TERMS: TREE-STRUCTURED REASON MAINTENANCE SYSTEM

THESIS SUBMITTED FOR
THE AWARD OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

SHINYMOL ANTONY KONDAY1L

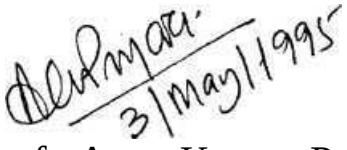
DEPARTMENT OF COMPUTER & INFORMATION SCIENCES
SCHOOL OF MATHEMATICS & COMPUTER /INFORMATION SCIENCES
UNIVERSITY OF HYDERABAD
HYDERABAD, INDIA

APRIL 1995

This is to certify that I, Shinymol Antony Kondayil, have carried out the research embodied in the present thesis for the full period prescribed under the Ph.D. ordinances of the University.

I declare, to the best of my knowledge, that, no part of this thesis was earlier submitted for the award of any research degree of any University.

Shinymol Antony Kondayil



Prof. Arun Kumar Pujari
Supervisor &
Head of the Department

Prof. V. Kannan
Dean of the School

To

GIGIE KARTHIK GEORGE

for all his love

Acknowledgements

This is by way of acknowledging my gratitude to those who have played a seminal role in the working and completion of this dissertation. Were it not for the help, support and encouragement extended by numerous individuals, the completion would have been impossible. There are those I MUST mention; yet my gratitude also extends to many not explicitly named.

Prof. Arun Kumar Pujari initiated me into the field of Computer Science and supervised the work all along. His unbounded patience and constant support at all times which easily exceeded the bounds of the professional duty helped me immensely. The personal example he set by way of academic brilliance provided inspiration when the going was tough. Words are inadequate to acknowledge my debt to him. But I record my deepest gratitude to him for making this long cherished dream come true. My heartfelt gratitude is due to his family members as well.

I thank all other faculty members of the Department of Computer Science and the staff of the AI lab for their encouragement and assistance. I respectfully acknowledge the helpful attitude of Prof. V. Kannan, the Dean of the School. I also record the co-operation of the Computer Centre staff.

My indebtedness to Prof. Jemmis and family who provided a home away from home is inexpressible. They were my family here and I could not have gone on without them. Prof. P. S. Zachariahs, Ms. Marina Abraham, Ms. Sushamma Abraham, Ms. Beena Bijoo, Ms. Gigi Dennis and all their families have been of immense help to me. Their love and encouragement helped me carry on, when giving up seemed to be the best thing to do. I thank Ms. Marina Abraham in special for her sensitivity, care and understanding.

I am obliged to Dr. Arindama Singh for some very productive discussions during the initial stages of this work. I also thank Dr. P. S. Rao and Dr. C. R. Rao for helpful

discussions. I record very special *Thank you* to Dr. Chitra Panikkar who patiently proof read the entire work.

Ms. Jeena, Ms. Shiji, Mr. Rajesh Kumar Verma, Ms. Jayashree and Ms. Durga Bhavani have earned my deepest respect by being true friends in times of need. I thank Ms. Reeba K. V. and Ms. Ann Abraham for being with me whenever I needed them. I am also grateful to Dr. A. Giridhar Rao, Mr. Abraham Mathew, Mr. E.V.R.C. Mohan Reddy, Mr. Bharadwaj, Mr. Vasudev Varma, Mr. Appajee, Ms. Priya, Mr. K. R. K. Varrior, Mr. K. Lakshmi Narayana, Ms. E. Uma, Mr. Ravindra Sharma and many other friends.

My heartfelt acknowledgement is due to Fr. Jayakumar, Fr. Anand, Br. Viji Joseph, Ms. Grace Mani and also Rev. Dr. Philips Vadackekalam, Sr. Martin, Ms. BeenaManoj, and Prof. K. E. Paulose who all proved that good will and friendship can transcend geographical distances.

I also thank Prof. George Joseph and family who were a constant source of encouragement. Not only did they have confidence in me, they also inspired me to have confidence in myself. I am deeply grateful to Peter Cheriyathundam for timely assistance and help which laid the foundation of all that I have been able to accomplish.

And now, my family. The joyful awareness of my parents' love, patience and encouragement have been an all-time inspiration. The same applies to the rapport I had with my brothers, Jacob and Joseph, my sister-in-law Jessy, and especially my nephew and godson Anton who helped in that unique only-children-can-way.

I place on record my gratitude to the Department of Atomic Energy for the Dr. K. S. Krishnan Junior Research Fellowship and the CSIR for a Senior Research Fellowship.

Shinymol Antony

Table of Contents

List of Notations

a, b,...	Variables
a	Negation of a
p, q	Paths
p ⊎ q	Concatenation of p and q
£	Set of literals
$\mathcal{F}$	Conjunctive formula
$\mathcal{F}_a$	Subformula of $\mathcal{F}$ obtained by deleting the literal *a* uniformly from those clauses having it.
$\mathcal{F}'_a$	Set of clauses in $\mathcal{F}$ not containing a.
$M(\mathcal{F})$	Binary matrix representing $\mathcal{F}$.
$\Pi(\mathcal{F})$	Set of prime implicates of $\mathcal{F}$
$M_\Pi(\mathcal{F})$	Matrix representing $\Pi(\mathcal{F})$.
C	Set of column numbers of $M(\mathcal{F})$.
S	Subset of *C*
T	Subset of *C*.
$M[S,T]$	Submatrix of $M(\mathcal{F})$ with columns in *S* and ignoring rows in T
$P[S,T]$	Set of prime paths in $M[S,T]$.
S_r	Set of $i \in S$ such that $M(r,i) = 1$.
$E(p,S_r)$	Extension of a path p to S_r.
$\mathbf{T}(\mathcal{F})$	Tree representing the formula $\mathcal{F}$.
h	Single hypothesis
$\mathcal{H}$	Set of hypotheses
$M(h)$	Matrix representing h.

List of Abbreviations

AI	Artificial Intelligence
ATMS	Assumption-based Truth Maintenance System
CV	Coefficient of variation
DF	Degrees of Freedom
DPP	Davis-Putnam Procedure
IPIA	Incremental Prime Implicant Algorithm.
JTMS	Justification-based Truth Maintenance System
LTMS	Logic-based Truth Maintenance System
PARPIAP	Parallel Prime Implicant algorithm using Paths
PIAPE	Prime Implicant Algorithm using Paths and Extension
PIAP	Prime Implicant Algorithm using Paths
RMS	Reason Maintenance System
SD	Standard Deviation
TMS	Truth Maintenance System

List of Tables

Abstract

Over the last decade, general purpose *Truth Maintenance Systems* (TMSs) have been developed and applied in a number of domains. One of the reasons for the development of TMS is that it enables search programmes to carry results obtained in one part of the search space to other parts. Among the non-monotonic reasoning systems, TMSs have a special place since these are the only systems which provide a constructive approach towards building practically useful inference engines working with incomplete knowledge.

Prime implicants play a significant role in many reasoning systems. There have been numerous techniques reported in the literature to compute the prime implicants of a formula. As a logical framework of *Assumption-based* TMS (ATMS), Reiter and de Kleer [Reiter 87] suggested a notion of *knowledge compilation* in a *Clause Management System* (CMS). It has also been suggested that for this purpose, the Slagle's method to compute the prime implicants is better than a simple consensus method. Recently, Socher [Socher 91] proposed an algorithm using a concept of *prime path* in a binary matrix. This method provides a better technique to compute the prime implicants. Nevertheless, Socher's algorithm perform far more computations than required.

In the present work, the properties of prime paths in a matrix are studied in detail. The paths which contain a literal and that which do not contain a literal are characterized. Based on this characterization, a scheme is proposed to partition the matrix representing the formula. The fact that the prime paths of a matrix can be obtained by the concatenation of prime paths of two submatrices, is established theoretically. A concept of the *extension* of a path in a submatrix to a larger matrix is proposed and the *Prime Implicant Algorithm using Paths and Extension* **(PIAPE)** is designed. It may be noted that subsumption is the crucial operation in any prime implicants algorithm and the actual execution time depends critically on the number and expense of

the subsumption checks that are required. It is established that extension is nothing but a sort of subsumption. In order to achieve efficiency, paths which neither subsume nor are subsumed by any other path are characterized. Further, the number of subsumptions can be reduced by generating less number of paths which are not prime. Hence, a better method to weed out paths that are not prime in a bigger matrix is proposed in this dissertation. The new algorithm *Prime Implicant Algorithm using Paths* (PIAP) a refinement of PIAPE is efficient in terms of number and expense of the subsumption checks required.

A new tree-structure for knowledge representation proposed in this dissertation naturally evolves from the partitioning scheme of PIAP. The structure of a node in the tree, and the implementation details of PIAP are explained. Experiments are conducted to compare the efficiency of the algorithm, and the results obtained substantiate that the proposed algorithm is far better than Socher's algorithm. The tree-structure is also used to maintain the prime paths. The efficiency of the method hinges on this tree-structure and the same structure helps design a novel RMS. Hence, it is named *TERMS: TreE-structured Reason Maintenance System.*

The PIAP is well suited for incremental computation of prime paths, which is necessary for RMS update problem. Different methods to update the tree representing the formula as well as methods for incremental knowledge compilation are discussed. The advantage of this method over other methods is that the additional knowledge is treated collectively for compilation. Thus, the method is better than other methods, both in global and incremental mode.

Apart from these advantages, the PIAP exhibits inherent parallelism. The potential for concurrency is explored, and the parallel algorithm, PARPIAP to compute prime paths is also designed. The different granularity levels are explored and the architecture suitable for each, and finally, a hybrid architecture suitable for PARPIAP is proposed.

Though the prime implicants paradigm is a widely used tool in many areas of AI, one particular application of it, in the area of computer vision, is discussed. The problem of shape from silhouettes is rephrased as a problem of computing prime implicants of a formula obtained from the object silhouettes. It is demonstrated that the proposed framework is better than the conventional algorithmic approach, namely, Volume Intersection.

Chapter 1

INTRODUCTION AND OVERVIEW

1.1 Introduction

Whatever intelligence may be, or be denned as, *reasoning and problem-solving* have traditionally been viewed as central subsets of it. Reasoning is the art of finding out what information follows from what other information, of finding what information is consistent with what other information, of finding what information is needed to answer a problem, and how to derive that answer. An important characteristic of reasoning is the combination of information items to form new information, usually in the process of deriving a particular conclusion by carefully considering the available facts.

Logic plays an important role in reasoning and problem-solving. It offers a formal mechanism for learning. The power of logic is based on three important features: Firstly, it provides a language for the accurate *representation of knowledge.* Secondly, a framework for *processing the represented knowledge* is given in the form of a calculus defining permitted rules for drawing conclusions. Thirdly, a mechanism for mechanicaJ *proofs* of truth values, or equivalence, of statements can be defined [Kurfe b 89]. Prepositional logic is appealing because it is simple to deal with and a decision procedure for it exists. Real-world facts can be represented as logical propositions written as *well-formed formulas*. (wffs) in prepositional logic. The molecules of logic are statements - and statements are chunks of knowledge or information.

1.1.1 Search methods

Search problems are ubiquitous in AI. Almost every AI program depends on a search procedure to perform its prescribed function. For every problem encountered there might be numerous alternatives to consider. The problem-solvers are constantly confronted with the necessity to select among these equally plausible alternatives. The frustrating property exhibited by most of the search problems is the exponential growth of the plausibilities. The computing time for the problem is hard to control as it grows exponentially due to combinatorial explosion. Though search is a general mechanism for intelligence, the efficiency with which it can be performe'd limits its applicability. The central issue in search is efficiency. Two important measures of efficiency are the amount of time and the amount of memory required to find the solution or to conclude absence of a solution.

Brute-force technique

Brute-force *method* of search guarantees a solution if there is one. This is a blind search in the sense that it uses no knowledge about the problem other than the problem space itself. All the alternatives in the search space are explored mechanically and tested until a solution is found (or all solutions are found), a time limit has been reached or failure occurs. At the worst case, it may be necessary to explore the whole search space before finding a solution.

' If each point of the search space is dissimilar, Brute-force method is the best that can be applied. However, in most cases, there is a great deal of similarity among the points of the search space. Hence, in order to achieve efficiency, the result obtained in one region of the search space has to be carried to other regions.

ability to perform dependency directed backtracking and so to support non-monotonic reasoning. TMS are also called *belief revision* or Reason Maintenance *Systems* (RMS). In this thesis the terms RMS and TMS are used interchangeably.

A RMS is the house-keeping subsystem of an overall reasoning system. The basic architectural presupposition is that the overall reasoning system consists of two components: a problem-solver and a TMS [de Kleer 86a]. The problem-solver usually includes all domain knowledge and inference procedures. Every inference made by the problem-solver is communicated to the TMS. An important characteristic of RMS is that it has no access to the semantics of the problem-solver behaviour. The RMS treats the expressions passed to it by the problem-solver purely syntactically. As a consequence of this, the problem-solver is held responsible for the correctness of the information passed to the RMS.

The uniqueness of RMS stems from maintaining records of the origins of labels assigned to database facts (dependency records), and subsequently using those dependency records to prune the search space and perform database updating. Different searching and reasoning programs had used many of the ideas previously, in a somewhat *adhoc* way. Hayes [Hayes 75] seems to be the earliest reference to what might be regarded as a RMS. However, Doyle [Doyle 78, Doyle 79] provided the first non-monotonic comprehensive implemented version of a TMS which automatically maintains consistency.

A context is normally determined by some set of hypotheses or assumptions and is expected to be consistent. Based on the type of dependency storage the systems are referred to as *justification* based or *assumption based.* Based on the type of access to information they are referred to as single *context* or *multiple context* systems.

The RMS insists on maintaining for the problem-solver a single context that has been passed to the RMS in a single *context system.* The *multiple context system* provides a facility for determining contexts dynamically, without enforcing the usage of any particular

one. Typically, justification based systems operate with single context and assumption based systems operate with multiple contexts.

Justification-based TMS (JTMS)

The JTMS associates a special data structure, called a node, with each problem-solver datum (formula in the database). These nodes are connected together in a web of data dependencies. The TMS uses Horn clauses as *justifications.* Each node has a status *in* or *out,* and a justification. If its justification is *valid,* then a node is *in,* and otherwise it is *out.* In Doyle's TMS [Doyle 79], a justification has the form (< *inlist* >< *outlist* >) and is valid if all the nodes in its inlist are *in* and all the nodes in the outlist are *out* The support for an in node must be *well-founded,* that is, circular, self-supporting networks of justifications are not valid.

The form of justifications permits non-monotonic inference. In other words, the coming *in* of a node that was previously *out* can result in the going *out* of a node that was previously *in.* This facilitates the creation of revisable assumptions as well as non-revisable premises. RMS performs two basic operations on the web of dependencies: *reason maintenance* and dependency directed backtracking. Reason maintenance is invoked whenever the problem-solver adds a new node or justification, and it ascertains which nodes are *in* and which are *out.* Dependency directed backtracking is invoked to resolve contradictions by backtracking through the thread of justifications for the contradictory node in search of an assumption which it can retract in order to restore consistency.

Shortly after Doyle, McAllester [McAllester 80] designed a single context *Logic-based* TMS (LTMS) which is significantly more efficient and comprehensible than Doyle's TMS. LTMS labels nodes as TRUE, FALSE or UNKNOWN and uses disjunctive clauses as justifications.

ATMS as an assumption. The basic structure on which the system depends is the ATMS node. Conceptually, the node has three parts. These are the *problem-solver datum, a label representing the assumptions under which the datum holds, and the justifications provided by the problem-solver which supports the datum.* A set of assumptions is an *environment* and the set of all data propositionally derivable from the assumptions using the justifications is the context of the environment. A datum is in a context if it is propositionally derivable (using the justifications) from the assumptions of the context. An environment is inconsistent, if falsity (symbolised as $\perp$) is propositionally derivable from the assumption set. An inconsistent environment is defined as not having a context. The efficiency of the ATMS is based on the observation that if a datum is derivable from a particular set of assumptions it is derivable from every superset as well. The ATMS associates with every datum the minimal set of environments from which it is derivable. This set is the label of the datum. By computing the label for each datum, the ATMS indirectly computes the contents of each context. Given the label, it is very easy to compute the contents of each context. A datum is in a context exactly when the assumptions of the context are a subset of any of the environments of the datum's label.

An ATMS offers an improvement over conventional RMS for search problems where all or many solutions are required. The problem-solver can explore many possibilities at once, and can compare solutions and potential solutions to problems. Furthermore, the resulting mechanism obviates the need for backtracking. When only one or a few solutions are required, the conventional RMS is more efficient. Acknowledging this and other deficiencies, de Kleer and Williams [de Kleer 86d] proposed a hybrid algorithm called *assumption based dependency directed backtracking.* ATMS, for many tasks, is more efficient than previous TMSs and has a more coherent interface between the TMS and the problem-solver without giving up exhaustivity. Unlike previous TMSs which are based on manipulating justifications, the ATMS is, in addition, based on manipulating

Backtracking

Chronological backtracking helps to increase the efficiency by reducing some of the futile search. The alternatives are selected following some order from one alternative to another and requires an additional machinery for controlling the search. If at any stage the search element is inconsistent or contradictory, the chronological backtracker retracts to the most recent selection made and proceeds from that point. Though this is better than the brute-force technique, much of the work undertaken by a chronological backtracking search procedure when it encountered a failure might be irrelevant to the particular problem discovered, and some of the backtracking may lead only to rediscover the contradictions. Moreover, if the failure depends only on choices made much earlier, all the work done in remaking the later choices is potentially irrelevant. Rather than retracting to the most recent selection made, the retraction to the selection which is responsible for the inconsistency could be much efficient. Information about the choices on which an inference rests is stored so that the culprits for the failure can be identified and the problem-solver need re-make only those choices. The dependency relations between choices and failure are used to direct the backtracking. This idea of *dependency directed backtracking* originated with Stallman and Sussman [Stallman 77].

1.1.2 Reasoning systems

In conventional reasoning systems, much of the work such as the caching of expensive inferences was implemented anew for each problem encountered. Clearly this is wasteful and, because these support mechanisms were not always separated from the problem-solving, it could lead to unnecessary confusion in the system design [Kelleher 88].

These concerns prompted the need to perform belief revision and motivated the development of *Truth Maintenance Systems* (TMS) [Doyle 79] as a way of providing the

In order to make recomputing of the status of an assertion and switching of contexts easier, McDermott [McDermott 83] attempted to bring together the ideas of data dependencies (as in Doyle's TMS) and contexts. The system works with a current context, defined by the user as a set of premises. It computes a label, a Boolean expression in the premises, for each assertion using the current justification set. The value of the label is found by assigning the value *true* to the premises in the current context. This value of the label .determines the current status of the assertion. The status of any assertion in all possible data contexts can be determined using this label. This makes switching to a different context simple. However, McDermott did not exploit this idea further to produce a true multiple context system. McDermott's system can be seen as overlapping the boundary between justification based systems and assumption based systems. The RMSs based on assumptions, appear to have started with the work of Martins and Shapiro [Martins 83, Martins 88], but the most widely used example of the assumption based approach to reason maintenance is probably de Kleer's [de Kleer 86a].

Assumption based TMS

In 1986, de Kleer introduced the idea of an *Assumption-based Reason Maintenance System* (ATMS) which solve several inherent problems in earlier TMS implementations. In Doyle's TMS, only one labelling of nodes with *in* or *out* is considered at any one time. The problem-solver can only focus on a single set of assumptions and their consequences. In contrast to this, the ATMS incrementally computes the assumptions on which each datum depends as each new problem-solver inference is received. The mechanisms of the ATMS revolve around its ability to determine, for any given item of data, the assumptions under which it holds.

All inferences made by the problem-solver are recorded and communicated to the ATMS as *justifications* and every problem-solving hypothesis is communicated to the

assumption sets.

Clause maintenance system

In a justification-based TMS, the database is always kept consistent; this makes it impossible to refer to problem-solving contexts explicitly and requires truth maintenance and dependency-directed backtracking to move to a different point in search space. On the other hand, in an ATMS each datum is labelled with the sets of assumptions (representing the contexts) under which it holds. These sets of assumptions are computed by the ATMS from the problem-solver-supplied justifications. The idea is that the assumptions are the primitive data from which all the other data are derived. These assumption sets can be manipulated far more conveniently than the datum sets they present. There is no necessity that the overall database be consistent; it is easy to refer to contexts, and moving to a different point in the search space requires very little work.

Conventional TMS is oriented towards finding one solution whereas ATMS is oriented towards problem-solving in multiple contexts simultaneously. This is efficiently achieved by labelling each datum with the assumptions upon which it ultimately depends. This idea and its ramifications radically alters the conception and technology of problem-solving. ATMS is not a panacea and is not suited to all tasks.

The basic ATMS [de Kleer 86a] provides a novel truth maintenance facility. Reiter and de Kleer [Reiter 87] proposed a generalization of the basic ATMS called the *Clause Management System* (CMS) and showed its applications to abductive reasoning. A CMS is intended to work together with a reasoner, which issues queries that take the form of clauses. The CMS is then responsible for finding *minimal supports* for the queries. Reiter and de Kleer [Reiter 87] show some relationships between prime implicants and minimal supports.

An ATMS is precisely intended to generate all and only minimal explanations simultaneously [Inoue 89], given a set of clauses. In the ATMS terminology, the set of minimal explanation of a node from the justifications and the assumptions is called the label of node, which is *consistent, sound, complete and minimal.* The basic ATMS is restricted to accept only Horn clause justification and atomic assumptions. If justification can contain non-Horn clauses, and the assumptions are allowed to be literals, then this generalization covers de Kleer's various extended versions of ATMS [de Kleer 86a, de Kleer 86b, de Kleer 86c], Dressler's extended ATMS [Dressier 90], and Reiter and de Kleer's Clause maintenance System (CMS) [Reiter 87].

There have been different algorithms to compute the minimal supports. However, Reiter and de Kleer [Reiter 87] consider two ways in which the CMS manages the knowledge base: keeping the set of clauses (denoted by $\mathcal{F}$) transmitted by the reasoner as it is (the *interpreted approach),* or computing the prime implicants/implicates of $\mathcal{F}$ (the *compiled approach).* When we are faced with a situation where we want to know explanations for many different queries, we must run the algorithm each time a query is issued. Instead of keeping the initial formula $\mathcal{F}$ as it is and doing the same deductions over and over again for different query clauses, some of these inferences can be made once and for all. That is the motivation for the compiled approach. One of the disadvantages of the compiled approach is the high cost of updating the knowledge base. When the reasoner adds a clause D to $\mathcal{F}$, we must compute all prime implicates of $\mathcal{F} \wedge D$. There are many approaches to compute the prime implicants/implicates. These methods are discussed in Section 1.3.

1.2 Hypothetical Reasoning

The information/knowledge available can be imperfect in one or more respects in the sense that it can be uncertain, incomplete, imprecise, inconsistent, or a combination of these. Most real world reasoning is performed in the context of *imperfect information.* Processing of imperfect information plays an important role in realizing advanced AI functions such as common sense, learning, automated reasoning etc. [Poole 87, Poole 88, Ishizuka 90]. A non-monotonic reasoning system is required to handle incomplete knowledge in the knowledge-base. Its formalism has a close connection with constraint satisfaction problem.

There are many approaches to the study of imperfect information processing especially when the knowledge is incomplete. *Hypothetical reasoning* is one of the reasoning schemes which handles incomplete knowledge as hypotheses. The central function of hypothetical reasoning is *abductive inference* which generates necessary combinations of hypotheses for proving a given goal. The hypothetical reasoning system is a logic-based one, where the knowledge is divided into two categories, i.e. complete knowledge $\mathcal{F}$ and set $\mathcal{H}$ of hypotheses. Complete knowledge is always true and has no possibility of inconsistency. On the other hand, set of hypotheses is incomplete, or defeasible knowledge, for which consistency checking is required in the inference process. The basic behaviour of the hypothetical reasoning [Ishizuka 91] is as follows. When a goal (or an observation) is given the system tries to prove the goal from the complete knowledge. If it fails, then the system selects a subset of the hypotheses so that the given goal is proved from the union of complete knowledge with this subset. The selected subset of the hypotheses should be consistent with complete knowledge, while inconsistency is allowed in the whole set of hypotheses. To efficiently exclude inconsistent combinations of hypotheses, truth maintenance is necessary in this inference process. A *reasoning system* based on logic

(logic-based reasoning system) can deal with incomplete knowledge as hypotheses. It is a useful framework because of its theoretical basis and applicability. In ordinary logic-based problem-solving, the success or failure of deductive proof becomes the answer. When the goal includes variables, the binding (unification) to the variables becomes an answer in the success case. On the other hand, a selected subset of the hypotheses becomes an answer in the logic-based hypothetical reasoning system, in which deductive inference mechanism is utilized in reverse direction to generate a solution hypotheses subset.

While it is a useful knowledge-processing framework applicable to many practical problems, the most crucial problem of logic-based hypothetical reasoning system is its slow inference speed. An immediate remedy for this problem is to incorporate heuristic knowledge which plays the role of guiding the inference. However, it is difficult to cover the whole problem domain by heuristic knowledge, which causes the well-known knowledge acquisition problem. Therefore, a fast inferencing mechanism not relying on heuristic knowledge is required. Backtracking caused by the inconsistency among selected hypotheses is the major factor of deteriorating the inference speed. A two-phase hypothetical reasoning system has been presented by Ishizuka [Ishizuka 91] where a *goal-directed inference-path network* is formed using the complete knowledge set but excluding hypotheses in the first phase. Hypotheses necessary for proving a given goal are synthesized in the second phase along this inference-path network in a forward inference fashion with no backtracking. The inference-path network also allows to reduce the number of computationally expensive hypotheses combination to a minimum. The formation of the inference-path network is based on a linear time algorithm for the satisfiability testing of prepositional Horn formula.

It is already indicated that the set of minimal supports for a query can be computed easily from the set of prime implicates of the RMS database. The negation of this minimal support clause becomes a solution hypotheses set for a given goal. Hence, by

such knowledge compilation, abductive synthesis of hypotheses is achieved very efficiently. Although the compilation of knowledge-base allows efficient abductive inference, the compilation process itself is very expensive in terms of computation time and memory.

Hence the core part of CMS as well as hypothetical reasoning is the computation of prime implicates. There have been many algorithms in the literature to compute the prime implicates of a set of clauses. These are the consensus method by *Bartee et al* [Bartee 62], methods by *Karnaugh* [Karnaugh 53], Quine [Quine 59], *McCluskey* [McCluskey 56], *Kohavi* [Kohavi 78], Semantic Resolution technique by *Slagle et al* [Slagle 70], Tison's method [Tison 67], Socher's method [Socher 91], [Jackson 92] etc. The next section gives a review of these methods.

Knowledge changes repeatedly from time to time and hence has to be updated and revised. Changes occur due to the addition, deletion, or change in the information. Due to the dynamic nature of the CMS, the most complicated, computationally expensive and essential operation is to update *(RMS update problem)* the existing database of prime implicates each time knowledge changes. Kean and Tsiknis' [Kean 90] proposed an *incremental prime implicant algorithm* (IPIA) that updates the set of prime implicates when the original corresponding knowledge is modified by addition of new. knowledge which is a single clause. But, generally, knowledge may be a set of clauses rather than a single clause. In this case, IPIA is not efficient since each of the clauses in the set has to be treated one by one. The algorithm proposed in this thesis (Section 3.3.4) treats the set of clauses collectively, and computes the set of prime implicants for the updated knowledge. It can be seen that the proposed method is efficient both in global and incremental modes.

1.3 Different Methods for Compilation

The different methods to compute the prime implicants/implicates of a formula are briefly reviewed here.

Quine's method

There exist numerous methods for reducing any formula to its simplest equivalent. Quine [Quine 52, Quine 55, Quine 59] proposed a mechanical procedure based on the *developed Disjunctive Normal Form. Prime implicants* of a formula are used to express the formula in its simplest equivalent. It is shown in [Quine 52] that any simplest equivalent of a formula is a disjunction of prime implicants of the formula. Any given formula has to be converted into an equivalent developed formula in which all the clauses have all the variables either in the positive or negative form from the set of variables. The prime implicants of the developed formula are computed following a mechanical routine. Initially consider the list of clauses in the formula as the list of prime implicants. This list is extended according to the following principle: whenever two entries can be found in the list which are identical except for the negation sign, add their common part *(consensus* of the two entries) as a new entry in the list; check marks are applied to those two entries which generate the new entry. The check mark is not treated as a disqualification for further consideration of those clauses. The list is extended by this process as far as possible. The entries which do not bear any check marks give the list of prime implicants.

The *shortest normal equivalent* is obtained by deleting from the alternation of all its prime implicants the largest possible combination of jointly superfluous clauses. Though this method gives a mechanical procedure to compute the prime implicants of a formula, the method is not systematic. One of the major drawbacks of this method is that it generates repeated clauses, and requires a large number of basic operations. This

routine, though not unmanageable, turns out to be far more laborious than the method of merely locating and cancelling redundancies. Moreover, the two methods are almost independent. The laborious method of finding simplest normal equivalents depends on a preliminary expansion into a developed normal formula, and this expansion is not affected by any previous cancelling of redundancies.

Tison's method

Tison [Tison 67] contributed a systematic method for determining the prime implicants of a Boolean function. The prime implicants are determined by generalization of the consensus operation which is performed systematically. It is to be noted that the method does not need the formula to be in the developed form as in the case of Quine's method. Consensus is extended from two to any number of clauses. A property of these generalized consensus relations is that the consensus of two implicants of a formula gives another implicant. This property helps to find prime implicants systematically. The method is simple because the application of consensus of order two is no longer iterative as in Quine's method.

One key note of Tison's method is that of a biform variable: a variable which occurs both positively and negatively in the formula. For each biform variable x and for every pair of clauses D_i, D_j in the formula $\mathcal{F}$, add the consensus of D_i and D_j with respect to the variable x, denoted as Con (D_i, D_j, x), to $\mathcal{F}$, (if such a consensus is possible) and delete every subsumed clause[1] from the formula. When all possible consensus with respect to all possible biform variables are computed, and if all subsumed clauses are deleted, then $\mathcal{F}$ will contain all the prime implicants of the original set of clauses, and *only* the prime implicants. The term consensus here is a restricted kind of resolution in which the attention is only to those resolvents of clauses in $\mathcal{F}$ which are tautologies.

[1] D_i subsumes D_j if $D_i \subseteq D_j$

The algorithm exploits the fact that each of the biform variable will be used exactly once in the algorithm. The consensus operation is equivalent to a resolution step and fundamentality test.

Ordered clause consensus method

The number of implicates grows exponentially in general as the resolution process proceeds, which is the serious problem in computing prime implicates. All the original and generated clauses remain to the end and become prime implicates if no subsumed clause appear during the resolution process. So the computational cost does not depend much on the ordering of the biform variables selected. However, the computational cost is influenced to a large extent by the order of the biform variable sequence used in resolution steps if there exists subsumed clauses which are to be deleted. Tsuruta and Ishizuka [Tsuruta 92] have developed a fast and efficient method named *Ordered Clause Consensus* (OCC) method for generating prime implicates utilizing the role of the biform variable. If a variable is a monoforrn one in a set, there is no consensus so that no resolution takes place with respect to the variable, and hence monoform variables are not considered for resolution. All possible unit resolutions regarding every unit variable is performed and all subsumed original clauses are deleted. This does not increase the number of clauses since the resolvent clause always subsumes its one parent clause, which is deleted from the set of prime implicates. This method uses a heuristic regarding the resolution order of biform variables while generating the prime implicates through successive steps. The number of consensi possible with each.of the biform variable is arranged in increasing order. The heuristic used is; *the biform variable corresponding to the least number* is considered; and consensi with respect to this variable are performed. This helps the set of prime implicates to grow slowly at the early stages, which in turn reduces the computational cost as reported in [Tsuruta 92].

Incremental method

All these methods that generate prime implicants/implicates are applicable to the RMS update problem. However, they are inefficient simply because they are concerned with the generation of prime implicants/implicates from an arbitrary Boolean expression. What is needed is an incremental method which generates the prime implicants/implicates using the already available prime implicants/implicates of the original formula and the new formula when the original Boolean expression is modified.

More formally, if the prime implicates denoted by $\Pi(\mathcal{F})$ of a formula $\mathcal{F}$ in the conjunctive form are known, the task is to find the set of prime implicates of $\mathcal{F}$ A $\mathcal{H}$ where $\mathcal{H}$ is another formula in the normal conjunctive form. Obviously, the prime implicates of $\mathcal{F}$ A $\mathcal{H}$ can be generated directly from $\mathcal{F}$ A *ft.* using any of the known conventional methods* discussed above. But this results in not utilizing the $\Pi(\mathcal{F})$ which is already available. Hence, ideally, one would like to generate the set of prime implicates of $\mathcal{F}$ A $\mathcal{H}$ from the $\Pi(\mathcal{F})$ A $\mathcal{H}$. Using the conventional methods to compute prime implicates of $\Pi(\mathcal{F})$ A *ft* results in lot of redundant computations simply because all the conventional methods do not exploit the fact that the elements of $\Pi(\mathcal{F})$ are already prime. Hence, an algorithm which can compute the prime implicates incrementally, and which reduces the redundant computations, is appreciable.

The **IPIA** proposed by Kean and Tsiknis [Kean 90] computes the prime implicates of $\mathcal{F}$ A $\mathcal{H}$ from $\Pi(\mathcal{F})$ and $\mathcal{H}$ when $\mathcal{H}$ is a single clause h. This method resembles Tison's method [Tison 67] except that it stores the new implicates of $\Pi(\mathcal{F})$ U $\{h\}$ in a new set. Incremental computation admits two simplifications which are (i) it needs only to perform consensus with respect to biform variables occurring in the input clause i; and (ii) it needs only to perform consensus between clauses from the sets $\mathcal{H}$ and $\Pi(\mathcal{F})$, but not within the same set since the clauses in $\Pi(\mathcal{F})$ are already prime.

A consensus tree is constructed with the input clause *h* as the root, and every arc is labelled by a clause from $\Pi(\mathcal{F})$, and every node (except the root) is labelled by the consensus of its parent and the associated arc label. Label of nodes are the implicates, and subsumption among these labels are performed to obtain the prime implicates. Various optimization techniques have also been proposed to improve IPIA. However, this method suffers from few drawbacks. In order to overcome the drawbacks of IPIA, Jackson [Jackson 92] proposed another algorithm which also resembles Tison's method, except that it computes prime implicates using a particular resolution strategy which concentrates on finding merges. Merges are resolutions involving a pair of clauses that contain literals of the same sign in addition to complimentary literals. A cost-function defined biases the search towards consensi that generate merges. The resulting merge contains fewer literals than non-merge resolvents derived from parents of the same size. This method computes the compliments of clauses of $\mathcal{F}$ in a particular order so as to avoid the duplication of steps.

