (1) it has the same vertex set as that of S
    (2) it contains one edge constructed between the first vertex and some other vertices in S}
    BEGIN
    case S of
    x: output none
    x1x2: output < x1x2 >
    y, xy, yx, y1y2: output < S, e >
    z1z2w: Routput OR(< z1z2, e >, < z1w. e >)
    END.
```

Remark: 'Routput' indicates that each following class should be processed recursively by calling its own T-procedure on its own v-list to generate the output.

```
    PROCEDURE type-1 (input S: v-list)
    {precedence=2}
```

{specification: generates a singlton set containing a path, such that a path belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) its edges are constructed by sequentially connecting vertices in S}

```
BEGIN
case S of
x: output none
x1x2: output < x1x2 >; y: output < y, 1 >
yx, xy: output AND(< S, 1 >, < y, 1 >)
y1y2: output AND(< S, 1 >, < y1, 1 >, < y2, 1 >)
z1z2w: Routput AND(< z1z2, 1 >, < z2w, 1 >)
END.
```

Example: The c-decomposition for class < x1x2x3y, 1 > is: AND(< x1x2 >, < x2x3 >, < x3y, 1 >, < y, 1 >).

PROCEDURE type-star (input S: v-list)

{precedence=3}

{specification: generates a singlton set containing a star, such that a star belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) its edges are constructed between the first vertex and all others in S}

```
BEGIN
case S of
x: output none
y: output < y, star >
x1x2: output < x1x2 >
xy: output < xy, star >
```

```
yx: output AND(< y, star >, < xy, join >)
y1y2: output AND(< y1y2, star >, < y1, star >)
z1z2w: Routput AND(< z1z2, star >, < z1w, star >)
END.
PROCEDURE type-clique (input S: v-list)
{precedence=4}
{specification: generates a singlton set containing a clique, such that a clique belongs to the set
if and only if
it has the same vertex set as that of S }
BEGIN
case S of
x: output none
y: output < y, clique >
x1x2: output < x1x2 >
wy: AND(output <wy, clique>, <y, clique>, Routput <w, clique>)
wx: Routput AND(<w, clique>, <xw, star>)
END.
PROCEDURE type-cycle (input S: v-list)
{precedence=5}
{specification: generates a singlton set containing a cycle, such that a cycle belongs to the set if
and only if
(1) it has the same vertex set as that of S
(2) its edges are constructed by circularly connecting vertices in S}
BEGIN
case S of
x: output none {for degenerate cycle}
```

```
xlx2: output < xlx2 > {for degenerate cycle}
y:output < y, cycle >
xy, yx: output AND(< xy, 1 >, < y, 1 > < yx, 1 >)
yly2: output AND(<yly2, 1>, <yl, 1>, <y2, 1> <y2yl, 1>)
zlwz2: Routput AND(< zlw, 1 >, < wz2, 1 >, < z2zl, 1 >)
END.
PROCEDURE type-join (input S: v-list)
{precedence=6}
{specification: generates a singlton set containing a graph, such that a graph belongs to the set
if and only if
(1) it has the same vertex set as that of S
(2) if it has more than one vertex then it has one edge constructed between the first two vertices
in S }
BEGIN
case S of
x: output none
xlx2: output < xlx2 >
y, xy, yx, yly2: output < S, join >
zlz2w: Routput < zlz2, join >
END.
PROCEDURE type-fork0 (input S: v-list)
{precedence=7}
{specification: generates a set of graphs, such that a graph belongs to the set if and only if
(1) it has the same vertex set as that of S
(2) its edges are constructed between the first vertex and zero or more other vertices in S}
BEGIN
```

```
case S of
x: output none
x1x2: output OR(< x1x2 >, none)
xy, yx, y1y2: output < S, fork0 >
z1z2w: Routput AND(< z1z2, fork0 >, < z1w, fork0 >)
END.
PROCEDURE type-fork (input S: v-list)
{precedence=8}
{specification: generates a set of graphs, such that a graph belongs to the set if and only if
(1) it has the same vertex set as that of S
(2) its edges are constructed between the first vertex and one or more other vertices in S}
BEGIN
case S of
x: failure
x1x2: output < x1x2 >
xy, yx, y1y2: output < S, fork >
z1z2w: Routput OR( AND(< z1z2, fork0 >, < z1w, fork >
AND(< z1z2, fork >, < z1w, fork0 >)
END.
PROCEDURE type-tree (input S: v-list, status: [basic, extended]) {precedence=12}
{specification: generates a set of rooted tree, such that a tree belongs to the set if and only if
(1) it has the same vertex set as that of S
(2) the left most vertex of S is the root of the tree, and any other vertex of S has a parent to its
left in S}
BEGIN
case S of
```

```
x: output none
y: output < y, tree >; x1x2: output < x1x2 >
yx: output AND(< y, tree >, < xy, e >)
wx: Routput AND(< w, tree >, < xw, e >)
if status=basic then
wy: output < S, tree >
else
xy: output OR(< S, tree >, AND(< y, tree >, < xy, join >))
wy: OR(output < S, tree >, Routput OR!( $\beta_{w1,w2}$ (wy)AND(<$w1$, tree>, <$w2$, tree>))
END.
```

Note that based on definition 2 the last statement Routput OR!(...), generates a disjunction of all possible AND(< $w1$, tree >, < $w2$, tree >) by decomposing the input **wy** into all possible forms of $w1$ and $w2$ which satisfy the conditions:

(1) $w1$ is obtained by removing zero or more elements from S and has pattern **xw**;

(2) $w2$ is obtained by removing all elements of w from S.

Example 1:

The c-decomposition for class < x1x2x3, tree > is:

OR(AND(< x1x2 >, < x1x3 >), AND(< x1x2 >, < x2x3 >))

and for class < yx, tree > is: AND(< y, tree >, < xy, e >)

Example 2:

Let w=y1xy2y3, the first round of the recursion on expression

OR!($\beta_{w1,w2}$(w) AND(< $w1$, tree >, < $w2$, tree >))

generates the following result:

OR(AND(< xy2, tree >, < y1xy3, tree >),

AND(< xy3, tree >, < y1xy2, tree >),

AND(< xy2y3, tree >, < y1x, tree >)).

Remark: when a T-procedure has extended status every class in its decomposition is associated with its own extended T-procedure.

For example, if status=basic, < y1x2y2, tree > will output:

< y1x2y2, tree >

but if status=extended, then < y1x2y2, tree > will output the following:

OR(< y1x2y2, tree >,

Routput(OR(AND(< x2y2, tree >, < y1, tree >, < x2y1, e >),

AND(< x2, tree >, < y1y2, tree >, < x2y1, e >))))

i.e.

OR(<y1x2y2, tree>, AND(<x2y2, tree>, <y1, tree>, <x2y1, e>),

AND(< y2, tree >, < x2y2, e >, < y1, tree >, < x2y1, e >),

AND(< y1y2, tree >, < x2y1, e >))

```
PROCEDURE type-acyclic (input S: v-list, status = basic)
{precedence=13}
{specification: generates a set of acyclics, such that an acyclic graph belongs to the set if and only if
(1) it has the same vertex set as that of S
(2) each connected portion of the graph is a tree, where each vertex has a parent to its left in S}
BEGIN
case S of
x: output none
y: output < y, acyclic >
x1x2: output OR(< x1x2 >, none)
wy: output < wy, acyclic >
wx: Routput OR(< w, acyclic >, AND(< w, acyclic >, < xw, e >))
END.
```

```
PROCEDURE type-acyclic (input S: v-list, status = extended)
{precedence=13}
{specification: generates a set of acyclics, such that an acyclic graph belongs to the set if and
only if
(1) it has the same vertex set as that of S
(2) each connected portion of the graph is a tree, where each vertex has a parent to its left in S}
BEGIN
case S of
x: output none
y: output < y, acyclic >
x1x2: output OR(< x1x2 >, none)
wx: Routput OR(< w, acyclic >, AND(< w, acyclic >, < xw, e >)
wy: OR(output < wy, acyclic >, AND(Routput < w, acyclic >, output < y, acyclic >))
END.
PROCEDURE type-tree(k) (input S: v-list, status=[basic, extended])
{precedence=14}
{specification: generates a set of rooted tree, such that a tree belongs to the set if and only if
(1) it has the same vertex set as that of S
(2) each node has at most k sons, the left most vertex of S is the root and any other vertex of S
has a parent to its left in S}
BEGIN
case S of
x: output none
y: output < y, tree >[tree(k)]
x1x2: output < x1x2 >
w: Routput < w, tree >[tree(k)]
```

```
END.
```

Remark: the constraint procedure, tree(k), will check each corresponding sub c-decomposition of the class to see if it generates no tree with less than k+1 sons, if so the c-decomposition is removed.

PROCEDURE type-fork(k) (input S: v-list k: integer ($k > 0$))

{precedence=15}

{specification: generates a set of graphs, such that a graph belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) its edges are constructed between the first vertex and k other vertices in S}

```
BEGIN
case S of
x: failure
x0x1x2..xt (t< k): output none
x0x1x2..xk: Routput < x0x1x2..xk, star >
x0x1x2..xt ($t > k$): Routput OR!$\delta_{x_i1,x_i2,\cdots x_ik}$< x0x$_i$1x$_i$2..x$_i$$k$, star >
w: output < w, fork > [fork(k)]
END.
```

Remark: the constraint procedure, fork(k), will check each corresponding sub c-decomposition of the class to see if it will generate no graph that has all edges constructed from the first vertex of the v-list to k other vertices in the v-list, if so the c-decomposition is removed.

PROCEDURE type-cubic (input S: v-list)

{precedence=16}

{specification: generates a set of cubic graphs, such that a cubic graph belongs to the set if and only if it has the same vertex set as that of S}

```
BEGIN
case S of
x: failure
```

```
x0x1x2: failure

x0x1x2x3: output < x0x1x2x3, clique >

x0x1x2..xt (t >3): output AND(< x0x1x2..$x_k$, fork(3) > (< x1x2..$x_k$x0, fork(3) >

......

(< $x_k$x1..$x_{k-2}x_{k-1}$, fork(3) >)[degree=3]

w: output < w, cubic >

END.
```

PROCEDURE type-regular(k) (input S: v-list, k: integer)

{precedence=17}

{specification: generates a set of k-regular graphs, such that a k-regular graph belongs to the set if and only if it has the same vertex set as that of S}

```
BEGIN

case S of

x0x1x2..xt (t< k): failure

x0x1x2..xk: output < x0x1x2..xk, clique >

x0x1x2..xt (t >k): output AND(< x0x1x2..xk, fork(k) > < x1x2..xkx0, fork(k) >

......

< xkx1..xk-2xk-1, fork(k) >)[degree=k]

w: output < w, regular(k) >

END.
```

Remark: the constraint procedure, degree=k, will check each corresponding sub c-decomposition of the class to see if it will generate no graph that has all vertices with degree k, if so the c-decomposition is removed.

PROCEDURE type-connected (input S: v-list, status=basic)

{precedence=18}

{specification: generates a set of connected graphs, such that a connected graph belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) it contains a spanning tree and the left most vertex of S is the root of the the spanning tree, any other vertex of S has a parent to its left in S}

```
BEGIN
case S of
x: none
y: output < y, connected >
x1x2: output < x1x2 >
wx: Routput OR(<w, connected>, <xw, fork >)
wy: output < wy, connected >
END.
```

PROCEDURE type-connected (input S: v-list, status=extended)

{precedence=18}

{specification: generates a set of connected graphs, such that a connected graph belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) it contains a spanning tree and the left most vertex of S is the root of the the spanning tree, any other vertex of S has a parent to its left in S}

```
BEGIN
case S of
x: none
y: output < y, connected >
x1x2: output < x1x2 >
wx: Routput OR(< w, connected >, < xw, fork >)
```

```
wy: OR(output < wy, connected >,
Routput OR!$\alpha_x$(w)(AND(< w, connected >, < y, connected >, < $xy$. e >),
AND(<w, connected>, <y, fork0>, <$xy$, star>))
END.
```

PROCEDURE type-disconnected (input S: v-list, status: [basic, extended])

{precedence=19}

{specification: generates a set of disconnected graphs, such that a disconnected graph belongs to the set if and only if

(1) it has the same vertex set as that of S

(2) there exists a one-to-one correspondence between the connected components and the v-list segments which form a partition of S. And each connected component of the graph contains a spanning tree. The left most vertex of the segments of S corresponding to the connected component is the root of the tree, and any other vertex of the segment has a parent to its left in the segment}

```
BEGIN
case S of
x: output failure
x1x2: output none
w1'yw2': output < S, disconnected >
wx: Routput OR(< w, connected >,
OR!$\gamma_{w1,w2}$(wx)OR( AND(< $w$1, connected >, < $w$2, connected >), AND(< $w$1, connected >, < $w$2, disconnected >),
AND(< $w$1, disconnected >, < $w$2, connected >), AND(< $w$1, disconnected >, < $w$2, disconnected >))
END.
```

PROCEDURE type-nonseparable (input S: v-list, status:[basic, extended])

{precedence=21}

{specification: generates a set of all nonseparable graphs containing a spanning tree that can be generated by calling the T-procedure on S}

```
BEGIN
case S of
x: none
x, x1x2: output none
y: output < y, connected > [no-cut-vertex]
x1x2x3: Routput < x1x2x3, cycle >
w: Routput < w, connected > [no-cut-vertex]
END.
```

Remark: the constraint procedure, [no-cut-vertex], will check each corresponding sub c-decomposition of the class to see if it generates no graph that has no cut vertex, if so the c-decomposition is removed.

PROCEDURE type-bipartite (input S1-S2: (v-list)-(v-list))

{precedence=22}

{specification: generates a set of all bipartite graphs with two mutually isolated vertex sets implied in two v-lists S1 and S2}

```
BEGIN
case S1-S2 of
x1-x2: output OR(< x1x2 >, none)
x-w, w-x: Routput < xw, fork0 >
y1-y2: output < y1-y2, bipartite >
xw1-w2, w1x-w2: AND(Routput < w1-w2, bipartite >,
output < xw2, fork0 >)
w1-xw2, w1-w2x: AND(Routput < w1-w2, bipartite >,
output < xw1, fork0 >)
y1w1-y2w2: AND(output < y1-y2, bipartite >,
Routput <y1-w2, bipartite>, <w1-y2, bipartite>, <w1-w2, bipartite>)
END.
PROCEDURE type-complete-bip (input S1-S2: v-list)
{precedence=23}
{specification: generates a set of all complete-bipartite graphs with two mutually isolated vertex
sets implied in two v-lists S1 and S2}
BEGIN
case S1-S2 of
x1-x2: output < x1x2 >
x-w, w-x: Routput < xw, star >
y1-y2: output < y1-y2, complete-bip >
xw1-w2, w1x-w2: AND(Routput < w1-w2, complete-bip >,
output < xw2, star >)
w1-xw2, w1-w2x: AND(Routput < w1-w2, complete-bip >,
output < xw1, star >)
y1w1-y2w2: AND(output < y1-y2, complete-bip >,
Routput < y1-w2, complete-bip >,
< w1-y2, complete-bip >, < w1-w2, complete-bip >)
END.
PROCEDURE type-symple (input S: v-list, status:[basic, extended])
{precedence=24}
{specification: generates a set of all simple graphs }
BEGIN
case S of
x: none
x1x2: output OR (< x1x2>, none)
```