Slagle's method

All the methods discussed so far compute the prime implicants when the formula is in disjunctive form, and the prime implicates when the formula is in the conjunctive form. Different from all these methods, Slagle [Slagle 79] describes a method which determine the prime implicants of a formula which is in the conjunctive form. The algorithm works as follows: All the clauses in the formula containing a complementary pair of literals are deleted. A *semantic tree* is constructed keeping this set of clauses thus obtained at the initial node. The literal which occurs in more number of clauses of the formula is given a higher frequency. *Sprouting* from the initial node with the frequency ordering is performed. If there is a nonterminating node, choose an ordering for that node and repeat the process. This sprouting is repeatedly done until there is

no nonterminating nodes to do sprouting. For each success node in the semantic tree, collect the product of all the literals at the branches on the path from the top down to the success node of the semantic tree. The set of all such products thus obtained gives all the prime implicants of the formula. The algorithm may possibly generate some non prime implicants. However, the use of frequency ordering of literals, helps to generate very few (possibly none) non prime implicants. The algorithm may also be used to find the minimal sums of a Boolean function. Reiter and de Kleer [Reiter 87] suggested this method as a well-disciplined method to compute the prime implicants. However, there are better methods to compute the prime implicants.

Though Tsuruta and Ishizuka [Tsuruta 92] also discussed frequency ordering, it has to be noted that these two orderings are different. Slagle considers all the literals for frequency ordering while Tsuruta and Ishizuka consider all the variables and the number of consensi possible with respect to this variable to fix frequency ordering. In Slagle's algorithm, the frequency ordering of literals at each node is required in order to achieve efficiency. But this is not the case with the ordering which Tsuruta and Ishizuka applies. The same ordering can be applied to the IPIA of Kean and Tsikins'. But, when the original formula is appended with more than one clause, computation of the prime implicates is not possible by performing the algorithm just once. The algorithm has to be performed as many times as there are clauses in the new input formula. This approach is obviously not efficient and hence a method which can handle the case when the original formula is appended with another formula will be of great use.

Socher [Socher 91] proposed an algorithm similar to that of Slagle *et al* [Slagle 70]. A concept of *path* in a matrix is introduced and shown that the set of prime paths is a matrix gives the set of prime implicants of a formula. The main difference is in the data structure used. Slagle uses semantic tree as the basic data structure while the data structure used in Socher's method is matrix. This makes Socher's method very suitable for

application in matrix methods for automated theorem proving. Though this method has advantages over other methods, it is not suitable for incremental computation of prime implicants. Apart from this, the algorithm performs certain redundant computations. This is discussed in detail in Chapter 2.

Present work

It is already seen that the prime implicants/implicates are significant in the context of RMS as well as hypothetical reasoning. Though there are different techniques to compute the prime implicants/implicates, as already indicated above, all techniques suffer from one drawback or the other. Hence there is a need to develop an efficient algorithm which overcomes the drawbacks of earlier methods. Besides applications in reasoning, prime implicants/implicates find applications in several areas such as switching theory, combinatorial optimization, computer vision etc. It would be difficult to design a very efficient algorithm for this hard problem suitable to all the applications. This prompts us to design a new algorithm which will overcome the drawbacks of earlier methods in the context of RMS. This thesis is concerned with the design of RMS based on the framework proposed by Reiter [Reiter 87] in CMS and also with the design of an algorithm to compute the prime implicants/implicates in the global as well as in the incremental mode. In this process, a new knowledge representation scheme is proposed.. Using the divide-and-conquer paradigm, an efficient technique for knowledge compilation.for the purpose of RMS is proposed in this thesis. The efficiency of the method hinges on the tree-structure representation for prepositional clauses, which also helps in a novel RMS design. The formula is subdivided into two subfonnulae and the set of prime implicants for both the subfonnulae are computed from where the prime implicants for the formula are computed. The prime implicants are maintained in a binary tree.

Unlike the earlier algorithms which are inherently sequential, the algorithm proposed

in this dissertation is naturally parallelizable. A parallel knowledge compilation technique, designed because of the special characteristic of **PIAP,** is an efficient tool for parallel RMSs.

1.4 Organization of the Thesis

The thesis comprising seven chapters is organized in the following manner:

In Chapter 2, the binary matrix representation of a formula and the three algorithms [Tison 67, Kean 90, Socher 91] to compute the prime implicants/implicates are explained using examples. The paths in a matrix are characterized and based on these characterizations, a new Prime Implicant *Algorithm using Paths* (PIAP) is proposed in Chapter 3. Theoretical arguments supporting the algorithm are also given in the same chapter.

The tree-structure representation of a prepositional formula suitable for the algorithm PIAP is presented in Chapter 4. The implementation details of PIAP and the experimental results that substantiate the theoretical arguments in Chapter 3 are presented in Chapter 4. Further, different methods to update a knowledge base, and to compile the knowledge incrementally, are discussed. Chapter 5 deals with the parallel algorithm to compute the prime paths of a formula.

In Chapter 6, an application of the results to computer vision is discussed. It is shown that reconstructing the three-dimensional shape from multiple silhouettes can be formulated as a problem in prepositional logic. All possible reconstructions of the object can be obtained since the problem of shape from silhouettes is viewed as the problem of computing the prime paths of a formula. It is also shown that this method provides a better way of shape reconstruction. Chapter 7 summarises the contributions and limitations of the work reported in this dissertation and considers possible routes for further research work.

Chapter 2

MATRIX AND PRIME IMPLICANTS

2.1 Introduction

In Chapter 1, the role of RMS and the significance of prime implicants in RMS have been discussed. In this chapter, the binary matrix representation of a prepositional formula is presented. Using the matrix representation scheme, the Tison's method [Tison 67] and the Incremental method (IPIA) of Kean and Tsiknis' [Kean 90] are elaborated here. The drawbacks of these algorithms are pointed out using examples. Socher's algorithm [Socher 91] which uses the concept of a path in a binary matrix to compute the prime implicants is also discussed in detail. An example is made use of to show that Socher's algorithm computes certain redundant computations. In brief, the motivation for development of a new algorithm for prime implicants is discussed here.

2.2 Definitions and Notations

Let the letters a,b,c,... with or without subscripts represent prepositional *variables.* If a is a variable, a denote the negation of a. Both a and *a* are referred to as *literals. A* clause *D* is called a *disjunctive clause* if it is a disjunction of literals. For example, $D = a_1 \vee a_2 \vee \ldots \vee a_q$ is a disjunctive clause. A clause *C* is called a conjunctive *clause* if it is a conjunction of literals. For example, *C* = fli A a_2 A ... A a_n is a conjunctive clause. A **propositional** formula $\mathcal{F}$ is said to be in *Conjunctive Normal Form* (CNF) if it is a conjunction D_1 A D_2 A D_3 A ... A D_m of disjunctive clauses D_i. A formula is in

Disjunctive *Normal Form* (DNF) if it is a disjunction $C_1 \vee C_2 \vee \ldots \vee C_m$ of conjunctive clauses C_i. Unless otherwise stated it is assumed in this dissertation that a formula is in CNF, and a clause is a disjunctive clause. In this thesis a formula $\mathcal{F}$ is denoted by the set $\{D_1, D_2, D_3, \ldots, D_m\}$ of clauses, and a clause by juxtapositioning the literals. For example, $D = a_1 \vee a_2 \vee \ldots \vee a_q$ is represented as $a_1 a_2 \ldots a_q$.

Two literals are said to be *complementary literals* if one is the negation of the other (for example, a and $\tilde{a}$). A clause is said to be *fundamental* if and only if it does not contain any pair of complementary literals. $ab\tilde{c}$ is fundamental whereas $abb\tilde{c}$ is not fundamental. A clause *Di* is said to subsume another clause *Dj* if every literal in *Di* occurs in *Dj* ($Di \subset D_j$). For example, if $Dj = ab\tilde{c}d$, and $D_i = a\tilde{c}d$, then D_i subsumes *Dj*.

Definition **2.2.1:-** *A disjunctive clause D is said to be an implicate of a formula* $\mathcal{F}$ *if* $\models \mathcal{F} \rightarrow D$. *D is a prime implicate of* $\mathcal{F}$ *if D is an implicate of* $\mathcal{F}$ *and there is no other implicate* D' *of* $\mathcal{F}$ *such that* $\models D' \rightarrow D$. *The prime implicate of a formula is not subsumed by any other implicate of the formula* $\mathcal{F}$.

Definition **2.2.2:-** *A conjunctive clause C is an implicant of a formula* $\mathcal{F}$ *if and only if C implies* $\mathcal{F}$. *C is a prime implicant of* $\mathcal{F}$ *if C is an implicant of* $\mathcal{F}$ *and there is no other implicant* C' *of* $\mathcal{F}$ *such that C implies* C'.

The prime implicants/implicates of a formula exist and are finite in number. The set of all prime implicants/implicates of $\mathcal{F}$, denoted by $\Pi(\mathcal{F})$, is unique and is logically equivalent to $\mathcal{F}$.

A variable is said to be a *biform variable* if the negation of the variable is present in atleast one of the clauses of the formula $\mathcal{F}$. If the negation of a variable is not present in any of the clauses of the formula, then such a variable is called a *monoform variable*. It is discussed in Section 1.3 that the set of prime implicates of a formula is obtained

by computing the generalized consensus of the components of the formula. Consensus between two clauses is possible only if there is exactly one complementary pair of literals in those two clauses, and consensus is obtained by taking the union of literals in the two clauses and cancelling out the complementary pair of literals with respect to which the consensus is computed. If there are more than one pair of complementary literals in the two clauses, the resulting clause will not be fundamental. Let p and q be two clauses for which consensus with respect to the variable x is possible. The consensus is defined as

$$\begin{cases} P \cup q - \{x \cup \tilde{x}\}\ ; & \text{if it is fundamental} \\ \text{not defined}\ ; & \text{otherwise.} \end{cases}$$

For example, if D_i = $ab\tilde{c}$, Dj = abc and D_k = abc then $Con(D_i, D_j, c)$= ab and $Con(D_i, D_k, c)$is not possible since abb is not fundamental. This is because there are two complementary pairs of literals, namely, c, c and b,b in D_i and D_k. No consensus is possible between Di = abc and ab since there is no complimentary pair of literals from these two clauses. In fact ab subsumes D_i. This need not be the case in general.

2.3 Matrix Representation

The binary matrix representation of a prepositional formula is introduced in this section. Any clause is considered a binary vector, or a one dimensional array. The occurrence.of a literal in the clause is denoted by an entry 1 or otherwise an entry 0. In the matrix representation, the number of rows in the matrix is equal to the number of literals in the formula and the number of columns in the matrix is equal to the number of clauses in the formula. Thus the formula $\mathcal{F}$ is represented as a binary matrix

$$\begin{cases} 1: & \text{if the literal } x \text{ occurs in the } i^{th} \text{ clause } D_i \text{ of } \mathcal{F} \\ 0\ ; & \text{otherwise.} \end{cases}$$

For a fundamental formula $\mathcal{F}$, the matrix representation is denoted by $M(\mathcal{F})$. Let C be the index set of clauses and £ be the set of literals over which $\mathcal{F}$ is defined. Then for the matrix $M(\mathcal{F})$, C is interpreted as the set of all columns and £ as the set of all rows. For example, for $\mathcal{F} = \{abc, a\tilde{b}c, ab\tilde{c}, ac, bc\tilde{c}\}$, the matrix $M(\mathcal{F})$ is

$$M(\mathcal{F}) = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ a & 1 & 1 & 1 & 1 & 0 \\ b & 1 & 0 & 1 & 0 & 1 \\ \tilde{b} & 0 & 1 & 0 & 0 & 0 \\ c & 1 & 1 & 0 & 1 & 1 \\ \tilde{c} & 0 & 0 & 1 & 0 & 1 \end{array}$$

Definition 2.3.1:- *A column j of $M(\mathcal{F})$ is said to absorb another column k of $M(\mathcal{F})$ if* $M(y,j) \leq M(y,k) \; \forall \, y \in$ £.

Definition 2.3.2:- *A column j of $M(\mathcal{F})$, and correspondingly a clause Dj of $\mathcal{F}$ is a tautology if there is an $x \in$ £ such that* $M(x,j) = M(\tilde{x},j) = 1$.

Absorption of columns in matrices is the same as subsumption of clauses. A column j absorbs another column k if the clause Dj corresponding to the column j subsumes the clause Dk corresponding to the column k. In the above matrix, column 4 absorbs columns 1 and 2. Column 5 of the matrix is a tautology.

2.4 Tison's Algorithm

In Chapter 1, we have discussed the generalized consensus method of Tison [Tison 67] to compute the prime implicants/implicates. This section gives more details of the method using the binary matrix representation introduced above.

A variable *x* is said to be a *biform variable* if there exists $i, j \in C$ such that $M(x,i)$ as $M(\tilde{x},j) = 1$. Let B be the set of biform variables in $\mathcal{L}$. For any such variable x, let C_x^+ C C be the set of columns i such that $M(x, t) = 1$ and C_-^- C C be the set of columns *j* such that $M(\tilde{x},j) = 1$. The consensus $Con(i,j,x)$ for t € C_x^+, $j \in C_-^-$ with respect to the variable *x* is represented by a vector A_{ij}^x defined as

$$A_{ij}^x(y) = \begin{cases} Max(M(y,i), M(y,j)) ; \forall\, y \in \mathcal{L} - \{x \cup \tilde{x}\} \\ 0 ; \text{ if } y = x \text{ or } y = x \end{cases}$$

Using these notations, the algorithm **CONSENSUS** outlines Tison's algorithm.

Algorithm CONSENSUS

INPUT: MATRIX $M(\mathcal{F})$ REPRESENTING THE FORMULA $\mathcal{F}$.

OUTPUT: MATRIX $M_\Pi(\mathcal{F})$ REPRESENTING

THE SET OF ALL PRIME IMPLICATES $\Pi(\mathcal{F})$ OF $M(\mathcal{F})$

STEP 0: INITIALIZE $M_\Pi(\mathcal{F}) = M(\mathcal{F})$

STEP 1: Delete all absorbed columns of $M_\Pi(\mathcal{F})$.

STEP 2: Let *B* be the set of biform variables in $M(\mathcal{F})$.

If $B \neq \phi$ SELECT a row r corresponding to a variable $x \in B$.

STEP 3; For each $i \in C_x^+$ and $j \in C_-^-$

STEP 3.1: COMPUTE A_{ij}^x

STEP 3.2: Augment to $M_\Pi(\mathcal{F})$ the vector A_{ij}^x

STEP 4: Update $B = B - \{x\}$

STEP 5: GO TO STEP 1

END

It is observed that Step 1 is computationally expensive and is performed many times.

As the number of consensi obtained in Step 3.1 increases, the number of columns of the matrix $M_{\Pi}(\mathcal{F})$ also increases. The algorithm is further explained with the help of an example.

Example 2.4.1:-

Let $\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg\}$ be the formula. The matrix $M(\mathcal{F})$ for this formula is given in Figure 2.1(a). For this matrix, $C = \{a, \tilde{a}, b, b, c, \tilde{c}, d,rf, e, \tilde{e}, f, f, g, \tilde{g}\}$, $C = \{1, 2, 3, 4, 5\}$ and $B = \{c, d, f\}$. Let A^x represent the matrix corresponding to the set of all consensi with respect to the variable x. Let c be the first biform variable taken to compute the consensi. The matrix A^c obtained by A^c_{12} and A^c_{23} is given in Figure 2.1(b). Augmenting the matrix A^c to the matrix $M_{\Pi}(\mathcal{F})$ gives the matrix in Figure 2.1(c). Columns 2 and 6 of the matrix in Figure 2.1(c) gets absorbed by column 7 and hence gets deleted. The new $M_{\Pi}(\mathcal{F})$ obtained after deletion of the two columns is given in Figure 2.2(a).

$M(\mathcal{F}) =$

	1	2	3	4	5
a	1	1	1	1	1
$\tilde{a}$	0	0	0	0	0
b	1	1	1	1	1
$\tilde{b}$	0	0	0	0	0
c	1	0	1	1	1
$\tilde{c}$	0	1	0	0	0
d	1	1	0	0	0
$\tilde{d}$	0	0	0	1	1
e	0	0	0	0	0
$\tilde{e}$	1	0	0	0	0
f	0	1	1	0	0
$\tilde{f}$	0	0	0	1	0
g	0	1	0	0	1
$\tilde{g}$	0	0	0	0	0

(a)

$A^c =$

	1	2
a	1	1
$\tilde{a}$	0	0
b	1	1
$\tilde{b}$	0	0
c	0	0
$\tilde{c}$	0	0
d	1	1
$\tilde{d}$	0	0
e	0	0
$\tilde{e}$	1	0
f	1	1
$\tilde{f}$	0	0
g	1	1
$\tilde{g}$	0	0

(b)

$M_{\Pi}(\mathcal{F}) =$

	1	2	3	4	5	6	7
a	1	1	1	1	1	1	1
$\tilde{a}$	0	0	0	0	0	0	0
b	1	1	1	1	1	1	1
$\tilde{b}$	0	0	0	0	0	0	0
c	1	0	1	1	1	0	0
$\tilde{c}$	0	1	0	0	0	0	0
d	1	1	0	0	0	1	1
$\tilde{d}$	0	0	0	1	1	0	0
e	0	0	0	0	0	0	0
$\tilde{e}$	1	0	0	0	0	1	0
f	0	1	1	0	0	1	1
$\tilde{f}$	0	0	0	1	0	0	0
g	0	1	0	0	1	1	1
$\tilde{g}$	0	0	0	0	0	0	0

(c)

The next iteration begins with $B = \{d,f\}$. The biform variable chosen is d and the matrix A^d representing the consensi with respect to the variable d is given in Figure 2.2(b). The augmented matrix is given in Figure 2.2(c) in which column 8 is absorbed by column 2. Hence this column is deleted and the resultant matrix is given in Figure 2.3(a). The algorithm proceeds to find the consensi with respect to the only biform variable left in B, namely, /. The matrix A^f is given in Figure 2.3(b).

$$M_\Pi(\mathcal{F}) = \begin{array}{c} \\ a \\ \tilde{a} \\ b \\ \tilde{b} \\ c \\ \tilde{c} \\ d \\ \tilde{d} \\ e \\ \tilde{e} \\ f \\ \tilde{f} \\ g \\ \tilde{g} \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \qquad A^d = \begin{array}{c} \\ a \\ \tilde{a} \\ b \\ \tilde{b} \\ c \\ \tilde{c} \\ d \\ \tilde{d} \\ e \\ \tilde{e} \\ f \\ \tilde{f} \\ g \\ \tilde{g} \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \left[\begin{array}{ccc} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

$$M_\Pi(\mathcal{F}) = \begin{array}{c} \\ a \\ \tilde{a} \\ b \\ \tilde{b} \\ c \\ \tilde{c} \\ d \\ \tilde{d} \\ e \\ \tilde{e} \\ f \\ \tilde{f} \\ g \\ \tilde{g} \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

(a) (b) (c)

Figure 2.2

The augmented matrix of matrices $M_\Pi(\mathcal{F})$ and A^f in Figure 2.3 is given in Figure 2.4(a). The columns 1, 6, 7 and 10 of this matrix are absorbed by column 9, and columns 3 and 4 are absorbed by column 8. These columns are removed from the matrix and the resulting matrix is given in Figure 2.4(b). At this point, $B = q$ and the algorithm terminates. The matrix $M_\Pi(\mathcal{F})$ in Figure 2.4(b) is the matrix representing the set of prime implicates, $\{abcf, abdfg, abed, abce\}$ of the formula $\mathcal{F}$ given by the matrix in Figure 2.1(a).

$M_\Pi(\mathcal{F}) =$

	1	2	3	4	5	6	7
a	1	1	1	1	1	1	1
$\tilde{a}$	0	0	0	0	0	0	0
b	1	1	1	1	1	1	1
$\tilde{b}$	0	0	0	0	0	0	0
c	1	1	1	1	0	1	1
$\tilde{c}$	0	0	0	0	0	0	0
d	1	0	0	0	1	0	0
$\tilde{d}$	0	0	1	1	0	0	0
e	0	0	0	0	0	0	0
$\tilde{e}$	1	0	0	0	0	1	1
f	0	1	0	0	1	0	0
$\tilde{f}$	0	0	1	0	0	1	0
g	0	0	0	1	1	0	1
$\tilde{g}$	0	0	0	0	0	0	0

(a)

$A^J =$

	1	2	3
a	1	1	1
$\tilde{a}$	0	0	0
b	1	1	1
$\tilde{b}$	0	0	0
c	1	1	1
$\tilde{c}$	0	0	0
d	0	0	1
$\tilde{d}$	1	0	0
e	0	0	0
$\tilde{e}$	0	1	1
f	0	0	0
$\tilde{f}$	0	0	0
g	0	0	1
$\tilde{g}$	0	0	0

(b)

Figure 2.3

$M_\Pi(\mathcal{F}) =$

	1	2	3	4	5	6	7	8	9	10
a	1	1	1	1	1	1	1	1	1	1
$\tilde{a}$	0	0	0	0	0	0	0	0	0	0
b	1	1	1	1	1	1	1	1	1	1
$\tilde{b}$	0	0	0	0	0	0	0	0	0	0
c	1	1	1	1	0	1	1	1	1	1
$\tilde{c}$	0	0	0	0	0	0	0	0	0	0
d	1	0	0	0	1	0	0	0	0	1
$\tilde{d}$	0	0	1	1	0	0	0	1	0	0
e	0	0	0	0	0	0	0	0	0	0
$\tilde{e}$	1	0	0	0	0	1	1	0	1	1
f	0	1	0	0	1	0	0	0	0	0
$\tilde{f}$	0	0	1	0	0	1	0	0	0	0
g	0	0	0	1	1	0	1	0	0	1
$\tilde{g}$	0	0	0	0	0	0	0	0	0	0

(a)

$M_\Pi(\mathcal{F}) =$

	1	2	3	4
a	1	1	1	1
$\tilde{a}$	0	0	0	0
b	1	1	1	1
$\tilde{b}$	0	0	0	0
c	1	0	1	1
$\tilde{c}$	0	0	0	0
d	0	1	0	0
$\tilde{d}$	0	0	1	0
e	0	0	0	0
$\tilde{e}$	0	0	0	1
f	1	1	0	0
$\tilde{f}$	0	0	0	0
g	0	1	0	0
$\tilde{g}$	0	0	0	0

(b)

Figure 2.4

Now let us assume that another formula $\mathcal{H}$ has been transmitted to the RMS by the problem solver. The RMS has to update its knowledge base by compiling the additional knowledge obtained. In the matrix representation scheme it can be said that another matrix $M(\mathcal{H})$ has to be appended to the original matrix $M(\mathcal{F})$, and the prime implicates of the new matrix corresponding to $\mathcal{F}$ A $\mathcal{H}$ are to be computed. Of course, the **Tison's** algorithm explained above can be used to compute the prime implicates of $M(\mathcal{F} \wedge \mathcal{H})$. However, the computation of prime implicates of $M(\mathcal{F}$A *ft)* using Tison's method does not make use of the fact that the set of prime implicates of $\mathcal{F}$ is already computed. The matrix $M_\Pi(\mathcal{F})$ is equivalent to the matrix $M(\mathcal{F})$ according to the definitions of prime implicates and therefore, it is ideal to compute the set of prime implicates of $M(\mathcal{F}$A $\mathcal{H})$ from the matrix $M_\Pi(\mathcal{F})$ corresponding to $\Pi(\mathcal{F})$, and the matrix $M(\mathcal{H})$. The computation time required to compute $\Pi(\mathcal{F} \wedge \mathcal{H})$ reduces if the primeness of the columns of the matrix $M_\Pi(\mathcal{F})$ are made use of.

The method of computing the prime implicates of a formula $\mathcal{F}$A$\mathcal{H}$ from $M_\Pi(\mathcal{F})$ and $\mathcal{H}$, due to the addition of the formula $\mathcal{H}$ is called the *incremental approach.* The Tison's algorithm will generate redundant columns which may be absorbed in later steps as it does not make use of the characteristic that the columns of the matrix $M_\Pi(\mathcal{F})$ are already prime. Hence Tison's algorithm is not suitable for incremental computation of prime implicates. For the RMS update problem, when the assumptions are transmitted to the RMS, what is needed is an incremental method to compute the prime implicates. Kean and **Tsiknis'** [Kean 90] proposed an incremental algorithm **(IPIA)** to compute the nrime implicates if the formula $\mathcal{H}$ is a single clause, say, h. The IPIA algorithm is

2.5 Kean and Tsiknis' Algorithm

In Chapter 1, we have noticed the need for revising a knowledge base. Each time the knowledge base changes, the inferences which can be obtained from the knowledge base also change. These inferences which are obtained by the compilation of knowledge base have to be computed each time the knowledge base changes. We have already seen in Chapter 1 that compilation of a given knowledge base in prepositional form means that the set of prime implicants/implicates of the prepositional formula are to be computed. Updating the compiled knowledge base whenever a change occurs in the original knowledge base is an essential operation. Due to the dynamic nature of the CMS, updating the compiled knowledge base is the most complicated, and computationally expensive operation.

As already mentioned, it is ideal to utilize the knowledge base which is already compiled rather than recompiling the entire knowledge base whenever a change occurs. For a logic based system, this is to use the primeness of the set of prime implicants/implicates which are already computed when the set is updated due to the additional formula given to the RMS. Kean and Tsiknis' [Kean 90] proposed an incremental method that updates the set of prime implicates when the original knowledge is appended by a single clause.

The set $\Pi(\mathcal{F})$ of prime implicates of a **propositional** formula $\mathcal{F}$ is the compiled knowledge base. When $\mathcal{F}$ is updated by an additional formula $\mathcal{H}$, correspondingly the set of prime implicates $\Pi(\mathcal{F})$ also has to be updated. Let the additional formula $\mathcal{H}$ be h, and let the matrix corresponding to the new clause h be represented by $M(h)$. Let $\mathcal{C}_\Pi$ and $\mathcal{C}_h$ be the set of columns of $M_\Pi(\mathcal{F})$ and $M(h)$, respectively. Let B_h be the set of biform variables x such that there are some t € $\mathcal{C}_h$, j € Cn such that $M(x,i) = M_\Pi(\tilde{x},j) = 1$, or there are some t € $\mathcal{C}_h$, j € Cn such that $M(\tilde{x},i) = M_\Pi(x,j) = 1$. For such x £ B_h let C_Π^{+x} C $\mathcal{C}_\Pi$ be the set of columns t such that $M_\Pi(x,0 = 1$, and C_Π^{-x} C Cn be the set

of columns i such that $M_\Pi(\tilde{x}, i) = 1$. Let us assume that consensus is possible between $M_\Pi(\mathcal{F})$ and $M(h)$ with respect to some variable *x*. Let $M_{new}(h)$ be a new matrix obtained by augmenting the matrix A^x representing the set of consensus with respect to x, between columns of $M(h)$ and columns of $M_\Pi(\mathcal{F})$. For the matrix $M_{new}(h)$, let C^{+x}_{new} and C^{-x}_{new} be the same as C^{+x}_Π and C^{-x}_Π defined for $M_\Pi(\mathcal{F})$. With these notations, the incremental algorithm IPIA of **Kean** and **Tsiknis'** is oulined below. The function append appends two matrices.

Algorithm IPIA

INPUT: MATRICES $M_\Pi(\mathcal{F})$ AND $M(h)$

OUTPUT: MATRIX REPRESENTING THE SET OF ALL PRIME IMPLICATES **of** *T* A *h*

STEP 0: Initialize $M_{new}(h) = M(h)$

STEP 1: Delete all absorbed columns from the matrices $M_\Pi(\mathcal{F})$ and $M_{new}(h)$.

If $M_{new}(h)$ is deleted STOP.

STEP 2: If $B_h \neq \phi$ SELECT a row r corresponding to a variable $x \in B_h$.

STEP 3: For each t € C^{+x}_Π and *j* € C^{-x}_{new}

STEP 3.1: COMPUTE $A^x_{\cdot}$

STEP 3.2: Augment to $M_{new}(h)$ the vector A^x_{ij}

STEP 3.3: For each *i* 6 C^{-x}_Π and *j* € C^{+x}_{new}

STEP 3.4: COMPUTE A^x_{ij}.

STEP 3.5: Augment to $M_{new}(h)$ the vector A^x_{ij}

STEP 4: Delete all absorbed columns from $M_\Pi(\mathcal{F})$ and $M_{new}(h)$.

STEP 5: Update $B_h = B_h - \{x\}$

STEP 6: IF $B_h \neq \phi$ GO TO STEP 2

else $M(\mathcal{F} \wedge h) =$ **append**$(M_\Pi(\mathcal{F}), M_{new}(h))$.

END

The algorithm IPIA is similar to that of the Tison's algorithm with two differences: Firstly, in this algorithm, in order to select the biform variables, only the newly transmitted clause *h* is considered, i.e., if the compliment of the literals in the new clause is present in any of the prime implicates of $\mathcal{F}$, then consensi between two columns are performed only with respect to those variables. The columns in the matrix representing set of prime implicates are already prime and hence the consensi between those columns are not computed. The consensi are performed between the columns in $M_{\Pi}(\mathcal{F})$ and the columns of the matrix $M_{new}(h)$. This helps to avoid most of the redundant computations. However, the IPIA algorithm in general is applicable only when the additional knowledge is a single clause. If the additional knowledge transmitted by the reasoner to the RMS is a set (say h_1 A $h_2 \ldots$ A h_m) of clauses which is normally the case in any real-life application, then the algorithm IPIA has to be repeatedly invoked for each of the clauses $h_1, h_2, \ldots, h_m$, sequentially. The following example will explain the working of the algorithm.

Example **2.5.1:-**

Let $\mathcal{H}$ = $\{abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ be the formula to be appended to the formula $\mathcal{F}$ considered in Example 2.4.1. In order to compute the $\Pi(\mathcal{F}$ A $\mathcal{H})$ using IPIA, the algorithm has to be applied five times so as to incorporate the five clauses of $\mathcal{H}$. First when h_1 = $\{abcef\}$, $M_{new}(h_1) = (1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0)^T$ where superscript *T* stands for the usual notation of matrix transpose, B_{h_1} = $\{e, f\}$, and $M_{\Pi}(\mathcal{F})$ is as given in Figure 2.4(b). The updated matrix $M_{new}(h_1)$ obtained by augmenting the matrix A^e with the matrix $M_{new}(h_1)$ is given in Figure **2.5(a).** In this matrix the column 1 is absorbed by the column 2 and hence it is deleted. The resulting $M_{new}(h_1)$ is given in Figure 2.5(b). The algorithm proceeds to find the consensi with respect to the variable /. The matrix $M_{new}(h_1)$ augmented with the matrix A^f is given in Figure 2.5(c). It can be observed that columns **1,** 3 and 4 of $M_{\Pi}(\mathcal{F})$ in Figure **2.4(b)** and columns 1 and 3 of

$M_{new}(h_1)$ in Figure 2.5(c) are absorbed by column 2 of $M_{new}(h_1)$ of Figure 2.5(c). Thus $M_{\Pi}(\mathcal{F})$ corresponds to the single clause *{abdf g}* and $M_{new}(h_1)$ corresponds to *{abc}*. These two matrices together give the prime implicates of $\mathcal{F}$ A *abcef* which is given in Figure 2.6(a).

$M_{new}(h_1) =$

	1	2
a	1	1
$\tilde{a}$	0	0
b	1	1
$\tilde{b}$	0	0
c	1	1
$\tilde{c}$	0	0
d	0	0
$\tilde{d}$	0	0
e	1	0
$\tilde{e}$	0	0
f	0	0
$\tilde{f}$	1	1
g	0	0
$\tilde{g}$	0	0

(a)

$M_{new}(h_1) =$

	1
a	1
$\tilde{a}$	0
b	1
$\tilde{b}$	0
c	1
$\tilde{c}$	0
d	0
$\tilde{d}$	0
e	0
$\tilde{e}$	0
f	0
$\tilde{f}$	1
g	0
$\tilde{g}$	0

(b)

$M_{new}(h_1) =$

	1	2	3
a	1	1	1
$\tilde{a}$	0	0	0
b	1	1	1
$\tilde{b}$	0	0	0
c	1	1	1
$\tilde{c}$	0	0	0
d	0	0	1
$\tilde{d}$	0	0	0
e	0	0	0
$\tilde{e}$	0	0	0
f	0	0	0
$\tilde{f}$	1	0	0
g	0	0	1
$\tilde{g}$	0	0	0

(c)

Figure 2.5

To this end we begin to incorporate the next clause from the additional formula $\mathcal{H}$. We begin with $M_{new}(h_2) = (1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,1)^T$ (the matrix representing the clause $h_2 = adfg$) and $M_{\Pi}(\mathcal{F})$ is $M_{new}(h_2)$ given in Figure 2.6(a). For this problem, the set $B_h = \{/\}$ and A^f is computed which represents the clause *abdg*. This subsumes column 2 of $M_{\Pi}(\mathcal{F} \wedge h_1)$ (Figure 2.6(a)) and hence, the prime implicates of the formula $\mathcal{F} \wedge h_1 \wedge h_2$ are $\{abc\}, \{adfg\}$ and $\{abdg\}$, which is given by the matrix in Figure 2.6(b). Proceeding in this manner, the prime implicates obtained after the incorporation of abf, $\tilde{a}\tilde{c}ef$,

and *aceg* are given by the matrices in Figure 2.6(c), Figure 2.7(a), and Figure 2.7(b), respectively. Thus, Figure 2.7(b) gives the matrix representing the prime implicates for the formula $\mathcal{F}$ A $\mathcal{H}$.

$M_{new}(h_2) = (M(\mathcal{F} \wedge h_1))$

	1	2
a	1	1
$\tilde{a}$	0	0
b	1	1
$\tilde{b}$	0	0
c	0	1
$\tilde{c}$	0	0
d	1	0
$\tilde{d}$	0	0
e	0	0
$\tilde{e}$	0	0
f	1	0
$\tilde{f}$	0	0
g	1	0
$\tilde{g}$	0	0

(a)

$M_\Pi(\mathcal{F} \wedge h_i) = i = 1,2.$

	1	2	3
a	1	1	1
$\tilde{a}$	0	0	0
b	1	0	1
$\tilde{b}$	0	0	0
c	1	0	0
$\tilde{c}$	0	0	0
d	0	1	1
$\tilde{d}$	0	0	0
e	0	0	0
$\tilde{e}$	0	0	0
f	0	0	0
$\tilde{f}$	0	1	0
g	0	1	1
$\tilde{g}$	0	0	0

(b)

$M_\Pi(\mathcal{F} \wedge h_i) = i = 1,2,3.$

	1	2	3	4
a	1	1	1	1
$\tilde{a}$	0	0	0	0
b	1	0	0	0
$\tilde{b}$	0	1	0	0
c	1	0	1	0
$\tilde{c}$	0	0	0	0
d	0	0	0	1
$\tilde{d}$	0	0	0	0
e	0	0	0	0
$\tilde{e}$	0	0	0	0
f	0	1	1	0
$\tilde{f}$	0	0	0	0
g	0	0	0	1
$\tilde{g}$	0	0	0	0

(c)

Figure 2.6

$M_\Pi(\mathcal{F} \wedge h_i) = i = 1,2,3,4.$

	1	2	3	4	5	6
a	1	1	1	1	0	0
$\tilde{a}$	0	0	0	0	1	0
b	1	0	0	0	0	0
$\tilde{b}$	0	1	0	0	0	1
c	1	0	1	0	0	0
$\tilde{c}$	0	0	0	0	1	1
d	0	0	0	0	0	0
$\tilde{d}$	0	0	0	0	0	0
e	0	0	0	0	1	1
$\tilde{e}$	0	0	0	0	0	0
f	0	1	1	0	1	1
$\tilde{f}$	0	0	0	0	0	0
g	0	0	0	1	0	0
$\tilde{g}$	0	0	0	0	0	0

(a)

$M_\Pi(\mathcal{F} \wedge \mathcal{H}) =$

	1	2	3	4	5	6	7	8
a	1	1	1	1	0	0	0	0
$\tilde{a}$	0	0	0	0	1	0	1	0
b	1	0	0	0	0	0	0	0
$\tilde{b}$	0	1	0	0	0	1	0	0
c	1	0	1	0	0	0	0	0
$\tilde{c}$	0	0	0	0	1	1	1	1
d	0	0	0	1	0	0	0	0
$\tilde{d}$	0	0	0	0	0	0	0	0
e	0	0	0	0	1	1	1	1
$\tilde{e}$	0	0	0	0	0	0	0	0
f	0	1	1	0	1	1	0	0
$\tilde{f}$	0	0	0	0	0	0	0	0
g	0	0	0	1	0	0	1	1
$\tilde{g}$	0	0	0	0	0	0	0	0

(b)

Figure 2.7

A *consensus tree* representation for the implementation of IPIA, and various refinements to IPIA have been proposed [Kean 90]. The different refinements suggested are,

(i) Root *optimization* that recommends to terminate the search for consensi with respect to the biform variable x, and makes the consensus of C and C_i the root of a new consensus tree if the consensus of C and C; with respect to the biform variable x subsumes C. Obviously, any further consensi involving C will be subsumable by consensi involving $Con(C, C_{i,x})$. However, the problem of finding such Con(C, C_i,x) quickly is not addressed by the algorithm.

(ii) *Single biform selection* restricts to consider only entries p from $\Pi(\mathcal{F})$ having the property $p \cap C = \{x_i\}$[1] when looking for consensi with C with respect to the ith biform variable x_i. Prime implicates from $\Pi(\mathcal{F})$ not having this property will not generate a consensus but a tautology.

(iii) The *history* [Kean 90] of a clause contains all the biform variables of C that were involved in the chain of consensus operation that generates the clause. The *history restriction* suggests that when looking for consensi with a clause C_i, any clause p in $\Pi(\mathcal{F})$ for which $p \cap history(C_i) = 0$ are only to be considered. For clause C and for each clause derived by consensus, its history is defined as

$history(C) = 0;$

$history(C_i, C_j, x) = history(C_i) \cup \{x\}.$

It is claimed that the cases where $p \cap history(C_i) \neq 0$ will introduce variables from C_i's history unnecessarily.

(iv) Local subsumption check suggests an early elimination of subsumed parent nodes and arc clauses. For each node of the tree if any node subsumes the parent node, then that node subsumes all the children of the parent node that are generated at that stage. Similarly, if the node subsumes an arc which connects it to the parent node, then that

[1] If $p = \{a_1, a_2, \ldots, a_k\}$, then $p = \{\tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_k\}$

node will subsume all the nodes resulting from any node of the tree and the arc. Hence, early elimination of such nodes and arcs are obviously advantageous.

But as cited by Jackson in [Jackson 92], the history restriction and the local subsumption checking interact to cause incompleteness. Moreover, it has already been demonstrated that the method is not suitable if the additional knowledge transmitted by the reasoner to the RMS is represented by more than a single propositional clause. Generally, in the RMS framework, the additional knowledge transmitted by the reasoner to the RMS may be a set of clauses rather than a single clause. In this case, IPIA algorithm is not efficient for the simple reason that each of the clauses in the set has to be treated individually, which is time consuming. Ideally the prime implicates $\Pi(\mathcal{F} \wedge \mathcal{H})$ should be obtained from $\Pi(\mathcal{F})$ and $\Pi(\mathcal{H})$. Further, an algorithm which treats the additional knowledge collectively would be good for RMS update problems.

This thesis suggests a more efficient algorithm which computes the prime implicants of an updated formula incrementally. Moreover, the algorithm proposed in this thesis can treat the additional set of clauses collectively, and compute the set of prime implicants for the updated knowledge. Thus the method proposed is efficient in global as well as in incremental modes.

2.6 Paths in a Binary Matrix

The consensus method of Tison and its modifications, compute the prime implicates of a formula. Socher [Socher 91] proposes an algorithm to compute the prime implicants of a formula, using a concept of paths in a binary matrix. This section introduces the definition of path in a binary matrix and few notations used in this dissertation.

Definition 2.6.1:- *A path in a binary matrix is a set p of literals such that*

$$\forall i \in \mathcal{C}, \exists x \in p \text{ such that } M(x,i) = 1$$

Note:- A path is a conjunctive clause if the matrix represents a CNF formula and disjunctive clause if the formula is in DNF.

In this thesis unless otherwise stated it has to be assumed that a path is a conjunctive clause since the formula $\mathcal{F}$ is assumed to be in CNF. All the properties of conjunctive clauses hold good for paths. A path *p subsumes* another path *q* if and only if *p* C q, and is *fundamental* if and only if for all $x \in$ p, $x \notin p$.

Definition **2.6.2:-** *A path p is complete if it is fundamental and has exactly one entry from the set C of literals corresponding to each column.*

Definition **2.6.3:-** *A complete path p is prime if and only if for all complete paths* q, q C *p if and only if q = p.*

Definition **2.6.4:-** *For* S C C, *T* C *C,* $M[S,$ T] *is the submatrix of* $M(\mathcal{F})$ *consisting of columns in S and ignoring the rows in* $\mathcal{T}$.

$P[S,\mathcal{T}]$ denotes the set of all prime paths of $M[S,\mathcal{T}]$. By definition 2.6.4 $M[\mathcal{C},\phi]$ represents the fundamental formula $\mathcal{F}$.

Definition **2.6.5:-** *With the usual union notation* U, **concatenation,** *denoted by the operator* $\uplus$, *of any two paths p and q is defined as*

$$p \uplus q = \begin{cases} \text{I p U q ;} & \text{if } p \text{ U q is fundamental} \\ \text{not defined ;} & \text{otherwise} \end{cases}$$

Definition **2.6.6:-** *For* $P[S',\mathcal{T}] \neq \phi$ *and* $P[S'',\mathcal{T}] \neq \phi$ *when* S', S'' *C S and T C C,*

$$P[S', T] \text{ y } P[S'',\mathcal{T}] = \{p \text{ U q} \quad p \in P[S',\mathcal{T}] \text{ and } q \in P[S'',\mathcal{T}]\}$$

Socher's algorithm uses this concept of path in a binary matrix. Properties of paths are discussed in Chapter 3. Using the above definitions and notations Socher's algorithm is elaborated in the next Section.

2.7 Socher's Algorithm

The multiplication algorithm Transform proposed by Socher [Socher 91] computes the prime implicants of a prepositional formula *T* by computing the prime paths in a binary matrix representing the formula. The main difference of Socher's method from the Semantic tree method of Slagle *et. al.* [Slagle 70] is in the data structure used. The Slagle's method computes the prime implicants using a concept of path in a Semantic tree where as Socher's algorithm computes the prime implicants of a formula using the concept of path in a binary matrix. This section discusses Socher's algorithm using the notations and definitions cited in Section 2.6.

For any literal r 6 £ define a set S_r *C S* as S_r = *{i* € *C* \ $M(r,i) = 1\}$. The following algorithm, TRANSFORM, outlines Socher's multiplication algorithm.

Algorithm TRANSFORM

INPUT: M[5,7], THE CURRENT MATRIX

OUTPUT: SET OF ALL PRIME PATHS *P[S,T]* OF *M[S,T]*

STEP 0: *T = ft* $P[S,T] = \phi$; INITIALIZATION

STEP 1: Delete absorbed columns; if *M* $[S,T]$ is zero matrix return *P[S, T]*

STEP 2: Select r $\notin T$

STEP 3: Compute $P[S - S_r, T \cup \{r, \tilde{r}\}]$

STEP 4: Update $P[S,T]$ = *{p* ⊎ $\{r\}$ | *p* € *P[S* - S_r, *T* U $\{r,\tilde{r}\}]\}$

STEP 5: Update $T = T$ U $\{r\}$.

If $T \neq \mathcal{L}$

Go to step 1.

STEP 6: Carry out subsumption on

P[S,T] to obtain the residue *P*

STEP 7: Update $P[S,T] = V$.
END

Remark 2.7.1 *The literal r in STEP S can be selected arbitrarily. However, a heuristic is recommended by Socher [Socher 91], to select* r *by finding the row having the maximum number of 1s in* $M(\mathcal{F})$.

The correctness of the algorithm has been proved by Socher in [Socher 91] where certain refinements of the algorithm are also proposed. However, certain properties of the algorithm are independently proved in this thesis. Since such analysis are more apt in the following chapter where a new algorithm is proposed, the theoretical analysis of the algorithm is given in Chapter 3. The following example further explains the working of the algorithm TRANSFORM.

Example **2.7.1:-**

Let $\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg, abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ represent a formula. The clause matrix $M[\mathcal{C},\phi]$ for this formula is given by the following matrix, where $\mathcal{C} = \{1,2,\ldots,10\}$, and £ = $\{a,a,b,b,c,\tilde{c},d,\tilde{d},e,\tilde{e},f,f,g,\tilde{g}\}$.

For the matrix $M[\mathcal{C},\phi]$, the algorithm TRANSFORM works as follows: Initially $T = fa$ $S = \mathcal{C}$, and $P[S,\phi] = \phi$. Applying the heuristic mentioned earlier (Remark 2.7.1), literal 'a' is chosen as r in Step 2. $S_r = S_a = \{1,2,3,4,5,6,7,8\}$ and $P[S-S_a, \{a,\tilde{a}\}] = P[\{9,10\}, \{a,\tilde{a}\}]$. In Step 3, the algorithm computes $P[\{9,10\}, \{a,\tilde{a}\}]$ which axe $\{\tilde{c}\}$,*{e}* and $\{f,g\}$. These are concatenated with $\{a\}$ in Step 4 to obtain the paths $\{a,\tilde{c}\}, \{a,e\}$ and $\{a,f,g\}$. These are the only paths containing a and hence in the Step 5, the row corresponding to 'a' is deleted from the matrix by updating $T = \{a\}$, and the algorithm starts afresh with the new matrix $M[\mathcal{C},T]$.

$$M(\mathcal{F}) = M(C, \phi) =$$

	1	2	3	4	5	6	7	8	9	10
a	1	1	1	1	1	1	1	1	0	0
$\tilde{a}$	0	0	0	0	0	0	0	0	1	1
b	1	1	1	1	1	1	0	0	0	0
$\tilde{b}$	0	0	0	0	0	0	0	1	0	0
c	1	0	1	1	1	1	0	0	0	0
$\tilde{c}$	0	1	0	0	0	0	0	0	1	1
d	1	1	0	0	0	0	1	0	0	0
$\tilde{d}$	0	0	0	1	1	0	0	0	0	0
e	0	0	1	0	0	1	0	0	1	1
$\tilde{e}$	1	0	0	0	0	0	0	0	0	0
f	0	1	1	0	0	0	0	1	1	0
$\tilde{f}$	0	0	0	1	0	1	1	0	0	0
g	0	1	0	0	1	0	1	0	0	1
$\tilde{g}$	0	0	0	0	0	0	0	0	0	0

In this iteration, at Step 2, the literal 'b' is chosen as r and P[{7,8,9,10}, $\{a,b,b\}$] is computed in Step 3. In order to compute $P[\{7,8,9,10\}, \{a,b,b\}]$ the algorithm first chooses the literal *a* and computes $P[\{7,8\}, \{a,$a, $b,b\}]$. The paths are $\{d,f\}$ and $\{f,g\}$, which are concatenated with *a* to get $\{\tilde{a},d,/\}$ and $\{\tilde{a},f,g\}$. These are the only paths in $P[\{7,8\},\{a,b,b\}]$ containing a. At this stage, the algorithm temporarily deletes the row corresponding to *a* and proceeds to compute the other paths in $P[\{7,8\}, \{a,$a, $b,b\}]$. For this, the literal c is chosen and computes $P[\{7,8\},\{a,$ a, $b,b,c,\tilde{c}\}]$ which are $\{d,f\}$ and *{/, g}*. This is the same as $P[\{7,8\}, \{a,$a, $b,b\}]$ computed earlier. These are concatenated with c to get $\{\tilde{c},d,f\}$ and $\{\tilde{c},f,g\}$.

Further, the algorithm deletes the row corresponding to the literal c and proceeds to find $P[\{7,8\},\{a,\tilde{a},b,b,c,\tilde{c},$ e, $\tilde{e}\}]$, on choosing e as the next literal. The paths are $\{d,f\}$ and $\{f,g\}$, which are also the same as paths in $P[\{7,8\},\{a,\tilde{a},b,\tilde{b}\}]$, and are concatenated with the literal e to get the paths $\{d,e,f\}$ and $\{e,f,g\}$. Thus the paths $P[\{7,8,9,10\},\{a,b,b\}]$ computed so far are $\{\tilde{a},d,f\},\{\tilde{a},f,g\},\{\tilde{c},d,f\},\{\tilde{c},f,g\},\{d,e,f\}$ and $\{e,f,g\}$. There are still other paths in $P[\{7,8,9,10\}, \{a,b,b\}]$, and in order to

compute them, the literal / is chosen and $P[\{7,10\},\{a,\tilde{a},b,\tilde{b},\tilde{c},e,f,\tilde{f}\}]$ is computed. g is the only path and it is concatenated with / to get $\{f,g\}$. The algorithm cannot proceed any further to compute $P[\{7,8,9,10\},\{a,b,b\}]$.

Thus the paths obtained are $\{\tilde{a},d,f\},\{\tilde{a},f,g\},\{\tilde{c},d,f\},\{\tilde{c},f,g\},\{d,e,f\},\{e,f,g\}$ and $\{f,g\}$. All the subsumed paths are weeded out and the prime paths obtained are concatenated with 6 to get the paths $\{\tilde{a},b,d,f\},\{b,\tilde{c},d,f\},\{b,$ rf,e,/$\}$, and $\{b,f,g\}$. At this stage the algorithm updates T as $\{a,$6$\}$ and proceeds to find $P[C,\{a,$a, $b,b,$ c, $\tilde{c}\}]$ on choosing c as the next r in Step 2. Proceeding thus, the prime paths obtained at the end of the algorithm are $\{a,\tilde{c}\}$, $\{a,e\}$, $\{a,f,g\}$, $\{b,f,g\}$, $\{c,f,g\}$, $\{\tilde{a},b,d,f\}$,$\{\tilde{a},b,c,g\}$, $\{\tilde{a},b,c,d\}$, $\{\tilde{a},c,d,f\}$, $\{b,\tilde{c},d,f\}$, $\{b,d,e,f\}$, $\{b,c,e,g\}$, $\{b,c,d,e\}$, $\{c,d,e,f\}$.

A simple inspection reveals that the paths $\{d,$/$\}$ and $\{f,g\}$ are computed many times. This is unnecessary and waste of computing efforts. These redundant computations can be easily avoided. Moreover, Socher's algorithm is not suitable to handle incremental computation of prime implicants.

For more efficient computation, a new algorithm is designed in this thesis which avoids redundant computations performed by Socher's algorithm. The algorithm designed is suitable to handle incremental computation of prime implicants. The theoretical aspects of the new algorithm, its advantages over Socher's algorithm as well as Kean and Tsiknis' algorithm are discussed in Chapter 3.

2.8 Conclusion

The consensus method of Tison and the IPIA of Kean and Tsiknis', to compute the prime implicates of a formula are discussed using the binary matrix representation. The concept of path in a binary matrix is introduced and Socher's algorithm which uses this concept of path is elaborated with example. Attempt is made to show that Socher's algorithm

performs certain redundant computations. In brief, the motivation for development of a new algorithm for prime implicants is discussed here. The theoretical analysis of Socher's algorithm can be seen in Chapter 3 where a new algorithm to compute the prime paths of a formula is presented.

Chapter 3

COMPUTATION OF PRIME PATHS

3.1 Introduction

The prime implicants/implicates have been of great importance in the context of RMS and hypothetical reasoning. So, there is the need to design efficient algorithm to compute the prime implicants/implicates of a formula, in global as well as incremental mode. Though the concept of path computation in a matrix representation of the formula is acceptable, the Socher's algorithm is inefficient as it involves redundant computations. In this chapter, a special scheme to partition the matrix representing the formula is introduced. Paths that contain the literal V and that do not contain the literal V are characterized in this chapter. A concept of *extension* of a path in a submatrix to a bigger matrix is also established in this chapter. Based on these characterizations, Prime Implicant Algorithm using Paths and Extension **(PIAPE)** is proposed, which evades the redundant computations involved in Socher's algorithm [Socher 91]. The major underlying concept of the present algorithm is the fact that the prime paths of a matrix can be obtained by the concatenation of prime paths of submatrices and subsequent removal of the subsumed paths.

The actual execution time of any prime implicants/implicates algorithm depends crucially on the number and expense of the subsumption checks required. Attempts are made by the researchers [Tison 67, Kean 90, Socher 91] to reduce the number of implicants/implicates generated, which in turn certainly reduces the number of subsumption

checks required. Generally, the number of subsumption checks grows faster [de Kleer 94] than the square of the final number of prime implicants. Unfortunately, none of the researchers has discussed how the subsumption checks, the real culprit for the high computation time, are carried out. Any prime implicants/implicates algorithm without such specifications is incomplete. The characterization of prime paths of the submatrix which become prime in a bigger matrix is obtained, and the Prime Implicant Algorithm using Paths **(PIAP),** the refinement of **PIAPE** is presented in Section 3.3.4. The number of subsumption checks reduces substantially since the number of paths considered for subsumption checks is less. The correctness of **PIAPE** and **PIAP** are given in Section 3.3.2 and Section 3.3.5, respectively.

Advantages of **PIAP** over **PIAPE,** Socher's algorithm [Socher 91], and **IPIA** of Kean and Tsiknis' [Kean 90] are discussed in Section 3.4. There are special cases which contribute to the improvement and better implementation, of **PIAP.** These cases are discussed in a later section. Prime paths can be used to check the consistency of a formula, provability of a query and to find the minimal support for a query, to name a few. These issues are discussed in subsequent sections. Based on these discussions, minimal support for a query is redefined in terms of paths. Due to the partition scheme, the algorithm **PIAP** presented here is suitable for the incremental computation of prime paths to facilitate hypothetical reasoning. The Section 3.7 recapitulates the highlights.

3.2 Properties of Prime Paths

In Section 2.6, it has been introduced that $P[S,T]$ is the set of prime paths in the binary matrix $M[S,T]$obtained by taking 5 C C of columns and deleting T C $\mathcal{L}$ of rows of the matrix $M(\mathcal{F})$. Any path in $P[S,T]$ is prime according to the notation and hence will be explicitly stated only when needed. The following definitions and theorems show that

all the prime paths of $M[S,T]$ can be obtained by the concatenation of prime paths of submatrices of $M[S,T]$, followed by the removal of the subsumed paths. In the following subsection, a specific type of partition is introduced which helps in characterizing prime paths containing a literal r and prime paths not containing r.

3.2.1 Partitioning

A property regarding concatenation of prime paths in two submatrices to get a path in the whole matrix is explored here.

Theorem **3.2.1:-** *Let* $S' \subset S \subset \mathcal{C}$ *and* $T \subset \mathcal{L}$, *then* $P[S,T] \subset P[S',T] \uplus P[S - S',T]$.

Proof:-

For any $p \in P[S,T]$ it is to be proved that there exists some path $q \in P[S',T]$ and some path $s \in P[S - S',T]$ such that $p = q \uplus s$. Let us define for any $p \in P[S,T]$,

$$p_1 = \{x \in p \mid M(x,i) = 1 \text{ for } i \in S'\}$$
$$p_2 = \{x \in p \mid M(x,i) = 1 \text{ for } i \in S - S'\}$$

Clearly, $p_1 \uplus p_2$ is a path in $M[S,T]$ in the sense that for every $x \in p_1 \uplus p_2$, there exists some $i \in S$ such that $M(x,i) = 1$.

p is prime in $M[S,T]$ and hence it is complete in $M[S,T]$. Therefore, the path p_1 is also complete in $M[S',T]$. Let us assume that p_1 is not prime. Then there is a complete path q_1 in $M[S',T]$ subsuming p_1. $q_1 \uplus p_2$ is a complete path in $M[S,T]$. If $q_1 \uplus p2$ subsumes p, then p is not prime which is a contradiction. If not, then $q_1 \uplus p2 = p$. Now if q_1 is not prime, then, following the same arguments, there is a q_2 in $M[S',T]$ subsuming q_1 and $q_2 \uplus P2 = p$. Repeating this way, the process results in arriving at a path $q \in P[S',T]$ such that $q \uplus P2 = P$-

Applying the same arguments for p2 from this point onward, a path s € $P[S - S', T]$ is obtained such that $q \uplus s = p$. This completes the proof. •

A more general result which takes into account any pair of subsets S' and S'' can be stated as the following corollary. The special case in Theorem 3.2.1 can be obtained by considering $S' \cap S'' = \phi$ and $S = S' \cup S''$.

Corollary **3.2.1:-** *For* $S', S'' \subseteq \mathcal{C}$ *and* $T \subset \mathcal{L}$, $P[S' \cup S'', T] \subset P[S', T] \uplus P[S'', T]$.

The proof of the Theorem 3.2.1 is independent of the fact that $S' \cap S - S' = \phi$. Hence the corollary directly follows from the Theorem 3.2.1.

From the above results, it is seen that if the set of clauses is divided into two sets, then the prime paths of each of the subsets, when concatenated, give rise to paths for the whole set of clauses. However, such a path may not be a prime path as it is complete but not necessarily defined due to violation of fundamentality. Moreover, the above process does not take into account the fact that these paths may be subsumed by some other paths.

Remark 3.2.1:- *For any* $p_1 \in P[S', T]$ *and* $p_2 \in P[S - S', T]$, $p_1 \uplus p_2$ *is not necessarily a path in* $P[S, T]$.

This may happen due to the following reasons:

1. $p_1 \uplus p_2$ may not be defined because $p_1 \cup p_2$ **may not be** fundamental even if p_1 and P2 are individually fundamental. For example, let $p_1 = \{c,/\}$ and p2 = $\{b,/\}$. Though both p_1 and p2 are individually fundamental, $p_1 \cup p_2 = \{b, c, f, \tilde{f}\}$ is not fundamental.

2. There may be some path $q \in P[S, T]$ which subsumes $p_1 \cup p_2$. For example, let $p_1 = \{c, d\}$, p2 = $\{\tilde{b}, \tilde{f}\}$ **and** $p_3 = \{b, d\}$. $p_1 \uplus p_2 = \{\tilde{b}, c, d, \tilde{f}\}$ and $p_1 \uplus p_3 = \{\tilde{b}, c, d\}$. It is evident that $p_1 \uplus$ P2 is subsumed by $p_1 \uplus p_3$.

The subset relationship in Corollary 3.2.1 cannot be made sharper as such because $p_1 \uplus$ P2 is not always prime. Precisely due to this reason, the characterization made in Theorem 3.2.1 is partial and is not expressed in terms of equality.

For the purpose of efficient computation, it is assumed that $S' \cap S'' = \phi$ as this avoids the overlapping of some clauses and hence avoids repeated computation. Thus, for $S \subset \mathcal{C}$, and $T \subset \mathcal{L}$ of any matrix $M[S, \mathcal{L}\text{-}T]$, define $S_r = \{i \in S : M(r,i) = 1\}$ for some (to be chosen later) literal r in $\mathcal{L} - \mathcal{T}$. By definition, S_r contains all columns in $M[S, \mathcal{L} - T]$ which has 1 in the r^{th} row. Using the above notations, the following section discusses certain properties of the set of prime paths not containing a literal r.

3.2.2 Prime paths void of the literal *r*

For any matrix $M[S, \mathcal{T}]$, 5V and $S - S_r$ are two nonintersecting subsets of 5. Hence, based on the partitioning technique described in Section 3.2.1, the prime paths which do not contain a literal r can be characterized. The specific partitioning scheme followed here is that the set S is partitioned into S_r and $S - S_r$.

This partitioning scheme does not directly affect the results here. However, it becomes useful later to characterize the prime paths containing the literal r.

Thus, the following corollary to Theorem 3.2.1 gives the partial characterization of prime paths which do not contain the literal r.

Corollary **3.2.2:-** *The set of prime paths of* $M[S, \mathcal{T}]$ *which do not contain r is a subset of* $P[S_r, \mathcal{T} \cup \{r\}] \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$.

Proof:-

A prime path in $M[S, \mathcal{T}]$ which does not contain r is a path in $P[S, \mathcal{T} \cup \{r\}]$. By definition, $S_r \subset S$. Therefore, by substituting S_r for S' and $\mathcal{T} \cup \{r\}$ for T in Theorem 3.2.1, we obtain $P[S, T \cup \{r\}] \subset P[S_r, T \cup \{r\}] \uplus P[S - S_r, T \cup \{r\}]$. Hence the proof. •

Due to the arguments in Remark 3.2.1, this characterization is partial. Except for the subsumption, the characterization is complete. However, special characteristics of the set of prime paths which contain the literal r can be completely identified. The properties of prime paths containing the literal r are discussed in the following subsection.

3.2.3 Prime paths containing the literal r

It is seen that partitioning 5 using the literal r as S_r and $S - S_r$ helps in partial characterization of paths which do not contain r. The same partitioning can be effectively used to completely characterize prime paths containing the literal r. The underlying principle is that the only paths in $P[S_r, \mathcal{T}]$ that contain r is $\{r\}$ and there is no path in . $P[S - S_r, \mathcal{T}]$ which contains r. This can be formally stated as the following theorem.

Theorem **3.2.2:-** *The set of prime paths of $M[S, \mathcal{T}]$ which contain r is a subset of*

$$\{r\} \biguplus P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}].$$

Proof:-

A path in $P[S, \mathcal{T}]$ that contains r does not contain f and hence it is also a path in $P[S, \mathcal{T} \cup \{\tilde{r}\}]$. By Theorem 3.2.1, $P[S, \mathcal{T} \cup \{\tilde{r}\}] \subset P[S_r, \mathcal{T} \cup \{\tilde{r}\}] \uplus P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}]$. Since no path in $P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}]$ contains f, every path in $P[S, \mathcal{T}]$ containing r is generated by the concatenation of some path in $P[S_r, \mathcal{T} \cup \{\tilde{r}\}]$ containing r and a path in $P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}]$. But the only path in $P[S_r, \mathcal{T} \cup \{\tilde{r}\}]$ that contains r is $\{r\}$ by the definition of S_r. Hence, the paths in $P[S, \mathcal{T}]$ that contain r are obtained by the concatenation of $\{r\}$ with paths in $P[S - S_r, T \cup \{\tilde{r}\}]$. Hence the proof. •

Remark 3.2.2:- ***The set $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}]$ contains only prime paths that contain the literal r and hence can be written equivalently as $\{r\} \uplus P[S - S_r, \mathcal{T}]$ or, $\{r\} \cup P[S - S_r, \mathcal{T} \cup \{\tilde{r}\}]$ or, $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$. In this thesis, the set of paths containing the***

literal r is denoted by $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$ *unless there is an explicit need for using other notations.*

The characterization of prime paths not containing the literal r and of prime paths containing the literal r are given in Corollary 3.2.1 and Theorem 3.2.2, respectively. Thus, by the above results, all the paths in $P[S, \mathcal{T}]$ can be obtained by the union of the sets $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$ and P[S,T U $\{r\}$] subject to fundamentally and subsumption criteria. Socher's algorithm [Socher 91] is essentially based on this principle.

The converse of Theorem 3.2.2 is, in general, not true. There may be paths in the set $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$ which are not prime paths, i.e., there may be paths in the set $\{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$ which are subsumed by some *other paths* in $P[S, \mathcal{T}]$. For example, if p = $\{a, b, c\} \in P[S, \mathcal{T} \cup \{r\}]$, then $p \in P[S, \mathcal{T}]$ since p is fundamental, complete and prime in $P[S,$ T U $\{r\}]$, and subsumes p te) $\{r\} = \{a, b,$ c, $r\}$. The *other paths* are precisely the complete paths in $M[S - S', \mathcal{T}]$ for some nonempty subset S' of 5 **which** admits the possibility of being complete when considered as paths in $M[S, \mathcal{T}]$. The properties of paths containing r which may be subsumed by some *other paths* in $P[S,$ T] are discussed in Section 3.3.3. If q is a path in $M[S - S', \mathcal{T}]$ which admits the possibility of being prime in $M[S, \mathcal{T}]$, then for every i G S', there is some $x \in q$ such that $M(x, i) = 1$. This motivates us to lay out the definition of *extension of* a *path* in the following subsection.

3.2.4 Extension of a path

The extension of a path p in a submatrix to a bigger matrix is formally stated as follows.

Definition **3.2.1:-** *Let* $p \in P[S - S', \mathcal{T}]$. *Then the extension of a path p to the matrix* $M[S', \mathcal{T}]$ *is defined as the set*

$$E(p, S') = \{i \in S' \mid M(x, i) = 1 \text{ for some } x \in p\}.$$

By extension of a path p in $P[S - S', T]$ we mean that it is the set of columns in S' having a nonzero entry for the literals in the path p. This concept is pictorially illustrated in Figure 3.1. All boxes represent a contiguous blocks of 1s along the row in the matrix. The dark boxes (▬) represent a path p = $\{r_1, r_2, r_3, r_4\}$ in $M[S - S', T]$. The shaded boxes (▨) represent the set of 1s in the submatrix $M[S', T]$ along the rows r_1, r_2 and r_4. In other words, the shaded boxes correspond to columns *i* such that $M(x, i) = 1$ for some $x \in p$. Hence, these columns represent the *extension* of p with respect to 5. In Figure 3.1, the set S'' is the extension $E(p, S')$. p is a path in $M[S - S', T]$ as well as in $M[S'', T]$. Hence the path p is extended to a larger submatrix $M[S - S' \cup S'', T]$ of the original matrix. In Figure 3.1, the boxes (▭) represent contiguous block of 1s for some literals in the formula.

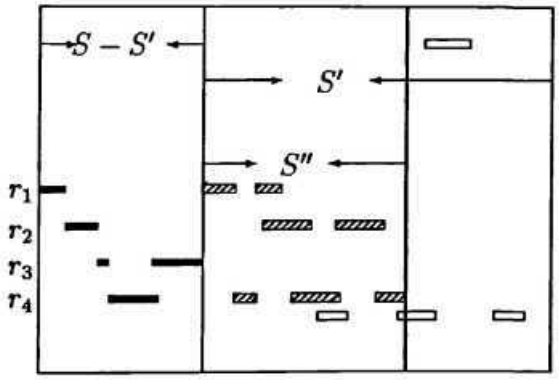


Figure 3.1: Extension of a path p ($E(p, S') = S''$)
▬ represents path p = $\{r_1, r_2, r_3, r_4\}$ in $S - S'$
▨ represents the set of 1s in the columns S'' along the rows r_1, r_2 and r_4.
▭ represents contiguous block of 1s for some literals.

Consider the Example 2.7.1 where $S_b = \{1, 2, 3, 4, 5, 6\}$, p = $\{f, g\} \in P[S - S_b, b]$. $E(p, S_b) = \{2, 3, 5\}$ since $M(f, 2) = M(f, 3) = M(g, 5) = 1$.

With this concept of extension of a path and the fact that the occurrence of a literal r in any clause precludes any occurrence of the literal f in it, some more properties of prime

paths are described below.

Theorem 3.2.3:- *If* $p \in P[S - S', T]$ *then* $p \in P[((S - S') \cup E(p, S')), T]$.

Proof:-

If $p \in P[S - S', T]$, then for all x 6 p there exists some $i \in 5 - S'$ such that $M(x, i) = 1$. By the Definition 3.2.1 of $E(p, S')$, if $j \in E(p, S')$, then there is some x 6 p such that $M(x,j) = 1$. Then, clearly, p is a path in $M[((S - S') \cup E(p, S')), T]$. Since p is a prime path in $M[S - S', T]$, it is complete. Let us assume that p is not prime in $M[((S - S') \cup E(p, S')), T]$. Then there is a complete path q in $P[((S - S') \cup E(p, S')), T]$ that subsumes p. Since $5 - S' \subset (5 - S') \cup E(p, S')$, there is a complete path $q_1 \subset q$ in $P[5 - S', T]$ such that q_1 subsumes p in $P[5 - S', T]$. This contradicts the **primeness** of the path p in $P[S - S', T]$. Hence the theorem is proved. •

For some prime paths p in $M[S - S_r, T \cup \{r\}]$, the extension of p to $M[S_r, T \cup \{r\}]$ is S_r. The following corollary to the Theorem 3.2.3 characterizes such prime paths.

Corollary **3.2.3:-** ***Let*** *p be a prime path in* $M[S - S_r, T \cup \{r\}]$. *Then the extension* $E(p, S_r) = S_r$ *if and only* ***if*** $p \in P[S, T]$.