```
w: Routput OR(< w, connected >, < w, disconnected >)
END.
```

4.3 The Features and The Correctness of T-Procedures

The important features of all T-procedures are given as follows:

1. If the input is a concrete vertex list (no unknowns), then all output classes are edges. Then each sub c-decomposition as well as each c-decomposition is a set of graphs which can be described by vertex sets and edge sets.

2. Each T-procedure accepts a v-list in any vertex-unknown configuration, and terminates its output in a finite number of steps.

3. Each T-procedure implements its specifications correctly for any inputs at any stage of recursion.

4. No non-trivial decomposition can be further made by OR operations (note that this feature is included for efficiency, it has no effect on correctness). For example, decomposing < S, T > into OR(< S, T >, AND(< S, T >, < S, T1 >)) is trivial if T subsumes T1.

5. Both basic and extended c-decompositions (see type-tree) are correct. But in basic status no 'special case' is explicitly included (by OR), i.e. no 'redundancy'. However, in extended status maximal number of 'special cases' are explicitly included (whether or not the number of special cases included is maximal has no effect on correctness). For example, for the case of S = xy in procedure type-tree, the set of graphs in < xy, tree > properly includes all graphs in AND(< y, tree >, < xy, join >).

These features can be verified directly from definitions or by using mathematical induction on elements of S in a straightforward manner. As examples, the following gives proofs for feature 3 on cycle-type and tree-type. The proofs for features 1 and 2 are apparent. The proofs for features 4 and 5 can be done by exhaustive checking.

4.3.1 A Proof for Cycle-Type T-Procedure

Fact: feature 3 is true for cycle-type (see T-procedure cycle-type).

Proof:

If S contains one vertex then it is represented by either **x** or **y**. The procedure finally represents the same graph, a vertex **x**.

If S contains two vertices, then it can only be represented in one of the forms: **x1x2**, **y**, **xy**, **yx**, **y1y2**. The output for each of these is a single edge. For example, in the case of S of **y1y2** replacing **y1** by **x1** and replacing **y2** by **x2**, the output:

AND(< **y1y2**, 1 >, < **y1**, 1 >, < **y2**, 1 > < **y2y1**, 1 >)

will be

AND(< **x1x2**, 1 >, < **x1**, 1 >, < **x2**, 1 > < **x2x1**, 1 >) = < **x1x2** >

If S contains three vertices, then it can only be represented in one of the forms: **y**, **xy**, **yx**, **y1y2**, **z1wz2**. There is finite number of ways to assign the patterns for each case. It is easy to check that all of them result in the same single output graph with three edges: < **x1x2** >, < **x2x3** >, < **x3x1** >. For example, let **z1** be **x1**, **w** be **x2**, **z3** be **x3** then the output:

Routput AND(< **z1w**, 1 >, < **wz2**, 1 >, < **z2z1**, 1 >)

will be:

Routput AND(< **x1x2**, 1 >, < **x2x3**, 1 >, < **x3x1**, 1 >) = < **x1x2** >, < **x2x3** >, < **x3x1** >.

Let S be $x_1x_2 \cdots x_n$ ($n \geq 3$) then S can be assigned for one of the following forms:

y, **xy**, **yx**, **y1y2**, **z1wz2**.

We show the proof for one of the forms (proofs for the others are similar):

Let S be assigned for $\mathbf{x}_1\mathbf{w}\mathbf{x}_n$

by the T-procedure it has output:

Routput AND(< $\mathbf{x}_1\mathbf{w}$, 1 >, < $\mathbf{wz}_n$, 1 >, < $\mathbf{x}_n\mathbf{x}_1$, 1 >)

here **w** contains n-2 vertices, assume that the correctness of the procedures with lower precedences has been proved, so each operand of AND will generate a sequencially connected path. For example, z1w will generate a path sequencially connecting vertices from x_2 to $x_(n-1)$. Thus the final output is the cycle circularly connecting vertices in S.

QED

4.3.2 A Proof for Tree-Type T-Procedure

Fact: the feature 3 is true for tree-type (see T-procedure tree-type).

Proof:

We first prove that any graph generated by the procedure is a tree described by the specification. We prove the claim by considering the patterns of S as follows:

1. S contains no unknown and has vertex, say **x**, at the right-most end of it. We use mathematical induction on n, the number of vertices S contains.

   (a) if n = 0, 1 and 2, the proofs are obvious.

   (b) assume that the claims are true when the number of vertices is less than n. Consider the case when the number of vertices is n, the output:

   wx: Routput AND(< **w**, tree >, < **xw**, e >)

   contains an AND of two classes. The first class has v-list with n_1 number of vertices and has type tree, thus being a tree by induction hypothesis. The second class: < **xw**, e >, by the correctness of T-procedures with lower precedences, represents only a single edge from the vertex **x** to one of vertices of **w**, i.e. to a vertex to its left in S. By the definitions of AND, each graph in the output is constructed by such a tree and such an edge. Thus the claim has been proved.

2. S has an unknown, say **y**. Examine all outputs of S's all possible patterns, it is clear that:

(a) either the same proof can be found when **y** is seen as instantiated vertex list (e.g. for S=**yx** case), or

(b) the output is < S, tree >, for which the proof can be converted to that for the non-unknown case, when **y**'s are seen as instantiated vertex lists, or

(c) the output class represents an AND with two classes, each of them has type tree and has shorter v-list (referred to special cases). By the induction hypothesis they are two trees satisfying the specification. because these two trees have only one common vertex so the union of them is a tree which satisfies the specification too.

We now prove that any tree given by the description can be generated by the procedure:

Since that a given tree is a concrete tree (i.e. no unknown in it), it is always possible to construct a v-list starting from the root of the tree and such that each vertex is a node of the tree and each node has a parent to its left in S. A level list of a tree defined in chapter 7 indicates the way to construct such a v-list. Considering the case of S: **wx** where **w** is a list of concrete vertices, it is easy to see that the output should contain the given tree since type-e generates a set of all possible edges from x to a vretex to its left in w.

QED

CHAPTER 5

Proof Process

Equipped with the algebraic and procedural knowledge representation scheme, GC can organize its proof mechanism as an **and/or** [Ni] proof tree with forward chaining search guidance. To produce a proof at each node GC employs a hierarchical proof processing. This chapter presents the complete local (at a node of a proof tree) proof process and associated knowledge handling.

5.1 Background Knowledge Formalization

In order to generate a decomposition of a class producing a simplified class expression in normal form, both the algebraic and procedural processes manipulate v-lists, graph types, and constraints. Such a manipulation is well guided under their background knowledge.

5.1.1 Conceptual Frames

The system GC provides a frame-based representation structure [Mi] to store and update the important background knowledge on graph concepts. A graph concept frame consists of the following slots (the correctness of the knowledge is proved mathematically):

- The pointer to its T-procedure (Type procedure) for recursively decomposing graphs of the type in a predefined order.
- Subsumption relation with other types, e.g. type path is subsumed by type tree, type tree is subsumed by type connected, type star is subsumed by type tree and so on.
- The indices of inference rules to rewrite formulas of the type with regard to some constraints used.

- The indices of axioms available to the type.
- Special algebraic properties, e.g. type clique has property:

  $COMP(< S, clique >) =< S, 0 >$.
- The pointers to algorithms available from graph theory, e.g the algorithm for finding all cycles from a concrete graph is available for type cycle.
- The extremal-cases of graphs in the type, e.g. for type cycle non-degenerated extremal-case is triangle.
- The equivalent types, e.g. type e is equivalent to type join.

Note that the subsumption relations must be proved to be applicable to any classes with the same v-lists based on the specification of graph type, for instance, < S, tree > subsumes < S, 1 >. The information in a slot of a frame can be updated either through human interaction or automatically done by the system after a theorem is proved.

5.1.2 Representation about Constraints

In addition to v-list and type, constraints are another kind of parameters to define a graph class, i.e. each graph generated from an unconstrained class formula must pass the corresponding constraint checking. Unsatisfied candidates are filtered out from the class. The constraints are handled as background knowledge. The reasoning processors related to constraints are a set of predefined procedures. The constraints are divided into the following two types:

(1) ME constraints It is a kind of constraints on the mutual relations between two v-list elements. For example:

x[= x1] : means ' x equal to x1' and the predicate symbol is '='.

x[not= x1] : means 'x does not equal to x1' and the predicate symbol is 'not='.

y1[in y] : means 'every vertex in y1 is in y' and the predicate symbol is 'in'.

y1[not-in y] : means 'every vertex in y1 is not in y' and the predicate symbol is 'not-in'.

y2[not= y1] : means 'y2 and y1 are not the same unknowns' and the predicate symbol is 'not='.

< S, T >[with < $x1x2$ >] : means 'each graph in < S, T > contains edge < $x1x2$ >' and the predicate symbol is 'with'.

(2) non-ME constraints Any other kind of constraints. For example:

x[degree> k] : means 'x has degree greater than k' and the predicate symbol is 'degree>'.

x[cut-vertex] : means 'x is a cut vertex' and the predicate symbol is 'cut-vertex'.

< S, T >[no-cut-vertex] : means 'each graph in < S, T > has no cut-vertex' and the predicate symbol is 'no-cut-vertex'.

To deal with ME constraints GC has a decision procedure called ME checking procedure, which is described below:

A set of v-list elements which are pairwise disjoint is called an ME-set (Mutual Excluded set). Based on the definition of a v-list (all vertices within a v-list are pairwise disjoint) and the semantic meaning of ME constraints, it can be proved that there exists a procedure which decides if two v-list elements are pairwise disjoint. This procedure is ME checking procedure. To simplify the discussion on this decision procedure it is necessary to have the following definitions:

- Definition 1. Direct-in(y) is a set of all v-list elements (vertices and unknowns) followed by constraints of the form [in y] or form [= y].

  {remark: the set consists of all elements, each of which has either 'in' or '=' constraint relations with y, thus is in y}

- Definition 2. If w1yw2 is a v-list then phy_out(y) is a set of all v-list elements contained in w1 and w2.

{remark: the set consists of all elements, each of which is in the same v-list as y is, thus is not in y}

- Definition 3. Sym_out(y) is a set of all v-list elements which are followed by constraints of the form [not_in y].

  {remark: the set consists of all elements, each of which has 'not-in y' constraint thus is not in y}

- Definition 4. Direct_out(y) is the union of phy_out(y) and sym_out(y).

  {remark: the set consists of all elements, each of which satisfies constraint 'not-in(y)' 'directly', and any element that satisfies the constraint 'directly' is in the set}

- Definition 5. Equal(x) is a set of vertices defined by:

  (a) initially equal(x) contains x;

  (b) if x1 is in equal(x) then any x2 which satisfies the constraint 'x2[= x1]' or 'x1[= x2]' is in equal(x);

  (c) repeat step 2 until no more such an x1 can be found.

  {remark: the set consists of all elements, each of which satisfies '=x' either 'directly' or 'indirectly' and any element that satisfies 'equal to x' is in the set (since exhaustive checking is used)}

- Definition 6. Unequal(x) is a set of vertices defined by:

  (a) initially unequal(x) is empty;

  (b) if x1 together with a vertex in equal(x) is argument of a 'not=' constraint, then x1 is in unequal(x);

  (c) if x1 is in unequal(x) then any x2 in equal(x1) is in unequal(x);

  (d) repeat steps 2 and 3 until no such x1 can be found.

  {remark: the set consists of all elements, each of which satisfies 'not=x' either 'directly' or 'indirectly', and any element that satisfies 'unequal to x' is in the set (since exhaustive checking is used)}

- Definition 7. Equal(y) is a set of unknowns defined by:

  (a) initially equal(y) contains y;

  (b) if y1 is in equal(y) then any y2 which satisfies the constraint 'y2[= y1]' or 'y1[= y2]' is in equal(y);

  (c) repeat step 2 until no more such a y1 can be found.

  {remark: the set consists of all elements, each of which satisfies '=y' either 'directly' or 'indirectly' and any element that satisfies 'equal to y' is in the set (since exhaustive checking is used)}

- Definition 8. Logic_in(y) is a set of v-list elements defined by:

  (a) initially logic_in(y) contains y;

  (b) if y1 is in logic_in(y) then any v-list element z which is in direct_in(y1) or in equal(y1) is in logic-in(y);

  (c) if x1 is in logic_in(y) then any v-list element x which is in equal(x1) is in logic-in(y);

  (d) repeat steps 2 and 3 until no more such a y1 and x1 can be found.

  {remark: the set consists of all elements, each of which satisfies in(y) either 'directly' or 'indirectly' and any element that satisfies 'in(y)' is in the set (since exhaustive checking is used)}

- Definition 9. Setequal(y) (set equivalent) is a set of unknowns defined by:

  (a) initially setequal(y) contains equal(y);

  (b) if y1 is in setequal(y), then if any y2 satisfies:

     y2 is in logic-in(y1) and y1 is in logic-in(y2)

     then y2 is in setequal(y);

(c) repeat steps 2 and 3 until no more such a y1 can be found.

{remark: the set consists of all elements, each of which satisfies 'having the same vertex set' either 'directly' or 'indirectly' and any element that satisfies such a condition is in the set (*since exhaustive checking is used*)}

- Definition 10. Logic_out(y) is a set of v-list elements defined by:

  (a) initially logic_out(y) is equal to direct_out(y);

  (b) if y1 is in logic_out(y2) and y2 is in setequal(y) then any v-list element z which is in logic_in(y1) is in logic_out(y);

  (c) repeat step 2 until no more such y1 can be found.

  {remark: the set consists of all elements, each of which satisfies 'not-in y' either 'directly' or 'indirectly' and any element that satisfies such a condition is in the set (since exhaustive checking is used)}

Algorithm: ME checking on v-list elements z1 and z2

(a) If z1 and z2 are both x type let z1=x1, z2=x2, and if x1 is in unequal(x2), then output ME and stop for x1 x2, else go to 4.

(b) If z1 is x type and z2 is y type let z1=x1, z2=y2, and if x1 is in logic-out(y2) then output ME and stop for x1 y2, else go to 4.

(c) If z1 and z2 are both y type let z1=y1, z2=y2, and if y1 is in logic-out(y2), then output ME and stop for y1 y2.

(d) Output NOT ME for z1 and z2.

Proof of the correctness of the procedure Notice that the proofs for above definition remarks (i.e. conclusions) can be simply done, and the correctness of the procedure can be proved directly based on these remarks.

5.2 Representation of Theorems

Using the class formula and the algebraic operations a theorem representation in graph theory can be divided into two parts:

(a) assertion part: represented by a class expression.

(b) goal part: represented by a class expression.

The meaning of a theorem in graph theory often takes one of the following eight (called Form-1 to Form-8):

(a) Form-1. Each graph in an assertion set is of type T with property (constraint) P.

(b) Form-2. No graph in an assertion is of type T.

(c) Form-3. The set of graphs in an assertion is equivalent to another set of graphs.

(d) Form-4. Each graph in an assertion has a subgraph of type T with property P.

(e) Form-5. Each graph in an assertion has no subgraph of type T.

(f) Form-6. Each graph in an assertion has an induced subgraph of type T with property P.

(g) Form-7. Each graph in an assertion has no induced subgraph of type T.

(h) Form-8. The set of graphs in an assertion is not equivalent to another set of graphs.

The following examples show the formula representations:

Example 1

The assertion of the theorem:

'Any two distinct cycles having a common edge e contain a cycle not passing e' is:

AND(< x1x2y1, cycle >, < x1x2y2[not= y1], cycle >)

The goal of the theorem is as follows:

GOAL-FORM: 4

$\forall$ G (i.e for any graph in the assertion) $\exists$ y, such that

< y, cycle >[not-with <x1x2>] partially subsumes G,

here 'partially subsumes G' means 'subsumes a subgraph of G'.

Remark: the common edge is $< x1x2 >$.

Example 2

The assertion of the theorem:

'If cycle (closed path) C1 contains edges e1, e2, and cycle C2 contains edges e2, e3, then there exists a cycle that contains e1 and e3' is:

AND(< x1y1x2, cycle >[with $< x3x4 >$], < x1y2x2, cycle >[with $< x5x6 >$])

Remark: $< x1x2 >$ represents e2, $< x3x4 >$ represents e1, and $< x5x6 >$ represents e3.

The goal of the theorem is:

GOAL-FORM: 4

$\forall$ G $\exists$ y, such that

< y, cycle >[with $< x3x4 >$][with $< x5x6 >$] partially subsumes G.

Example 3

The statement: 'The union of a tree and a path is a tree if they have only a single common vertex' has

FACT (or assertion):

AND(< y1xy2, tree >, < xy3[not-in y1, y2], 1 >)

CONJECTURE : such a graph is a tree (GOAL-FORM: 1)

$\forall$ G $\exists$ y, such that

< y, tree> subsumes G.

5.3 Proof of Existence

Any one of the above eight theorem forms basically can be seen as a claim of existence or non-existence, because it can be seen that either there exists a subset (may not be proper subset) of the goal set (a subset of the universal graph set) which subsumes the assertion graph set, or there exists no subset of the goal which subsumes the assertion set. This research provides a constructive approach to prove existence for a theorem or a subtheorem based mainly on the following ideas:

(a) Use c-algebraic representation to make next candidate for subset g_1 of the goal set g based on exhaustive search or heuristic guidance.

(b) Try to prove that g_1 subsumes (or partially; see later) the assertion graph set using algebraic manipulation and type decomposition.

(c) Go to step 1 repeatedly until step 2 succeeds (proved) or fails to find a candidate (unproved).

5.3.1 Subsumption Primitives

The approach has a simple and efficient method based on pattern matching to prove the subsumption relation between graph class expressions. This subsection discusses the subsumption primitives. To simplify the discussion assume only AND/OR operations involved. For cases involving other kinds of operations the discussions are similar. The following shows subsumption proof primitives of the class algebra:

(a) To prove that a class c2: < S2, T2 > subsumes class c1: < S1, T1 >, it is necessary to convert c1 and c2 to unconstrained classes, and then check whether

   i. their character strings are exactly the same , or

   ii. S2 is the same as S1, and that T2 subsumes T1 can be found in their type frames.

(b) To prove set subsumption (or partial subsumption) between two sub c-decompositions, say g_2 subsumes (or partially subsumes) g_1, we need to find a 1-to-1 correspondence between all classes of g_2 and all classes of g_1 (or part of classes of g_1), and then prove that g_2's class subsumes g_1's class for each correspondence.

(c) To prove set subsumption (or partial subsumption) between two c-decompositions, say N_2 subsumes N_1, we need to prove that each sub c-decomposition of N1 is subsumed (or partially subsumed) by a sub c-decomposition of N2.

5.3.2 Goal-Assertion Subsumption Method

The Goal-Assertion Subsumption Method (GASM) is the most powerful and general proving process for subsumption relation between two AND/OR normal forms provided by this approach. The applications of GAMS on different theorem forms are given below:

5.3.2.1 GASM on Form-1 Theorems

GASM as applied to any theorem with form-1 goal is described in six steps below (assume that the type stated in the goal is T):

(a) Invoke ME checking to determine an ME-set containing all vertices in assertion, and denote the ME-set as R.

(b) Construct a v-list, say S, by permuting the elements of R.

(c) Expand the assertion to obtain normal form N1.

(d) Expand the class $< S, T >$ to obtain normal form N2 for the goal.

(e) Prove that the normal form N2 subsumes the normal form N1. If N2 contains type-1 classes then it may be necessary to try various permutations of reversals of unknowns in these classes using the rule that $< yx, 1 >$ is equal to $< x\bar{y}, 1 >$.

(f) If step 5 fails then an exhaustive or guided search on the permutation of S will be performed to construct a new S and then go back to step 2.

It is easy to verify that if the subsumption relation is proved and constraint checking is passed, the theorem of form-1 is proved. Apparently, the cardinality of R should be fairly small, fortunately it is usually true for many theorems represented using unknowns. A technique dealing with large cardinality is under study, for example, subsumption checking might be replaced by an isomorphism checking using domain specific knowledge, thus permutation will not be needed.

5.3.2.2 GASM on Form-3 Theorems

GASM as applied to any theorem with form-3 goal is described below (assume that the type stated in goal is T):

(a) Expand the assertion to obtain normal form N1.

(b) Expand the goal to obtain normal form N2.

(c) Prove that the normal form N2 subsumes the normal form N1. If N2 contains type-1 classes then it may be necessary to try various permutations of reversals of unknowns in these classes using the rule that $< yx, 1 >$ is equal to $< x\bar{y}, 1 >$.

(d) Prove that the normal form N1 subsumes the normal form N2. If N1 contains type-1 classes then it may be necessary to try various permutations of reversals of unknowns in these classes.

5.3.2.3 GASM on Form-4 Theorems

GASM as applied to any theorem with form-4 goal is described below (assume that the type stated in goal is T):

(a) The same as that for form-1 except step 4, that is replaced by the following: prove that the normal form N2 partially subsumes the normal form N1.

5.3.2.4 GASM on Form-6 Theorems

GASM as applied to any theorem with form-6 goal is described below (assume that the type stated in goal is T):

(a) The same as that for form-4.

(b) Partition N1's classes into two parts, the first part consists of all classes subsumed by N2. Prove that the second part has no edge with two end vertices on the first part.

An equivalent GASM on form-6 theorems, which uses stronger condition but often simplifies the proof process, is:

(a) The same as that for form-4.

(b) Partition N1's classes into two parts, the first part consists of all classes subsumed by N2.

(c) Partition the second part into two subparts, the first subpart consists of all classes, each of which has no more than one vertex that is also in the first part. Prove the second subpart has no edge with two end vertices on the first part.

(d) If the second subpart is empty then proved, else prove that the second subpart has no edge with two end vertices on the first part.

5.3.2.5 GASM on Theorems with Form 2, 5, 7 or 8

The current solution for theorems with these forms is:

(a) if there is unknowns involved then stop (unproved).

(b) Prove form 2, 5, 7, 8 in the same manner as form 1, 4, 6 and 3 respectively.

(c) If the corresponding proof succeeds then output 'proved', otherwise output 'disproved'.

5.4 Leveled Methods

The approach adopts a technique used in Boyer-Moore theorem prover [BM]: applying different proving methods leveled from 'simple' to 'complex' (i.e. only when it fails to prove at level 1 the level 2 is reached). In GC proving methods are divided into four levels.

5.4.1 Level 1

Level 1 is restricted to be applied for concrete graphs; in other words, the assertion should not contain an unknown. The system is able to utilize most of algorithms existing from graph theory for proving processes at this level, such as for a concrete graph:

(a) Finding all paths between any two vertices.

(b) Finding all cycles.

(c) Finding a spanning tree rooted at a specified vertex for a connected graph.

(d) Finding a maximal clique.

(e) Finding all cut vertices.

(f) Finding a cut set.

(g) Compute the degree of a vertex.

(h) Compute the maximum matching.

(i) Compute the maximum matching in a bipartite graph.

5.4.2 Level 2

Level 2 is devoted to the use of a set of decision procedures with input containing unknowns. The system has several general and useful decision procedures such as:

(a) Finding all paths on class expression containing only type 1, type cycle and type clique.

(b) Finding all cycles on a class expression containing only type 1 and type cycle.

5.4.3 Level 3

Another proof method that is closely related to the GASM is using a proved theorem as an axiom to help prove. This is applied at level 3. A proved theorem is judged. If the generality is appreciated, then it will be installed into the axiom base where its assertion (with variable elements) is represented in normal form. Once a theorem's goal matches an axiom's goal, the assertion of the axiom is instantiated, and the further process will be the same as that of the Goal-Assertion Subsumption Method.

5.4.4 Level 4

This level is devoted to the GASM.

CHAPTER 6

Proof Search: Case-Split-Based Inferencing

The graph classification can also be done by case-split-based inferencing rather than by c-decomposition. A set of inference rules is designed to rewrite a class into a logical expression (an and/or form but not to be confused with AND/OR as defined in c-algebra) of several subclasses in terms of constraint predicate language. In order to guide a proof search this kind of process is applied after a proof (or a sub proof) fails in the root (or a node).

6.1 Inference Rule and Case Splitting

The case splitting caused by meta inferencing can be seen from the example below:

The unknown y1[not= y2] shown in example 1 in 5.1 can be split into three cases by:

INFERENCE RULE 1:

y1[not= y2]

---

remove [not= y2] and generate

case 1: y′1xy3 substitutes y1, y′1x substitutes y2

and (conjunction) case 2: y′1xy′3 substitutes y1, y′1x1[not= x]y′4 substitutes y2

and (conjunction) case 3: y′1x substitutes y1, y′1xy3 substitutes y2.

Here case 1 represents : y1 is longer, case 3 represents : y2 is longer, and case 2 represents : two vertices are different at same position on the lists y1 and y2 (remark: the integrity of naming in rewritting is guaranteed in GC, also here operators 'and' and 'or' represent the operators to construct an and/or tree).

Substantially, the use of such a rule is a kind of class decomposition, where the newly generated classes are more constrained thus often making subtheorems simpler to be proved (recall that c-decomposition does not decompose constraints).

The rules may be used repeatedly to generate an and/or structured proof tree with the given theorem allocated at the root. Proving processes are applied at each node in an order controlled by a recursive backtracking procedure.