Proof:-

($\Rightarrow$) Let us assume that $E(p, S_r) = S_r$. Then, $5 - S_r \cup E(p, S_r) = 5$. Substituting S_r for S' and $T \cup \{r\}$ for T in Theorem 3.2.3, it can be seen that $p \in P[S, T \cup \{r\}]$. $P[S, T \cup \{r\}]$ contains prime paths in $M[S, T]$ void of literal r, and hence is a subset of $P[5, T]$. Therefore, p 6 $P[S, T]$.

($\Leftarrow$) Let $p \in P[S, T]$. p does not contain the literal r since $p \in P[5 - S_r, T \cup \{r\}]$. Since p G $P[S, T]$, for all x 6 p there exists some $i \in 5$ such that $M(x, i) = 1$. By Definition 3.2.1, $E(p, S_r) \subset S_r$. Hence it is only required to prove that $S_r \subset E(p, S_r)$. To this end, let j 6 S_r. Then, since $S_r \subset 5$ and p is complete in $M[S, T]$, there is some $y \in$ p such that $M(y, j) = 1$. Hence $j \in E(p, S_r)$. This completes the proof. •

In Figure 3.2, it can be seen that $E(p, S_r) = S_r$ and hence p is a prime path in the whole matrix.

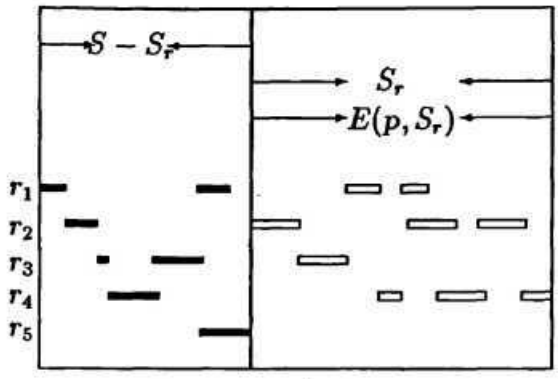


Figure 3.2: Illustration of case where $E(p, S_r) = S$
▬ represents path $p = \{r_1, r_2, r_3, r_4, r_5\}$ in $S - S_r$
▭ represents the set of 1s in the columns S_r along the rows r_1, r_2, r_3 and r_4.

The complete characterization of prime paths having the property $E(p, S_r) — S_r$ for some p € $P[S - S_r, \mathcal{T} \cup \{r\}]$ is established by Theorem 3.2.3 and Corollary 3.2.3. The *other paths* in $P[S, \mathcal{T}]$ mentioned earlier in Section 3.2.3 are the paths having the property $E(p, S_r) = S_r$ for some $S_r \subset 5$. Moreover, any path p € $P[S — S_r, \mathcal{T} \cup \{r\}]$ satisfying $E(p, S_r) = S_r$ need not be concatenated with any q € $P[S_r, \mathcal{T} \cup \{r\}]$ so as to get a path in $M[S, \mathcal{T}]$. This is because any path thus obtained is definitely subsumed by p in $P[S, \mathcal{T}]$. This motivates us to characterize the prime paths which do not satisfy $E(p, S_r) = S_r$. The following theorem characterizes such paths.

Theorem **3.2.4:-** *Let* p € $P[S - S_r, \mathcal{T} \cup \{r\}]$. *Then* $p \uplus \{r\}$ € $P[S, \mathcal{T}]$ *if and only if* $E(p, S_r) \neq S_r$.

Proof:-

($\Rightarrow$) follows from the Corollary 3.2.2 by contraposition since, if $E(p, S_r) = S_r$, then p is a prime path in $P[S, \mathcal{T}]$. This implies that $p \uplus \{r\}$ is not prime.

(<=) Assume that $E(p,S_r)$ + S_r. Since p € $P[S - S_r, \mathcal{T} \cup \{r\}]$ and $\{r\} \in P[S_r, \mathcal{T}]$, $p \uplus \{r\}$ is **defined,** and fundamental if f $\notin$ p, and is a complete path in $M[S, \mathcal{T}]$ **since** for all $x \in p \uplus \{r\}$ there is some i 6 5 such that $M(x, i) = 1$.

Let us assume that p $\uplus$ $\{r\}$ is not prime in $M[S, \mathcal{T}]$. Then there is a complete path q $\in P[S, \mathcal{T}]$ that subsumes $p \uplus \{r\}$. Since $E(p, S_r) \neq S_r$, there is some $i \in S_r$ such that $M(x, i) = 0$ for all x G p. Therefore, if q subsumes p, then $M(x, i) = 0$, for all $x \in q$ and for some $i \in S_r$, contradicting the fact **that** q is a path in $P[S, \mathcal{T}]$. Otherwise, there is a complete path $q_1 \subset$ q - $\{r\}$ in $P[S - S_r, \mathcal{T}]$ $(P[S - S_r, \mathcal{T} \cup \{r\}])$ such that q_1 subsumes p. This contradicts the primeness of p in $P[S - S_r, \mathcal{T}$ U $\{r\}]$. Hence p $\uplus$ $\{r\} \in P[S, \mathcal{T}]$. This completes the proof. •

For example, consider the Example 2.7.1 where S_b = $\{1,2,3,4,5,6\}$ and p = $\{f,g\}$. p $\in P[S - S_b, b]$ and $E(p, S_b)$ = $\{2,3,5\}$. $\{2,3,5\} \neq \{1,2,3,4,5,6\}$, and therefore, $E(p, S_b) \neq S_b$. Hence, p $\uplus$ $\{b\}$ = $\{b,f,g\}$ is a path in $P[S, \mathcal{T}]$.

In Figure 3.3, $E(p, S_r) \neq S_r$ and hence in order to extend the path p to S_r, some literals from $M[S_r, \mathcal{T}]$ have to be concatenated with p. The literal r gives a nonzero entry in all the columns of S_r and hence is concatenated to give a trivial prime path in the whole matrix.

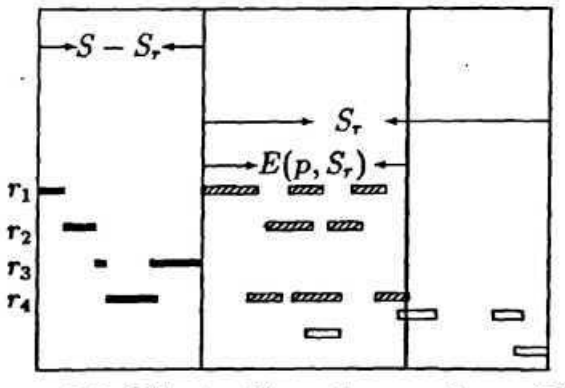


Figure 3.3: Illustration of case where $E(p, S_r) \neq S_r$
— represents path $p = \{r_1, r_2, r_3, r_4\}$ in $S - S_r$
▨ represents the set of 1s in the columns S_r along the rows r_1, r_2 and r_4.
▭ represents contiguous block of 1s for some literals.

Thus, Theorem 3.2.4 establishes the properties of paths p € $P[S - S_r, T \cup \{r\}]$ satisfying $E(p,S_r) \neq S_r$. Moreover, it gives the complete characterization of prime paths in P[5, *T]* containing the literal r. It is to be noted that there are prime paths in the other partition satisfying similar conditions. The following theorem gives the characterization of paths q € $P[S_r, T \cup \{r\}]$ satisfying $E(q, S - S_r) \neq S - S_r$.

Theorem **3.2.5:-** *Let* $p \in P[S - S_r, T \cup \{r\}], q \in P[S_r, T \cup \{r\}]$ *such that* $E(p, S_r) \neq S_r$ *and* $E(q, S - S_r) \neq S - S_r$. *Then* $p \uplus q$ *is a path in* $M[S, T]$ *not containing r.*

Proof:-

p and q are prime paths in $M[S - S_r, T \cup \{r\}]$ and $M[S_r, T \cup \{r\}]$, respectively. Since $S_r \cup S - S_r = S$, p ⊎ q is fundamental and complete in $M[S, T]$. Both p and q are void of the literal r. Therefore, $p \uplus q$ is a path in $M[S, T]$ not containing r. This completes the proof. •

This is a partial characterization of paths not containing the literal r because all the paths so obtained may not always be prime due to the arguments given in Remark 3.2.1. Based on these discussions, the recursive algorithm to compute the prime paths of $M[C, \phi]$ is given in the following section.

3.3 Algorithms to Compute Prime Paths

In the previous section, the properties of prime paths have been established by partitioning the matrix representing the set of clauses into two submatrices and introducing the concept of extension of a path in a submatrix to the other submatrix. Based on these, the Prime Implicant Algorithm using Paths and Extension (PIAPE) which computes the prime implicants of M[S,T] is given below.

3.3.1 Algorithm PIAPE

Algorithm PIAPE

INPUT: *M[S,T]* , THE CURRENT MATRIX

OUTPUT: SET OF ALL PRIME PATHS *P[S,T]* OF *M[S,T]*

STEP 0: Initialization *P[S,T] = fa*

STEP 1: Select a row r $\notin T$

STEP 2: Compute $P[S_r, \mathcal{T} \cup \{r\}]$ and $P[S - S_r, \mathcal{T} \cup \{r\}]$

STEP 3: For **all** $p \in P[S - S_r, \mathcal{T} \cup \{r\}]$, do

if $E(p, S_r) = S_r$,

then do

3.1 Update P[S, T] = P[S, T] U $\{p\}$

3.2 Update P[5 - $S_r, \mathcal{T} \cup \{r\}] = P[S - S_r, \mathcal{T} \cup \{r\}] - \{p\}$

3.3 else Update $P[S,$ T] = $P[S,$ T] U $\{p \uplus r\}$

STEP 4: For all q 6 $P[S_r, \mathcal{T} \cup \{r\}]$, do

if $E(q, S - S_r) = S - S_r$,

then do

4.1 Update P[5, *T]* = $P[S,$ T] U $\{q\}$

4.2 Update $P[S_r, T$ U $\{r\}]$ = P[5 - $S_r, \mathcal{T}$ U $\{r\}]$ - $\{q\}$

4.3 else for all p $\in$ P[5 - $S_r,$ T U $\{r\}]$ such that $E(p, S_r) \neq S_r$

4.4 **Compute** $\mathcal{R} = \{p \uplus q \mid p \in P[S - S_r, \mathcal{T} \cup \{r\}], q \in P[S_r, \mathcal{T} \cup \{r\}]\}$

STEP 5: Update $\mathcal{R} = \mathcal{R} \cup P[S, \mathcal{T}]$

STEP 6: Delete all subsumed paths in ft to obtain the residue $P[S, T]$

END

3.3.2 Correctness of the algorithm PIAPE

The algorithm **PIAPE** correctly computes the set *P[S,T]* of all prime paths in *M[S, T]*. The correctness of the algorithm can be derived from the theorems proved earlier in this chapter. By Theorem 3.2.1, for a partition of *S* as S_r and $S - S_r$, every path in *P[S,T]* must be in the form of $p_1 \uplus$ p2 where $p_1 \in P[S_r, T]$ and p2 $\in P[S - S_r, T]$. By Theorem 3.2.2, every path in $P[S_r, T]$ is either {r} or a path not containing *r*. So, every path in $P[S, T]$ is either of the form $\{r\} \uplus$ p2 or, $p_1 \uplus$ P2 when r $\notin p_1$. By Theorem 3.2.3 and Corollary 3.2.3, if $E(p_1, S - S_r) = S - S_r$, then $p_1 \uplus q$ is subsumed by p_1 for any $q \in P[S_r, T]$ and hence, no concatenation should take place with p_1. Using the same argument, if $E(p_2, S_r) = S_r$, then no concatenation should take place with p_2. When the above two conditions are not satisfied, p_1 and p2 can be concatenated due to Theorem 3.2.5. There is no other method to generate a path p in $M[S, T]$. However, all the paths which are generated so far, may not be in $P[S, T]$ as subsumption is not completely taken into account. Hence, as the last step, all paths which are subsumed by some other paths are deleted from the paths thus generated to get $P[S, T]$.

3.3.3 Extension as subsumption

In order to check the possibility of extending the prime path p in $M[S - S', T]$ as & prime path in $M[S, T]$, the computation of set $E(p, S')$ for some $S' \subset S$ is the crucial step in the algorithm **PIAPE.** From the foregoing discussions, it is seen that this step is essential to avoid certain subsumptions. However, this need not be checked explicitly as directed by the algorithm because of the following remark.

Remark **3.3.1:-** *Corollary 3.2.3 allows us to decide whether a path* $p \in P[S - S_r, T \cup \{r\}]$ is a *path in* $P[S, T]$ *or whether it should be extended trivially to a path in* $P[S, T]$. *This leads to a general problem of checking* $E(p, S_r) - S_r$, *for some* $S_r \subset S$. *This can be*

accomplished by a simple technique if $P[S_r, \mathcal{T} \cup \{r\}]$ is *known. The technique is not only simple, but it also identifies a path in the other partition which is subsumed,*

Thus, in order to check $E(p, S_r) = S_r$, it is enough to check whether there is any $q \in P[S_r, \mathcal{T} \cup \{r\}]$ subsuming $p \in P[S - S_r, \mathcal{T} \cup \{r\}]$, in which case p is in $P[S, \mathcal{T}]$. For p € $P[S - S_r, T \cup \{r\}]$, if $E(p, S_r) = S_r$, then p is in P[S, $\mathcal{T}$] and hence p need not be concatenated with any of q € $P[S_r, \mathcal{T} \cup \{r\}]$ so as to get a path in $P[S, \mathcal{T}]$. This is because any path so obtained by concatenation is definitely subsumed by p. Further, it is to be noted that due to the characterization of prime paths, those paths which have the property $E(p, S_r) = S_r$ are never subsumed by any path in $M[S,T]$ and hence need not be considered for subsumption check. Let $\mathcal{P}1$ be the set of such paths in $P[S, \mathcal{T}]$. A similar argument is used for path q in $P[S_r, T \cup \{r\}]$ having the property $E(q, S - S_r) = S - S_r$. Let $\mathcal{P}2$ be the set of such paths in $P[S, T]$ which are not considered for subsumption check.

By the above discussions, the check involved in Steps 3 and 4 of the algorithm **PIAPE** can be accomplished by the following two checks.

- Whether p ∈ $P[S - S_r, T \cup \{r\}]$, is subsumed by some $q \in P[S_r, T \cup \{r\}]$
- Whether $q \in P[S_r, T \cup \{r\}]$, is subsumed by some p ∈ $P[S - S_r, T \cup \{r\}]$

This checking is similar to that of subsumption checking. Hence, extension check is nothing but a sort of subsumption check. Thus, though these two steps avoid certain subsumptions later, this fact is not properly utilized in the last step of **PIAPE** where subsumption check is carried out for all the paths generated by the algorithm. Further, Theorem 3.2.4 and Theorem 3.2.5 characterize the paths which are to be trivially extended to a path in P[S, T]. These are accomplished by the following steps.

- Extend p as $p \uplus \{r\}$ if f $\notin p$, for p 6 $P[S - S_r, \mathcal{T} \cup \{r\}]$which satisfy $E p, S_r) \neq S_r$.

- Extend p as p ⊎ q, for p € $P[S - S_r, T \cup \{r\}]$ and q e $P[S_r, T \cup \{r\}]$ which satisfy $E(p, S_r) \neq S_r$ and $E(q, S - S_r) \neq S - S_r$.

Further, it is to be noted that those paths p ⊎ {r} obtained when $E(p, S_r) \neq S_r$ are already prime and need not be considered for subsumption. If any path in $P[S, T]$ subsumes p ⊎ {r}, then it has to be p or a subset of p which contradicts the fact that $E(p, S_r) \neq S_r$. If p ⊎ r subsumes any path, then it has to be of the form $p' \uplus q'$ such that p' C p. This contradicts the primeness of p in $P[S - S_r, T \cup \{r\}]$. Thus, in fact, these paths which contain r neither subsume nor is subsumed by any other path and hence can be treated separately. Let $\mathcal{P}3$ be the set of such paths.

Thus, the only paths which are to be considered for subsumption are paths of the form $p \uplus q$ when $E(p, S_r) \neq S_r$ and $E(q, S - S_r) \neq S - S_r$. We have already identified that the paths in $\mathcal{P}1, \mathcal{P}2$ and $\mathcal{P}3$ are never subsumed by any other paths. Hence, the paths of the form p ⊎ q may be subsumed by the paths in $\mathcal{P}1$ or $\mathcal{P}2$. Moreover, earlier deletion of paths which may be subsumed by other paths will definitely reduce the number of paths generated, and as a consequence, the number of subsumption checks required as well as the time taken reduce considerably. Let $\mathcal{P}4$ be the set (initially empty) of paths of the form p ⊎ q. In order to accomplish earlier subsumption, each path of the form p ⊎ q is subjected to the following set of checks,

1. whether there is any p' € $\mathcal{P}1$ such that p' C p ⊎ q.
2. whether there is any p' e $\mathcal{P}2$ such that p' C p ⊎ q.
3. whether there is any $p' \in \mathcal{P}4$ such that p' C p ⊎ q.

The paths of the form p ⊎ q are generated one by one and are subjected to the above three comparisons. If the check fails, then the ptelq is included in the set *P4*. If it is a success, the next path of the form p ⊎ q is generated and so on. This process is continued

till all the paths of the form $p \uplus q$ are computed. Though most of the paths of the form $p \uplus q$ which are subsumed by some other paths are deleted, there may be few paths which are subsumed by some other paths of the form p ⊎ q. These paths are the paths which are generated later. Such paths are removed by performing the following step.

- for all p_i, p_j G $\mathcal{P}4$, (t < *j)* check whether p_j C p_i. The subscripts denote the order in which the paths are generated.

3.3.4 Algorithm PIAP

Based on the above discussions the PIAPE to compute the prime implicants can be restated as follows.

Algorithm PIAP

INPUT: $M[S,T]$, THE CURRENT MATRIX

OUTPUT: SET $P[S,T]$ OF ALL PRIME PATHS OF $M[S,T]$

STEP 0: Initialization $\mathcal{P}1=\phi; \mathcal{P}2=\phi; \mathcal{P}3=\phi; \mathcal{P}4=\phi;$ $P[S,T] = \phi;$

STEP 1: Select a row r ∉ *T*

STEP 2: Compute $P[S - S_r, T \cup \{r\}]$ and $P[S_r, T \cup \{r\}]$

STEP 3: For all p € $P[S - S_r, T \cup \{r\}]$ do

if ∃ q ∈ $P[S_r, T \cup \{r\}]$ such that q C p

then do

3.1 Update $\mathcal{P}1 = \mathcal{P}1 \cup \{p\}$

3.2 Update $P[S - S_r, T \cup \{r\}] = P[S - S_r, T \cup \{r\}] - \{p\}$

STEP 4: For all q € $P[S_r, T \cup \{r\}]$ do

if ∃ p € $P \backslash S - S_r, T \cup \{r\}$ such that p C q

then do

4.1 Update $P2 = P2 \cup \{q\}$

4.2 Update $P[S_r, T$ U {r}] = $P[S - S_r, T$ U {r}] - $\{q\}$

STEP 5: For all $p \in P[S - S_r, T$ U $\{r\}]$ do

5.1 If f $\notin$ p, Compute $p \uplus \{r\}$.

5.2 Update $\mathcal{P}3 = \mathcal{P}3 \cup \{p \uplus \{r\}\}$.

STEP 6:

6.1 For all $p \in P[S - S_r, T \cup \{r\}]$ do

6.2 For all $q \in P[S_r,$T U $\{r\}]$ do

6.3 Compute p $\uplus$ q.

6.4 If *fl* $p' \in \mathcal{P}1$ U $\mathcal{P}2 \cup \mathcal{P}4$ such that p' *C* $p \uplus$ q, then

Update $\mathcal{P}4 = \mathcal{P}4$ U{p $\uplus$ $q\}$.

STEP 7: For all $p_i \in \mathcal{P}4$ do

If 3 *pj* $\in \mathcal{P}4$ such that p_j C p_i $(i < j)$, then

Update $\mathcal{P}4 = \mathcal{P}4 - \{p_i\}$

STEP 8: P[5,T] = $\mathcal{P}1 \cup \mathcal{P}2 \cup \mathcal{P}3 \cup \mathcal{P}4$

END

3.3.5 Correctness of the algorithm PIAP

The correctness of the algorithm PIAP is derived from the correctness of the algorithm PIAPE in Section 3.3.2. Like PIAPE, the algorithm PIAP, at any iteration, generates two submatrices of the current matrix and in turn, generates the prime paths for the whole matrix from the prime paths of both the submatrices. The prime paths in submatrices which are prime paths in the whole matrix are separated out *as PI* and *P2* in Steps 3.1 and 4.1. The paths in these sets are not subjected to subsurnption due to the complete characterization given in Theorem 3.2.3 and Corollary 3.2.3. The correctness of Step 5.1 which computes the prime paths containing the literal r is due to the complete

characterization of such paths given in Theorem 3.2.4. Supported by Theorem 3.2.5, Step 6 computes the set *P4* of paths of the form $p \uplus q$, which do not contain r. The paths which are definitely not prime are deleted, and the paths which may be prime are kept in *P4.* Since the comparison of these paths with those in *P1* and *P2* for subsumption is already performed by Step 6.4, it is only required to know whether there is any path in *P4* which is subsumed by a path in *P4* itself, which is generated later. Hence, the comparisons in Step 7 are performed. Finally, as a last step, the union of the sets *P1, P2, P3* and *P4* thus obtained is taken to get *P[S,T}.*

3.4 Analysis of PIAP

In this section, it is shown that the algorithm **PIAP** is a refinement of the algorithm **PIAPE.** Apart from discussing the advantages of the algorithm **PIAP** over **PIAPE,** an attempt is also made to explain the efficiency of **PIAP** over Socher's algorithm [Socher 91], Experimental results substantiating the superiority of **PIAP** over Socher's algorithm are given in Chapter 4.

3.4.1 PIAP as refinement of PIAPE

In the algorithm **PIAPE,** *P[S,T]* is updated every time a path in M[S,T] is computed, and subsumption is performed only at the end for all the paths thus generated. The fact that the paths satisfying $E(p, S_r) = S_r$ obtained by Step 3.1 admit the possibility of being prime in M[S, T], is not properly utilized while performing subsumption. Similarly, the primeness of the paths obtained by Step 4.1 is also not properly made use of. In order to compare the number of subset checks required in **PIAPE** and **PIAP,** let us assume that among the paths in $P[S_r, T \cup \{r\}]$ there are k_1 paths satisfying $E(p, S_r) = S_r$ and there are m_1 paths not satisfying this property. Hence, $P[S_r, T \cup \{r\}]$ has $m_1 + k_1$

paths. Similarly, let there be $m_2 + k_2$ paths in $P[S, T]$ of which k_2 paths satisfy, and m_2 paths do not satisfy $E(q, S - S_r) = S - S_r$. Then by Steps 3.1 and 4.1 of PIAPE we get $k_1 + k_2$ paths in P[S,T] using $O((m_1 + k_1)(m_2 + k_2))$ number of subsumption checks. It is to be noted that in PIAP, k_1 paths in *P1* and k_2 paths in *P2* are also obtained by $O((m_1 + k_1)(m_2 + k_2))$ subsumption checks which is equal to the number of checks required in PIAPE to get the same prime paths. Hence, this is not considered here for the comparison of number of subsumptions required in PIAPE and PIAP. Assuming that $p \uplus \{r\}$ and $p \uplus q$ are fundamental for all $p \in P[S - S_r, T \cup \{r\}]$ and for all $q \in P[S_r, T \cup \{r\}]$, by Steps 3.3 and 4.4 of PIAPE, we get m_l paths of the form $p \uplus \{r\}$ and $m_1 m_2$ paths of the form $p \uplus q$ in $P[S, T]$. Thus, the total number of paths generated is $k_1 + k_2 + m_1 + m_1 m_2$, and are subjected to subsumption check. The number of subsumption checks at this stage is $O((k_1 \cdot k_2 + m_1 + m_1 m_2)^2)$.

In the algorithm PIAP, the P[S,T] is not updated each time a path in $M[S, T]$ is computed. The prime paths obtained by Step 3.1 and 4.1 having the property of being prime in M[S,T] are treated separately. Thus, as it can be seen, the k^1 paths satisfying $E(p, S_r) = S_r$ are considered as the set $\mathcal{P}1$, and k_2 paths satisfying $E(q, S - S_r) = S - S_r$ are considered as the set $\mathcal{P}2$. The m_1 paths of the form $p \uplus \{r\}$ are considered as the set *P3*. Thus there are $k_1 + k_2 + m_1$ paths which are identified to be prime in P[S, *T]* of which $k_1 + k_2$ paths are used, for subsumption comparison. By Step 6.4, when the first path of the form $p \uplus q$ is computed, $k_1 + k_2$ comparisons are performed to check whether $p \uplus q$ is subsumed by any path which is already computed. If $p \uplus q$ is not subsumed, then it is included in $\mathcal{P}4$. For the next path of the form $p \uplus q$, $k_1 + k_2 + 1$ comparisons are performed, and the process is continued. In the worst case, if none of the paths thus generated is subsumed, the number of comparisons performed is $(k_1 + k_2) + (k_1 + k_2 + 1) + \ldots + (k_1 + k_2 + m_1 m_2)$. Thus $m_1 m_2$ paths of the form $p \uplus q$ are obtained with $O((m_1 m_2 + -1)(2k_1 + 2k_2 + m_1 m_2))$ subset checks. Finally, $(m_1 m_2)(m_1 m_2 - 1)$ comparisons

are performed on P4 in Step *7*.

Thus, in the worst case, the number of subsumption checks required for **PIAPE** and **PIAP** are $O((k_1 + K_2 + m_1 + m_1m_2)^2)$ and $O((k_1 + k_2 + m_1m_2)(m_1m_2 +1))$, respectively. Obviously, **PIAP** requires less number of subsumption checks and hence requires considerably less computation time. Admittedly, no amount of algorithmic improvement can avoid the complexity produced by the sheer number of prime paths.

Though both the algorithms **PIAPE** and **PIAP** compute the same prime paths, **PIAP** is computationally more efficient simply because the subsumption checks performed repeatedly at different levels in **PIAPE** are judiciously avoided in **PIAP.** Even if PI, *P2* and *P3* are empty the algorithms PIAP requires less number of subset checks. If none of the paths generated subsumes any other path generated by the algorithms **PIAPE** and **PIAP,** then both are equally efficient.

3.4.2 Subsumptions in PIAP

Facilitating subsumption checks is the crucial step in all the algorithms to compute prime implicants/implicates. Though a lot of research has been going on to improve the algorithms to compute prime implicants/implicates [Tison 67, Kean 90, Socher 91, Jackson 92, de Kleer 94], the focus has been only in reducing the number of implicants/implicates generated by the algorithms. None of the algorithms except [de Kleer 94] indicates how subsumption is to be performed, and any prime implicants/implicates algorithm is incomplete without such specifications simply because the actual running time depends critically on the number and expense of the subsumption checks required. We have already seen in Section 3.4.1 that the algorithm **PIAP** reduces the number of paths subjected to subset checks. It will be shown in this section that subsumptions performed at earlier stages of the algorithm **PIAP** help to reduce the running time considerably.

In Tison's method, the subsumptions are performed after computing all consensi

possible with respect to a chosen variable. These consensi are of full length in the sense that each clause has the highest possible number of literals in it. In Socher's algorithm, subsumptions are performed after computing all paths in *M[S,T]*. Subsumption check is accomplished by comparing each path with other paths. To perform this comparison, each of the literals in the path has to be compared with literals of other paths. Less is the number of literals, small is the number of comparisons. The subsumption checks at the submatrices level, that is, subsumptions of partial (not complete) paths of *M[S,T]* are expected to handle less number of literals.

In PIAP, the subsumptions are performed at two stages, one at the submatrices level and the other at the whole matrix level. The concept of partial subsumption, in the sense that if a path *p* subsumes another path q, then the path p ⊎ *s* subsumes *q* ⊎ s, is also made use of. If a path is not prime in a submatrix, it is not prime in the whole matrix. Hence, all such paths need not be considered in order to compute the prime paths of $M[S,T]$. This concept is made use of in PIAP, and for each submatrix, only the prime paths are considered to compute the prime paths for the whole matrix. The number of literals present in a path in P[S,T] is, in general, more than the number of literals present in the paths in $P[S - S_r, T \cup \{r\}]$ and $P[S_r, T \cup \{r\}]$. Therefore, the subsumption is performed at the submatrices level, where the number of literals to be compared is less.

If a path *p* in $P[S - S_r, T \cup \{r\}]$ is subsumed by a path *q* in $P[S_r, T \cup \{r\}]$, as we have already seen any path concatenated with p is definitely subsumed by p. Hence, such paths need not be considered for any further computations. This is accomplished in PIAP by considering such paths separately as *P1* and *P2*. If this is not done, then, p will be concatenated with paths in $P[S_r, T \cup \{r\}]$ just to get it subsumed at a later stage and thus increases the number of subsumptions. Here the subsumption is performed on paths which are of lesser length in the sense that these paths have less number of literals.

The algorithm PIAP proposed in this chapter separates out those paths which need to undergo subsumption check from those which do not have to. Moreover, in Step 6, the set *P4* contains only those paths which are not subsumed by paths so far generated. This set may contain certain paths which may be subsumed by paths that are generated later. In order to remove such paths, the set *P4 is* subjected to subset check in Step 7. Thus, the number of paths for which subset checks are required is much less compared to other algorithms. Hence, PIAP makes use of partial subsumption, early subsumption, and efficient computation of subsumption.

de Kleer [de Kleer 94] proposes a new algorithm which uses a discrimination net called *tries* to perform subsumption. Conceptually, *trie* is a tree, all of whose edges are literals and whose leaves are clauses. Each variable is given a special *id* and is represented in the trie according to the ascending order of the *id*. The database of paths can be represented as a trie by giving special *ids* to the literals and the method proposed by de Kleer can be used to perform subsumption efficiently.

3.4.3 Socher's algorithm Vs PIAP

In Chapter 2, we have already explained that Socher's algorithm [Socher 91] performs certain redundant computations. The efficiency of Socher's algorithm hinges on the choice of the literal r. Here we briefly state the cause of redundant computation in Socher's algorithm and then show that the algorithm PIAP is efficient irrespective of the choice of the literal r.

Why there are redundant computations in Socher's algorithm?

Socher's algorithm identifies only the paths which contain r, using the Theorem 3.2.2. It is already seen that Theorem 3.2.2 is only a partial characterization of the prime paths containing r. A better and complete characterization of paths containing r is proposed

subsequently. Moreover, Theorem 3.2.4 and Theorem 3.2.5, which characterize paths of P[S, *T]* which do not contain r are judiciously utilized in the present algorithm. This concept does not exist in Socher's algorithm and hence, there are a lot of redundant computations. The reason for redundant computations can be understood by the following analysis of Socher's algorithm. The paths in *M[S, T]* are computed by the algorithm **findP** given below.

The algorithm findP *(M[S,T])*

1. Delete all absorbed columns in M[S,T]
2. If M[S,T] = 0, then STOP,
   else
   Select *r* having maximum number of Is
3. Fmd $P[S - S_r, \mathcal{T} \cup \{r, \tilde{r}\}]$
4. If $P[S - S_r, T \cup \{r, f\}] \neq fa$ then
   Compute {r} $\uplus P[S - S_r, T \cup \{r, \tilde{r}\}]$
5. Delete row r to get a new matrix M[S,T U {r}]
6. Go to 1

The algorithm starts with *S* and selects r = r_1 Assume 5 — S_{r1} = S^1, for r_1 at the first step. Delete row r_1 and let the next r be r_2 and $S'' = S — S_{r2}$. S' C 5" (Figure 3.4) because of the heuristic choice of the literal r. While computing $P[S'', \mathcal{T} \cup \{r_1, \tilde{r}_1, r_2, \tilde{r}_2\}]$, it is possible that a literal r_3 is chosen to get S_{r_3} C S'' such that S''' U S' = S'' where $S''' = S'' — S_{r_3}$. The paths for the submatrix having S''' columns are already computed while computing P[S',T]. It is again computed while computing P[S",T] which is sheer waste of computation time.

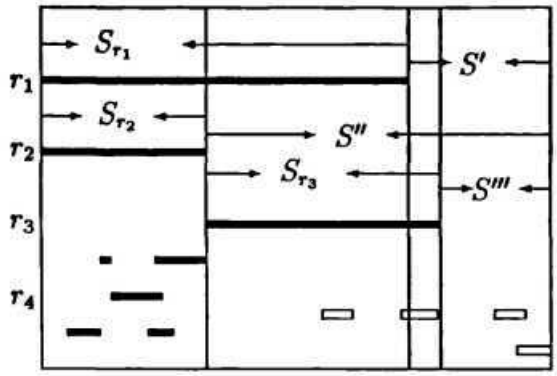


Figure 3.4: Illustration of redundant computations in Socher's algorithm

▬ represents contiguous blockof 1s for literals r_1, r_2, etc.

▭ represents contiguous block of 1s for some literals.

Example 3.4.1:-

Consider the Example 2.7.1 where $r_1 = a$ and $r_2 = b$. $S' = S - S_a$ = {7,8} and S'' = 5 - S_b = {7,8,9,10}. In computing P[S'',T U $\{a, \tilde{a}, b, \tilde{b}\}$], c is chosen as r_3 such that $S''' = S'' - S_{\tilde{c}}$ = {7,8}. It can be seen that $S''' \cup S' = S''$. In fact, here S' = S''' and P[S',T] and P[S''',T] are computed separately.

Moreover, the absorption check invoked for each matrix at Step 1 of findP is evaded in PIAP. The special partitioning scheme in PIAP takes care of this automatically. If $M[S - S_r, T \cup \{r\}]$ is a zero matrix then M[S,T] has columns containing only zeroes and the entire matrix is absorbed by those columns. Similarly, if $M[S_r, T \cup \{r\}]$ is a zero matrix, then the columns in $M[S, T]$ are all the same and represent only a single literal. Thus, PIAP avoids all the redundant and expensive computations performed by Socher's algorithm.

The Choice of the literal r

The choice of the literal r in Socher's algorithm is based on the heuristic: choose the row having the maximum number of Is in the matrix M[S,T]. The number of redundant computations increases if this heuristic is not applied in Socher's algorithm. Whereas in

PIAP, the literal can be chosen arbitrarily, and the choice does not affect the algorithm. This is because the redundant computations are anyway avoided by the partition of the matrix. In fact, the r can be chosen so as to partition the matrix into two equal submatrices.

3.4.4 IPIA Vs PIAP

We have already seen in Chapter 2 that the IPIA algorithm of Kean and Tsiknis [Kean 90] is not a suitable incremental algorithm when a set of clauses has to be appended to the original set of clauses. The reasoner transmits informations, which are, in general, a set of clauses rather than a single clause. In such cases, the IPIA has to be invoked as many number of times as there are clauses. IPIA does not hold the concept of computing the prime implicates of the newly transmitted clauses separately, and use the prime implicates thus computed to find the prime implicates of the whole set. The algorithm **PIAP** facilitates this concept to compute the prime implicants in the incremental mode.

If *H,* is the set of clauses transmitted by the reasoner, the prime implicants of $\mathcal{F}$ A $\mathcal{H}$ can be computed from the prime implicants of $\mathcal{F}$ and $\mathcal{H}$ using **PIAP.** If the prime implicants of $\mathcal{F}$ are already computed, it is enough to compute the prime implicants of $\mathcal{H}$ using **PIAP** and then to invoke **PIAP** to compute the prime implicants for the combined set of clauses from the prime implicants of $\mathcal{F}$ and $\mathcal{H}$. The implementation details of **PIAP** for the incremental mode is dealt in Section 4.6.3. Thus the **PIAP** is well suited for reasouer-CMS architecture.

3.5 Special Cases for Algorithmic Improvement

The special partitioning scheme introduced in this chapter has some special cases which help to improve the algorithm. These are (1) $S - S_r$, in which case, $M[S - S_r,T]$ is

empty, and (2) $S \neq S_r$, and $M[S - S_r, T]$ is zero matrix. The properties of paths in these special cases are discussed in this section.

Theorem 3.5.1:- *If* $S = S_r$ *then* P[S,T] = $\{r\} \cup P[S_r, T \cup \{r\}]$.

Proof:-

When S = S_r, *r* is present in all columns of *M[S,T]* and hence $\{r\} \in P[S,T]$. All the other prime paths in *M[S, T]* are void of the literal r and hence it is in $P[S, T \cup \{r\}]$. But $S = S_r$, and hence the proof.

When $M[S - S_r, T \cup \{r\}]$ is a zero matrix, $P[S - S_r, T \cup \{r\}]$ is empty as no path is possible in $M[S - S_r, T \cup \{r\}]$. The following theorem states that in such a case, there . is no prime path in the full matrix also.

Theorem 3.5.2:- *If* $M[S - S_r, T \cup \{r\}]$ *is a null matrix (zero), then* $P[S,T] = \phi$.

Proof:-

We get $P[S,T] \subseteq \{r\} \uplus P[S - S_r, T \cup \{r\}] \cup P[S - S_r, T \cup \{r\}] \uplus P[S_r, T \cup \{r\}]$ by the results in Section 3.2. When $P[S - S_r, T \cup \{r\}]$ is empty, both the components of the above relation are empty, i.e, both $\{r\} \uplus P[S - S_r, T \cup \{r\}]$ and $P[S - S_r, T \cup \{r\} \uplus P[S_r, T \cup \{r\}]$ are empty, and hence, P[S,T] has no paths. This completes the proof.

Theorem 3.5.3:- *A nonempty matrix M [S, T] has empty paths if there exists some literal* r *such that* $P[S - S_r, T \cup \{r\}] = \phi$.

Proof:-

$P[S,T] \subset \{r\} \uplus P[S - S_r, T \cup \{r\}] \cup P[S - S_r, T \cup \{r\}] \uplus P[S_r, T \cup \{r\}]$ by the results in Section 3.2. Clearly, if $P[S - S_r, T \cup \{r\}] = \phi$, then both $\{r\} \uplus P[S - S_r, T \cup \{r\}]$ and $P[S - S_r, T \cup \{r\}]$fc|$P[S_r, T \cup \{r\}]$ are not defined, and hence, $P[S,T]$ has no paths. This completes the proof.

The converse of the Theorem 3.5.3 is, in general, not true. P[S, *T]* may be empty even if $P[S - S_r, \mathcal{T} \cup \{r\}] \neq \phi$. This may happen when all the paths in $P[S - S_r, \mathcal{T} \cup \{r\}]$ contain $\tilde{r}$, and some literal which is the complement of some literal in all the paths in $P[S_r, \mathcal{T} \cup \{r\}]$.

Corollary 3.5.1:- If $P[S_r, \mathcal{T} \cup \{r\}] = \phi$, *then* $P[S, \mathcal{T}] = \{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}]$.

Proof:-

$P[S, \mathcal{T}] \subset \{r\} \uplus P[S - S_r, \mathcal{T} \cup \{r\}] \bigcup P[S - S_r, \mathcal{T} \cup \{r\}] \uplus P[S_r, \mathcal{T} \cup \{r\}]$. The corollary is a direct consequence of the above relation where $P[S_r, \mathcal{T} \cup \{r\}] = \phi$, in which case the only concatenation possible is of the type r $\uplus p$ where $p \in P[S - S_r, \mathcal{T} \cup \{r\}]$. None of the paths obtained thus subsumes each other and hence the subset relation is made equal and hence the proof.

These results are used to improve the implementation of the algorithm.

3.6 More Properties of Paths

The properties of prime paths discussed in the earlier sections contribute to the design of an efficient algorithm to compute the prime implicants of a formula. Inference is the process by which one prepositional clause is arrived at and affirmed on the basis of one or more other propositions accepted as the starting point of the process. The set of prime implicants of a formula gives all the possible inferences from the formula. Apart from these, there are other properties of prime paths which help to decide the consistency of a formula, provability of a goal, and the minimal support for a given query, to name a few. In this section, these properties of prime paths are explored.

3.6.1 Consistency

A formula is consistent when there is no contradiction in the formula. There are different methods in the literature [Frost 86, Carney 74, Copi 78] to check the consistency of a formula. In the reasoner-CMS architecture, when a new set of clauses is transmitted by the reasoner to the CMS, it is required that the knowledge-base is consistent. As and when the reasoner transmits clauses, the consistency has to be checked incrementally. Every time a new clause is transmitted, checking the consistency by the conventional methods is computationally expensive. It is already indicated in Section 3.4.4 that the prime paths for the augmented knowledge-base can be computed efficiently using **PIAP.** It is shown here that a formula is consistent if the set of prime paths in M[C, ϕ] is nonempty. Thus, if it is only required to check the consistency of a formula $\mathcal{F}$, it is enough to check whether there is a path in $M[\mathcal{C}, \phi]$.

Theorem 3.6.1:- *A formula* $\mathcal{F}$ *represented as* $M[\mathcal{C}, \phi]$ *is consistent if and only if the set of prime paths* $P[\mathcal{C}, \phi] \neq \phi$.

Proof:-

(=>) Assume that $\mathcal{F}$ is **consistent.** i.e, $M[\mathcal{C}, \phi]$ is consistent and there is no contradiction among the columns of $M[\mathcal{C}, \phi]$. Therefore, atleast one literal can be taken from each of the columns so as to make a path in $M[\mathcal{C}, \phi]$. When there is a path there is a prime path also. Hence the proof.

($\Leftarrow$) Assume that $P[\mathcal{C}, \phi] \neq \phi$; that is, there is atleast one prime path p in $M[\mathcal{C}, \phi]$. p is complete and fundamental in $M[\mathcal{C}, \phi]$, and so, for every $i \in \mathcal{C}$, there exists some literal $x \in$ p such that $M(x, i) = 1$. Thus, there is exactly one literal x from each clause in $\mathcal{F}$ such that x is in a path p. Hence, the clauses are not contradictory. Therefore $\mathcal{F}$ is consistent. This completes the proof.

3.6.2 Provability

A goal *g* is provable from a formula $\mathcal{F}$, represented as $M[\mathcal{C},\phi]$, if *g* is true whenever $\mathcal{F}$ is true; in other words, if $\mathcal{F}$ implies *g*. Here we prove that a goal *g* is provable from $\mathcal{F}$ if all the paths in $P[\mathcal{C},\phi]$ have nonempty intersection with g.

Theorem 3.6.2:- *Let g be a goal g is provable from* $\mathcal{F}$ *represented as* $M[\mathcal{C},\phi]$ *if and only if there does not exist any path p in* $P[\mathcal{C},\phi]$ *such that* $p \cap g = \phi$.

Proof:-

Here *g* can be a single literal, a disjunctive clause or, a set of disjunctive clauses.

Case 1. Let us assume that *g is &* single literal, and let the literal be a.

It is assumed that $\mathcal{F}$ is consistent and hence $P[\mathcal{C},\phi] \neq \phi$. Let $P[\mathcal{C},\phi] = \{p_1, p_2, \cdots, p_m\}$. The formula $\mathcal{F}$ is true if and only if atleast one of the prime paths p_i *(say* p_k) is true, and *pk* is true if and only if all the literals in p_k are true simultaneously.

($\Leftarrow$) If $pi \cap a \neq \phi$ for all $p_i \in P[\mathcal{C},\phi]$, then when any p_k is true, *a* is true. Hence *a* is provable.

($\Rightarrow$) If $pi \cap a \neq \phi$ for all $p_i \in P[\mathcal{C},\phi]$ is not true, then there exists a path p_k such that *pk* n a = ϕ. If p_k is true, then a need not be true even if $\mathcal{F}$ is true. Hence a is not provable. Therefore, the assumption that $p_i \cap$ a $\neq \phi$ for all $p_i \in P[\mathcal{C},\phi]$ not true is false and hence there does not exist any path p_i in $P[S,T]$ such that p_i n *a* = ϕ.

Case 2. The goal *g* is a disjunctive clause, and let it be *ab*.

By the argument in the case 1, if we consider the clause *ab* as a single entity, then it forces not to have any path p_i in $P[\mathcal{C},\phi]$ which is free of a or 6. Thus every path will contain either a or 6 or both. Hence the proof.

Case 3. *g* is a conjunctive clause. In this case for *g* to be derivable from $\mathcal{F}$, each of its components should be derivable. Therefore, if $g = \{g_1, ff2, ff3i. —, g_m\}$, then for each *ffj*

to be true, by case 2, there should be nonempty intersection with all p_i. i.e. $p_i \cap g_j \neq \phi$ for all $p_i \in P[\mathcal{C}, \phi]$ and for all $g_j \in g$. This completes the proof. •

Thus, in order to find whether a goal is provable from a formula, it is enough to check whether all the prime paths of the formula have nonempty intersection with the given goal. The prime implicants provide better facility to check the provability of queries. When the provability of a number of goals from a formula has to be checked, the prime implicants give an easy and efficient method to accomplish it.

As indicated before, a given goal *g* may not be provable from a given formula $\mathcal{F}$. *11 g* is not provable, then there may be some additional information which together with the formula $\mathcal{F}$ proves *g*. It is often required to know the different additional set of clauses that are necessary to prove the goal *g*. Such processes give rise to the concept of abductive reasoning and it is also related to the concept of minimal support. To this end, we establish how the prime implicants can be used to find the minimal support for a given goal.

3.6.3 Minimal support

In the reasoner-CMS architecture, the reasoner transmits clauses and the CMS records these clauses if they are fundamental. In addition, the reasoner can query the CMS whenever required. The query *g* is a prepositional formula and the CMS must respond with every minimal clause s such that s U *g* is a fundamental logical consequence of the clauses so far transmitted by the reasoner. Such a clause *s* is called the *minimal support* for *g* with respect to the CMS database. Following Kean and Tsiknis [Kean 90] minimal support can be defined as follows.

Definition 3.6.1:- *Let $\mathcal{F}$ be a set of clauses and g be a single clause. A disjunctive clause s is a fundamental support (or support) for g, with respect to $\mathcal{F}$ if*

(1) $\mathcal{F} \models s \wedge g$

(2) $\mathcal{F} \not\models s$

(3) s U g is *fundamental.*

Any support s is a *minimal fundamental support (or* minima] *support)* if no other support of *g* subsumes s. Let *S* be the set of minimal supports for *g* with respect to $\mathcal{F}$.

It is shown in [Reiter 87, Tsiknis 88] that the set of minimal supports for a query *g* can be computed trivially from the set of prime implicates of the CMS data base. More formally, if $\mathcal{F}$ denotes the database, $\Pi(\mathcal{F})$, the set of prime implicates of $\mathcal{F}$, and g, the query clause set, let

$$\Delta(\mathcal{F}, g) = \{p \text{ — } g \ \backslash' P \ 6 \ \Pi(\mathcal{F}),\ p \cap g \neq \phi,\ \textit{and}\ p \cup g\ \textit{is fundamental}\}$$

Then the set of minimal support for g is defined as

$$\Gamma(\mathcal{F}, g) = \{s \mid s \in \Delta(\mathcal{F}, g)\ \textit{and no}\ s' \in \Delta(\mathcal{F}, g)\ \textit{subsumes}\ s\}$$

Kean and Tsiknis [Kean 90] insist that s U g is fundamental and the minimality is defined with respect to a different ordering among clauses. However, we insist s U g to be fundamental so as not to consider the trivial support g for g, and the minimality defined by Reiter and de Kleer [Reiter 87]. For any clause p, let $p = \{\tilde{x} \mid x \in p\}$. If p is a disjunctive clause, then p is a conjunctive clause and vice versa, and (p) = *p.* With this notation, the definition can be interpreted and restated as follows.

Remark 3.6.1:- *A disjunctive clause s is a support for g if (1)* $\mathcal{F}$ A s $\Rightarrow g$, *(S)* $\mathcal{F}$ A s is *consistent and (3) s* U g *is fundamental.*

Thus the conjunctive clause s is the additional information, when appended to $\mathcal{F}$, derives g. When a goal is not provable, how the minimal support for the goal can

be computed is our main concern in this section. To this end, we begin the following subsection which gives insight into it.

3.6.4 Invalidation of a path

In Section 3.2.4, the concept of extension of a path to a bigger matrix has already been introduced. Here, the concept of *invalidation* of a path in a bigger matrix is introduced. A path p in a submatrix is said to be invalidated, if *p* ceases to be a path in a bigger matrix. In other words, if p is not a path in the bigger matrix.

Let $p \in P[\mathcal{C}, \phi]$. Then p is a disjunctive clause since all the paths in $M[\mathcal{C},\phi]$ are in the conjunctive form. Let h, the binary vector corresponding to the clause p be called *hypothesis*, and $\mathcal{H}$ be the set of such hypotheses. When there is no confusion we denote h by $M[h,\phi]$ and use the same letter *h* to denote the column corresponding to the clause h in matrix $M[h,\phi]$. Further, the set *H* of hypotheses is denoted by $M[\mathcal{H},\phi]$. By appending the matrix $M[h,\phi]$ and $M[\mathcal{H},\phi]$ to the matrix $M[\mathcal{C},\phi]$ we get $M[\mathcal{C}\cup h,\phi]$ and $M[\mathcal{C}\cup\mathcal{H},\phi]$, respectively. With these notations and construction of h, we prove here that the p is not a path (invalid) in $M[C \cup h,\phi]$.

Theorem 3.6.3:- For $p \in P[\mathcal{C},\phi]$, *let* $M[h,\phi]$ *represent the matrix for the clause p. Then* $p \notin P[\mathcal{C} \cup h,\phi]$.

Proof:-

By Corollary 3.2.3, $p \in P[\mathcal{C},\phi]$ can be extended to a bigger matrix $M[C \cup h,\phi]$ if E(p, h) = h. It will be proved here that E(p, *h)* $\neq h$.

By the construction of h, $x \in b$ for any $x \in p$. Hence, $M(\tilde{x}, h) = 1$. $E(p,h) = h$ if there exists atleast one $x \in p$ such that $M(x, h) = 1$. But for all $x \in p$, $x \in h$ and hence, $x \notin b$ and $M(x,h) = 0$. **Therefore,** $E(p,h) \neq b$. This completes the proof.

It is already seen in Section 3.2.4 that when $E(p,S') \neq S'$, for some $'$ C $\mathcal{C}$, the paths possible from two submatrices are of the form $p \uplus q$, where p is a prime path in $M[C - S', \phi]$ and, q a prime path in $M[S', \phi]$. The following corollary states that $p \uplus q$ is not a path in Af[C U h, ϕ] under the above construction of h.

Corollary 3.6.1:- *Let* $p \in P[\mathcal{C}, \phi]$, h = p, *and* $q \in P[h, \phi]$. *Then,* $p \uplus q \notin P[C \cup h, \phi]$.

Proof:-

Since h is a single clause, every literal in *h* is a prime path. Therefore, any path $q \in P[h, \phi]$ is the negation of some literal $x \in$ p. Thus, for any literal $x \in$ p, $x \in$ q and hence, $p \uplus q$ is not fundamental. Therefore, $p \uplus q$ is not a path in $M[C \cup h, \phi]$ and hence the proof.

Thus, by appending a column representing the negation of the literals in the path p to the original matrix, the path p is invalidated in the new matrix. Let $Q = \{q^1, q_2, \ldots, q_m\}$ be a subset of prime paths in $P[\mathcal{C}, \phi]$. Let $h_1, h_2, \ldots, h_m$ be the negation of $q_1, q_2, \ldots, q_m$, respectively, and let $\mathcal{H} = \{h_1, h_2, \ldots, h_m\}$. With these notations, the following theorem explains the way to invalidate more than one path.

Theorem 3.6.4:- *Let the paths in* $Q = \{q_1, q_2 \ldots, q_m\}$ *are to be invalidated. The path* $q_i \uplus s_j$, *for any* $q_i \in Q$ *and for* $s_j \in P[\mathcal{H}, 4]$ *is not a prime path in* $M[C \cup \mathcal{H}, \phi]$.

Proof:-

According to the definition of prime path, $q_i \uplus \tilde{s}_j$ is a prime path if $q_i \cup \tilde{s}_j$ is fundamental, complete and not subsumed by any other path.

In order to prove that $q_i \cup s_j$ is not a prime path it is enough to prove that $q_i \cup s_j$ is either not fundamental or not complete. Since q_i is complete in $M[\mathcal{C}, \phi]$, and $\tilde{s}_j$ is complete in $M[\mathcal{H}, \phi]$, $q_i \cup Sj$ is complete if it is fundamental. Here it will be proved that $q_i \cup s_j$ is not fundamental.

Let $x \in q_i$. By the construction of $\mathcal{H}$, the i^{th} column of $\mathcal{H}$ contains $\tilde{x}$. In order for $q_i \cup \tilde{s}_j$ to be complete, it should have exactly one literal from each of the columns. Any literal in the i^{th} column of *H is* the negation of some literal in q_i. Therefore, s_j has atleast one literal which is the complement of some literal x in q_i. Hence, $q_i \cup \tilde{s}_j$ is not fundamental, and hence not a path. This completes the proof.

Thus, by Theorem 3.6.4, it is stated that if there are m paths to be invalidated, m columns are to be added to the original matrix. The reason for this is established here. Let us assume that there are two paths p_1 and p_2 to be invalidated. According to the Theorem 3.6.4, p_1 and p_2 are the two clauses which are to be appended to the original matrix in order to invalidate p_1 and p_2. Let us assume that p_1 alone is appended. Clearly, p_1 is invalidated by the process. p_1 may contain some literal x whose complement is not present in p_2. This literal can be concatenated with p2 to give a path in the augmented matrix and hence not invalidated. In this way, in order to invalidate p2, one more clause has to be appended. Otherwise, if all the literals in p2 are in p_1 then P2 will also become invalid. But in this case the primeness of the paths p_1 and p2 is violated.

Therefore it is necessary to add as many number of columns as there are paths to be invalidated. With this concept of invalidation of a path, how the minimal supports can be obtained from the prime paths is explored in the following subsection.

3.6.5 Computation of minimal support

It is already established in Section 3.6.2 that in order to prove a goal g, p_i should have nonempty intersection with g for all $p_i \in P[C, \phi]$. If g is not provable, then there exists some q $\in P[C,\phi]$ such that $q \cap g = \phi$. Let all paths $q_i \in Q \subset P[C, \phi]$ have the additional property that $q_i \cap g = \phi$. Therefore, in order to prove g, it is required to invalidate all the paths in the set Q, without affecting the provability of g. The trivial case of g itself added as a clause is not considered here. By Theorem. 3.6.4, all paths in Q are invalid in

$M[\mathcal{C} \cup \mathcal{H}, \phi]$. It will be proved here that atleast one p € $P[\mathcal{C}, \phi]$ such that p $\cap g \neq \phi$ can be extended to a path in $M[C \cup \mathcal{H}, \phi]$. This guarantees that the goal *g* is provable from $M[\mathcal{C} \cup \mathcal{H}, \phi]$.

Theorem **3.6.5:-** *Let* $P = \{p_1, p_2, \cdots, p_k\}$ *be the set of prime paths in* $P[\mathcal{C}, \phi]$ *such that* $Pj \cap g \neq \phi$. *Let* $n = \{h_1, h_2, \ldots, h_m\}$ *where* $h_j = q_j$; $q_j \in P[\mathcal{C}, \phi]$ *such that* q_j n $g = \phi$. *Then* $P[C \cup \mathcal{H}, \phi] \neq \phi$ *and g is provable from* $M[C \cup \mathcal{H}, \phi]$.

Proof:-

First it will be proved that $P[C \cup \mathcal{H}, \phi] \neq \phi$.

$\mathcal{H}$ is consistent since it is the set of negation of some prime paths in $M[\mathcal{C}, \phi]$. Therefore, by Theorem 3.6.1, $P[\mathcal{H}, \phi]$ is nonempty. By Theorem **3.6.4,** for every path $s \in P[\mathcal{H}, \phi]$, $qj \uplus s$ is not defined. We prove that atleast one path p_i € *P* can be extended to a path of the form $p_i \uplus s$ in $M[\mathcal{C} \cup \mathcal{H}, \phi]$.

If $p_i \uplus s$ is not possible for all *pi* ∈ P, then it means that every *s* has atleast one literal 6 for some a € p_i. This is true for all path p_i ∈ P. If a appears (let us say) in the j^{th} column of $M[\mathcal{H}, \phi]$ (value of *j* is not unique), then *qj* has *a* as one of its literals, and *a* is also in p_i. However, because of the primeness of q_j and p_i, they satisfy noninclusion property (*The* property *that qj* $\not\subseteq p_i$ *and* $p_i \not\subseteq$ qj). Hence there exists atleast a literal 6 in *qj* which is not in p_i and hence there exists a path in M [H, ϕ] containing b. Taking into account the primeness of the paths in $P[\mathcal{C}, \phi]$, consistency of the formula $\mathcal{F}$, and assuming $\mathcal{H}$ is not tautology, it can be shown that there is a path in $\mathcal{H}$ which contain 6 but cannot contain 6. Extending this argument for every literal of the form a in p_i, we can prove that $p_i \uplus s$ is possible. Hence $P[C \cup \mathcal{H}, 4] \neq \phi$.

Now it is to prove that all paths in $P[C \cup \mathcal{H}, \phi]$ have nonempty intersection with g. P is the set of prime paths in $M[\mathcal{C}, \phi]$ having nonempty intersection with *g*. By Theorem 3.6.4, all paths in $M[\mathcal{C}, \phi]$ having empty intersection with *g* are invalidated in

$M[C \cup H, 4]$ due to the construction of $\mathcal{H}$. Thus all the paths in $M[C \cup \mathcal{H}, \phi]$ are of the form $p_i \uplus s$ where $p_i \in P$ and $s \in P[\mathcal{H}, \phi]$. Hence all paths in $M[C \cup \mathcal{H}, \phi]$ have nonempty intersection with *g*.

Thus, all the paths which have empty intersection with a given goal *g* can be invalidated by the addition of few clauses to the original set of clauses. We now show here that the set of prime paths in $M[H, \phi]$ is the set of minimal supports for the goal g.

Theorem 3.6.6:- *Let g be the goal, and P,and H are as defined above. Then* s 6 $P[\mathcal{H}, \phi]$ *is a minimal support for g with respect to* $P[\mathcal{C}, \phi]$.

Proof:-

It is enough to prove that $(p \uplus s) \cap g \neq \phi$ for all p fcl s $\in P[C \cup \mathcal{H},$ fl.

By the construction of $\mathcal{H}$, any clause in $\mathcal{H}$ has empty intersection with *g* and so is with s $\in P[\mathcal{H}, \phi]$. Therefore, in order for nonempty intersection with g, p should contain *g*. By Theorem 3.6.4 and Theorem 3.6.5, all the prime paths in $M[C \cup \mathcal{H}, \phi]$ are of the form p $\uplus$ s such that p n g $\neq \phi$. Hence s is a support. Since s is a prime path, the minimality property is satisfied and hence s $\in P[\mathcal{H}, \phi]$ is a minimal support for g. Hence the proof.

Example 3.6.1:-

Let us consider the Example 2.4.1. The prime implicates for the formula in Figure 2.1(a) are given by Figure 2.4(b). If the reasoner wants the minimal explanation for the query c, by the Kean and Tsiknis [Kean 90] method, the minimal supports are $\{abf, abd, ab\tilde{e}\}$. The prime implicants for the same formula are $\{a\}, \{b\}, \{c, d\}, \{c, f\}, \{c, g\}, \{\tilde{d}, \tilde{e}, f\}$. The set of paths which do not contain c is $Q = \{\{a\}, \{b\}, \{d, \tilde{e}, f\}\}$. Therefore, the set *H* - $\{\tilde{a}, \tilde{b}, de\tilde{f}\}$. The prime paths $P[\mathcal{H}, \phi]$ for the set *H* axe $\{\tilde{a}, \tilde{b}, \tilde{f}\}$, $\{\tilde{a}, b, d\}$, $\{\tilde{a}, 6, e\}$ which is nothing but the negation of minimal supports obtained by Kean and Tsiknis' method.

Thus the definition of minimal support can be restated in terms of paths as follows.

Definition 3.6.2:- *A conjunctive clause s is a minimal support for a query g with respect to a formula* $\mathcal{F}$ *represented as* $M[\mathcal{C}, \phi]$ *if s is a prime path in the formula* $\mathcal{H}$ *defined as* $\mathcal{H} = \{\tilde{p} \mid p \in P[\mathcal{C}, \phi],\ p \cap g = \phi\}$.

Thus, the minimal support of a query *g* with respect to a formula $\mathcal{F}$ can be computed using the prime paths of the formula $\mathcal{F}$.

3.6.6 Hypothetical reasoning

Hypothetical reasoning, is a reasoning style where we proceed to an inference by considering matters which are not clearly true or false, but tentatively true (i.e, generating a hypothesis). A hypothesis is regarded true if the inference successfully reaches a goal. When the knowledge is incomplete, and inadequate to answer a given goal, hypothetical reasoning, which can handle incomplete knowledge comes into picture. If a given query *g* is not provable from a complete knowledge $\mathcal{F}$, then it is to find what additional information from the hypotheses set $\mathcal{H}$ when appended to $\mathcal{F}$ would prove the goal *g*. Thus, the basic behaviour [Poole 88, Ishizuka 91] of hypothetical reasoning is as follows.

When a goal *g* is given, the system first tries to prove the goal from complete knowledge $\mathcal{F}$. If it fails, then the system selects a subset of hypotheses so that the given goal is proved from the union of complete knowledge and this hypotheses subset. The selected hypotheses must be consistent with complete knowledge, while inconsistency is allowed in the whole set of the hypotheses*. This structure (pictorially depicted in Figure 3.5) of the hypothetical reasoning system can be summarized to find a solution *h* satisfying the following conditions [Ishizuka 91, Kondo 91, Ishizuka 93].

- $h \subset \mathcal{H}$ *(h* is a subset of *H)*

- $\mathcal{F} \cup h \Rightarrow g$ (*g* can be derived from $\mathcal{F}$), and
- $\mathcal{F} \cup h \not\Rightarrow D$. ($\mathcal{F} \cup h$ is consistent).

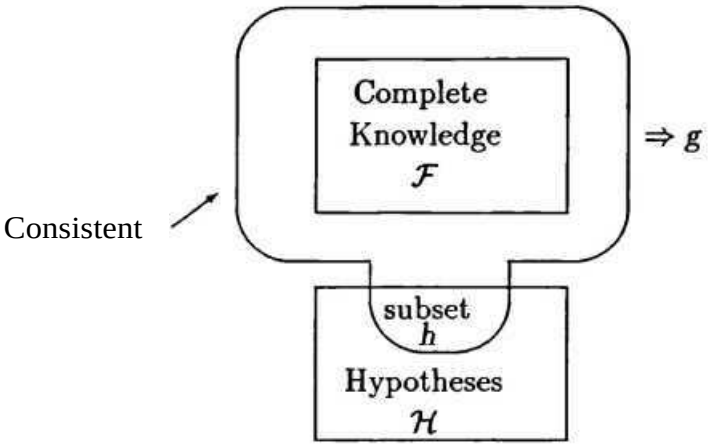


Figure 3.5: Structure of the logic-based hypothetical reasoning system

There may be many hypotheses which together with the knowledge $\mathcal{F}$ prove the goal. What is required is the minimum set of hypotheses required to answer the query. It is already seen that minimal support for a query gives the minimal explanation for the query. Hence, if the minimal support for a goal is known, then it is to find those hypotheses from $\mathcal{H}$ which contribute literals to the minimal support. The computation of hypotheses to prove a given goal *g* is explained with the help of the example taken from [Ishizuka 91].

Example **3.6.2:-**

Let $\mathcal{F}$ — $\{ab\tilde{c}, abd, b\tilde{e}f, f, e\tilde{g}h, e\tilde{g}i, jh\}\mathcal{H}$ = {c, $d, g, h, i\}$ and *a* be the complete knowledge, hypotheses and goal, respectively. The inconsistent set of hypotheses are $\{c, g\}$ and $\{d, h\}$.

First the prime paths for the formula $\mathcal{F}$ is obtained. The set of minimal supports for the query a is obtained as /, $b\tilde{c}$, $\tilde{c}\tilde{g}i$, $\tilde{c}\tilde{g}h$, ce, $e\tilde{g}h$, $b\tilde{g}h$, bf, $b\tilde{g}i$, $e\tilde{g}i$, hj, $d\tilde{e}$, $d\tilde{g}h$, $d\tilde{g}i$, bd from the prime paths thus computed. As indicated before, the negation of the minimal

supports are the possible hypotheses, which on adoption prove the goal a. Among the possible hypotheses, *cgi, cgh,* and *dgh* are inconsistent and *f, bc, ce, egh, bgk, bf, bgi, egi, hj, de,* and *bd* have literals which are not present in the hypotheses set. Thus the only clause remaining is *gid.* Therefore, the goal a can be proved by adopting the hypotheses g, *i* and *d.* In this case, this is the only minimal hypotheses which together with the formula $\mathcal{F}$ prove a.

3.7 Conclusion

In this chapter, the characterization of prime paths containing the literal *r* and of prime paths which do not contain the literal *r* are established. Hinged on this characterization, a correct and sound algorithm PIAP is proposed to compute the prime implicants of. a formula $\mathcal{F}$. This algorithm is superior to the most recent algorithm, namely, Socher's algorithm for the same problem. The redundant computations performed by Socher's algorithm is evaded here by making use of a divide-and-conquer paradigm. Besides this, the algorithm PIAP has the additional advantage of being suitable for incremental computation of prime paths especially when a set of clauses is appended to the formula rather than a single clause at a time. The algorithm PIAP is efficient in handling any number of clauses for the incremental approach. The worst case of the proposed algorithm is when the set of prime paths is empty, in which case consensus-subsumption methods perform better. The algorithm works more efficiently compared with the earlier methods when the number of prime paths is large. The implementation details and the experimental results can be seen in Chapter 4.

The number of subsumptions performed in PIAP is less compared to the other algorithm, resulting in substantial reduction of the computation time. The algorithm can be efficiently used to find the minimal support for queries and can also be used to find the

consistency of a formula, provability of goal, and also for efficient hypothetical reasoning.

The algorithm **PIAP** is therefore superior to other methods in the following sense.

- It avoids the redundant computations performed by Socher's algorithm.
- It hinges on the divide-and-conquer paradigm
- The method is suitable for incremental computation of prime implicants, especially when the reasoner transmits a set of clauses rather than a single clause.
- The number of subsumptions as well as the length of the paths which undergo subsumption check are less resulting in substantial reduction of the computation time.
- The method helps for easy computation of minimal support for a query when it is not provable from the given formula.
- Facilitates efficient hypothetical reasoning.

Chapter 4

TREE AND REASON MAINTENANCE

4.1 Introduction

The significance of prime implicants and the need for efficient technique for computing prime implicants have been discussed in the previous chapters. The algorithm **PIAP** as proposed in Chapter 3 employs a divide-and-conquer technique which naturally gives rise to a tree structure for representation. In this chapter, this concept is extended further to utilize the tree structure not only for computation of prime implicants but also to have full-fledged RMS. It is shown here that a binary tree is suitable to represent prepositional formula in CNF and also to represent the prime paths as a result of knowledge compilation. The implementation of **PIAP** makes use of the proposed tree representation.

In the later part of this chapter, the implementation details of the tree structure and the **PIAP** algorithm are discussed. The computational experiments carried out for **PIAP** are also reported in this chapter. It is shown here that the computing time as well as the number of subsumptions required by **PIAP** is much less than that of Socher's algorithm. The experimental results substantiate the theoretical arguments given in Chapter 3. Based on the discussions in Chapter 3 as well as the experiments reported in this chapter, it can be seen that the proposed algorithm **(PIAP)** is better than the Socher's algorithm for computation of prime implicants. Moreover, it is shown that the structure used for the algorithm can be extended to have a full-fledged RMS. Updating the data base of the RMS as and when the reasoner transmits clause(s) is **very**

important. Different methods to accomplish this are discussed in Section 4.6. Due to the dynamic nature of the RMS, updating the prime paths incrementally is essential. The same structure is suitable for incremental computation of prime paths as well as other reasoning paradigms such as hypothetical reasoning.

4.2 Tree Representation of a Formula

A binary tree representation of a formula suitable for satisfiability problem is discussed in [Horowitz 83, Frost 86]. In this representation, each node denotes an operator V, A or, ¬ *(~)* and the leaf nodes represent literals. The formula is obtained by traversing the tree. The value of the formula represented by the tree can be computed by assigning *true* or *false* to the variables present in the formula. To evaluate a formula, traverse the tree in postorder evaluating subtrees until the entire expression is reduced to a single value. However, in the present context, keeping in view the algorithm **PIAP,** this representation scheme is not suitable. In the present work, a new tree representation scheme is developed to represent a prepositional formula. Such a scheme naturally evolves from the algorithm PIAP.