The next section illustraits some typical inference rules. Because typed rules (suit to a designated type) are well indexed the increasing use of this kind of rules does not rapidly cause referencing problem [BM], thus many other explicitly typed rules (e.g. type 1 or type clique) can be added.

6.2 Inference Rule Set

INFERENCE RULE 2:

$< wx, T >$[with x1]

---

case 1: $< wx1, T >$; x1 substitutes x

and case 2: $< w$[with x1]$x, T >$.

The rule splits out two cases for a type T (any type) class that contains vertex x1.

INFERENCE RULE 3:

$< w, 1 >$[with <x1x2>]

(case 1: $< w'1x1x2, 1 >$

and case 2: $< w'1x2x1, 1 >$

and case 3: < w1[with <x1x2>]x, 1 >)

or

(case 4: < x1x2w′1, 1 >

and case 5: < x2x1w′1, 1 >

and case 6: < xw′1[with <x1x2>], 1 >).

The rule splits out six cases (an and/or subtree) for a type-1 class that contains edge <x1x2> (assume the edge is newly named otherwise 12 cases (exchaning x1 and x2 for another 6 cases) will be involved).

INFERENCE RULE 4:

< wx, clique >[with < x1x2 >]

---

case 1: < w[with x1]x2, clique > ; w[with x1] substitutes w and x2 substitutes x

and case 2: < w[with < x1x2 >]x, clique >.

The rule splits out two cases for a type-clique class that contains edge < x1x2 > (assume the edge is newly named otherwise three cases will be involved).

INFERENCE RULE 5:

< xw, star >[with < x1x2 >]

---

case 1: < x1w[with x2], star > ; w[with x1] substitutes w and x2 substitutes x.

The rule splits out one case for a type-star class that contains edge < x1x2 > (assume the edge is newly named otherwise two cases will be involved).

INFERENCE RULE 6:

x

y

---

case 1: x[not-in y]

and case 2: y=y′1xy′2.

The rule splits out two cases for vertex x and unknown y regarding to their mutual relation.

INFERENCE RULE 7:

y1

y2

---

case 1: y1[not-in y2]

and case 2: y1=y′11[not-in y2]x[in y2]y′

or case 3: y1=y′11[not-in y2]x1[in y2]y′x2[in y2]y22′[not-in y2].

The rule splits out three cases for unknown **y1** and unknown **y2** regarding to their mutual relation.

INFERENCE RULE 8:

y

---

case 1: y′1x substitudes y

or case 2: xy′1 substitudes y.

The rule splits out two disjunctive cases for a nonempty unknown y which is replaced by y′1x or xy′1.

INFERENCE RULE 9:

y'

---

case 1: y' is replaced by empty

and case 2: y.

The rule splits out two disjunctive forms of a nonempty unknown **y**.

INFERENCE RULE 10:

AND(< y[universal] , connected >

< w'1x1[not= x2]w'2, T >)

---

add < x1y'1x2, 1 > to the assertion.

The rule says that in a connected graph there exists a path between two different vertices. And it is added to the assertion (note that the 'y[universal]' means 'any element in the assertion is in y').

INFERENCE RULE 11:

AND(< y[universal] , connected >

< w'1x1[not= x2]w'2, T >)

---

replace the assertion by < x1y'1x2, 1 >.

The rule says that in a connected graph there exists a path between two different vertices. It replaces the assertion and it is not a complete rule (see next section).

6.3 Soundness of The Rules and Equivalent Substitutions

The first 10 rules have the following features: the conjunction ('and') of split cases are logically equivalent to the original case; the disjunction ('or') of split cases are equivalent each other, i.e. each of them is equivalent to the original case. For example, in RULE 3: the 'and' of first three cases is equivalent to the original case. the 'and' of last three cases is also equivalent to the original case. Here equivalence refers to set (of graphs) equality between the assertion of the original theorem and the assertions which are generated by substitutions defined in each case. In other words, if the original theorem is true then each subtheorem is true, and if each subtheorem is proved the original theorem is proved. We also call the applications of rules with the above property as equivalent substitutions.

The application of rule 11 is not an equivalent substitution (called non-equivalent substitution), but all these rules are logically sound [Ni]. This is because any theorem derived from an assertion by applying these inference rules are also logically followed from that assertion.

If a reasoning system uses only equivalent substitutions then it can generate lemmas during a proof process since unproved subtheorems are lemmas. However, equivalent substitutions may not be efficient due to involving too much useless information. GC's strategy on choosing rules is based on: (1) if there is a non-equivalent substitution rule (may not be complete) available then use it to conduct efficient proof search: (2) if all major efforts fail or no such a rule is available then use equivalent substitution rules, and the generation of interesting lemmas is also considered during the proof search.

6.4 Induction Formalization

The richness of the semantic network provided by class representation enables GC to construct a mathematical induction formulation which performs induction proving on the number of vertices of a chosen unknown. This method has resulted in many improvements on GC's performance. In GC the induction formalization is obtained by the application of an inference rule which has the following process:

Let a subtheorem contain an unknown y (chosen arbitrarily or by heuristics), the following two cases (in conjunction) are generated for the subtheorem:

1. Base case: for each instance of y in the theorem substitute y by some minimal value or by null.

2. Induction step: generate a temporary axiom corresponding to the subtheorem; replace y by yx for all instances in the subtheorem to generate a new subtheorem. The temporary axiom will be used as follows:

(a) instantiate variables in the axiom,

(b) check whether the temporary axiom's assertion subsumes (or partially subsumes) the new subtheorem's assertion (by GASM),

(c) if yes then instantiate the axiom's goal and substitute (or partial substitute) the goal for each c-decomposition of the new assertion, (after the replacement the original constraints are still effective).

(d) prove the new obtained theorem based on the GASM methods.

The correctness of the formalization is proved directly by the observation that the method is just a special form of mathematical induction. The induction variable is the number of vertices contained in a v-list.

CHAPTER 7

The Implementation of System GC

GC has been built up on layer 2 of the Logical Machine Architecture (LMA) in UNIX^{TM}[1] system using programming language PASCAL. Currently the implemented version of GC is restricted to applicability to theorems involving path, cycle and tree. The c-algebra and Goal-Assertion Subsumption Method and some important mechanisms have not been implemented in the system as yet. However, the current system does reflect a special case of the approach, for example, an algebra called path-set as a subset of c-algebra has been implemented, also the decision procedure proving method (instead of leveled methods) has been implemented. With such a simple structure, GC is still able to prove many basic graph theory theorems related to path and cycle. GC consists of six well organized modules, which are interconnected to support the and/or tree search proof mechanism similar as that in chapter 6 (see Figure 1 at page 88). The following sections give a brief view of the structure of the system.

7.1 Overview of The System's Modules

7.1.1 Module 1 : The Symbolic Processor

Figure 2 at page 89 shows the structure of module 1. This module accepts an input theorem, recognizes the path-set (the set of path type classes), and separates the assertion part from the goal part of the theorem. During the symbolic passing the module assigns each distinct v-list element a unique integer (called property in LMA). Several scans of the theorem formula are designed to generate corresponding logic-in and logic-out sets thus providing the required ME knowledge.

[1]UNIX is a trademark of Bell Laboratory

7.1.2 Module 2 : The Algebraic Processor

This module manipulates a set of constrained classes (with path type) to obtain a normalized form. It also processes different kinds of constraints to obtain a simplified constraint representation. The structure of module 2 is shown in Figure 3 at page 90.

7.1.3 Module 3 : The Candidate Generator

Figure 4 at page 91 shows the structure of module 3. This module is able to generate the next candidate v-list of a goal either in exhaustive or in selective order by using the ME checking procedure. In order to choose a better candidate, some heuristic guidances have been designed, and a backtracking mechanism associated with the generator has been implemented. Backtracking must be used whenever there is no candidate that can be used to satisfy a subtheorem. In that case the tree search will be retried at its parent node.

7.1.4 Module 4 : The Proof Engine or Local Processor

Figure 5 at page 92 shows the structure of module 4. This module consists of two procedures:

(a) A procedure which finds all cycles (or paths) in the path set.

(b) A procedure which uses constraint knowledge to decides whether the cycle (or path) found is satisfactory, that is to say, it decides whether the corresponding constraints are satisfied by the cycle (or path).

7.1.5 Module 5 : Global Processor to Perform Tree Expansion

Figure 6 at page 93 shows the structure of module 5. This module consists of a heuristic guidance unit, a rule rewriting processor, and a set of inference rules based on case-splitting. The rules are chosen by the control unit. The heuristics used include knowledge about the neighborhood of an unknown, and the 'Mutual Exclusion' relations between an unknown and elements in other v-lists. The more neighbors an unknown has the higher is the priority for it to be chosen as a target for application of a related rule.

7.1.6 Module 6 : The Control Unit

This module controls input and output, generates proof information. If a proof is found this unit will generate the output proof tree. If a proof process fails it will clear up the current process records and get the system ready to start a new process. The structure of this module is shown in Figure 7 at page 94.

7.2 Descriptions of The System GC

The currently implemented system GC is mainly a stand-alone automated reasoning system in graph theory with interactive components preserved. This software is built in UNIX 4.3 BSD on VAX 11/785. It is constructed upon the Logic Machine Architecture (LMA) between layer 2 and layer 3. The partially implemented reasoning components can be used to provide constructive proofs for revealing the existence of cycles from factual input.

7.2.1 The Syntax of The Graph Language Subset

The language used in this implementation can be seen as a modified subset of the graph classification language restricted on the specific domain of graph theory. The language can be described in terms of both c-algebraic expressions and a BNF (a Backus-Naur Form) grammar. In this implementation the following BNF grammar is used to describe the language:

theorem :: assertion $\models$ goal

assertion :: g path $\models$ paths

pathname :: p chardigits | q chardigits

constraints :: constraint constraints |

constraint :: [in b-element]| [ni b-element]| [ke element]| [eq element]

constrained-elements :: constrained-element constrained-elements |

constrained-element :: a-element constraints | b-element constraints

a-element :: a chardigits

b-element :: b chardigits

chardigits :: chardigit chardigits |

chardigit :: a..z | A..Z | 0..9

goal :: c a-element | c b-element | c a-element a-element

7.2.2 The Diagrams of The Software Modules

Note: Major procedures are indicated by parenthesized names in the various blocks.

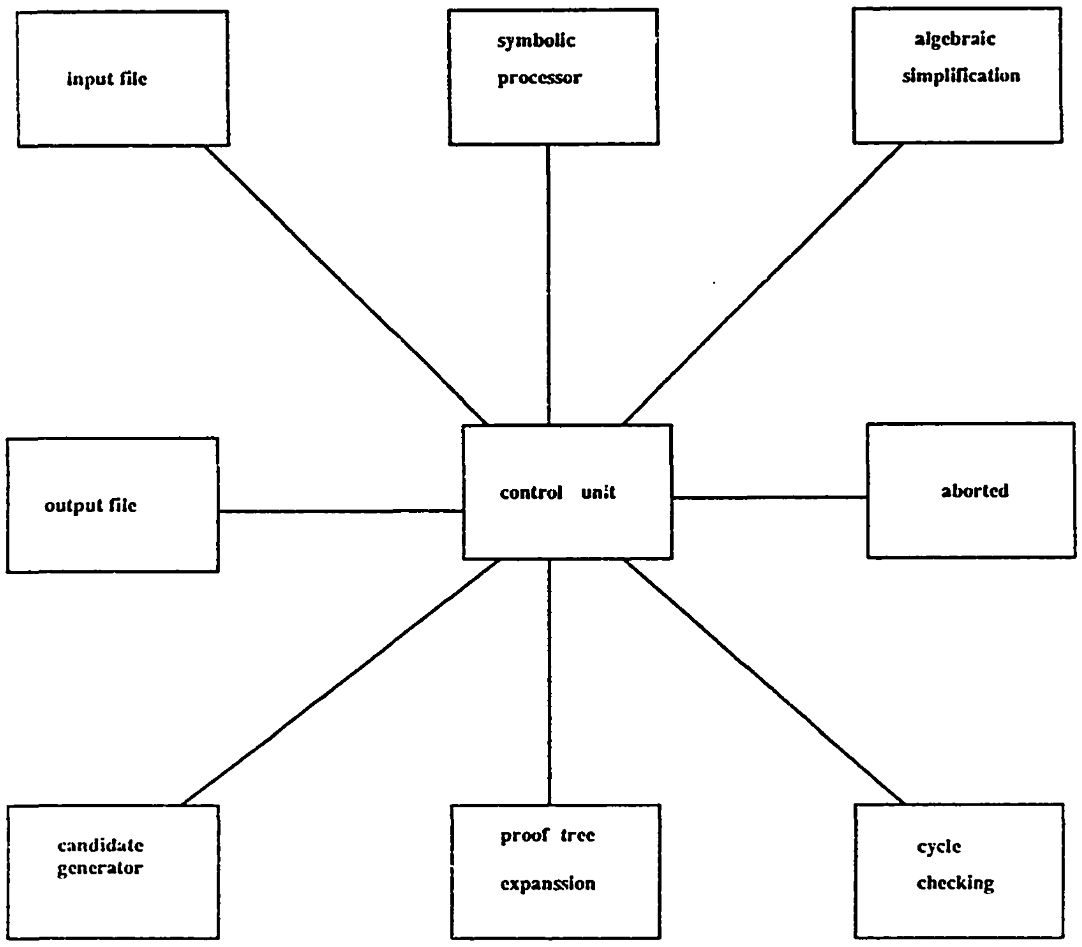

Figure 1 The overall structure of the system GC

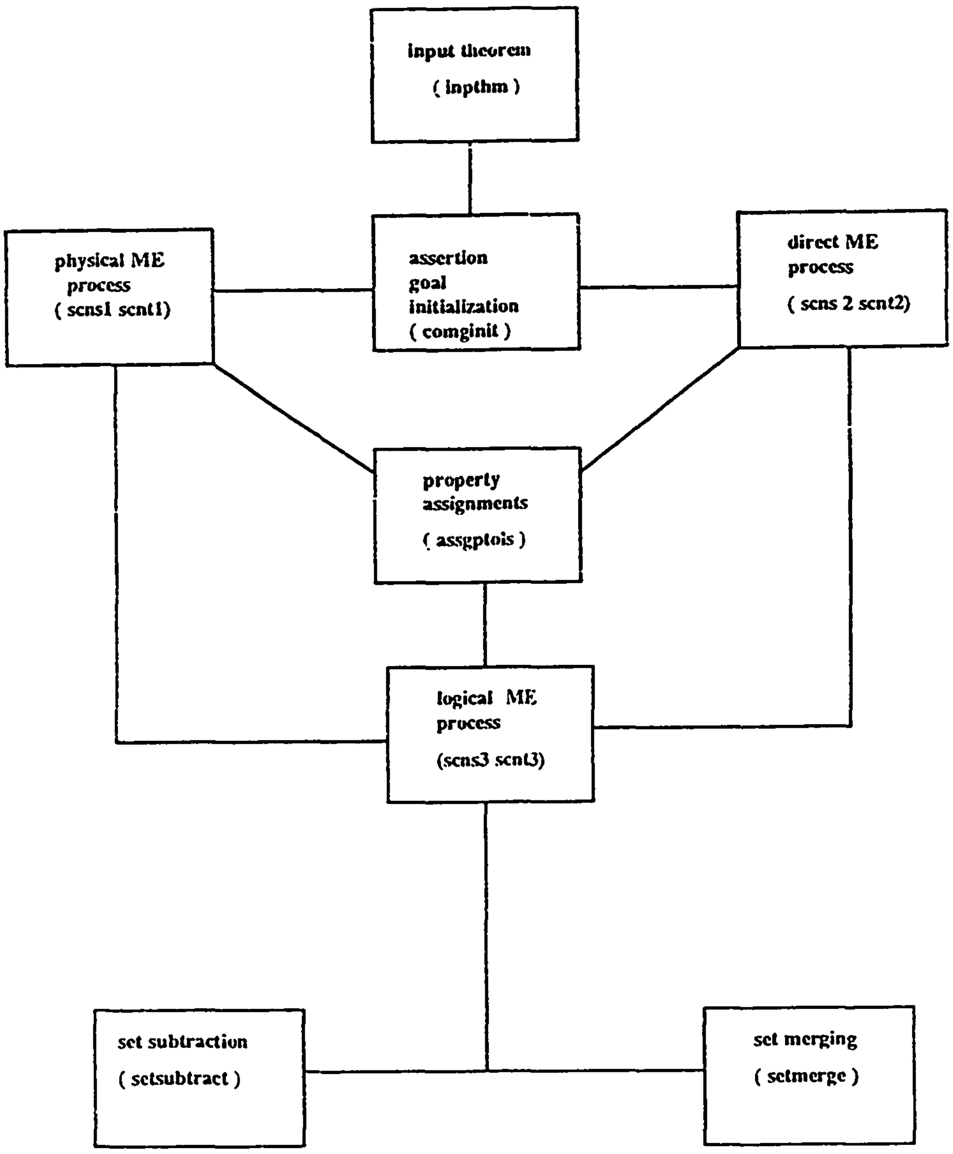

Figure 2 The structure of the symbolic processor

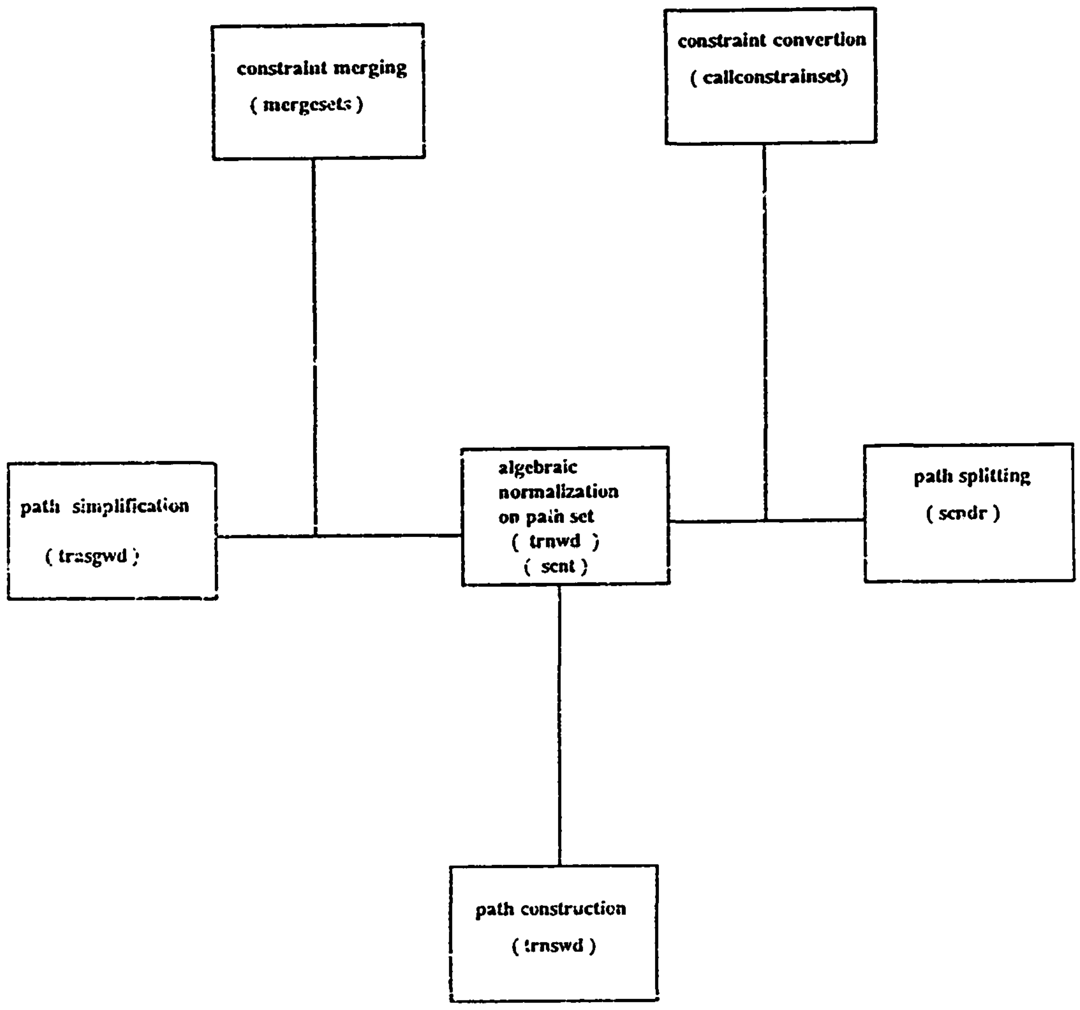

Figure 3 The structure of the algebraic processor

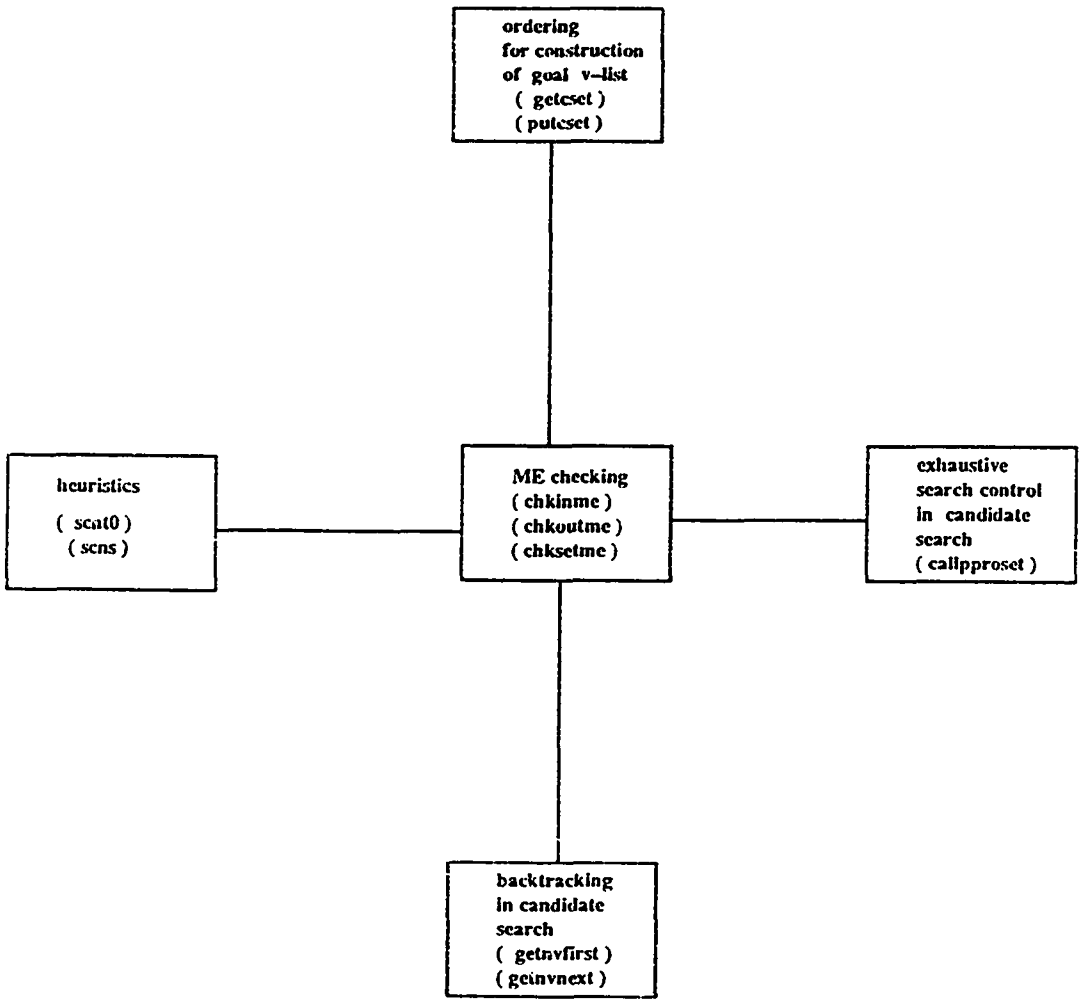

Figure 4   The structure of the goal candidate generator

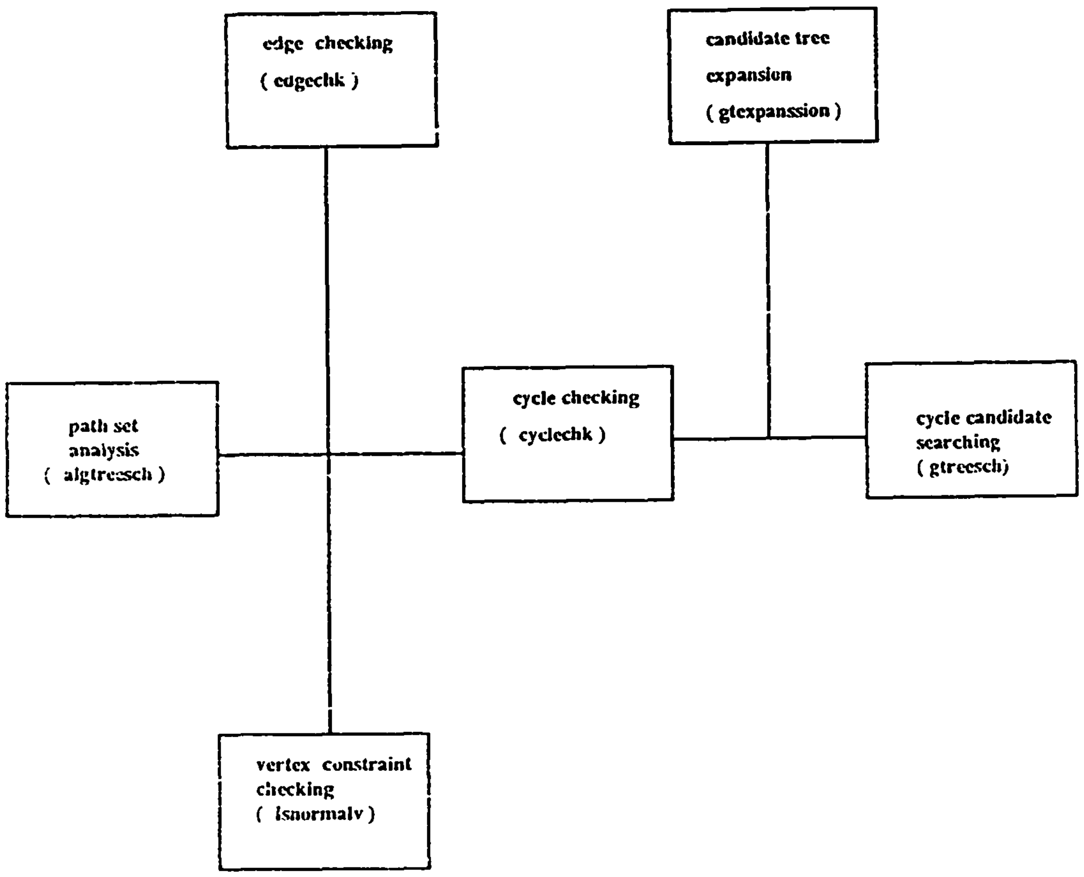

Figure 5 The structure of the local proof process (cycle checking engine)

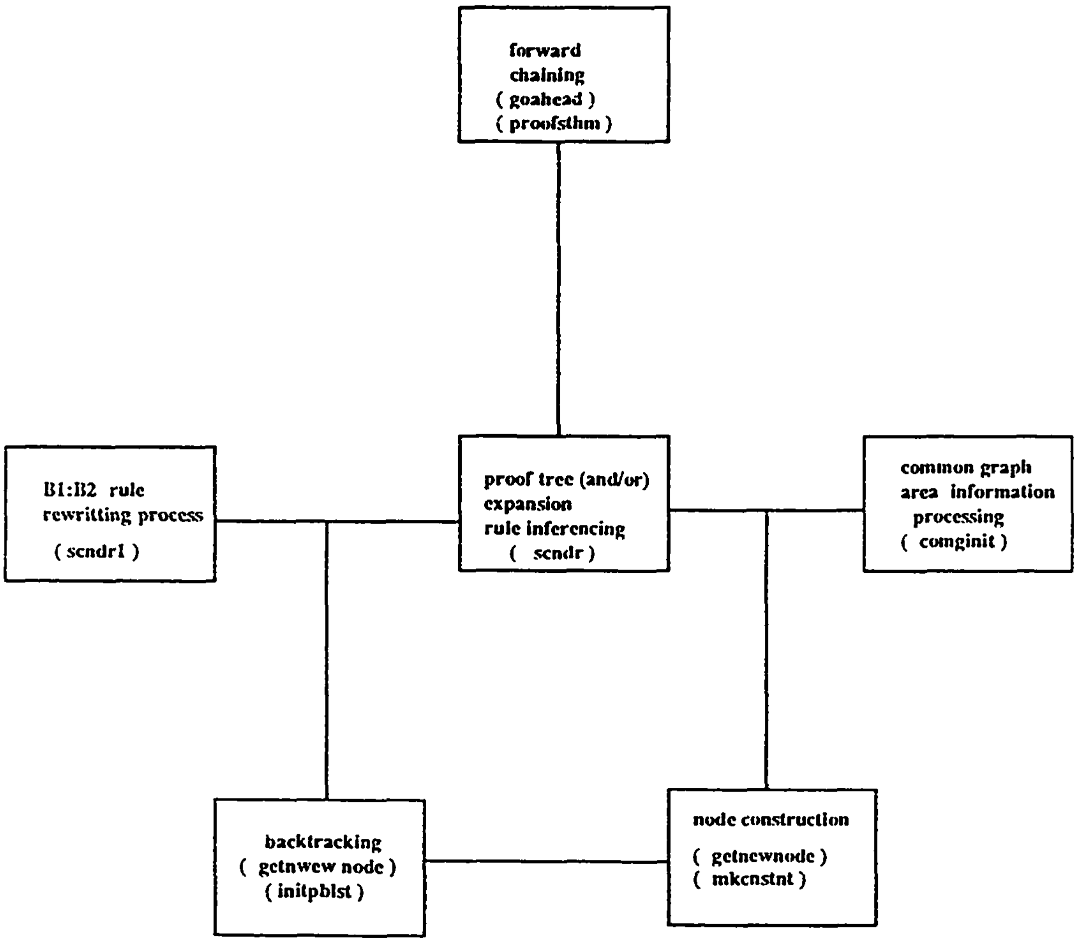
forward chaining (goahead) (proofsthm)
B1:B2 rule rewritting process (scndrl)
proof tree (and/or) expansion rule inferencing (scndr)
common graph area information processing (comginit)
backtracking (getnwew node) (initpblst)
node construction (getnewnode) (mkcnstnt)

Figure 6 The structure of the global proof process (tree expanssion)

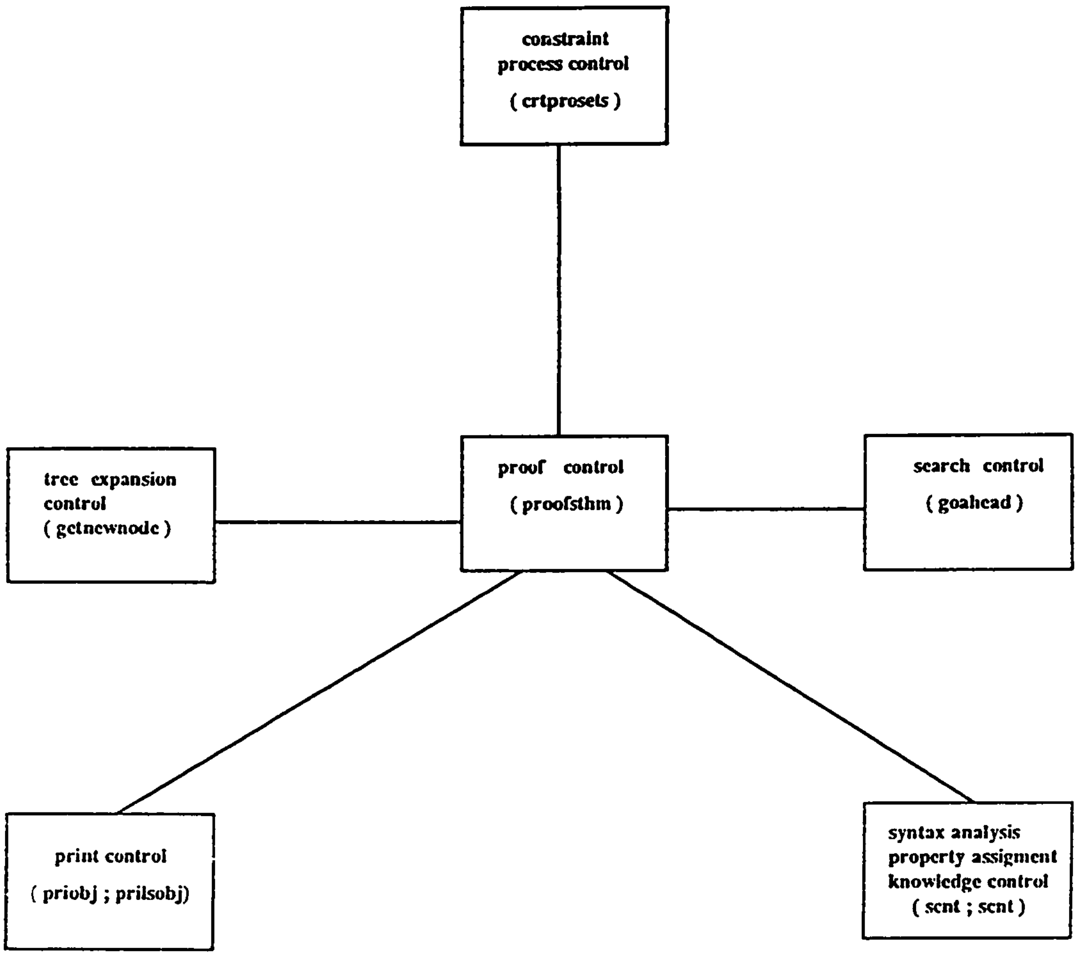
constraint process control (crtprosets)
proof control (proofsthm)
tree expansion control (getnewnode)
search control (goahead)
print control (priobj ; prilsobj)
syntax analysis property assignment knowledge control (scnt ; scnt)

Figure 7   The structure of the control unit

7.2.3 The File Structures of GC

Two kinds of files are used in constructing GC:

1. Logic Machine Architecture (LMA) system files. For example, p2linkur.o, p3trutil.o, l2demodlib, and layer1lib.

2. Files dedicated for the implementations of the approach. For example, tg.o, tg11.o, tg12.o, and tg62.o.

The arguments to the program are: infile - the file containing the formalized theorem; outfile - the file containing the output descriptions of the proof search.

7.2.4 Example Proofs

The following illustrates five test examples of the system GC. Each theorem is represented using the language given in section 7.2.1. Each vertex is represented by letter 'a' followed by letters and/or digits, and each unknown is represented by letter 'b' followed by letters and/or digits. Each path is represented by a list of vertices and unknowns, and it has a name denoted by letter 'p' followed by digits.

```
(test 1) The theorem is:

**assertion:  A graph consisting of three paths (p1, p2, p3) **
***********   is given. Several predicate constraints **
***********   satisfied by the graph are also given. **

g    (p1: a1  a7 a3 end )
  (p2: a1  a2  a4(ni p1)(ni p3)  b2  a3 end  )
  (p3: a1  a2 a6(in p1)(ni p2)  b3  a3 end  )

**goal:  Prove that in the graph there exists a cycle passing **
******   vertex a2. **

EXIST a cycle  passing vertex a2

PROOF:

The node number is           1
 The theorem assertion is
   (p1: a1  a7 a3 end  )
  (p2: a1  a2  a4(ni p1)(ni p3)  b2  a3 end   )
  (p3: a1  a2 a6(in p1)(ni p2)  b3  a3 end   )

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b2;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b2;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b3;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b3;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a3;b2;a4;a2;a6;b3;a3;


***** application of rule B1:B2  *****

The node number is           2
 The theorem assertion is
   (p1: a1  a7 a3 end  )
  (p2: a1  a2  a4(ni p1)(ni p3)  b2  a3 end   )
  (p3: a1  a2 a6(in p1)(ni p2)  b3(ni b2)  a3 end   )

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b2;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b2;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b3;a6;a2;a1;
```

```
 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b3;a6;a2;a1;

 The  retcd from cyclechk is :         0
 The  cycle is :
a3;b2;a4;a2;a6;b3;a3;

***** application of rule B1:B2  *****

The node number is           3
 The theorem assertion is
   (p1: a1  a7 a3 end  )
  (p2: a1  a2  a4(ni p1)(ni p3)  b21  an7  b22  a3 end   )
  (p3: a1  a2 a6(in p1)(ni p2)  b31(ni b21)(ni b22)  an7  b32  a3 end   )
(qb2: b21  an7  b22   end )
(qb3: b31(ni b21)(ni b22)  an7  b32   end )

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b22;an7;b21;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b22;an7;b21;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b22;an7;b31;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b22;an7;b31;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b32;an7;b21;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b32;an7;b21;a4;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b32;an7;b31;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a7;a3;b32;an7;b31;a6;a2;a1;

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a3;b22;an7;b32;a3;

 The  retcd from cyclechk is :         0
 The  cycle is :
a2;a4;b21;an7;b31;a6;a2;


The theorem is proved

11.7u 2.5s 0:32 44% 104+88k 5+126io 18pf+0w
```

```
 (test 2) The theorem is:

**assertion:  A graph consisting of six paths: (p1, p2, p3, p4 **
***********   p5, p6) are given. Several pradicate constraints **
***********   satisfied by the graph are also given. **

g (p1: a1 a2 a3 a4 a5 a6 end )
  (p2: a2 a4 a6 end  )
  (p3: a1 a6 a2 end  )
  (p4: a1 ak2 ak3 ak4 ak5 ak6 end )
  (p5: ak2 ak4 ak6 end  )
  (p6: a1 ak6 ak2 end  )