Any set of clauses representing a formula $\mathcal{F}$ can be divided, with respect to a literal r, into two subsets of clauses, namely, a set of clauses containing *r* and another set of clauses void of r. If $\mathcal{F} = \{D_1, D_2, \ldots, D_m\}$, let $\mathcal{F}_r = \{D - \{r\} \mid r \in D,\ D \in \mathcal{F}\}$. Obviously, $\mathcal{F}_r$ is the set of clauses obtained by uniformly deleting the literal r from the clauses in $\mathcal{F}$ containing r. Thus $\mathcal{F}_r$ represents a **subformula** of $\mathcal{F}$. Though $\mathcal{F}_r$ does not contain any clause as it is in $\mathcal{F}$, we consider $\mathcal{F}_r$ a subset of $\mathcal{F}$. Similarly, $\mathcal{F}'_r = \{D \mid r \notin D, D \in \mathcal{F}\}$ is another subset of $\mathcal{F}$ which does not contain r. Thus $\mathcal{F}'_r$ and $\mathcal{F}_r$ are two subsets of $\mathcal{F}$. with empty intersection.

Each of these two sets of clauses can be further divided in a similar manner. The

literal with respect to which these two sets are partitioned need not be the same. That is, the set $\mathcal{F}_r$ can be divided with respect to some literal (say, r') whereas $\mathcal{F}'_r$ can be divided with respect to some other literal (say, r''). This branching process can be continued till the formula gets reduced to a single literal.

Thus, any set $\mathcal{F}$ of clauses is divided into two subsets $\mathcal{F}_r$ and $\mathcal{F}'_r$ of clauses, with respect to some literal r and hence can be visualized as a binary tree. Any node of this binary tree corresponds to a subset of $\mathcal{F}$, the root obviously corresponds to $\mathcal{F}$ itself. The literal r with respect to which the set $\mathcal{F}$ is divided to obtain two subsets, is associated with the node representing $\mathcal{F}$. This literal associated with every node is called the *label* of the node. The label of the nodes at any level of the tree need not be the same. In the binary tree representation of the formula $\mathcal{F}$, if *r* is the label of the node corresponding to the formula $\mathcal{F}$, then its left child corresponds to the formula $\mathcal{F}'_r$ whereas the right child corresponds to $\mathcal{F}_r$. For example, let the formula be given as $\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg, abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$. The root corresponds to the formula $\mathcal{F}$. The literal *a* is chosen and the formula is split with respect to a, giving $\mathcal{F}'_a = \{\tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ represented by the left child and $\mathcal{F}_a = \{bcd\tilde{e}, b\tilde{c}dfg, bcf, bcdf, bcdg, bcef, dfg, bf\}$, represented by the right child of the root. These two nodes are further split independently, and the complete binary tree is obtained. The binary tree representation of the above formula thus obtained is given in Figure 4.1. The label of each node is circled, and the formula corresponding to the node is on the right side. If the subset $\mathcal{F}'_a$ is to be split with respect to $\tilde{a}$, then according to the partitioning method, the left child will be a NULL node. The NULL node is represented by D and • represents an empty formula in the Figure 4.1.

The matrix representation of a formula has already been discussed in Chapter 2. Following the matrix representation scheme, any node of the binary tree corresponds to a submatrix *M[S,T]* of $M[\mathcal{C}, \phi]$, the root obviously corresponds to $M[\mathcal{C}, \phi]$ itself.

Since the rows of the matrix correspond to the literals in the formula, the binary matrix associated to a given node is partitioned with respect to the row corresponding to the literal *r*. If *M[S, T]* corresponds to a formula $M(\tilde{\mathcal{F}})$ at a given node where the label is r, then the matrices *M[S - S_r,T U {r}]* and $M[S_r, \mathcal{T} \cup \{r\}]$ correspond to $M(\hat{\mathcal{F}}'_r)$ and $M(\hat{\mathcal{F}}_r)$, respectively. As discussed earlier, if r is the label of the node in the tree representing the formula *M [S , T]*, then the submatrix *M[S — S_r*, T U {r}] is the left child of the node. Similarly, the submatrix $[S_r, \mathcal{T} \cup \{r\}]$ corresponds to the right child of the node. Both $M[S — S_r, \mathrm{T} \cup \{r\}]$ and $M[S_r, \mathrm{T} \cup \{r\}]$ have one row (the row corresponding to the chosen literal r) less than that of the matrix *M[S,T]*.

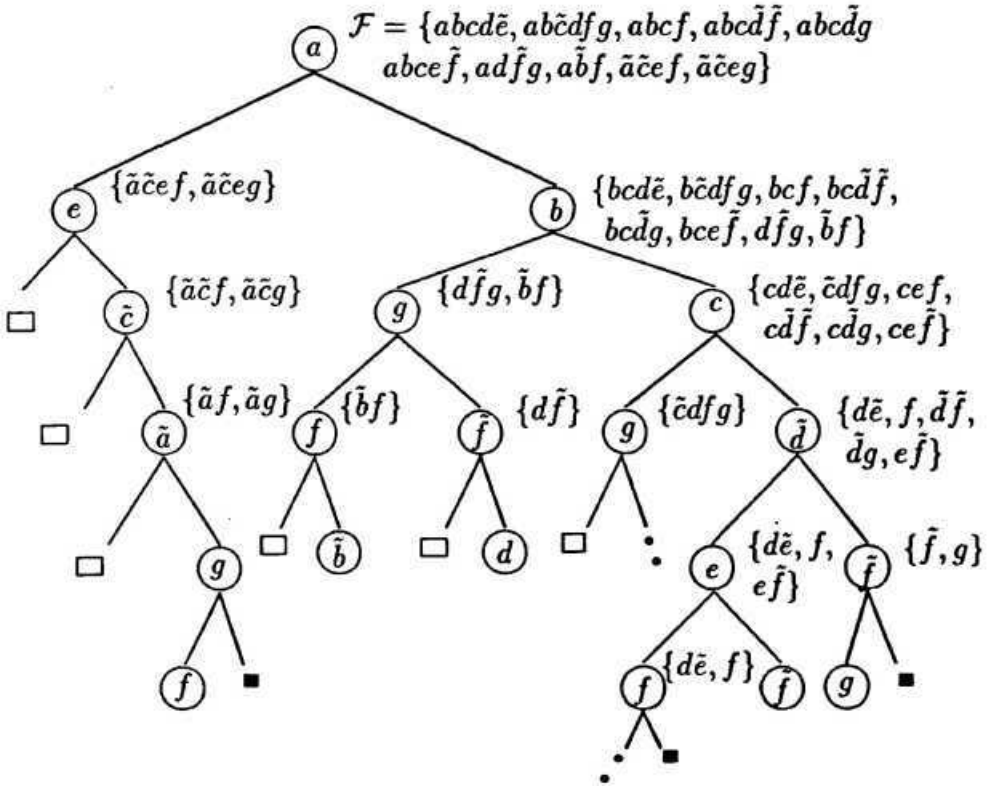


Figure 4.1: Tree-representation of the formula $\mathcal{F}$ for TERMS
□ represents NULL node and • represents zero matrix (empty formula).
Labels are circled and clauses corresponding to the nodes
are on the right side of the node

The main aim of this representation scheme is to compute the prime paths of each subset of clauses so as to compute the prime paths of a formula. It is already mentioned that all the literals in a disjunctive clause are prime paths for the clause. Hence, the branching process is continued until the submatrix becomes a column matrix, or a row matrix or, a zero matrix. The column matrix corresponds to a unit clause whereas a row matrix corresponds to a literal. The zero matrix corresponds to an empty formula. It may be noted that this occurs if there is redundancy among the set of clauses corresponding to the node. If the left child of a node is zero (empty), then by Theorem 3.5.2, the node has an empty set of prime paths. If the right child of the node is zero, it indicates that there is redundancy in the set of clauses corresponding to the node, and has no affect on the set of prime paths of the node. In Socher's algorithm, the redundant clauses are removed after each iteration by removing the absorbed columns (Definition 2.3.1). In Socher's algorithm, in order to accomplish the absorption check, each of the columns has to be compared with the other columns in the matrix which is obviously expensive especially when the matrix is of large size. However, this absorption check is avoided here since the redundancy among clauses leads to a node representing a zero matrix.

In this ramification process, the left child of the node corresponding to *M[S,T]* is NULL if all the entries in the r^{th} row of *M [S , T]* are 1. The right child of a node is NULL only when the node corresponds to a conjunctive clause. The right child of a node can not be NULL in other cases because it means that the formula does not have any literal present. If the formula does not have any literal, then the matrix corresponding to it is zero. If no node in the tree is zero, then the number of leaf nodes is equal to the number of clauses in the formula. Having chosen binary tree as the representation scheme for a formula, the question is how to represent the binary tree? The implementation details of the construction of the binary tree and the algorithm **PIAP** are presented in the following section.

4.3 Implementation Details

There are different representations for a binary tree, for example, representation by sequential numbering, array representation, linked representation etc. [Horowitz 83, Tremblay 84]. The binary tree representing a formula is, in general, not balanced and hence the representation by sequential numbering and array representation are not suitable. Moreover, insertion or deletion of nodes from the tree requires more quantum of changes in the level number of nodes. These problems are easily overcome through the use of linked representation. Hence, the linked representation of the binary tree is adopted here.

4.3.1 Structure of a node in the tree

Any node of the binary tree for TERMS has the following nine fields which is pictorially depicted in Figure 4.4.

Field 1 Integer *c* :- This field acts as a flag to denote whether the current node is a root node, intermediate node or, a leaf node. Separate integer *ids* are given to represent the root (0), intermediate left child (-2), intermediate right child (+2), left leaf node (-1) and right leaf node (1). These *ids* are used while computing the prime paths of the tree.

Field 2 Integer *1*:- This integer gives the row number corresponding to the label r of each node and the row number is computed as follows: The number of Is in each of the row of the matrix corresponding to the node is computed, and the row having the maximum number of Is is selected. If there is a tie between the rows, then arbitrarily one of these rows is chosen. However, it is to be noted that the choice of r does not affect the efficiency of the algorithm PIAP significantly as in the case of Socher's algorithm. If the node is a leaf node, then some flag is given as the

label so as to identify it since the computation of prime paths for leaf nodes is very simple. Further, if the node represents a zero matrix, a different label is given so that the computation of paths is made easy.

Field 3 Integer *nrow* :- The number of rows in the submatrix corresponding to the node is stored in this field. This helps to check the stopping criteria and also to compute the number of rows of its children. The root at the 0^{th} level has the maximum number (equal to the number of literals present in the formula) of rows and the number of rows in the submatrix decreases as the level of the tree increases. Though the number of rows and the level of the tree are related, the level of a node is not required for the computation of prime paths. Hence, only the number of rows in the submatrix is stored to check the stopping criteria.

Field 4 Rows:- There is a one to one correspondence between the rows in the matrix and the literals. This correspondence must be maintained throughout and hence, the row numbers of the original matrix which are in the submatrix corresponding to the node are stored as a list of integers. Though the number of rows are same at a given level, the literals present in the clauses corresponding to the node may be different and hence it is necessary to know the literals which appear in the submatrix corresponding to the node. This information is obtained by the list of row numbers stored in each node. It is a linked list with two fields; one an integer representing the row number and the other, a pointer to the next element in the list. The structure of the list is given in Figure 4.2. This also helps when the tree is updated when there are new literals in the clauses transmitted by the reasoner.

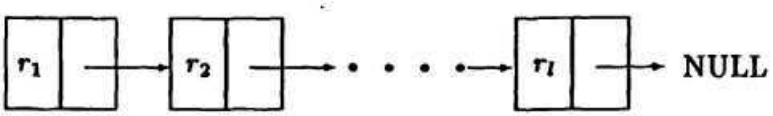


Figure 4.2: Structure of a path

Field 5 Cols:- This is the field giving the list of columns of the original matrix which are columns of the submatrix corresponding to the node. This list is similar to that of the list of rows. The correspondence between the clauses and the columns of the matrix is maintained throughout by storing the column numbers. This is required to rebuild the original matrix from the constructed tree. This field also helps in easy deletion or, addition of clauses to the original formula which is required in .RMS.

Field 6 Leftc :- Pointer to the left child of the node.

Field 7 Rightc :- Pointer to the right child of the node.

Field 8 Par :- Pointer to the parent of the node. This pointer makes it possible to traverse the tree which is needed in the computation of prime paths.

Field 9 Path:- This field gives the prime paths in the matrix corresponding to the node. This field is NULL during the construction of tree. It is used while computing the prime paths at each node. Each of the paths is stored as a list of row numbers representing the literals present in the path. The set of paths at each node is stored as a list of these paths. This data structure is suitable due to the dynamic nature of the paths as well as the set of paths. The row numbers in the lists representing the paths are stored in ascending order so as to facilitate easy subsumption check. Moreover, this representation makes deletion and addition of literals from a path easier. Further, the deletion and addition of paths from and to the set of prime

paths are made easy because of the linked representation of the set of paths. The structure of the set of prime paths is shown in Figure 4.3.

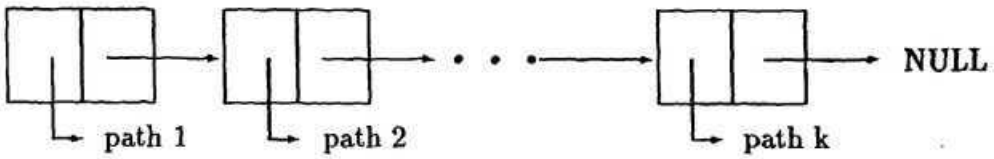


Figure 4.3: Structure of a set of paths

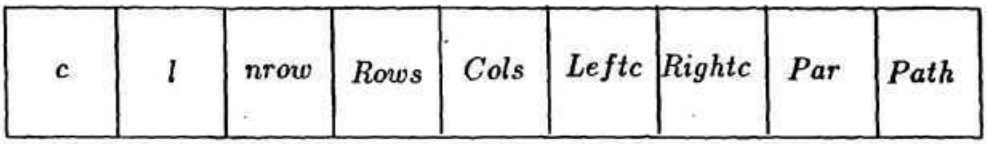


Figure 4.4: Structure of a node in the binary tree for TERMS

4.3.2 Creation of binary tree

Based on the structure of a node in the binary tree for TERMS discussed above, the pseudo code for the construction of the binary tree is given here followed by the function which computes the left child of a node.

Procedure CREATE_TREE $(M[\mathcal{C}, \phi])$

Input: $M[\mathcal{C}, \phi]$, the m x n matrix corresponding to the formula $\mathcal{F}$

Output: $\mathbf{T}(\mathcal{F})$, the binary tree corresponding to the formula $\mathcal{F}$

Step 0: Initialize the root node

Step 1: root→c =0

Step 2: root→nrow= n.

Step 3: root→Col = *create_list(n)* * Creates the list of integers $0, 1, \ldots, n-1$ *\

Step 4: root→Row =*create_list(m)* * Creates the list of integers 0, 1, ..., m — 1 *\

```
Step 5: root—>1 = maxrow (root—>row, root—>col)
                        \* Computes the row having the maximum number of Is *\
Step 6: root—>Leftc =compute_]eft (root)      \* Computes the left child of the root *\
Step 7: root—>Rightc =compute_right (root)\* Computes the right child of the root *\
Step 8: root->Par =NULL
Step 9: root->Path =NULL
END

Procedure  Compute-left  (NODE)          \* Computes the left child of a NODE *\
        Input : The node NODE
       Output : The left child of the NODE
NODE->Leftc-->Cols = compute_left_col (NODE–>Par–>1, NODE-->Par->Cols)
                        \* Computes the list of columns of the left child of the NODE *\
If (NODE->Leftc-->Cols is NULL) then
   return NULL
else
      NODE->Leftc-->nrow=NODE->nrow-l.
      NODE-->Leftc-->Rows=compute_chid_row(NODE-+Par-*l, NODE0->Par–>Rows)
                        \* Computes the list of rows in the left child of the NODE *\
     If (NODE-->Leftc-->Rows or NODE->Leftc-->Cols has single element) then
        NODE-->Leftc--> c = -1                                  \* left leaf node *\-
        NODE-->Leftc--> 1 = -1                                  \* flag for leaf node *\
        NODE-->Leftc-->Leftc =NULL
        NODE->Leftc-->Rightc = NULL
      else
          NODE—>Leftc—> c = —2                                  \* left intermediate node *\
```

```
NODE→Leftc→l = maxrow (root→Rows, root→Cols)
                    \* Computes the row having the maximum number of 1s *\
If (NODE→Leftc→l is negative ) then  \* Case where the matrix is zero *\
   NODE→Leftc→Leftc =NULL
   NODE→Leftc→Rightc = NULL
else
   NODE→Leftc→Leftc =compute_left (NODE→Leftc)
                    \* Computes the left child of the NODE *\
   NODE→Leftc→Rightc =compute_right (NODE→Leftc)
                    \* Computes the right child of the NODE *\
NODE→Leftc→Par =NODE
NODE→Leftc→Path =NULL
:ND
```

The right child of a node is computed using a similar function. The function corn-*mteJeft-Col* is replaced by another similar function cojnpute_riglit_col to compute the columns of the right child of the node. Further, the flags for the right intermediate node and leaf node are 1 and 2, respectively. Therefore, NODE→Rightc→c and NODE→Rightc→c are 1 and 2, respectively.

4.4 Path Computation Using Tree

The aim of this section is to recast the **PIAP** for implementation using tree-representation. At any node (with label r) which represents a subformula, the corresponding prime paths are computed by making use of the sets of prime paths of the child nodes. As already mentioned in Section 4.2, the left child and right child of a node corresponding to $M[S, T]$

represent $M[S - S_r, T \cup \{r\}]$ and $M[S - S_r, T \cup \{r\}]$, respectively. In the tree representation of paths, this node (corresponding to $[S, T]$) contains $P[S - S_r, T \cup \{r\}]$ in the field *Path* of the tree. Similarly, field *Path* in the right child of the node contains $P[S_r, T \cup \{r\}]$. The prime paths of the node can be computed by performing simple operations like concatenation and merging of the prime paths of the left child and the right child of the node. Step 3 of PIAP is accomplished by checking whether any path of the right child of a node subsumes any path of the left child of the node. In the similar way, Step 4 is implemented by checking whether any path of the left child of the node subsumes any path of the right child of the node. By concatenating the prime paths of the left child with the label of the node, the Step 5 can be implemented. The concatenation of prime paths of the left child with the paths of the right child gives the paths obtained by Step 6. Thus the prime paths of a formula can be computed with the help of the tree constructed for the formula. Moreover, the tree structure can also be used to represent the prime paths of the formula.

4.4.1 Working of the algorithm PIAP

In this subsection, the computation of prime paths using the tree representation is illustrated with the help of an example. The example is so chosen that possible adverse cases are highlighted. Let $\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg, abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ be the formula chosen. The nodes of the tree representing $\mathcal{F}$ are sequentially numbered as seen in Figure 4.5.

The node *15* at level 4 has prime paths *{f, g}*. The node *14* is NULL and so is the set of prime paths. Since the left child of node 9 is NULL, by Theorem 3.5.1, the label itself is a prime path for the node; i.e. $\{\tilde{a}\}$ is a prime path of node *9*. The other paths of this node are those paths in the right child of the node. Thus the prime paths for the *node 9* are $\{\tilde{a}\}$ and {f,g}. Similar is the case of *node 5* and node 2. Traversing up the

tree and computing the prime paths at each node the prime paths of *node 2* are $\{\tilde{a}\}$, $\{\tilde{c}\}$, $\{e\}$ and $\{f,g\}$. The prime paths are given on the right side of each node in Figure 4.5.

We have already seen in Section 3.6.1 that a formula is consistent if it has nonempty prime paths. Here it can be seen that the set of prime paths at *node 13* is empty, even though its child nodes have nonempty set of prime paths. For the node 13, r = $\tilde{d}$, $P[S - S_{\tilde{d}}, T \cup \{r\}] = \{d, e, /\}$ and $P[S_{\tilde{d}}, T \cup \{r\}] = \{f,g\}$. The path $\{d, e, /\}$ of the left child neither subsumes nor is subsumed by $\{f,g\}$ giving $\mathcal{P}1 = \phi$. By Step 5, *d* has to be concatenated with $\{d,e,f\}$. The concatenation is not possible since fundamentality is violated and therefore $\mathcal{P}3 = \phi$. In Step 6, though $\{d, e, /\}$ has to be concatenated with $\{f,g\}$, it is not possible since the union of two paths is not fundamental and hence $\mathcal{P}1$ remains to be ϕ. Thus there is no prime path for the node 13 at level 3. In fact, one can see that the set $\{d\tilde{e}, /, df, dg, ef\}$ of clauses corresponding to the node 13 is not consistent.

The *node 7* is the case where the right child of the node is empty and hence the paths are obtained only by Step 5 (concatenating the prime paths of the left child with the label of node 7). These paths are $\mathcal{P}3 = \{\{c,d\},\{c,f\},\{c,g\}\}$. All the paths in $\mathcal{P}3$ are identified to be prime and hence subsumption is not required. Thus $\{c,d\}$, $\{c,f\}$, *{c,g}* are the prime paths of node 7. The prime paths $\{b,g\},\{f,g\},\{b,d\},\{b,f\},\{d,f\}$ of node *6* are computed from the prime paths of the nodes 10 and 11.

So far we have seen a few simple cases. The prime paths for the node 3 are computed from the prime paths of node *6* and node 7. There is no path satisfying the conditions in Step 3 and Step 4, and hence $\mathcal{P}1 = \phi$ and $\mathcal{P}2 = \phi$. By concatenating the label b with the prime paths of node 6, $\mathcal{P}3 = \{\{b,f,g\},\{b,d,f\}\}$ is obtained. By Step 6, $\mathcal{P}4$ is obtained as $\{b,c,d,g\},\{b,c,f,g\},\{b,c,g\}$, $\{c,d,f,g\}$, $\{c,f,g\}$, $\{\tilde{b},c,d\}$, and $\{c,d,f\}$ of which $\{b,c,d,g\}$, $\{b,c,f,g\}$ and $\{c,d,f,g\}$ are subsumed in Step 7. Thus, the prime paths of *node 3* are $\{b,f,g\}$, $\{b,d,f\}$, $\{\bar{b},c,g\}$, $\{c,d,f,g\}$, $\{c,f,g\}$, $\{b,c,d\}$ and $\{c,d,f\}$.

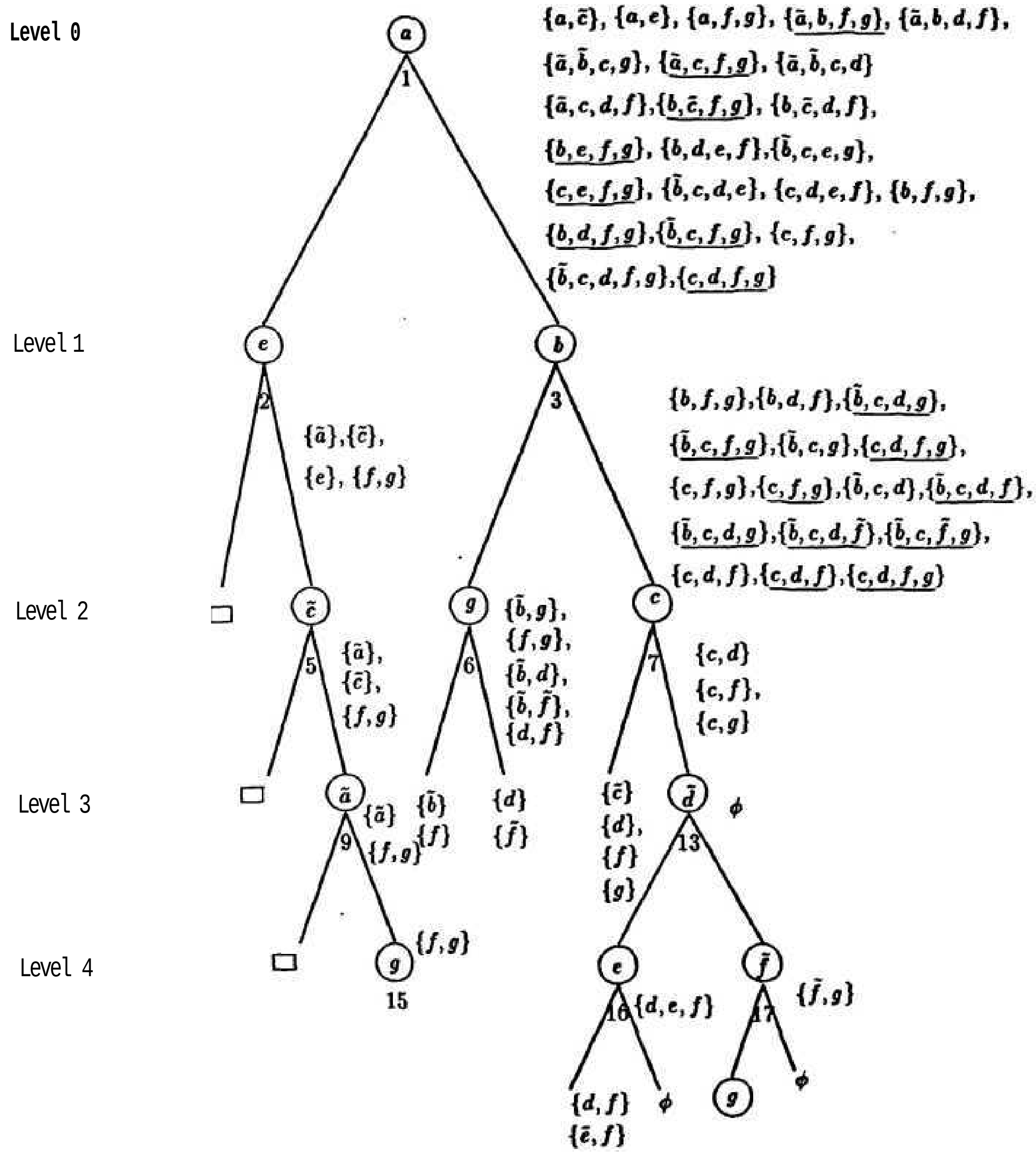


Figure 4.5: Binary tree and prime paths for the formula

$\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg$]&cf, $abcdf, abcdg, abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$

Thus the prime paths of the left subtree and right subtree of the root node are obtained. The root node gives a case where $\mathcal{P}2 \neq \phi$. The prime path $\{f, g\}$ of the left subtree subsumes the prime paths $\{b, f, g\}$ and $\{c, f, g\}$ of the right subtree. Therefore, by Step 4, the paths $\{b, f, g\}$ and $\{c, f, g\}$ in $\mathcal{P}2$ are prime paths for the root node also. The set of prime paths obtained by Step 5 is $\mathcal{P}3 = \{\{a, \tilde{c}\}, \{a, e\}, \{a, f, g\}\}$. By Step 6, the concatenations of the prime paths of the left subtree with that of the right subtree are obtained. Finally, all the subsumed paths are deleted and the prime paths of the formula are $\{a, \tilde{c}\}$, $\{a, e\}$, $\{a, f, g\}$, $\{b, f, g\}$, $\{c, f, g\}$, $\{\tilde{a}, b, d, f\}$, $\{\tilde{a}, b, c, g\}$, $\{\tilde{a}, b, c, d\}$, $\{\tilde{a}, c, d, f\}$, $\{b, \tilde{c}, d, f\}$, $\{b, d, e, f\}$, $\{b, c, e, g\}$, $\{b, c, d, e\}$, $\{c, d, e, f\}$.

4.5 Experimental Results

Based on the foregoing discussions, experiments were carried out to compare the efficiency of the proposed algorithm with respect to Socher's algorithm. The algorithms were coded in C language and were run in the same computing platform[1]. In order to have a uniform measure of performance, the literals were taken from a fixed set of size 20 (10 variables together with its negations form the set of literals). The problem size is identified as the number of clauses in the formula. The output size is the number of prime paths computed by the algorithm. The length of a path is the number of literals present in a path.

If there are n variables, then the maximum possible length of a path is n. The problems are generated randomly from a large set of consistent clauses. In other words, initially a very large set of clauses which is consistent is identified, and then from this set, arbitrarily, certain number of clauses are picked up to define a formula. This method was adopted because the automatic generation of random problems following some probability

[1]The computing platform is IBM PC 386 Compatible

distribution as well as having a consistent set of clauses was found to be a complex task. The number of problems generated by the proposed method being very large, it automatically is devoid of the dements of deviating from random generation.

In order to study the performance of the algorithm, several experiments were carried out. It is observed that the behaviour of the algorithm depends on the number of subsumptions carried out, overall time taken by the algorithm, and the number of candidate paths subjected to subsumption operation. It has already been pointed out that subsumption is the crucial operation in any prime implicants computation algorithm. Hence, the number of subsumption checks gives a measure of the performance of the algorithm. In PIAP, certain paths are identified (namely, the set *P*3) which are not subjected to any subsumption operation. Moreover, the subsumption operation is performed at different levels of the tree and hence the length of the candidate paths for subsumption are expectedly different. Thus the paths having smaller number of literals naturally take less time for subsumption than that of paths having more number of literals. So it is necessary to know the candidate path for subsumption at different levels of the tree. In the third experiment, the study is made to find the number of candidate paths at the root level. This is compared with the candidate paths in Socher's algorithm. The experimental results are summarized in the form of tables and are also pictorially depicted in the form of-graphs. The details of each experiment are described below.

4.5.1 Experiment 1

In this experiment, the aim is to illustrate that the algorithm PIAP requires less number of subsumptions as compared to Socher's algorithm. This is so because certain subsumptions are carried out at the level of intermediate nodes where the paths of submatiices are subsumed. In one sense, a path is subsumed at a much earlier stage where expectedly the number of literals in the path is less. Moreover, at any given node, certain paths

which never get subsumed are identified in advance and this saves the computational effort. It is to be noted that it is emphasized again and again in literature [de Kleer 94, Jackson 92, Kean 90] (and here too) that subsumption operation is the most crucial operation in prime implicants computation. Hence, reducing number of subsumptions naturally results in a better and more efficient algorithm. In the present experiment, the subsumption operation is carried out in the usual way of subset checking. The same function of subsumption operation is used for both the algorithms.

The experimental results are summarized in Table 4.1 to Table 4.7. The first column of any of Table 4.1 to Table 4.7 gives the problem size and the sample number. For example, 15-12 means, this is the 12^{th} sample data for the problem having 15 clauses. The second column gives the number of subsumptions carried out by Socher's algorithm. The third column gives the total number of paths that are generated and subjected to subsumption operation in Socher's algorithm. The fourth column gives the total number of subset checking carried out at intermediate nodes by PIAP. i.e., at nodes other than the root of the tree. This figure depicts the number of subsumptions carried out over paths in the submatrices and not on paths having full length. So, normally the subsumption operations on these paths take less computational time (as there are less number of literals) than the subsumption operation at the root. The number of subsumptions carried out at the root level is given in column 5. The total number (column 4 + column 5) of subsumptions by the PIAP is given in the sixth column. The seventh column gives the number of paths subjected to subsumption operation at the root level. The number of prime paths obtained after subsumption is given in column 8. These are the prime paths void of the label of the root. It is already mentioned that certain paths are identified to be prime at an early stage, and are not subjected to subsumption operation. The number of such paths is given in column 9. The number of paths in column 7 and column 9 together gives the number of paths that are generated

by PIAP so as to compute the prime paths of the formula. This is given in column 10. The total number of prime paths (column 8 + column 9) for the problem is given in the last column of the tables (Table 4.1 to Table 4.7). In the case of Socher's algorithm, this is the outcome of performing subsumption operation on the number of paths given in column 3.

It is to be noted here that if the left child of the root node is NULL, then column 9 gives the number of paths obtained by appending the label of the root node to the set of prime paths of the right child of the root. These paths are never subjected to subsumption checks since they are already prime. There are certain samples (15-3, 15-5, 15-6, 15-10, 15-13, 25-1, 25-19, 30-6, 30-13, 301-8, 35-1, 35-2, 35-8, 40-1, 40-2, 40-4, 40-18, 45-3, 45-6) where the entries in columns 5, 7 and 8 are zero. In all these cases, the left child of the root node is NULL and so is its set of prime paths. If the left child is not NULL, column 9 gives the prime paths obtained by concatenating the label of the root node to the paths in the set of prime paths of the left child of the root node. There are few cases where there was memory allocation problem for Socher's algorithm. Such cases are denoted by * in the tables. Further, SOCH stands for Socher's algorithm in the tables given in this chapter.