**goal  Prove that in the graph there exists a cycle passing **
******  edge <ak4,ak6>. **

EXIST a cycle passing edge <ak4,ak6>

PROOF:

The node number is           1
 The theorem assertion is
 (p1: a1 a2 a3 a4 a5 a6 end  )
  (p2: a2 a4 a6 end   )
  (p3: a3 a1 a6 a2 end   )
  (p4: a1 ak2 ak3 ak4 ak5 ak6 end  )
  (p5: ak2 ak4 ak6  end   )
  (p6: ak3 a1 ak6 ak2 end   )

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a3;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a3;a4;a5;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a3;a4;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a5;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a5;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a5;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a1;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a1;a6;a5;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a3;a1;a6;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a4;a3;a1;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a4;a5;a6;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a4;a3;a1;a6;a4;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a4;a3;a1;a6;a5;a4;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a4;a5;a6;a4;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak3;a1;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak3;ak4;ak5;ak6;a1;

 The  retcd from cyclechk is :         0
 The  cycle is :
a1;ak2;ak3;ak4;ak6;a1;


The theorem is proved

84.8u 2.7s 1:47 81% 104+144k 5+146io 4pf+0w
```

```
 (test 3) The theorem is:

**assertion:  A graph consisting of three paths (p1, p2, p3) **
***********   is given. Several pradicate constraints **
***********   satisfied by the graph are also given. **

g (p1: a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 end )
  (p2: a2 a4 a6 a8 a10  end  )
  (p3: a1 a10 a2 end  )

**goal:   Prove that in the graph there exists a cycle passing **
******    edge <a9,a10>). **

EXIST a cycle  passing vertex <a8,a10>

PROOF:

The node number is          1
 The theorem assertion is
 (p1: a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 end  )
  (p2: a2 a4 a6 a8 a10  end   )
  (p3: a1 a10 a2 end   )

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a7;a6;a5;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a7;a6;a5;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a7;a6;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a7;a6;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a6;a5;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a6;a5;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a6;a4;a2;

 The  retcd from cyclechk is :         0
 The cycle  candidate (not a goal ) is :
a2;a1;a10;a9;a8;a6;a4;a3;a2;

 The  retcd from cyclechk is :         0
 The  cycle is :
a2;a1;a10;a8;a7;a6;a5;a4;a2;
```

```
The theorem is proved

4.3u 2.0s 0:21 29% 96+104k 6+119io 15pf+0w


 (test 4) The theorem is:

**assertion:  A graph consisting of two paths (p1, p2) **
***********   is given. Several pradicate constraints **
***********   satisfied by the graph are also given. **

g  (p1: a1 a2[ni b02] b01 a3 end )
   (p2: a1  b02 a3 end  )

**goal:   Prove that in the graph there exists a cycle passing **
******    vertex a2. **

EXIST a cycle  passing vertex <a8,a10>

PROOF:

The node number is          1
 The theorem assertion is
   (p1: a1 a2[ni b02] b01 a3 end  )
  (p2: a1  b02 a3 end   )

 The  retcd from cyclechk is :         1
 The cycle  candidate (not a goal ) is :
a1;a2;b01;a3;b02;a1;


***** application of rule B1:B2  *****

The node number is          2
 The theorem assertion is
   (p1: a1 a2[ni b02] b01 a3 end  )
  (p2: a1  b02[ni b01] a3 end   )

 The  retcd from cyclechk is :         0
 The  cycle is :
a1;a2;b01;a3;b02;a1;


***** application of rule B1:B2  *****

The node number is          3
 The theorem assertion is
   (p1: a1 a2[ni qb02] b011 an4 b012 a3 end  )
  (p2: a1  b021[ni b011][ni b012] an4 b022 a3 end   )
(qb01: b011 an4 b012  end )
(qb02: b021[ni b011][ni b012] an4 b022  end )

 The  retcd from cyclechk is :         0
 The  cycle is :
a1;a2;b011;an4;b021;a1;


The theorem is proved

3.6u 2.0s 0:26 21% 96+112k 4+119io 26pf+0w
```

```
 (test 5) The theorem is:

**assertion:  A graph consisting of six paths: (p1, p2, p3, p4 **
***********   p5, p6) are given. Several pradicate constraints **
***********   satisfied by the graph are also given. **

g (p1: a1 a2 a3 a4 a5 a6 end )
  (p2: a2 a4 a6 end  )
  (p3: a4 a1 a6 a2 end   )
  (p4: a1 ak2 ak3 ak4 ak5 ak6 end )
  (p5: ak2 ak4 ak6 end  )
  (p6: a1 ak6 ak2 end  )

**goal  Prove that in the graph there exists a cycle passing **
******  edge <ak2,ak6>. **

EXIST a cycle passing edge <ak2,ak6>

The node number is           1
 The theorem assertion is
 (p1: a1 a2 a3 a4 a5 a6 end  )
  (p2: a2 a4 a6 end   )
  (p4: a1 ak2 ak3 ak4 ak5 ak6 end  )
  (p5: ak2 ak4 ak6 end   )
  (p6: a1 ak6 ak2 end   )

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a4;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a4;a3;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a4;a5;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a4;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a5;a4;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a5;a4;a3;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a4;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a1;a6;a4;a3;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a5;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a3;a4;a1;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a4;a5;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a2;a4;a1;a6;a2;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a4;a5;a6;a4;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a4;a5;a6;a1;a4;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak3;ak4;ak5;ak6;a1;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak3;ak4;ak6;a1;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak4;ak5;ak6;a1;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak4;ak6;a1;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
a1;ak2;ak6;a1;

 The  retcd from cyclechk is :          0
 The cycle  candidate (not a goal ) is :
ak2;ak3;ak4;ak2;

 The  retcd from cyclechk is :          0
 The  cycle is :
ak2;ak3;ak4;ak5;ak6;ak2;


The theorem is proved

49.6u 2.5s 1:22 63% 104+120k 5+136io 17pf+0w
```

7.3 Discussion and Future Implementation Plan

The system as implemented is applicable to a restricted set of classification theory only. However, the simple structured GC can prove many basic theorems in graph theory with acceptable search space size. The test proofs shown in the previous section indicate that the system's performance is acceptable; in other words, the search space size is reasonable and the cpu time used is also reasonable for those examples. These results have partially shown that the approach developed by this research potentially has much value in providing a model to attack problems related to 'existence'.

The future implementation of GC is planed to:

(a) Complete implementation of the core of the c-algebraic representation scheme.

(b) Build up a rich background knowledge base including most T-procedures for graph types algorithmically defined in graph theory.

(c) Implement the Goal-Assertion Subsumption Method and all the proving methods.

(d) Build up a well organized inference rule set, adopt all recognized heuristics to utilize type knowledge for better choice of inference rules.

(e) Construct a flexible interactive knowledge-based reasoning mechanism based on the classification theory for the purpose of solving widely ranged basic problems and attacking some hard problems in graph theory.

CHAPTER 8

Examples and Some Results

Using the mechanism and strategies of the classification approach many proofs can be generated in step by step fashion. This chapter presents such examples and some results. To simplify the demonstration, most failed proofs of subtheorems are omitted. Only correct choices in using inference rules are presented. Several individual results obtained from the research, which are relevant to tree type graphs will be discussed in the final part of the chapter.

8.1 Examples of Proving Theorem and Verifying Types

The following examples demonstrate how GC proves a theorem and verifies a graph type from factual input.

Example 8.1 proof of the theorem in example 1 of 5.2 (see section 5.2).

Theorem:

'Any two distinct cycles having a common edge e contain a cycle not passing e'.

The assertion is:

AND(< x1x2y1, cycle >, < x1x2y2[not= y1], cycle >)

The goal of the theorem is:

GOAL-FORM: 4

∀ G (graph in the assertion) ∃ y, such that

< y, cycle >[not-with <x1x2>] partially subsumes G.

GC starts the proof search by applying the proving methods at the root of the and/or tree (in this example only the Goal-Assertion Subsumption Method is illustrated). Since the generated ME-set is (x1, x2), and no v-list constructed from the ME-set can generate a non-degenerate cycle by applying cycle T-procedure, GC applies inference rule 1 on y1 and y2 (suggested by ME-check) and then proves the subtheorems at the newly created three nodes (represent three cases):

case 1 {y1 is replaced by y′1xy3, y2 is replaced by y′1x}

ASSERTION:

AND(< x1x2y′1xy3, cycle >, < x1x2y′1x, cycle >)

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < y, cycle >[notwith <x1x2>] partially subsumes G

PROOF:

The c-decomposition of the ASSERTION is:

AND(<x1x2>, <x2y′1, 1>, <y′1x, 1>, <xy3, 1>, <y3x1, 1>, <xx1>)

The generated ME-set is (x1, x2, y′1, x, y3), and one of v-lists constructed from the ME-set is: xy3x1

The c-decomposition of < xy3x1, cycle > is:

AND(< xy3, 1 >, < y3x1, 1 >, < x1x >)

Since this result partially subsumes the c-decomposition of the ASSERTION, the system infers that x2, x, y3 and x1 are in a given v-list, thus x2 satisfies 'not-in xy3x1' and therefore, the cycle satisfying 'not-with <x1x2>' is verified by the constraint checking procedure. Hence the subtheorem is proved.

case 2 {The same as case 1 except y2 is replaced by y′1xy3, y1 is replaced by y′1x}

case 3 {y1 is replaced by y′1xy′3, y2 is replaced by y′1x3[not=x]y′4}

ASSERTION:

AND(< x1x2y′1xy′3 , cycle >, < x1x2y′1x3[not=x]y′4, cycle >)

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < y, cycle >[notwith <x1x2>] partially subsumes G

By singleton reduction rule (see remark below) the assertion is rewritten to:

ASSERTION:

AND(< x11y′11x22xy′3 , cycle >, < x11y′11x22x3[not=x]y′4, cycle >)

with substitution of x11y′11x22 to x1x2y′1.

Since the generated ME-set is (x11, x22, y′11, x, and x3), and no v-list constructed from the ME-set can be applied by cycle T-procedure to generate a non-degenerate cycle, GC applies inference rule 6 on y′3 and x3 (suggested by ME-check) and then proves the subtheorems at the newly created two nodes (representing two cases).

Remark:

The following defines rules of singleton sub v-list and singleton reduction:

A singleton sub v-list is a sub v-list which satisfies:

(1) it contains more than two list elements and at least two vertices are in the sub v-list,

(2) all the elements in the sub v-list do not appear in anywhere else of the theorem.

Singleton Reduction Rule: If w is a singleton sub v-list then w can be replaced by a pattern **x11y′x22** to improve the efficiency of class algebraic manipulation.

It is easy to see that the singleton reduction rule does not lose information on v-list elements but makes v-lists either shorter or better structured (only three elements remained with two vertices at its ends).

case 3.1 {y′3 is replaced by y′31x3y′32 }

ASSERTION:

AND(< x11y'11x22xy'31x3y'32, cycle >, < x11y'11x22x3[not=x]y'4, cycle >)

PROOF:

The c-decomposition of the ASSERTION is:

AND(< x11y'11, 1 >, < y'11x22, 1 >, < x22x, 1 >, <xy'31, 1 >, < y'31x3, 1 >, < x3y'32, 1 >, < y'32x11, 1 >, < x22x3 >, < x3y'4, 1 >, < y'4x11, 1 >)

One of the generated ME-sets is (x22, x, y'31, x3), and one of the v-lists constructed from the ME-set is: x22xy'31x3

The c-decomposition of < x22xy'31x3, cycle > is:

AND(<x22x>, < xy'31, 1 >, < y'31x3, 1 >, <x3x22>)

The above class expression partially subsumes the assertion, also it is verified by the constraint checking procedure that 'not-with <x1x2>' is true, (based on the singleton reduction rule), x1 is in x11, and x2 is in y'11x22, Hence the subtheorem is proved.

case 3.2 {x3 is replaced by x3[notin y'3]}

ASSERTION:

AND(< x11y'11x22xy'3, cycle >, < x11y'11x22x3[notin y'3][not=x] y'4, cycle >)

Since from the generated ME-sets no v-list constructed can be used by cycle T-procedure to generate a class expression that partially subsumes the assertion and satisfies the constraint, GC applies inference rule 6 on y'4 and x (suggested by ME-check). Then GC proves the subtheorems at the newly created two nodes (represent two cases):

case 3.2.1 {y'4 is replaced by y'41xy'42 }

ASSERTION:

AND(< x11y'11x22xy'3, cycle >, < x11y'11x22x3[notin y'3][not=x] y'41xy'42, cycle >)

PROOF:

The c-decomposition of the ASSERTION is:

AND(< x11y'11, 1 >, < y'11x22, 1 >, < x22x, 1 >, <xy'31, 1 >,

< y'31x3, 1 >, < x3y'32, 1 >, < y'32x11, 1 >, <x22x3>, < x3y'41, 1 >,

< y'41x, 1 >, < xy'42, 1 >, < y'42x11 >)

One of the generated ME-sets is (x22, x3, y'41, x), and one of the v-lists constructed from the ME-set is: x22x3y'41x

The c-decomposition of < x22x3y'41x, cycle > is:

AND(<x22x3>, < x3y'41, 1 >, < y'41x, 1 >, <xx22>)

The fact is that this expression partially subsumes the assertion, also 'not-with <x1x2>' is verified by the constraint checking procedure. Hence the subtheorem is proved.

case **3.2.2** {x is replaced by x[notin y'4]}

ASSERTION:

AND(< x11y'11x22x[notin y'4]y'3, cycle>, <x11y'11x22x3[notin y'3][not=x]
y'4, cycle >)

Since from the generated ME-sets no v-list constructed can be used by cycle T-procedure to generate a class expression that partially subsumes the assertion and satisfies the constraint, GC applies inference rule 7 on y'3 and y'4 (suggested by ME-check) and then proves the subtheorems at the newly created three nodes (represented by three cases where the conjunction of the first two is disjunctive to the third, thus the proofs of the first two cases are sufficient to prove case 3.2.2).

case **3.2.2.1** {y'3 is replaced by y'3[notin y'4] }

ASSERTION:

AND(< x11y'11x22x[notin y'4]y'3[notin y'4, cycle >, < x11y'11x22x3[notin y'3][not=x]y'4, cycle >)

One of the generated ME-sets is (x22, x, y'3, x22, y'4, x3), and one of the v-lists constructed from the ME-set is: x22xy'3x11$\overline{y}$'4x3

The c-decomposition of < x22xy'3x11$\overline{y}$'4x3, cycle > is:

AND(<x22x>, < xy'3, 1 >, < y'3x11 >, <x11$\overline{y}$'4, 1 >, < $\overline{y}$'4x3, 1 > <x3x22>)

The expression partially subsumes the assertion, also the truth of 'not-with <x1x2>' is verified by the constraint checking procedure, hence the subtheorem is proved.

case **3.2.2.2** {y'3 is replaced by y'31[notin y'41]x31y'32 and y'4 is replaced by y'41x31y'42 }

ASSERTION:

AND(< x11y'11x22x[notin y'4]y'31[notin y'41x31y'32, cycle >,

< x11y'11x22x3[notin y'3][not=x]y'41x31y'42, cycle >)

One of the generated ME-sets is: (x22, x, y'31, x31, y'41, x3), and one of the v-lists constructed from the ME-set is:

x22xy'31x31$\overline{y}$'41x3

The c-decomposition of < x22xy'31x31$\overline{y}$'41x3, cycle > is:

AND(<x22x>, <xy'31, 1>, < y'31x31, 1>, <x31$\overline{y}$'41, 1>, <$\overline{y}$'41x3, 1>, <x3x22>)

The expression partially subsumes the assertion. The truth of 'not-with <x1x2>' is verified by the constraint checking procedure, thus the subtheorem is proved.

QED

To generate v-lists of case 1 the exhaustive search will result in 96 v-list candidates from the ME-set, but with a simple heuristic guidance, such as 'turn around lists are redundant for cycle type', the number of candidates will be reduced to 36. If another heuristic rule 'neighborhood relation must be maintained for cycle type' is added, then only 3 candidates remain.

Example 8.2 verification of a graph type

FACT : AND(< y1xy2, tree >, < xy3[not-in y1, y2], 1 >)

CONJECTURE (suggested by heuristics): the graph is type tree (or GOAL-FORM: 1)

PROOF :

The generated ME-set is (x, y1, y2, y3). The c-decomposition of the FACT is:

OR(AND(< y1xy2, tree >, < xy3, 1 >))

One of the v-lists constructed from the above ME-set is: y1xy2y3. And the extended c-decomposition of the class < y1xy2y3, tree > has the following sub c-decompositions:

OR(< y1xy2y3, tree >,

AND(< xy2, tree >, < y1xy3, tree >),

AND(< xy3, tree >, < y1xy2, tree >),

AND(< xy2y3, tree >, < y1x, tree >)).

Since < xy3, tree > subsumes < xy3, 1> and the system infers that constraint y3[not-in y1, y2] is satisfactory (x, y1, y2, y3 are in an ME-set), the c-decomposition of the GOAL subsumes the c-decomposition of the ASSERTION. Thus the conjecture is verified.

8.2 Typical Theorems Proved

Using proper rules (indexed by types) combined with the Goal-Assertion Subsumption Method GC could generate many proofs for the theorems and lemmas as shown in the following (some are very simple and some are not very simple for human provers, while all of them are from general graph theory textbooks [ST] [Ha] [Bo]):

(a) Theorem: If cycle (closed path) C1 contains edges e1, e2, and cycle C2 contains edges e2, e3, then there exists a cycle that contains e1 and e3.

ASSERTION:

AND(< x1y′1x3x4y′2x2, cycle > , < x1y′3x5x6y′4x2, cycle >)

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < y, cycle >[with <x3x4> <x5x6>] partially subsumes G

(b) Theorem: In a connected graph, for any two disjoint paths p1 and p2 there exists a path that connects p1 and p2 and does not intersect with p1 and p2 except at their two end vertices.

ASSERTION:

AND(< y1, 1 >, < y2[notin y1], 1>, < y3[universal], connected >)

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, x1, x2 such that: < x1[in y1]y[notin y1y2]x2[in y2], 1 > partially subsumes G

(c) Theorem: If there exist two distinguished paths between two vertices then there exists a cycle.

ASSERTION:

AND(< xy1x2, 1 >, < x1y2[not= y1]x2, 1>)

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < y, cycle > partially subsumes G

(d) Theorem: Any two vertices of a nonseparable graph lie on a common cycle.

ASSERTION:

< y[with x1x2], nonseparable >

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < x1yx2, cycle > partially subsumes G

(e) Theorem: For any two vertices and an edge of a nonseparable graph there exists a path joining the vertices which contain the edge.

ASSERTION:

< y[with x1x2][with < $x3x4$ >]1, nonseparable >

GOAL-FORM: 4

∀ G (in the assertion) ∃ y, such that: < x1yx2, 1 >[with < $x3x4$ >] partially subsumes G

(f) Theorem: If a graph G=(V, E) is connected, then the graph G′ = (V, $E - e$), which results after removing a cycle edge e, is also connected.

ASSERTION:

AND(< y2[universal], connected>, RE(< x1x2y1, cycle>, < $x1x2$ >))

GOAL-FORM: 1

∀ G (in the assertion) ∃ y, such that: < y[with x1x2y1], connected> subsumes G

8.3 Examples of Induction Proving

A simple example of induction proving which is discussed in chapter 6 is illustrated below:

Thorem:

In a connected graph if a vertex v and a path p are disjoint then there exists a path p1 from v to p, and p1 does not intersect p except at an end vertex.

ASSERTION:

AND(< y1, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >)

GOAL-FORM: 4

∀ G (graph in the assertion) ∃ y, x such that

< x1y[notin y1]x[in y1], 1> subsumes G

Induction proving of the theorem:

The unknown y1 in class < y1, 1 > is chosen as the induction argument and split out two cases:

case 1: Based on the frame knowledge of type path the minimum value of the type is a vertex. Replace y1 by x2 everywhere in the theorem.

ASSERTION:

AND(< x2, 1 >, < x1[notin x2], 0 >, < y2[universal], connected >)

Since from the generated ME-sets no v-list constructed can be used by path T-procedure to generate a class expression that partially subsumes the assertion, GC applies inference rule 11 on x1 and x2 (suggested by ME-check) and then proves the subtheorems at the newly created theorem:

ASSERTION:

AND(< y1, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >, < x1y3x2 >)

The c-decomposition of the ASSERTION is:

AND(< y1, 1 >, < x1, 0 >, < y2, connected >, < x1y3, 1 >, < y3, 1 >, < y3x2, 1 >)

One of the generated ME-sets is (x1, y3, x2), and a v-list constructed from the ME-set is: x1y3x2 (where y3 is matched with y, x2 is matched with x), and The c-decomposition of < x1y3x2, 1 > is:

AND(<x1y3, 1>, <y3, 1>, <y3x2, 1>)

This c-decomposition partially subsumes the assertion. Because y1 is x2 (that is made by substitution), x2[in y1] is satisfied, y3 and x2 are in ME-set, and y3[notin y1] is satisfied, the theorem (base case) is proved.

case 2: Generate a temporary axiom (the temporary axiom will be used in further process):

AXIOM (temporary):

ASSERTION:

AND(< y1, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >)

(c-decomposition form of the ASSERTION is the same as the assertion.)

GOAL-FORM: 4

∀ G (graph in the assertion) ∃ y, x such that < x1y[notin y1]x[in y1], 1 > subsumes G

The new theorem is:

ASSERTION:

AND(< y1x2, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >)

GOAL-FORM: 4

∀ G (graph in the assertion) ∃ y, x such that < x1y[notin y1x2]x[in y1x2], 1> subsumes G

The c-decomposition of the ASSERTION is:

AND(< y1, 1 >, < y1x2, 1 >, < x1, 0 >, < y2, connected >, < x1y3, 1 >, < y3, 1 >, < y3x2, 1 >)

After referencing the temporary axiom, the assertion of the axiom partially matches the new assertion (it requires corresponding instantiation, but for this problem the instantiation results names unchanged), thus the conclusion (goal of the axiom) is instantiated and is added to the assertion:

ASSERTION:

AND(< y1x2, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >, < x1y[notin y1]x[in y1], 1 >)

No v-lists constructed from the generated ME sets, when operated on by the path T-procedure, can generate a class expression which partially subsumes the assertions and also satisfies the constraints. Therefore, GC applies inference rule 6 on y and x2 (suggested by ME-check) and then proves the subtheorems at the newly created two nodes (two cases):

case 1 {x2 is replaced by x2[notin y]}

ASSERTION (is replaced by):

AND(< y1x2, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >, < x1y[notin y1]x[in y1], 1 >)

The c-decomposition of the ASSERTION is:

AND(< x1y, 1 >, < y, 1 >, < yx, 1 >)

One of the generated ME-sets is (x1, y, x), and a v-list constructed from the ME-set is: x1yx, and the c-decomposition of < x1yx, 1 > is:

< x1y, 1 >, < y, 1 >, < yx, 1 >

The expression subsumes the assertion. Because x2[notin y] is satisfied, y[notin y1x2] is satisfied, all constraints are satisfied, thus the subtheorem is proved.

(Remark: when a constraint parameter is a multi-element v-list such as in y[notin y1x2] or x[in y1y2], ME checks if the constraint is satisfied for each parameter element. Also some simple rules like: if x[not= x1] then x[notin x1] and if x[notin y] then y[notin x] are recognized by ME checking procedure.

case 2 {y is replaced by y'3x2y'4 }

ASSERTION:

AND(< y1x2, 1 >, < x1[notin y1], 0 >, < y2[universal], connected >, < x1y'3x2y'4[notin y1]x[in y1], 1 >)

The c-decomposition of the ASSERTION is:

AND(< x1y'3, 1 >, < y'3, 1 >, < y'3x2, 1 >, < x2y'4, 1 >, < y'4, 1 >, < y'4x, 1 >)

One of the generated ME-sets is (x1, y'3, x2), and a v-list constructed from the ME-set is: x1yx, and The c-decomposition of < x1y'3x2, 1 > is:

< x1y'3, 1 >, < y'3, 1 >, < y'3x2, 1 >.

The above class expression partially subsumes the assertion. Because y'3x2y'4[notin y1] is satisfied, y'3[notin y1] is satisfied too, all constraints are satisfied, thus the subtheorem is proved.

QED

8.4 A Result on The Tree Representation

A by-product from this research is an interesting tree representation and its application. This section is devoted to this result.

8.4.1 Level Sequence Representation of Trees

For a rooted ordered labeled tree (simply say tree) T the following definitions are formalized:

Definition 1. The level sequence L is defined as:

Let the sequence of nodes visited in preorder traversal of T be numbered 1, 2, .., n, then L(i) denotes the level number of the ith node (i.e. the node numbered i is at level L(i)).

Definition 2. B is said to be a label sequence if B(1), B(2), $\cdots$ B(n) are the values of the labels of the nodes as visited in preorder traversal.

Definition 3. Two trees are said to be isomorphic if we can map one tree into the other by permuting the order of the sons of vertices [AHU]).

Definition 4. A tree CT with level sequence CL and label sequence CB is said to be a canonical tree iff any tree T that has level sequence L, label sequence B and is isomorphic to CT satisfies the condition:

CL $\geq$ L and if CL=L then CB $\geq$ B holds in lexicographic order

(Note that a similar definition for canonical level sequence can be found in [BH]).

It is clear that a level sequence plus a label sequence uniquely determines a rooted ordered labeled tree, and any such a tree has a unique canonical form. The following gives an algorithm to compute the canonical tree by modifying an algorithm given by Aho, Hopcroft, and Ullman [AHU, 3.2], which tests isomorphism of trees and runs in O(n) time.

8.4.2 Canonical Tree Algorithm

The canonical form of the tree is defined as:

Algorithm A. Converting a rooted ordered labeled tree T to its canonical form.

Input: L, B {L, B are level sequence and label sequence of tree of T respectively}

Output: CL, CB {CL, CB are level sequence, label sequence of the canonical tree of T}

Step 1. Associate each leaf node (denote its node number by w) of the tree in the level k with following three tuples:

L-tuple(w) = (0);

B-tuple(w) = B(w);

W-tuple(w) = (k) for all k in (0, h);

where h is the highest level. And let k=h.

Step 2. If k=0 goto Step 4. else do lexicographic sort [AHU] on all L-tuples of this level in nonincreasing order. Then assign each node of the level with a grade g (an integer) if there exist $g-1$ distinct L-tuples which are less than the tuple of the node.

Step 3. Do the same as step 2 on B-tuples, thus assigning each node with a rank r.

Step 4. Let k=k-1. For each non-leaf node (note number w) in the level do bucket sort [AHU] on its sons and the following two conditions must be satisfied:

(a) If the sorted sequence is denoted as (w1, w2, ..., wt), then its corresponding grade sequence (denoted as (g1, g2, ..., gt)) is in nonincreasing order.

(b) For those nodes with same grade (say wi, ..., wj), their corresponding rank sequence (denoted as (ri, ..., rj) is in nonincreasing order.