Table 4.1: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total # of Paths	Total Prime Paths
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$		
15-1	6914	163	767	1003	1770	30	22	16	46	38
15-2	5595	142	906	2553	3495	71	25	24	95	49
15-3	300	27	56	0	56	0	0	8	8	8
15-4	11146	271	596	2661	3257	67	39	7	74	46
15-5	1267	65	348	0	348	0	0	17	17	17
15-6	4122	115	1101	0	1101	0	0	31	31	31
15-7	1795	57	928	873	1801	26	26	6	32	32
15-8	7212	122	562	1394	1956	34	30	12	46	42
15-9	1086	60	164	289	453	11	11	9	20	20
15-10	1511	65	194	0	194	0	0	17	17	17
15-11	1719	67	240	472	712	28	14	11	39	25
15-12	2105	95	163	347	510	16	13	9	25	22
15-13	391	34	161	0	161	0	0	11	11	11
15-14	2885	103	555	504	1059	16	15	13	29	28
15-15	2693	114	578	402	980	13	11	15	28	26
15-16	938	61	264	195	459	9	9	11	20	20
15-17	835	52	62	150	212	10	9	5	15	14
15-18	2306	72	646	548	1194	22	11	24	46	35
15-19	7096	229	1292	2312	3604	55	34	7	62	41
15-20	2574	93	424	1149	1573	37	28	6	43	34

Table 4.2: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							Total Prime Paths
	# of		# of Subsumptions			# of Paths			Total	
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$	# of Paths	
20-1	30647	276	5438	9545	14983	109	69	36	145	105
20-2	33358	291	7725	9558	17283	87	65	50	137	115
20-3	60243	649	27832	11128	38960	85	37	67	152	104
20-4	1764	66	5982	625	6607	10	6	20	30	26
20-5	1069	80	1!	559	2090	15	9	8	23	17
20-6	*	*	20303	35333	55636	258	123	103	361	226
20-7	2948	131	1765	1129	2894	23	14	8	31	22
20-8	14174	333	4794	2553	7347	43	25	15	58	40
20-9	3265	136	3444	1167	4611	26	11	12	38	23
20-10	*	*	23129	16364	39493	340	145	98	438	243
20-11	4929	118	887	1491	2378	31	27	17	48	44
20-12	6665	162	1684	1111	2795	23	23	15	38	38
20-13	4157	98	829	914	1743	21	17	22	43	39
20-14	5620	167	397	1295	1692	34	31	4	38	35
20-15	1412	55	231	339	570	15	15	5	20	20
20-16	9507	234	899	620	1519	17	15	21	38	36
20-17	1206	65	62	150	212	10	9	5	15	14
20-18	3257	88	642	825	1467	25	25	6	31	31
20-19	2304	70	275	662	937	24	18	9	33	27
20-20	8706	187	1501	1670	3171	34	27	18	52	45

Table 4.3: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total	Total
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$	# of Paths	Prime Paths
25-1	27	6	46	0	46	0	0	5	5	5
25-2	432	43	341	101	442	8	8	3	11	11
25-3	1500	88	521	283	804	12	10	8	20	18
25-4	17535	455	2846	1643	4489	40	28	14	54	42
25-5	4733	172	666	2028	2694	53	29	7	60	36
25-6	5805	190	386	605	991	23	17	11	34	28
25-7	3150	138	310	404	714	17	13	9	26	22
25-8	1102	88	150	152	302	10	9	7	17	16
25-9	1408	87	490	257	747	11	10	9	20	19
25-10	2010	94	491	258	749	7	5	13	20	18
25-11	13400	276	1582	443	2025	8	8	29	37	37
25-12	64126	468	23129	16364	39493	177	58	68	245	126
25-13	3065	111	197	404	601	15	15	6	21	21
25-14	3065	149	88	128	216	10	8	8	18	16
25-15	11931	326	797	499	1296	19	11	20	39	31
25-16	11031	477	902	693	1595	18	13	17	35	30
25-17	7328	197	736	1209	1945	36	23	11	47	34
25-18	10276	288	915	692	1607	21	19	18	39	37
25-19	17621	301	4355	0	4355	0	0	50	50	50
25-20	117395	955	2396	10560	12956	114	87	29	143	116

Table 4.4: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total	Total
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$	# of Paths	Prime Paths
30-1	24066	450	3494	3916	7410	74	46	11	85	57
30-2	13441	324	3031	1523	4554	31	31	10	41	41
30-3	30453	511	1679	2197	3612	36	33	15	51	48
30-4	7645	238	1415	743	2158	20	19	19	39	38
30-5	21869	518	1211	1238	2449	34	16	31	65	47
30-6	13899	325	1840	0	1840	0	0	36	36	36
30-7	59776	487	5967	4824	10791	58	39	56	114	95
30-8	4116	117	864	185	1049	5	5	29	34	34
30-9	4069	191	728	815	1543	30	16	10	40	26
30-10	8498	206	371	522	893	25	14	19	44	33
30-11	7482	248	513	523	1036	15	15	11	26	26
30-12	9973	226	1419	1367	2786	34	23	20	54	43
30-13	28	7	47	0	47	0	0	5	5	5
30-14	680	46	431	180	611	10	7	7	17	14
30-15	2371	104	853	343	1196	11	11	13	24	24
30-16	6013	180	757	665	1422	22	22	6	28	28
30-17	13666	367	838	999	1837	24	18	22	46	40
30-18	29901	500	4721	0	4721	0	0	60	60	60
30-19	13324	258	1811	1892	3703	35	23	23	58	46
30-20	15837	399	1807	944	2751	26	21	19	45	40

Table 4.5: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total	Total
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$	# of Paths	Prime Paths
35-1	1735	125	249	0	249	0	0	10	10	10
35-2	3101	220	343	0	343	0	0	14	14	14
35-3	6645	241	658	755	1413	23	15	16	39	31
35-4	13399	420	781	710	1491	22	14	20	42	34
35-5	8487	226	729	937	1666	21	21	12	33	33
35-6	3536	99	560	488	1048	15	15	15	30	30
35-7	8082	209	1471	1127	2598	25	14	19	44	33
35-8	11588	488	688	0	688	0	0	16	16	16
35-9	9319	222	805	990	1795	26	26	7	33	33
35-10	8583	249	792	906	1698	26	24	9	35	33
35-11	17219	405	3582	1478	5060	33	17	21	54	38
35-12	⋆	⋆	2415	1218	3633	29	20	28	57	48
35-13	17865	348	2326	1516	3842	34	34	8	42	42
35-14	6449	210	1192	565	1757	13	12	20	33	32
35-15	2171	123	186	172	358	10	10	5	15	15
35-16	2550	129	420	201	621	7	7	14	21	21
35-17	7328	197	550	400	950	14	10	9	23	19
35-18	20035	425	2795	1088	3883	16	13	36	52	49
35-19	12737	246	1250	1249	2499	23	23	19	42	42
35-20	10783	360	883	1259	2142	36	22	7	43	29

Table 4.6: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total # of Paths	Total Prime Paths
	Subs.	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $\mathcal{P}3$		
40-1	26867	583	2867	0	2867	0	0	33	33	33
40-2	6066	225	799	0	799	0	0	18	18	18
40-3	35782	668	2510	1325	3835	27	21	32	59	53
40-4	25408	447	3326	0	3326	0	0	44	44	44
40-5	25020	385	2750	2044	4794	37	20	30	57	50
40-6	6529	216	974	166	1140	5	5	22	27	27
40-7	5510	269	682	266	948	8	7	17	25	24
40-8	10091	348	806	347	1153	9	9	14	23	23
40-9	20270	412	2417	1787	4204	34	34	11	45	45
40-10	26210	665	2093	1148	3241	24	19	28	52	47
40-11	16982	419	1703	1904	3607	22	22	15	37	37
40-12	11369	279	2705	678	3383	12	11	24	36	35
40-13	11319	306	728	383	1111	10	7	22	32	29
40-14	14061	321	1487	1300	2787	26	26	18	44	44
40-15	4997	229	600	297	897	9	9	15	24	24
40-16	17551	342	1316	1525	2841	45	28	9	54	37
40-17	16663	436	1138	1578	2716	19	18	22	41	40
40-18	1671	179	258	0	258	0	0	8	8	8
40-19	6217	191	1051	627	1678	15	15	19	34	34
40-20	20949	557	1547	1380	2927	32	31	7	39	38

Table 4.7: Table giving the number of subset checking and the number of paths using Socher's algorithm and PIAP

Prob. Size & Samp. No.	SOCH		PIAP							
	# of		# of Subsumptions			# of Paths			Total	Total
	Subs. Checks	paths for subs.	Till root	At root	Total	for subs.	after subs.	in $P3$	# of Paths	Prime Paths
45–1	31064	496	3235	2338	5573	34	34	22	56	56
45–2	9660	298	1249	729	1978	13	13	15	28	28
45–3	32788	650	3615	0	3615	0	0	37	37	37
45–4	10382	372	1005	263	1268	20	14	16	36	30
45–5	15661	619	1545	193	1738	14	13	15	29	28
45–6	12159	570	989	0	989	0	0	22	22	22
45–7	32939	640	1620	1556	3176	34	31	12	46	43
45–8	28817	570	1922	1697	3619	30	29	22	52	51
45–9	18245	417	1123	1062	2185	18	17	28	46	45
50–1	8552	420	823	351	1174	14	9	18	32	27
50–2	15889	450	1389	932	2321	26	18	20	46	38
50–3	3459	278	291	20	311	0	0	11	11	11
50–4	20616	399	1198	1297	2495	31	20	19	50	39
50–5	⋆	⋆	5183	1479	6662	33	25	27	60	52
50–6	15117	530	527	345	872	15	15	8	23	23
50–7	⋆	⋆	6125	4079	10204	49	43	33	82	76

From these tables, it is clear that the number (column 5) of subsumption operations carried out by **PIAP** at the root level is much less compared to the number (column 2) of subsumption operations carried out by Socher's algorithm. In fact, the total number (column 6) of subsumptions required by **PIAP** is less compared to Socher's algorithm except for few stray cases, (15-7, 20-4, 20-9, 25-1, 25-2, 30-13). However, for these samples also, the number of subsumptions required at the root level is less compared to Socher's algorithm. The number of subsumptions are more in these cases due to the following reason. In order to obtain the prime paths at any node, subsumption check is performed between the set of prime paths of its children so as to obtain the paths which are prime for the parent node (Step 3 and Step 4 of **PIAP).** If there is no path in the set of prime paths of children which hold the property of being prime at the parent node, then these subsumptions are of no use. If most of the nodes in the tree are of this type, then the number of subsumptions required by **PIAP** can be more compared to Socher's algorithm. However, these subsumptions are performed at the intermediate levels where the length of paths are less and hence take less execution time, ex. 15-7, 20-9, 25-1, 25-2, and 30-13. Problem 20-4 and 25-12 are the only two cases among the samples tested, where the number of subsumptions as well as the execution time is more for **PIAP.** However, it can be seen that the number of paths generated by **PIAP** for all samples (including 20-4 and 25-12) is less than paths generated by Socher's algorithm.

The average number of subsumptions required by both the algorithms for samples of fixed size is calculated. In order to find the significance of the algorithm **PIAP** over Socher's algorithm, *paired* t-test has been performed and the results obtained are given in Table 4.8. Apart from the average number of subsumptions required by both the algorithms (column 2 and column 5), the standard deviation (SD) and coefficient of variation (CV) of the samples for both the algorithms are also given. The degrees of freedom (DF), computed *t* value (paired t), and the significant level **(≈** probability *p),*

are given respectively in the last three columns of Table 4.8. $p < .0005$ means that the computed *t*-value is greater than the tabled *t* value [Rao 75] of *t—distribution* at the 0.05% level of significance. In most of the cases $p < .0005$ in Table 4.8 and this substantiates that the algorithm PIAP perform significantly better than Socher's algorithm.

Table 4.8: Table giving the # of input clauses and the average # of subsumptions with SD and CV, by Socher's algorithm and PIAP

# of	SOCH			PIAP			DF	*paired*	*p*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
15	3224	2899	.899	1244	1115	.896	19	4.460	< .0005
20	10846	15513	1.43	6179	9455	1.529	17	2.944	< .005
25	14847	27910	1.879	3903	8849	2.267	19	2.139	< .025
30	14355	13919	.969	2820	2554	.905	19	4.432	< .0005
35	9032	5441	.602	1794	1316	.733	18	7.433	< .0005
40	15476	9254	.597	2425	1313	.541	19	7.164	< .0005
45	21301	9991	.469	2683	1450	.541	8	6.343	< .0005
50	12726	6732	.529	1434	943	.657	4	4.318	< .01

The behaviour of the algorithm not only depends on the input size but also on the output size - in fact more on the output size than the input size. In order to study the behaviour of the algorithms based on the number of prime paths, the samples were grouped according to the output size. The average number of subsumptions required for each of the groups is calculated and the same is given in Table 4.9. The results obtained by the paired *t-test* performed to study the performance of both the algorithms are quite significant and substantiates positively that PIAP is significantly better than Socher's

algorithm.

The first order regression curve for the data in Table 4.9 is obtained[2] and is given in Figure 4.6. It is evident from the Figure 4.6 that the performance of **PIAP** is better than Socher's algorithm.

Table 4.9: Table giving the # of Prime Paths and the average # of subsumptions with SD and CV, by Socher's algorithm and PIAP

# of	SOCH			PIAP			DF	*paired*	*p*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
10–15	1534	1219	.794	331	146	.44	7	2.991	< .025
15–20	2954	3170	1.073	669	474	.707	13	2.838	< .01
20–25	4494	3314	.737	1321	1213	.918	11	3.616	< .005
25–30	7086	3980	.561	1633	1332	.815	16	6.559	< .0005
30–35	7348	3941	.536	1718	613	.356	19	7.112	< .0005
35–40	13977	6699	.479	2783	1368	.491	19	7.910	< .0005
40–45	15030	7751	.515	3211	841	.261	13	6.010	< .0005
45–50	19030	7938	.417	3643	676	.185	8	6.038	< .0005

[2] **Package used for this purpose is Regression Analysis for Tme Series (RATS) Ver. 2.0.**

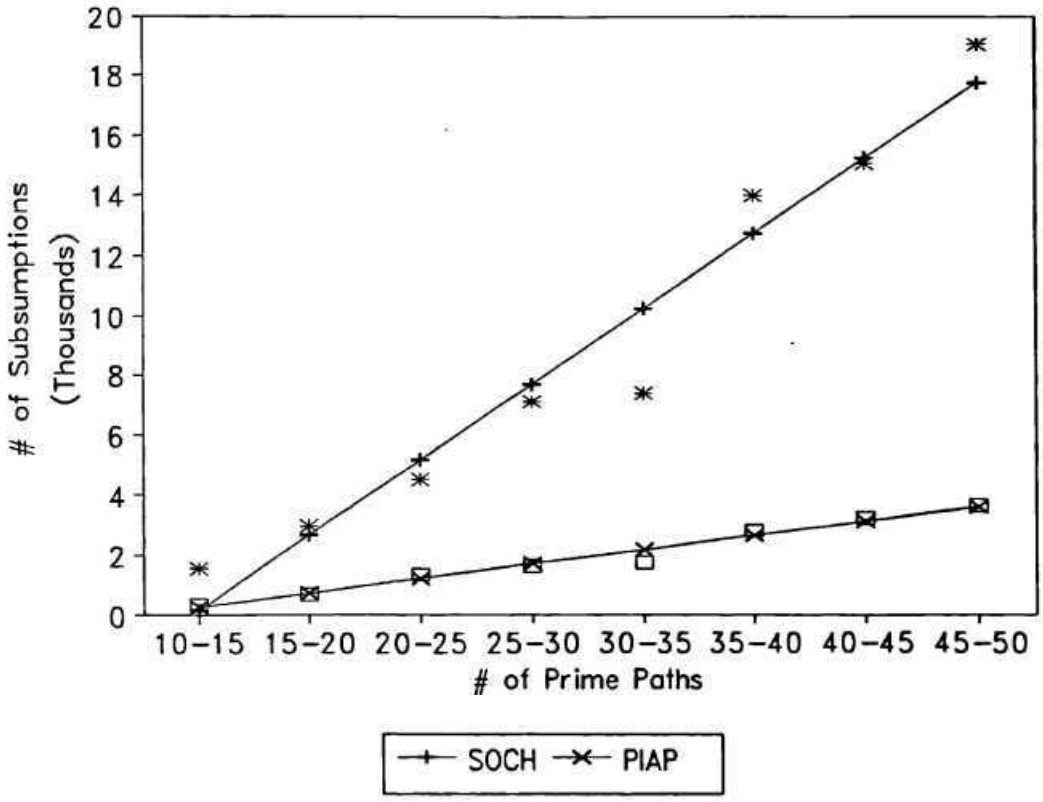


Figure 4.6: The graph depicting the # of subsumptions required by Socher's algorithm and PIAP.

4.5.2 Experiment 2

It is to illustrate that the execution time taken by PIAP is much less compared to Socher's algorithm. The time taken for the same samples considered in Experiment 1 are given in Table 4.10 to Table 4.16. From these tables it is clear that the execution time of PIAP is less than that of Socher's algorithm. The samples 20-4 and 25-12 are two cases where the execution time for PIAP is more than that of Socher's algorithm.

The average execution time for the samples of same input size is computed. As in the case of Experiment 1, *paired t-test* is performed and the results obtained are in given in Table 4.17. The values in the last column of the Table 4.17 gives the level of significance ($\approx$ probability *p)*. It can be seen that the value of p is $< .0005$ for all of the cases, supporting the claim that PIAP is efficient than Socher's algorithm.

It is already mentioned that the performance of algorithms depend on the size of the output also. Therefore, the average execution time taken by the algorithms depend on the number of prime paths. Hence, the average execution time required by both the algorithms for groups formed based on the output size are also calculated, and the *paired t-test* is performed for the data thus obtained. The results are given in Table 4.18. From the values of p in the last column of Table 4.18, it can be seen that the execution time taken by PIAP is less compared to that of Socher's algorithm.

The first order regression curve for the data in Table 4.18 is obtained and is given in Figure 4.7. The graph substantiate that PIAP is significantly efficient than Socher's algorithm. Thus, the execution time for PIAP proposed in Chapter 3 is less than that of Socher's algorithm.

Table 4.10: Table giving the execution time of Socher's algorithm and PIAP

Size 15		Execution time	
Prob.No.	PIs	SOCH	PIAP
15–1	38	.879121	.219780
15–2	49	.879121	.494505
15–3	8	.219780	.054945
15–4	46	1.263736	.274725
15–5	17	.329670	.054945
15–6	31	.439560	.109890
15–7	32	.329670	.164835
15–8	42	.659341	.219780
15–9	20	.274725	.109890
15–10	17	.329670	.054945
15–11	25	.329670	.109890
15–12	22	.389615	.054945
15–13	11	.274725	.054945
15–14	28	.494505	.164835
15–15	26	.494505	.109890
15–16	20	.384615	.054945
15–17	14	.384615	.054945
15–18	35	.384615	.219780
15–19	41	1.043956	.384615
15–20	34	.549451	.164835
Average Time		.516733	.1565392

Table 4.11: Table giving the execution time of Socher's algorithm and PIAP 1

		Execution time	
Prob. No.	PIs	SOCH	PIAP
20-1	105	2.032967	1.483516
20-2	115	2.087912	1.538462
20-3	104	4.890110	5.000000
20-4	26	.769231	.879121
20-5	17	.769231	.384615
20-6	226	*	5.659341
20-7	22	.989011	.494505
20-8	40	2.252747	1.153846
20-9	23	.879121	.604396
20-10	243	*	5.604396
20-11	44	.604396	.219780
20-12	38	.879121	.274725
20-13	39	.604396	.164835
20-14	35	.769231	.164835
20-15	20	.439560	.054945
20-16	36	1.263736	.219780
20-17	16	.434505	.054945
20-18	31	.494505	.164835
20-19	27	.439560	.109890
20-20	45	.989011	.384615
Average Time		1.199353	.741758

Table 4.12: Table giving the execution time of Socher's algorithm and PIAP

Size 25		Execution time	
Prob. No.	PIs	SOCH	PIAP
25-1	5	.219870	.054945
25-2	11	.549451	.109890
25-3	18	.659341	.164835
25-4	42	2.692308	.659341
25-5	36	1.153846	.384615
25-6	28	1.263726	.164835
25-7	22	.989011	.109890
25-8	16	.714286	.054945
25-9	19	.769231	.109890
25-10	18	.659341	.164835
25-11	36	1.263736	.274725
25-12	23	3.681319	5.054945
25-13	21	.769231	.109890
25-14	16	.824176	.054945
25-15	31	2.252747	.274725
25-16	30	2.472527	.274725
25-17	34	.934066	.219780
25-18	37	1.538462	.219780
25-19	50	1.813187	.329670
25-20	116	6.043956	1.043956
Average Time		1.563191	.491758

Table 4.13: Table giving the execution time of Socher's algorithm and PIAP

Size 30	I	Execution time	
Prob. No.	PIs	SOCH	PIAP
30-1	57	2.582418	.714286
30-2	41	1.813187	.384615
30-3	48	2.692308	.329670
30-4	38	1.263736	.219780
30-5	47	3.076923	.329670
30-6	36	1.758242	.219780
30-7	95	3.571429	1.153846
30-8	34	.769231	.164835
30-9	26	1.923077	.219780
30-10	33	1.428571	.109890
30-11	26	1.593407	.164835
30-12	43	1.758242	.384615
30-13	5	.549451	.054945
30-14	14	.494505	.054945
30-15	24	.879121	.219780
30-16	28	1.263736	.109890
30-17	40	1.978022	.274725
30-18	60	2.967033	.439560
30-19	46	1.428571	.439560
30-20	40	2.362637	.329670
Average Time		1.806479	.315954

Table 4.14: Table giving the execution time of Socher's algorithm and PIAP

Size 35		Execution time	
Prob. No.	PIs	SOCH	PIAP
35-1	10	.879121	.054945
35-2	14	1.813181	.109890
35-3	31	1.758242	.219780
35-4	34	2.582418	.274725
35-5	33	1.318681	.219780
35-6	30	.879121	.109890
35-7	33	1.538462	.384615
35-8	16	2.582418	.109890
35-9	33	1.758242	.219780
35-10	33	1.923077	.164835
35-11	38	2.582418	.659341
35-12	48	*	.439560
35-13	42	2.252747	.384615
35-14	32	1.373626	.274725
35-15	15	.769231	.054945
35-16	21	1.208791	.109890
35-17	19	.934066	.164845
35-18	49	3.791209	.549451
35-19	42	1.978022	.219780
35-20	29	2.637363	.329670
Average Time		1.818970	.2429153

Table 4.15: Table giving the execution time of Socher's algorithm and PIAP

Size 40		Execution time	
Prob. No.	PIs	SOCH	PIAP
40-1	33	3.956044	.329670
40-2	18	1.593407	.109890
40-3	53	4.890110	.439560
40-4	44	3.131868	.329670
40-5	50	2.967033	.714286
40-6	27	1.538462	.219780
40-7	24	1.648352	.164835
40-8	23	2.362637	.219780
40-9	45	2.692308	.384615
40-10	47	4.055495	.384615
40-11	37	2.857143	.439560
40-12	35	1.813187	.494505
40-13	29	2.967033	.164835
40-14	44	2.087912	.274725
40-15	24	1.703297	.164835
40-16	46	1.978022	.439560
40-17	34	2.637363	.329670
40-18	8	1.263736	.054945
40-19	34	1.318681	.164835
40-20	38	4.010989	.274725
AverageTime		2.573654	.304945

Table 4.16: Table giving the execution time of Socher's algorithm and PIAP

Size 45 & 50		Execution time	
Prob. No.	PIs	SOCH	PIAP
45-1	56	3.406593	.549451
45-2	28	2.142857	.439560
45-3	37	4.725275	.439560
45-4	30	2.692308	.274725
45-5	28	4.175824	.274725
45-6	22	4.230769	.219780
45-7	43	4.450549	.329670
45-8	.51	3.846154	.329670
45-9	45	2.857143	.329670
Average Time		3.614164	.354090
50-1	27	4.175824	.274725
50-2	38	3.516484	.384515
50-3	11	2.637363	.109890
50-4	39	3.131868	.384614
50-5	52	*	.989011
50-6	23	3.241758	.109870
50-7	76	*	.879121
Average Time		3.340660	.252727

Table 4.17: Table giving the # of input clauses and the average execution time with SD and CV, by Socher's algorithm and PIAP

#of	SOCH			PIAP			DF	*paired*	*P*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
15	.517	.285	.551	.157	.120	.766	19	8.089	< .0005
20	1.199	1.085	.905	.742	1.162	1.566	17	7.308	< .0005
25	1.563	1.362	.872	.492	1.099	2.235	19	4.107	< .0005
30	1.806	.857	.474	.316	.252	.798	19	9.850	< .0005
35	1.819	.785	.432	.243	.162	.667	18	10.475	< .0005
40	2.574	1.036	.403	.305	.157	.515	19	10.815	< .0005
45	3.614	.888	.246	.354	.103	.291	8	11.919	< .0005
50	3.341	.565	.169	.253	.138	.546	4	13.913	< .0005

Table 4.18: Table giving the # of Prime Paths and the average execution time with SD and CV, by Socher's algorithm and PIAP

#of	SOCH			PIAP			DF	*paired*	*P*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
10-15	.989	.890	.900	.0785	.0294	.374	6	2.785	< .025
15-20	.780	.599	.767	.114	.089	.764	14	4.401	< .0005
20-25	1.365	1.069	.782	.215	.166	.775	11	3.913	<.005
25-30	1.878	1.186	.631	.252	.185	.735	16	6.107	< .0005
30-35	1.210	.698	.576	.207	.068	.332	16	6.332	<.0005
35-40	2.112	1.167	.552	.363	.230	.633	17	6.764	< .0005
40-45	2.030	1969	.477	.349	.111	.318	13	6.884	<.0005
45-50	2.394	1.086	.453	.429	.132	.306	9	6.193	<.0005

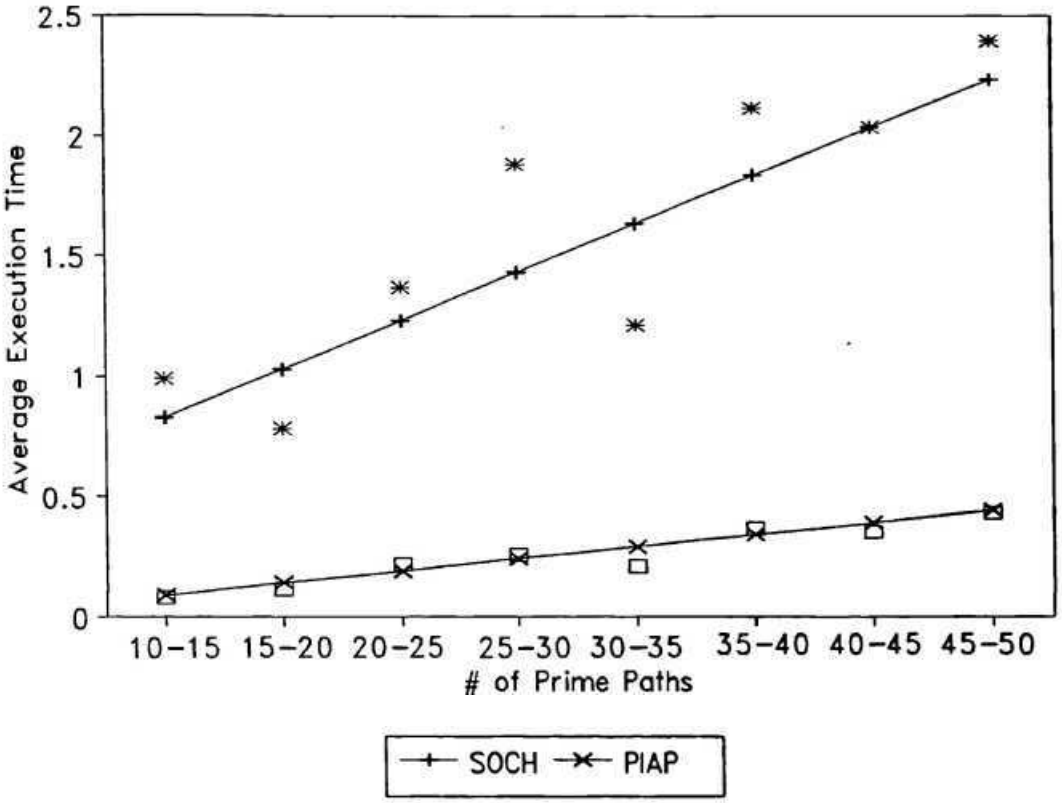


Figure **4.7:** The **graph depicting the execution time taken by Socher's** algorithm **and PIAP**

4.5.3 Experiment 3

While the execution time for the algorithms is machine dependent, the number of paths generated is independent and hence a study on the number of paths generated by both the algorithms for each of the data is also carried out. As already discussed in Chapter 3, and substantiated here by Experiment 1, the number of subsumptions required by PIAP is much less compared to that required in Socher's algorithm. This is so because the number of paths generated by PIAP itself is much less. The comparison of column 3 and column 10 of Table 4.1 to Table 4.7 establishes that the number of paths generated by PIAP is much less compared to that generated by Socher's algorithm. This is because the generation of paths which may be subsumed at a later stage is avoided in PIAP. Hence the paths which are not prime in a bigger set (lower level of the tree) are few in number.

In order to check the efficency of PIAP, the *paired t test* is carried out. The results obtained based on the input size and the output size are given in Table 4.19 and Table 4.20, respectively. It is observed (last columns of Table 4.19 and Table 4.20) that the algorithm PIAP is significantly better than Socher's algorithm since $p < .0005$ in most of the cases. The first order regression for. the data in Table 4.20 is obtained and is depicted in Figure 4.8. The Table 4.20 and the Figure 4.8 substantiate the claim that PIAP is efficient than Socher's algorithm.

Thus, all the three experiments substantiate the theoretical arguments provided in Chapter 3 regarding the efficiency of PIAP over the Socher's algorithm.

Table 4.19: Table giving the # of input clauses and the average # of paths with SD and CV, by Socher's algorithm and PIAP

#of	SOCH			PIAP			DF	*paired*	*p*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
15	100	62	.618	35	22	.617	19	6.231	<.0005
20	178	144	.810	54	43	.794	17	4.681	<-0005
25	245	219	.890	47	55	1.161	19	4.836	< .0005
30	285	157	.551	46	24	.520	19	7.712	< .0005
35	260	115	.440	33	13	.386	18	9.539	< .0005
40	374	150	.401	37	13	.356	19	10.720	< .0005
45	515	126	.245	39	11	.285	8	11.972	< .0005
50	387	75	.194	35	17	.493	3	11.680	< .0005

Table 4.20: Table giving the # of prime paths and the average # of paths with SD and CV, by Socher's algorithm and PIAP

# of	SOCH			PIAP			DF	*paired*	*p*
PIs	Mean	SD	CV	Mean	SD	CV		*t*	
10-15	108	92.734	.862	14	2.326	.171	7	2.926	<.025
15-20	129	115.998	.901	19	2.154	.112	13	3.580	< .005
20-25	194	143.829	.742	27	6.171	.232	11	4.142	<.001
25-30	255	156.314	.614	32	4.671	.147	16	5.978	<.0005
30-35	213	124.645	.584	37	5.284	.141	19	6.487	<.0005
35-40	336	137.803	.411	44	7.499	.170	19	9.773	<.0005
40-45	321	145.631	.454	48	5.919	.124	13	7.166	<.0005
45-50	386	161.547	.418	62	14.774	.240	8	6.515	<.0005

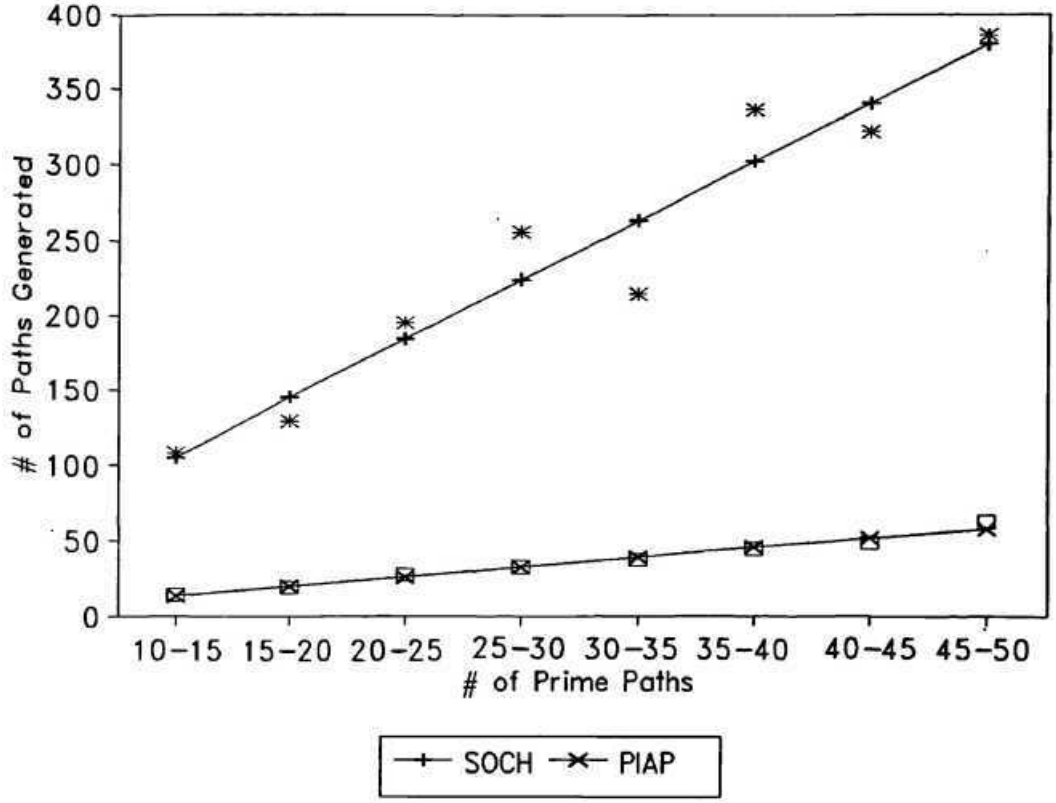


Figure 4.8: The graph depicting the # of paths generated by Socher's algorithm and PIAP.

4.6 Editing of Tree

When the reasoner transmits the set $\mathcal{H}$ of clauses to the RMS there is the need to update the RMS database. In order to accomplish this, the binary tree corresponding to the clauses that are already transmitted to the RMS has to be modified. Addition and deletion of clauses and the changes thereof in the binary tree are being discussed in this section. Updating RMS database means that the prime paths for the updated database has to be obtained. The incremental computation of prime paths using the tree structure is also discussed in this section.

4.6.1 Addition of a clause

The need for updating a formula, and the incremental algorithm to update the compiled knowledge base has been discussed in earlier sections. It has already been discussed that the existing algorithms [Kean 90, Jackson 92, Socher 91] does not incorporate efficiently the set $\mathcal{H}$ of clauses transmitted by the reasoner. If $\mathbf{T}(\mathcal{F})$ denotes the binary tree corresponding to the formula $\mathcal{F}$, then the problem is to obtain $\mathbf{T}(\mathcal{F} \wedge \mathcal{H})$ making use of $\mathbf{T}(\mathcal{F})$. The different ways of obtaining $\mathbf{T}(\mathcal{F}$A $\mathcal{H})$ are discussed here.

Method 1:-

In this method, the set $\mathcal{H}$ is partitioned into $\mathcal{H}_r$ and $\mathcal{H}'_r$ with respect to the label r of the root node of the tree $\mathbf{T}(\mathcal{F})$. The left subtree $\mathbf{T}(\mathcal{F}'_r)$ of the root node is updated with those clauses $(\mathcal{H}'_r)$ in $\mathcal{H}$ which do not contain r. The right subtree $\mathbf{T}(\mathcal{F}_r)$ of the root node is updated with the subset $(\mathcal{H}_r)$of $\mathcal{H}$ obtained by removing the literal r from the clauses in $\mathcal{H}$ containing r. This process is continued until leaf node is reached. If the leaf node of the tree $\mathbf{T}(\mathcal{F})$ is to be updated with a nonempty subset of $\mathcal{H}$, then the node is no more a leaf node, and hence the corresponding formula is split until further partitioning is not possible. The pseudo-code to update a tree $\mathbf{T}(\mathcal{F})$ and an example are given below.