Step 5. Do the following assignments:

L-tuple(w) = (g1, g2, ..., gt),

B-tuple(w) = (r1, r2, ..., rt),

W-tuple(w) = (k, w1, w2, ..., wt).

Goto Step 2.

Step 6. Push the root node onto an empty stack. For t=0 to n-1, do the following: Pop the node (denoted w) on the top of the stack, assign the first item of W-tuple(w) to CL(t), and assign B(w) to CB(t). If there are more items of W-tuple, sequentially push them onto the stack from right to left.

The proof of correctness of the algorithm A can be obtained based on the above definitions and the operations given by the algorithm. It is omitted here. Also that the algorithm takes O(n) time can be proved in a similar way as the proof given for theorem 3.3 in [AHU].

The algorithm has been applied on databases query standardization to improve performance [ML]. The following illustrates two examples of generations of the canonical trees using algorithm A.

Example 1: Calculate the canonical form for tree T, which has L=(0, 1, 1, 2, 2, 1), B=(1, 2, 1, 1, 2, 3).

The L-tuples, W-tuples and grades created after execution of the algorithm A are listed as follows:

node number	L-tuple	*grade*	B-tuple	*rank*	W-tuple
1	2,1,1	1	2,3,1	1	0,3,6,2
2	0	1	1	1	1
3	1,1	2	2,1	2	1,5,4
4	0	1	1	1	2
5	0	1	2	2	2
6	0	1	3	3	1

Thus T has the canonical form: CL=(0, 1, 2, 2, 1, 1), CB=(1, 1, 2, 1, 3, 2)

Example 2: A tree T has L=(0, 1, 1, 2, 2, 1, 2, 2), B=(1, 1, 1, 2, 3, 1, 1, 2). Its canonical form is: CL=(0, 1, 2, 2, 1, 2, 2, 1), CB=(1, 1, 3, 2, 1, 2, 1, 1)

CHAPTER 9

Discussion And Conclusions

9.1 Discussion of The Approach

This section discusses and evaluates the reasoning approach in comparison with other approaches in the domain of graph theory. The generality of the formalization of this work is also analyzed.

9.1.1 Why The Approach Is Useful

Currently, there are only two previously published approaches to automated reasoning in graph theory. One is Epstein's GT [Ep] and another is Cvetkovic's GRAPH [CP].

The major shortcoming of GT's approach is that it treats a graph type as an atomic object so it is unable to reveal the components of a type, i.e subgraphs and their relations. However, proof of conceptual relations involving components or subgraphs is an important task in mathematical research. The ability to prove relations on subgraphs provided by this work is therefore significant to the state of the art.

The system GRAPH is an interactive type of reasoning system. It provides flexible guidance to users in graph theory research, but does not have adequate reasoning mechanisms for stand-alone proof processes. Since GC is a well formalized stand-alone system it can be used in conjunction with GRAPH.

In comparison with general purpose theorem provers, GC is efficient in generating proofs for graph theory theorems. This is because GC effectively organizes domain specific knowledge in an algebraic and procedural manipulable form. Such modeling is hard to obtain from general purpose theorem provers. General purpose theorem provers such as resolution type provers are inadequate in proving existence [Bul]. It is clear that the classification approach has successfully illustrated a mechanism to overcome this particular difficulty of reasoning, thus providing constructive proofs for theorems related to existence.

9.1.2 The Proof Process in GC is Simple

The proof process in GC is simple because of the following:

In local processes GC uses only two kinds of knowledge:

(a) the pattern matching technique,

(b) the subsumption technique.

These techniques already exist in well developed form. Thus the knowledge base required for GC's local processing is rather simple.

In global process GC uses case-splitting which is also a well understood technique. The process is organized as a simulation of human proof process using case analysis method. Therefore global as well as local processes in GC are simple.

9.1.3 The Conceptual Formalization Is General

This section describes a proposition which indicates that the representation power of the approach is equivalent to the first-order predicate logic in graph theory. Therefore the approach is general enough to be used to deal with basic concepts in graph theory.

Proposition: In the domain of graph theory, theorem representations in terms of the first-order predicate calculus and theorem representations in terms of the classification method are equivalent, i.e. there exists a formal transformation from one form to the other.

In the following two parts we construct such a transformation. Part 1 describes the process to transform a theorem represented in first order predicate calculus to a theorem in class expression form. Part 2 describes the process to transform a theorem in class expression form to a theorem in first order predicate calculus.

PART 1 We modify language AGT (Arithmetical Graph Theory, see section 2.2.3) to formalize the graph theory in terms of first order logic, and call the modified language MAGT.

In MAGT the only object is set. Number, vertex, edge, v-lists, graph, and set of graphs are represented as a set satisfying certain predicates. Note that functions mapping from v-lists to sets of graphs (e.g. type-functions: pathf, treef) are also sets.

MAGT is now able to quantify graphs (or subgraphs). For example:

$\models_U \forall$ G belong(G, treef(x1x2x3)) $\rightarrow$ path(G)

The formula is interpreted as follows: U is the universal graph set, $\models$ represents tautological implication, $\forall$ is universal quantifier symbol, $\rightarrow$ is logical implication, treef(x1x2x3) is a type-function that represents a set of graphs generated in the same way as that used by the tree-procedure defined in chapter 4. The symbol 'belong' is a predicate that means that the graph G belongs to the graph set treef(x1x2x3). The predicate path(G) means G is a path.

We now construct the transformation which converts a formula of two MAGT expressions (assertion and goal) into two classification expressions. The process is shown as the following ten steps, which are to be executed in this order:

Step 1: Convert both the assertion and the goal to clause form [Ni].

Step 2: Convert each edge name, say W, to an edge in GC, say <x1x2>, and convert vertex name, say Z, to x (note that naming consistency is assumed).

Step 3: Convert a predicate to a class: We first assume that the parameter list of a predicate is ordered such that graphs are always in right most positions and they are followed toward the left by edges, vertices and other type parameters. Let P′ be a predicate having graphs as its parameters, and let the right most graph parameter be G. Then form a class < y, simple > with constraint: [P], where P is defined as the same predicate as P′ but with parameter

G explicitly removed (note it is implicitly replaced by each graph in the class < y, simple >, see section 3.1). It is easy to see that GC interprets the constraint P in the same way as GRAPH interprets P′. For example GC converts P′5(Q) to < y, simple >[P5], where P′5 means connected, and Q is a graph variable. The connected graph is converted to the class < y, simple > with constraint [connected].

Step 4: If the predicate has no graph name as a parameter but has an edge name (as the right most edge parameter), say W, then form a class < x1x2, e > with constraint: [P], here P represents the same predicate as P′ but with parameter W removed from its parameter list. GC can interpret the predicate P in the same way as GRAPH interprets the predicate P′. For example, GC converts S′1(X, Y, W) to S′1(x1, x2, <x3x4>) then further to class < x4x5, e >[S1(x1, x2)] which in turn is interpreted as class < x4x5, e > (this is a single edge) with constraint S1(x1, x2) meaning that it is formed by joining vertices x1 and x2.

Step 5: If the predicate has neither graph name nor edge name but has a vertex name, say Z (as the right most vertex parameter), then form a class < x, 0 > with constraint: [P], here P represents the same predicate as P′ but with parameter Z removed from its parameter list. It is easy to see that GC will interpret the constrained class in the same way as GRAPH interprets P′. For example, GC converts R4(X, Y) to R4(x1, x2) then further to class < x2, 0 >[R4(x1)] which is interpreted as class < x2, 0 > (this is a single vertex) with constraint R4(x1) which means vertex x2 is joined to x1 by a walk.

Step 6: Form a list of predicates by linking predicates which do not have graphs, edges or vertices as parameters but whose parameter sets have a non-empty intersection. If a predicate in the predicate list has a parameter which appears in a predicate P′ that has a graph or edge or vertex as parameter, then the conjunction of the predicates in the list and P will be the constraints of the corresponding class generated from predicate P′. If the condition is not satisfied then all predicates in the list are unrelated to graph theory and will not be processed.

Step 7: The predicate connective $\vee$ is converted to the operator '+'. It is clear that the same interpretation will be made by GRAPH and GC on corresponding forms.

Step 8: The connective $\wedge$ between two predicates P' and Q' is processed as follows:

(a) if two unconstrained classes generated from these two predicates are the same, say $< S, T >$, then GC converts $P' \wedge Q'$ to $< S, T >[P \wedge Q]$

(b) if two unconstrained classes generated from these two predicates are different, say $<$ S1, T1 $>$ and $<$ S2, T2 $>$, then GC converts $P' \wedge Q'$ to $<$ S1, T1 $>$[P] $\cdot$ $<$ S2, T2 $>$[Q]. Based on the definition of operator '$\cdot$' GC will interpret the class expression in the same way as GRAPH interprets the conjunction of those two predicates.

Step 9: The predicate connective $\neg$ remains unchanged.

Step 10. Convert the assertion and goal separately thus obtaining a theorem form in GC with meaning that the goal set subsumes the assertion set.

PART 2 We now construct a transformation which converts a formula in two class expressions to two MAGT expressions in clause form. The process is shown as follows:

Step 1: Convert each class, say $<$ Si, Ti $>$[wffj], to a well formed formula wff', where wff' is made by the following process:

(a) the constraints on elements of Si are converted to $wff_{i1}, \wedge \cdots \wedge wff_{in}$

(b) let $wffi' = wff_{i1}, \wedge \cdots \wedge wff_{in}$

(c) each predicate in wffi' is modified by adding a new parameter which is the predecessor of the predicate (see section 3.1) in its parameter list.

(d) a predicate Ti(Gi) (meaning: Gi is type Ti) is created, and let $wffi' = wffi' \wedge Ti(Gi)$

(e) let the predicates in wffj be $wff_{j1}, \wedge \cdots \wedge wff_{jm}$ each predicate in wffj is modified to add a parameter (in the right most position of its parameter list) Gi (with meaning 'in Gi')

(f) let $wffj' = wff_{j1}, \wedge \cdots \wedge wff_{jm}$

(g) create a predicate P(Gi, Ti(S′))

In substep (g) the predicate P is the same as the predicate 'belong' used in the PART 1. P(Gi, T′i(S′)) means that the graph Gi belongs to the class of graphs generated by the function T′i, which has the same output as that generated by the T-procedure T′ with input S′. S′ is the unconstrained v-list with respect to S.

(h) let wff′ = P(Gi, Ti(S′)) ∧ wffi′ ∧ wffj′

Example: < x1x2y1[not= y2], cycle >[with <x3x4>] is equivalent to the wff:

(P(G, cycle(x1x2y1)) ∧ not=(y1, y2) ∧ with(G, < $x3x4$ >))

where the predicate not=(y1, y2) means that each vertex in y1 is not in y2, the predicate with(G,< $x3x4$ >) means that the edge < $x3x4$ > is in G and the function cycle(x1x2y1) results in the set of cycles.

Step 2: Conversion of operators:

(a) AND: the transformation for the formula $<S_1, T_1>$. $<S_2, T_2>$ is:

R(G, G_1, G_2) ∧ P(G_1, $T_1(S_1')$) ∧ P(G_2, $T_2(S_2')$) ∧ wff1′ ∧ wff2′ where the predicate R means graph G is the union of G_1, G_2.

(b) operators OR, RV, RE, IV, IE and CMPL can be processed similarly. The corresponding predicates are for OR, RV, IV, IE, and COPL.

The proof of the equivalence of the two representations can be obtained directly from the definitions.

9.2 Conclusions

This thesis research has developed an artificial intelligent approach for automated reasoning on basic concepts of graph theory through using knowledge-based conceptual classification.

It addresses:

(a) Representing and manipulating knowledge based on both an algebraic theory and a recursive procedural theory.

(b) Reasoning relations among basic graph concepts using multiple non-resolution theorem proving methods.

(c) Developing a model of proving theorems of existence.

(d) Simulating human processes on constructive proofs.

(e) Simulating human processes on case-split-based reasoning.

(f) Modeling completeness and soundness of reasoning on graph concepts.

(g) Modeling an automated reasoning system, GC (Graph Classification), based on the approach.

A summary of the approach is as follows:

- LRC (Linear Recursive Constructivity), a graph attribute, defines its values by algebraic formulas implicitly associated with the background knowledge of graph types.
- LRC is utilized to recursively classify graph concepts.
- Subsumption between two class expressions which represent two graph concepts is investigated in their decomposed forms, where the smaller objects are generated by invoking T-procedures.
- The GASM (Goal-Assertion Subsumption Method) and forward chaining search guidance (by case-split-based inference rules) are used in proof process.
- Linear forms of graph concepts simplify symbolic processes thus they are easy for human to read.
- The approach has been partially implemented and generated many constructive proofs for graph theory theorems.

The main contributions of this thesis are:

(a) Development of a proof mechanism in graph theory, which is able to reveal the relation involving components of a mathematical object, i.e subsets of a graph type or subgraphs of a graph.

(b) Development of an algebraic representation theory and a recursive procedural representation theory for simplifying, normalizing, and decomposing graph concepts. This is significant in dealing with graph concept manipulation.

(c) Development of a proof search and case-splitting technique with the guidance of graph type knowledge, which simulates human efficient modeling in proving theorems.

(d) Illustration of a new proving mechanism that can generate constructive proofs for existence by manipulating only simple linear forms of theorems.

The future research will be on completely implementing the system GC for this approach; extending the graph language for directed graphs; developing isomorphism checking method to overcome permutation problem concealed in subsumption method; improving the representations for more complex key concepts; investigating a proof organization that can derive the goal by non-constructive deduction; applying the approach to attack hard problems in graph theory and to investigate the fields other than graph theory.

References

[AHU] Aho, A.V. Hopcroft, J.E. Ullman, J.D. The design and analysis of computer algorithm. Addison-Wesley Reading, Mass, Oct 1975.

[BE1] Blasius, K., N. Eisinfer, J. Siekmann, G. Smolksa, A. Herld and Walther. The Markgraf Karl refutation procedure. Proc. 7th IJCAI, Aug. 1981, 511-518.

[BH] Terry Beyer and Sandra Mitchell Hedetniemi. Constant time generation of rooted tree. SIAM J. COMPUT. Vol. 9. No 4. Nov 1980.

[BH1] Bledsoe, W.W. and L.J. Henschen What is Automated Theorem Proving? Journal of Automated Reasoning Vol. 1 No. 1 1985, 23-28.

[Bi1] Bibel W. Automated Theorem Proving. Friedr. Vieweg & Sohn. Braunschweig/Wiesbaden, 1982.

[Bl] Bledsoe, W.W. Non-Resolution Theorem Proving. Artif. Intell., 1977, 1-35.

[BM] Boyer, R.S. and Moore, J.S. A Computational Logic. Academic Prell, New York, 1979.

[Bo] Bondy, J. and Murty, U. Graph Theory with Applications. New York, North-Hollan, 1976.

[BT1] Bledsoe, W.W. and M. Tyson. The UT Interactive Theorem Prover. Univ. Texas Math Dept. Memo AIP-17A, May 1978.

[Bu1] Bundy, A. Discovery and Reasoning in Mathematics. Proc. 9th IJCAI, Aug. 1985, 1221-1230.

[Bu] Bundy, A. The Computer Modeling of Mathematical Reasoning. Academic Press, New York, 1983.

[CF2] Cohen, P.R. and Feigenbaum, E.A. The Hand Book of Artificial Intelligence. Volume 3.

[Cl] Clancey, W.J. Classification Problem Solving. Proceedings of the National Conference on Artificial Intelligence. Austin, TX: William Kaufmann, Inc.

[Cs1] Chou, Shang-Ching. Proving Elementary Geometry Theorems Using Wu's Algorithm. Contemporary Mathematics. Volume 29. 243-286.

[deB] deBruijn, J.G., 'A Survey of The Project AUTOMATH.' In Essays on Combinatory Logic, Lambda Calculus and Formalism (J.P. Seldin and J.R. Hindley, eds) Academic Press, 589-606 (1980).

[Ep] Epstein, S.L. On The Discovery of Mathematical Theorems. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milano, Italy. 1987. Volume 1, pp. 194-197.

[FS1] Fernley-Sander, D. Using and Computing Symmetry in Geometry Proofs. Proceeding of ATSB-85 1985.

[HB] Hanson, S.J. and Bauer, M. Conceptual clustering, Semantic Organization and Polymorphy. in Uncertainty in Artificial Intelligence, Kanal, L.N. and Lemmer, D. (eds.), North Holland, 1986.

[Ha] Harary, F. Graph Theory. Reading, MA, Addison -Wesley, 1972.

[La1] Lawrence C. Paulson Interactive Theorem Proving with Cambridge LCF . A user Manual.

[Le] Lenat, D.B. AM: An Artificial Intelligence Approach to Discovery in Mathematics. Ph.D dissertation, Stanford University, 1976.

[Li1] Lin, R., 'A Classification Approach for Reasoning Systems – A Case Study in Graph Theory', published in Proceeding of the Fifth Israeli Symposium on Artificial Intelligence, Vision and Pattern Recognition (page 45-62), Tel Aviv, Israel, December 1988.

[Li2] Lin, R. An Algebraic structure for Classification of Graph Concepts. in preparation. Old Dominion University, Norfolk, Va.

[Lo1] Loveland, D.W. Automated Theorem-Proving: A Qarter-Century Review. Automated Theorem Proving after 25 Years (Bledsoe, W.W. and D.W. Loveland, Eds.), Proc. Special Session on Automatic Theorem Proving, 89th Ann. Meeting of the Amer. Math. Soc., Denver, CO, Jan. 1983.

[LO1] Lusk, E.L. and Overveek, R.A. The Automated Reasoning System ITP. ANL-84-27 Argonne national laboratory. Apr. 1984.

[LO2] Lusk, E.L. and Overveek, R.A. An LMA-Based Theorem Prover ANL-82-75 Argonne national laboratory. Dec 1982.

[LO3] Lusk, E.L. and Overveek, R.A. Logic Machine Architecture Inference Mechanisms-Layer 2 User Reference Manual. ANL-82-84 Argonne national laboratory. Apr 1984.

[Ma] Manna, Z. Introduction to The Mathematical Theory of Computation. McGraw-Hill, New York, 1973.

[MKK] Mitchell, T. M., Keller, R. and Kedar-Cabelli, S. Explanation-Based Generalization. A Unifying View, Machine Learning 1, 1 (January 1986), pp. 47-80.

[MS] Michalski, R. and Stepp, R. Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy. IEEE Transactions on Pattern Analysis and Machine Intelligence, 5.

[ML] Mukkamala, R., and Lin, R., Improving Database Performance Through Query Standardization, accepted for Publication, IEEE Southeastcon'89, Columbia, SC, April 1989.

[Ne1] Nevins, A.j. A Human Oriented Logic for Automatic Theorem Proving, J. ACM, 1974, 606-621.

[Ne2] Nevins, A.J. Plane Geometry Theorem Proving Using Forward Chaining. Artif. Intell., 1975 1-23.

[NS1] Newell, A., J.C. Shaw and M.A. Simon, Empirical Explorations of The Logic Theory Machine: A Case Study in Heuristics. Computers and Thoughts (Feigenbaum and Feeldman, Eds.), McGraw-Hill, 1963, 134-152.

[Ro1] Robinson, J.A. A Machine-Oriented Logic Based on The Resolution Principle. J. ACM, 1965, 23-41.

[SB1] Sterling, L. and Bundy, A. Meta-Level Inference and Program Verification. In Loveland, D.W. (editor), 6th conference on Automated deduction, pages 144-150 Springer verlag, 1982.

[SM] Stepp, R. and Michalski, R. Conceptual Clustering: Inventing Goal-oriented Classification of Structured Objects In book Machine Learning Volume 2. pp. 472. Morgan Kaufmann Publishers, Inc.

[ST] Swamy, M.N.S. and Thulasiraman, K. Graph, Networks, and Algorithms. John Wiley & Sons, Inc, 1981.

[Wo] Wos, L. What Is Automated Reasoning. Journal of Automated Reasoning 1, 1 (January 1985), pp. 6-9.

[WW1] Wos, L., S. Winker and E. Lusk. An Automated Rea- soning System. AFIPS Conf. Proc.: Natl. Comp. Conf., AFIPS Press. Montvale, 1981, 697-702.