Procedure **Update1_Tree** $(\mathbf{T}(\mathcal{F}), \mathcal{H})$

Initialize root of $\mathbf{T}(\mathcal{F}$ A ft) as root of $\mathbf{T}(\mathcal{F})$.

r = the label of the root of $\mathbf{T}(\mathcal{F})$.

Compute $\mathcal{H}'_r = \{h \in H \setminus r \notin h\}$

Compute $\mathcal{H}_r = \{h - r \setminus b \in \mathcal{H}, r \in h\}$

If $\mathbf{T}(\mathcal{F}'_r)$ is NULL, then

$\mathbf{T}(\mathcal{F}'_r$ A $\mathcal{H}'_r)$ = **CREATE_TREE** $(\mathcal{H}'_r)$.

else

Update1_Tree $(\mathbf{T}(\mathcal{F}_r'), \mathcal{H}_r')$

If $\mathbf{T}(\mathcal{F}_r)$ is NULL, then

$\mathbf{T}(\mathcal{F}_r$A $\mathcal{H}_r)$ = CREATE_TREE $(\mathcal{H}_r)$.

else

Update1_Tree $(\mathbf{T}(\mathcal{F}_r), \mathcal{H}_r)$

END

Example 4.6.1:-

Let $\mathcal{H} = \{abcef, adfgabf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ be the formula to be appended to the formula $\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg\}$.

The set $\mathcal{H}$ is incorporated at the root of the tree $\mathbf{T}(\mathcal{F})$. This is depicted in Figure 4.9. The clauses in the box has to be appended to the node to which the arrow points. The literal *a* is the label of the root of the tree $\mathbf{T}(\mathcal{F})$ and this literal is present in some clauses of $\mathcal{H}$. Hence $\mathcal{H}_a = \{bcefdfgbf\}$is to be appended to the right subtree of node 1 (root node) in Figure 4.10. The remaining clauses, $\mathcal{H}_a' = \{\tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$ are to be appended to the left subtree (node 2) of node 1 (Figure 4.10). The node 2 of $\mathbf{T}(\mathcal{F})$ is NULL and hence, the tree for the formula $\mathcal{H}_a'$ is created. One[3] of the trees thus obtained is given in Figure 4.11. In order to update the node 3 of $\mathbf{T}(\mathcal{F})$, the right subtree of node 3 has to be updated with $\{cef\}$ and the left subtree is updated with $\{dfg, bf\}$.This process is continued and one of the complete binary tree thus constructed for $\mathbf{T}(\mathcal{F}$ A $\mathcal{H})$ is the same as the tree already given in Figure 4.1. It is to be noted that in this method, if $\mathcal{H}$ has any literal which is not present in $\mathcal{F}$, then these literals become the labels of nodes only when a leaf node or NULL node of $\mathbf{T}(\mathcal{F})$ is reached.

A simple case of the addition of the formula is when the formula has to be updated by a single clause *h*. For updating the binary tree, if the label of the root is present in h then the right child is updated by the clause *h* — *{label}*. On the other hand, if the label

[3]**There are different trees depending on the choice *of* the label**

of the node is not present in *h* then the left child is updated by the clause *h.*

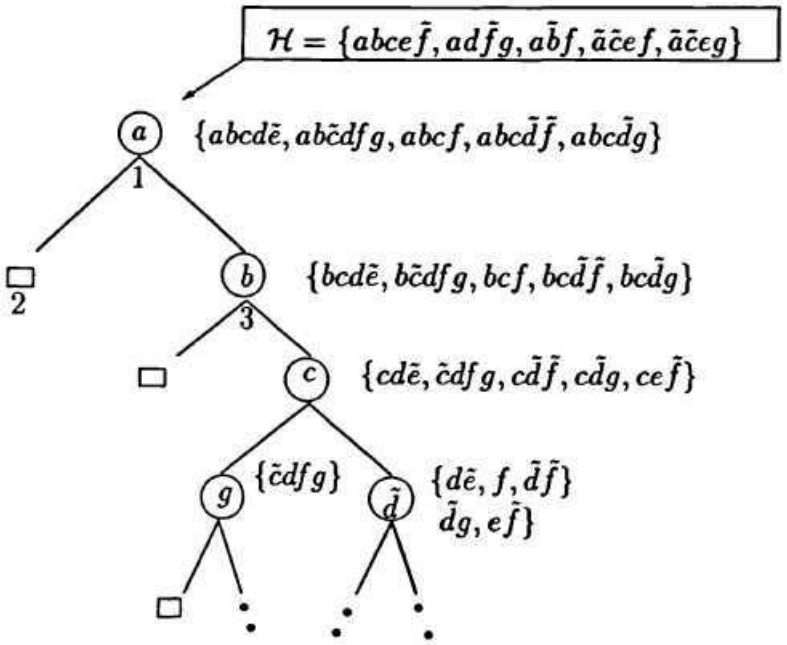


Figure 4.9: Incorporation of $\mathcal{H}$ to the root of $\mathbf{T}(\mathcal{F})$

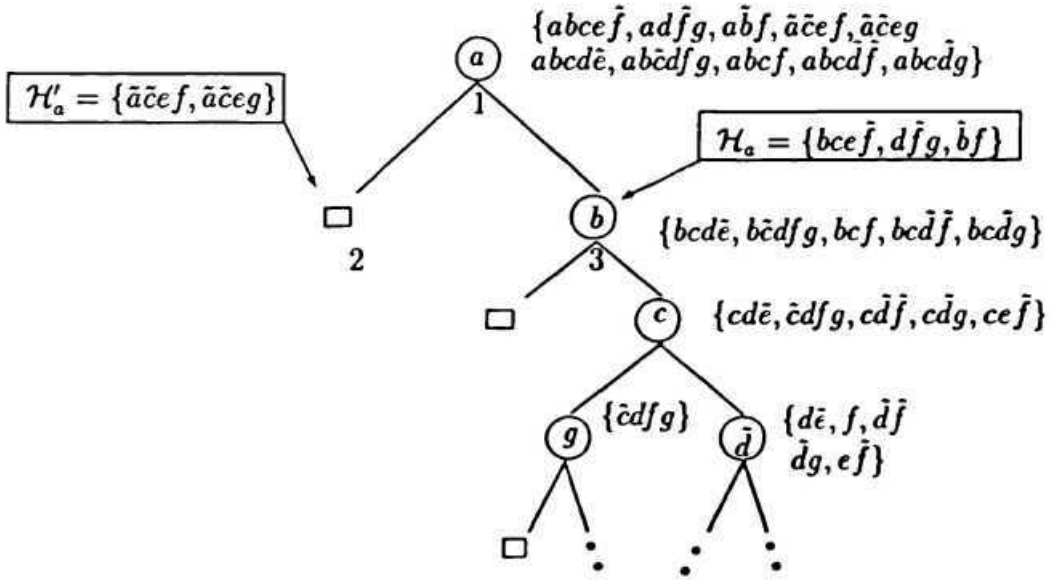


Figure 4.10: Incorporation of $\mathcal{H}_a$ and $\mathcal{H}'_a$ to the right child and left child of the root node of $\mathbf{T}(\mathcal{F})$.

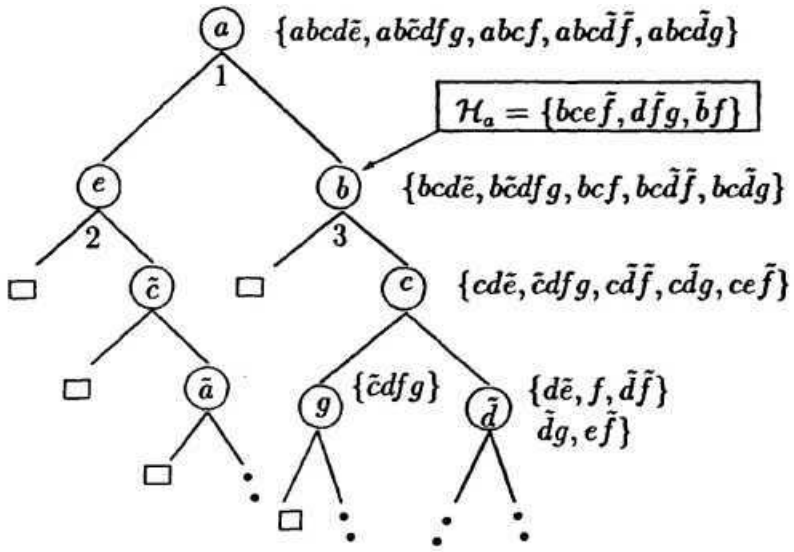


Figure 4.11: Binary tree after incorporation of $\mathcal{H}'_a$ to $\mathbf{T}(\mathcal{F}'_a)$

Method 2:-

The main aim of tree-representation of $\mathbf{T}(\mathcal{F} \wedge \mathcal{H})$ is to facilitate computation of prime implicants incrementally as and when the reasoner transmits clauses. The Method 1 used to construct $\mathbf{T}(\mathcal{F} \wedge \mathcal{H})$ is not suitable for the incremental computation of prime paths for the simple reason that in this method, the prime paths already available for $\mathbf{T}(\mathcal{F})$ are not made use of. Moreover, Method 1 of appending $\mathcal{H}$ to $\mathcal{F}$ depends on $\mathbf{T}(\mathcal{F})$ and requires updating of most of the nodes in it. Another method is proposed here to construct $\mathbf{T}(\mathcal{F}\mathrm{A}\,\mathcal{H})$ keeping $\mathbf{T}(\mathcal{F})$ intact. In this method, the binary tree $\mathbf{T}(\mathcal{H})$ for the formula $\mathcal{H}$ is constructed independent of $\mathbf{T}(\mathcal{F})$. The binary tree for $\mathbf{T}(\mathcal{F} \wedge \mathcal{H})$ is created with $\mathbf{T}(\mathcal{H})$ as the right subtree and $\mathbf{T}(\mathcal{F})$ as the left subtree with the root node being dummy, if $\mathcal{F}$ and $\mathcal{H}$ are defined over the same set of literals. If there is any literal in $\mathcal{H}$ foreign to $\mathcal{L}$, the set of literals, then the root of $\mathbf{T}(\mathcal{F}\,\mathrm{A}\,\mathcal{H})$ is not dummy. If the root node is dummy, a dummy literal (say, x) is considered the label of the root node of $\mathbf{T}(\mathcal{F}\mathrm{A}\,\mathcal{H})$. In this case, it is not necessary to consider the tree corresponding to the additional formula as the

right subtree of the new root node created. However, if the root node is not dummy, any of the literals foreign to £ is considered the label of the root node of $\mathbf{T}(\mathcal{F}$ A $\mathcal{H})$. In this case, the right subtree of $\mathbf{T}(\mathcal{F} \wedge \mathcal{H})$ must necessarily be $\mathbf{T}(\mathcal{H})$. The binary tree obtained by this method for the Example 4.6.1 is given in Figure 4.12, where the right subtree represents the formula $\mathcal{H}$. The particular case where $\mathcal{H}$ is a single disjunctive clause is so simple; the new clause becomes the right child leaf node of the dummy node.

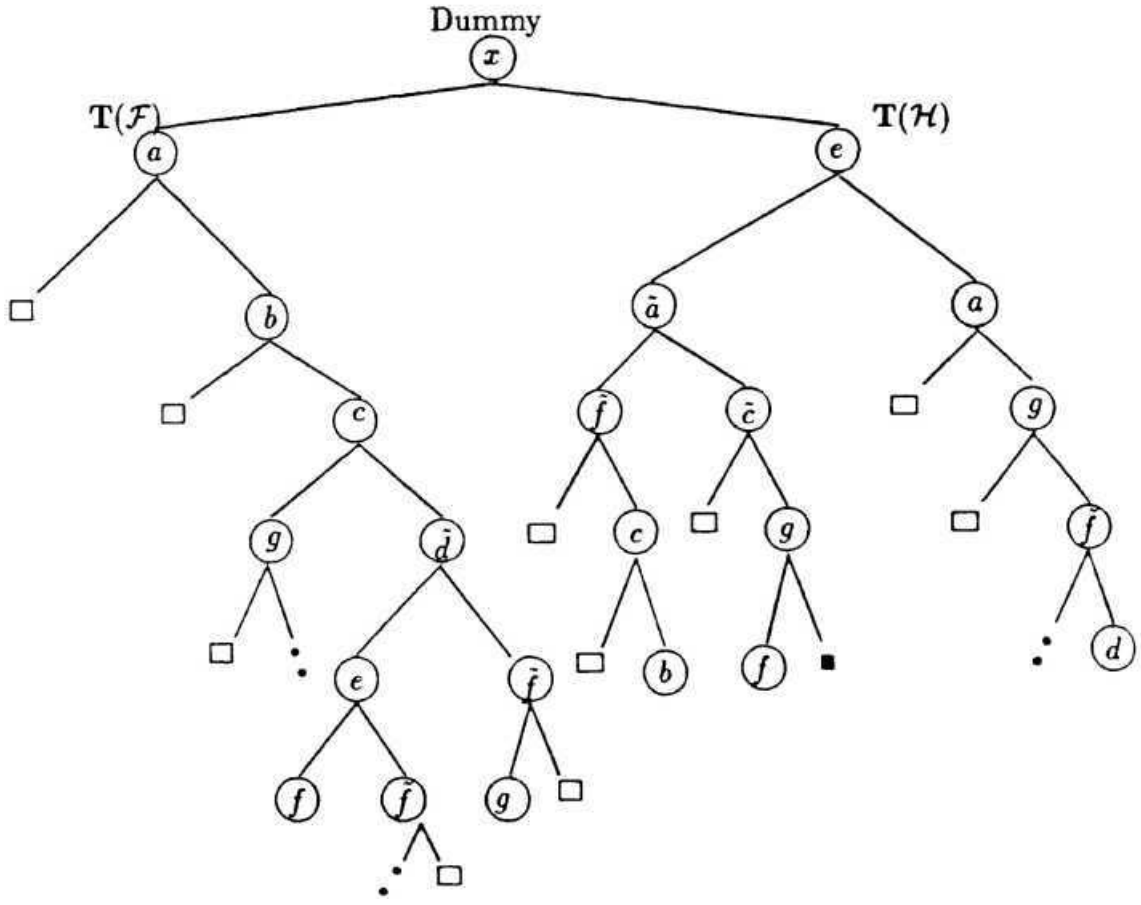


Figure 4.12: Tree-representation of formula $\mathcal{F} \wedge$ ft by Method 2

$\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abf, abcdf, abcdg\}$.

$\mathcal{H} = \{abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$

Method 3:-

This method is the combination of Method 1 and Method 2. The tree $\mathbf{T}(\mathcal{H})$ is constructed separately as in Method 2 whereas the literals chosen for partitioning are the same as those in $\mathbf{T}(\mathcal{F})$. i.e., if r is the label of the root of $\mathbf{T}(\mathcal{F})$, then the same r is the label of the root node of $\mathbf{T}(\mathcal{H})$. In this process, at any level of the tree the corresponding nodes of both trees will have the same label. The tree thus constructed for the Example 4.6.1 is given in Figure 4.13. It can be seen that the labels of the corresponding nodes of the trees are the same.

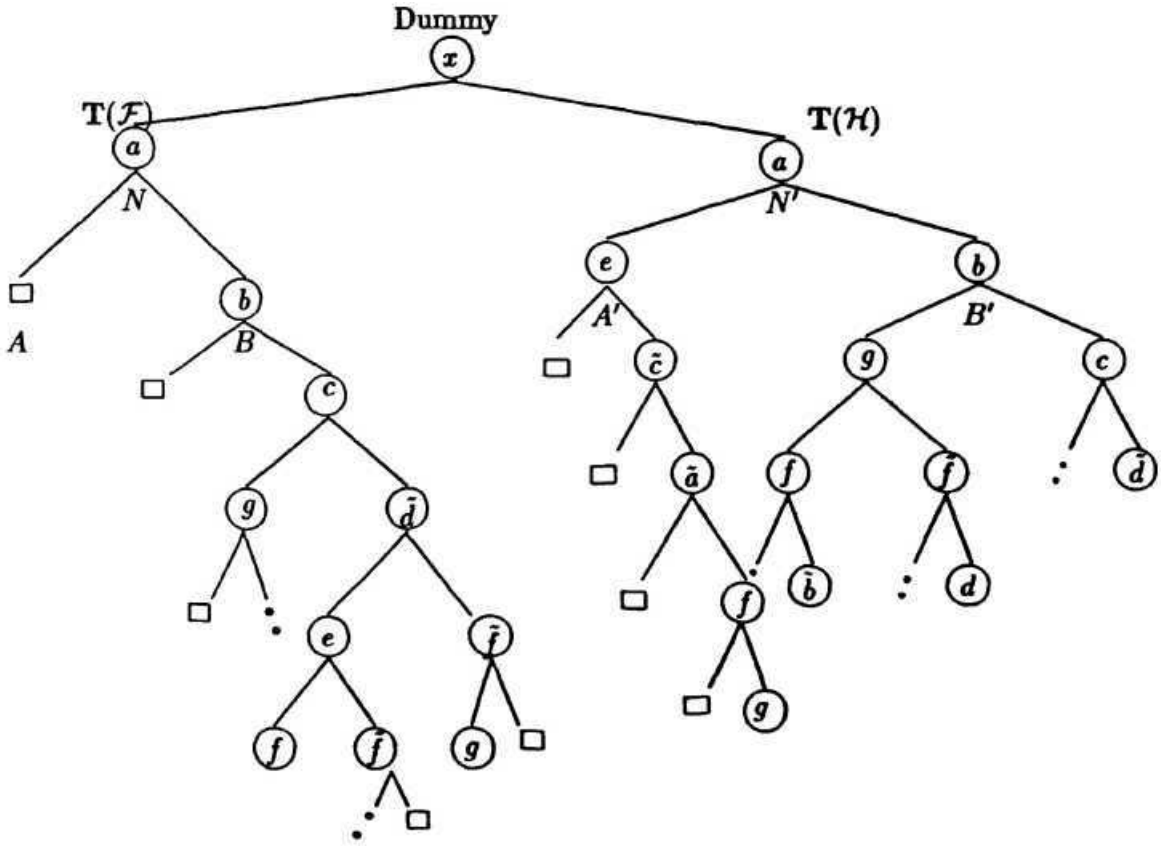


Figure 4.13: Tree-representation of $\mathcal{F} \wedge \mathcal{H}$ using Method 3.

$\mathcal{F} = \{abcd\tilde{e}, ab\tilde{c}dfg, abcf, abcdf, abcdg\}$.

$\mathcal{H} = \{abcef, adfg, abf, \tilde{a}\tilde{c}ef, \tilde{a}\tilde{c}eg\}$

The advantages of Method 3 over Method 1 and Method 2 are explained below. In Figure 4.13, let *A* and *B* are the left and right child, respectively of the root node *N* with label *a* representing $\mathcal{F}$. Similarly, let *A'* and *B'* be the left" and right child of the root node *N'* representing $\mathcal{H}$. Since the label of the nodes *N* and *N'* are the same, it can be visualized as branching from a single node (say, NN^r). i.e., the nodes *A* and *A'* can be visualized as the left child of a single node *NN'*. Similarly, *B* and *B'* is the right child of the node NN^f. Further, *A* and *A'* itself can be visualized as the left and right child of a dummy node (Figure 4.14). Similarly the nodes *B* and *B'* form the left and right child of another dummy node. In this visualization, the root node label of $\mathbf{T}(\mathcal{F}$ A $\mathcal{H})$ is the same as that of $\mathbf{T}(\mathcal{F})$ and $\mathbf{T}(\mathcal{H})$. This visualization helps in the incremental computation of prime paths which is discussed in Section 4.6.3.

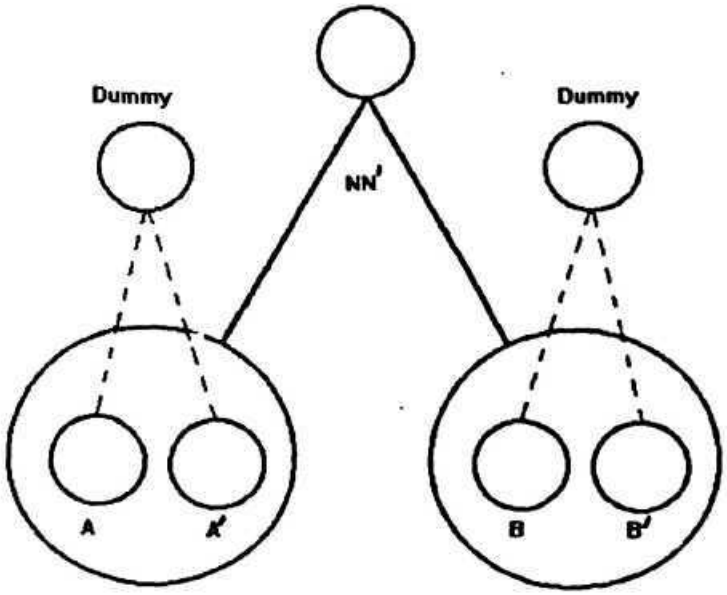


Figure 4.14: Visualization of nodes *A, A'* (the left child nodes) and *B, B'* (the right child nodes) of two nodes with same label at same level of tree as the left child and right child, respectively of a dummy node

4.6.2 Deletion of a clause

Different methods of updating a tree has been discussed in the above section. Deletion of a clause is equally important as the reasoner might wish to transmit a clause temporarily and later delete it. Deleting a clause from the binary tree corresponding to a formula is the issue in this subsection.

If a set of clauses has to be deleted from a formula, choose the corresponding columns in the binary matrix representation of the formula. If these columns have 1 in the row (label) associated to the node representing the formula, then delete the chosen columns from the right child of the node; otherwise, delete the chosen columns from the left child of the node.

The deletion of clauses is easy if Method 2 or Method 3 is followed for updating a tree. If the reasoner transmits a clause temporarily and later wish to delete it, it is only required to delete the right subtree (since this corresponds to the formula temporarily added) of the root of the binary tree .

4.6.3 Incremental computation of prime paths

It is already mentioned in Section 3.4.4 that the algorithm PIAP is well suited for computing the prime implicants incrementally. We have also seen that when the reasoner transmits a set *ft* of clauses, there are different ways of obtaining the binary tree representation of the formula $\mathcal{F}$ A $\mathcal{H}$. The computation of prime paths in each of these cases is discussed here.

Case 1: Tree is *updated using Method 1:-* In this case the prime paths for the new tree representing $\mathcal{F} \wedge \mathcal{H}$ is obtained exactly following the steps in PIAP. This method is definitely not efficient as it does not make use of the prime paths already computed for a portion of the submatrix representing each of the nodes.

Case 2: $\mathcal{H}$ *is defined over* $\mathcal{L}$ *and* $\mathbf{T}(\mathcal{H})$ *is constructed by Method 2:-* In this case the binary tree for $\mathcal{F}$ is kept intact and so is its set of prime paths. The prime paths for the formula $\mathcal{H}$ are computed independently using PIAP. Since the root of the new tree constructed has dummy label the prime paths for the formula $\mathcal{F}$ A $\mathcal{H}$ is computed following PIAP except Step 4. Moreover, since the label of the root is dummy, it is immaterial whether $\mathbf{T}(\mathcal{F})$ is the left or right child.

Case 3: $\mathcal{H}$ *is defined over* $\mathcal{L}$ *and* $\mathbf{T}(\mathcal{H})$ is constructed by *Method 3.* It is already discussed that in this case, the two trees $\mathbf{T}(\mathcal{F})$ and $\mathbf{T}(\mathcal{H})$ can be merged together. This visualization makes it easy to compute the prime paths of $\mathcal{F}'$ A $\mathcal{H}'$ from the prime paths of $\mathcal{F}'$ and $\mathcal{H}'$, where $\mathcal{F}'$ and $\mathcal{H}'$ are the **subformulae** of $\mathcal{F}$ and $\mathcal{H}$, respectively. Thus, the prime paths of any node in the binary tree can be computed incrementally, making use of the prime paths already computed for the corresponding node in the original formula. Each of the intermediate node can be considered a tree with a dummy root node (as in the case of Method 2) and hence the prime paths of each of the intermediate node can be computed incrementally using the method adopted for Case 2.

Case 4: $\mathcal{H}$ *has some literal (say, r) foreign to C and* $\mathbf{T}(\mathcal{H})$ is constructed *by Method 2 or Method 3:-* In this case also the binary tree for $\mathcal{F}$ is kept intact and so is its set of prime paths and the prime paths for the formula $\mathcal{H}$ are computed independently using PIAP. The root of the tree $\mathbf{T}(\mathcal{F}$ A $\mathcal{H})$ has the literal r (any one, in the case of more than one) as the label and hence the prime paths for the formula $\mathcal{F}$ A $\mathcal{H}$ are computed exactly following PIAP with $\mathbf{T}(\mathcal{F})$ as the left child and $\mathbf{T}(\mathcal{H})$ as the right child. Further, Case 2 or Case 3 is applicable if $\mathbf{T}(\mathcal{H})$ is constructed using Method 2 or Method 3, respectively.

4.7 Conclusion

In this chapter, a new binary tree representation of a formula is introduced. This representation naturally evolved from the partitioning scheme involved in PIAP proposed in Chapter 3. The basic structure of a node in the tree, the creation of the tree, and the implementation details of PIAP are also discussed in this chapter. Different experiments were carried out to compare the performance of PIAP and Socher's algorithm. The experimental results reported in this chapter substantiate positively that the algorithm PIAP is very efficient compared to Socher's algorithm. Apart from being efficient in sequential computatuion of prime implicants, PIAP has many other advantages. The binary tree representation is well suited for representing the updated knowledge as well as for compiling the knowledge incrementally. Different methods to accomplish this are also discussed in this chapter. The compiling method is global in the sense that all the clauses in the knowledge-base are treated collectively, and the newly-added information which can be a set of clauses, is also treated collectively for incremental computation. Further, unlike the earlier algorithms which are inherently sequential, the proposed algorithm is naturally parallelizable. The parallel algorithm is described in Chapter 5. Moreover, the tree structure is utilized to have a full-fledged RMS.

In a nut-shell form, the advantages of PIAP are :

- Number of subsumptions, execution time and the number of candidate paths for subsumption are less compared to Socher's algorithm.
- Well suited for knowledge compilation in global as well as incremental mode.
- The algorithmm is naturally parallelizable.

Chapter 5

PARALLELIZATION OF PIAP

5.1 Introduction

Parallel processing has been a subject of interest for the computer scientists and has always fascinated them. In the context of today's technology, which has reached a limit where it seems impossible to gain higher performance from sequential machines, parallel processing is being accepted as an alternative architectural approach to overcome this technology barrier.

The problem of computing prime implicants for a prepositional formula in CNF is computationally intractable [Provan 90]. Except for certain specific cases, the algorithms to solve the problem are known to be exponential in nature. Hence, for reasonably big problem size, prime implicants computation may not be practical with the existing com* putational methods. Therefore, it is necessary to resort to high performance computing.

This chapter explores the possibility of developing a parallel algorithm for the present problem. As discussed in earlier chapters, the importance of prime implicants in RMS have been established by [Reiter 87, Inoue 90, Provan 90], and hence most of the attempts have been focused on developing an efficient and fast algorithm to compute prime implicants. However, there has not been any attempt so far to utilize parallel computing technique for this purpose. The new algorithm, PIAP proposed in this dissertation has the additional advantage of being parallelizable. The aim of the present study is to investigate different aspects of parallelization of PIAP.

The coarse-grain parallelism can be seen trivially due to the tree structure proposed in Chapter 4. The prime paths for different nodes at the same level of the tree can be computed simultaneously and independently. The medium-grain parallelism can be achieved when concatenation and subsumption operations are performed between sets of paths with respect to two sibling nodes of the same parent node. Finally, the fine-grain parallelism can be achieved when the subsumption between the sets of paths is carried out as vector computation. Thus, three different levels of parallelism are explored here and the suitable architecture for each of these is also investigated.

The principles of parallel computing, the parallel architectures, and earlier works which deal with propositional formula using parallel computers are briefly reviewed. In later sections, each of the levels of parallelization are dealt with separately. The hybrid architecture to solve the problem of computation of prime implicants is also proposed.

5.2 Review of Architecture

This section is concerned with a brief review of the parallel architecture and parallel algorithms. There are many ways of classifying machine architecture. The first classification of parallel computers is given by Flynn's [Flynn 66] taxonomy in 1966. In this classification, multiprocessors are distinguished according to multiplicity of instructions and data streams. Though this classification is not comprehensive for parallel processors today, it spans the complete spectrum of the processor organization. According to this classification, there are four categories of parallel processors.

SISD- Class of computations with Single Instruction stream and Single Data stream.

SIMD- Class of computations with Single Instruction stream and Multiple Data stream.

MISD- Class of computations with Multiple Instruction stream and Single Data stream.

MIMD- Class of computations with Multiple Instruction stream and Multiple Data stream.

Flynn's classification does not cover all aspects of interconnection in the parallel processor. However, it is general enough for the design of many parallel algorithms. Generally, SIMD and MIMD classes of parallel computers are relevant in the present context. SIMD computation which exhibits data parallelism may be employed when the data is regular and calculation tends to be uniform. MIMD machine further falls into two categories; shared memory and distributed memory machines, though many architectures exhibit both parallelism. Finally, there are systolic arrays, pipe-line computation, mesh-connected arrays, hypercube network, shuffle exchange method, and partitionable and hierarchical multiprocessor architecture [Hwang 89] that have been proposed and studied for varieties of applications.

Pipeline Computation

Pipelining is one of the most primitive forms of synchronous computation. It consists of processors connected in a certain fashion by which on each machine cycle each processor receives data from its input ports, performs the required computations and passes the result and data through its output ports. Once the pipeline is filled, all the processing elements operate in parallel and one output is produced per cycle. Therefore, pipeline parallelism is effective for handling batches of data when operations on them can be broken down into distinct suboperations.

Shared Memory

Shared memory systems share a common block of the memory, and the processor cannot be used as a stand alone computer since the memory is shared. High interaction between various processing units and fast data communication is feasible in shared memory

multiprocessors. In a shared memory SIMD computer, data comirninication takes place through shared memory location. In this, information flowing from a processing unit to a memory module contains data, address and other memory controls. In a local memory computer, information exchange between two processing units is through discrete messages which flow on the network wires in either serial or parallel order. The processing units in a MIMD computer interact with each other to solve a problem collectively and the computer organization is such that the interaction between various processing units is high and the data communication is fast. These are highly coupled multiprocessors and an interaction of this type is only possible with multiprocessors using the concept of shared memory. An important issue in shared-memory systems is data synchronization. That is, processors must synchronize with each other to avoid write conflicts, and to enforce the data precedence relationship inherent in the algorithm. The synchronization between processors in a shared memory multiprocessor is difficult and the basic mechanism of synchronization in such type of multiprocessors is by setting a lock, either in hardware or in software, before changing a critical variable shared by two or more processors.

Vector Processor

Vector processors are useful when an identical function is repeated many times for different data values, namely the elements of vectors. The efficiency of vector processing is primarily determined by the way the vector or matrix is handled. The vector processing can be achieved through a pipeline where a set of data is computed one after another in a pipeline mode, or through array processing where all data items can be computed in parallel.

A vector operand contains an ordered set of n elements, where *n* is the length of the vector. Four basic types [Ghosh 95] of primitive vector processing instructions are:

1. Those that operate on one vector and produce another vector. The input to these instructions is, therefore, only one vector.
2. Those that operate on one vector to produce a scalar. The input to these instructions is only one vector and the output is a scalar.
3. Those that operate on one vector and one scalar to produce one vector output.
4. Those that operate on two vectors to produce one vector.

The vector processing required for the computation of prime paths is of the fourth primitive type, where two vectors are given as input to produce one vector output.

A generic vector processor requires two input vector streams and provides one output vector stream. For the vector processor to work, two inputs must be simultaneously present on the input ports. Vector processor performs the designated operation on the vector elements and sends the result through the output port.

5.3 Design of Parallel Algorithms

The designing of efficient parallel algorithms must be guided by the architectural organizations that may support at a particular time. However, for a generalized approach, the extent of theoretical studies on parallelism exhibited by a problem should be far more developed to absorb any new architectural innovations that augment the technology of parallel computation. The emphasis of theoretical studies lies in extracting inherent parallelism of a problem. This study results in designing efficient parallel algorithms taking into consideration one of the following.

1. Mapping an existing sequential algorithm to a suitable parallel processor.
2. Designing afresh a new algorithm exploiting fully the problem's inherent characteristics.

It is observed in the approach of mapping a sequential algorithm to a suitable parallel machine that the best sequential algorithm need not be the best parallel algorithm. In the present context, the new algorithm, PIAP, is superior to other known algorithms in sequential computing, and is also suitable for parallelization. In this context, it is worthwhile to study the different parameters of the nature of parallelism.

Nature of Parallelism: Nature of parallelism has a number of attributes that depend on what kind of parallelism is used and the way in which the algorithms and/or data can be decomposed [Jamieson 88]. The following are some important parallelisms.

1. Data parallelism and function parallelism.
2. Data granularity.
3. Module granularity.
4. Degree of parallelism
5. Uniformity of operations.
6. Synchronous requirement.
7. Static and Dynamic characters of the algorithm.
8. Data dependencies.

The most widely used theoretical model for parallel algorithms is paralle random access machine (PRAM). PRAM skirts the communication overheads entirely by enforcing communication through shared memory wherein the access is allowed to be made in a transparent manner simultaneously by all processors as long as they avoid clash for a specific location of the memory. Even the clashes or memory conflicts are allowed under some predetermined protocol which the designer of the algorithm may have to tackle.

5.4 Motivation and Earlier Work

The problem of computing prime implicants of a prepositional formula is NP-hard. So, in any real life application, when the number of prime implicants is very large, it would be difficult to compute the prime implicants in acceptable time. Therefore, it may be necessary to resort to multiprocessor computing to solve this problem. Since **PIAP** has an inherent tree structure, mapping the algorithm to a parallel computer becomes easy. Since the set of clauses representing the formula is successively divided into subsets and the paths of the subsets are concatenated to get the paths of the original formula, it is natural to process the subsets simultaneously. Thus different nodes of the tree can be processed in a parallel computer. On the other hand, when the paths of two nodes are combined to get the paths of the common parent node, the concatenation process for different paths can also be done in parallel. Every path consists of a set of literals and the subsumption operation between two paths is essentially comparing the clause set of the respective paths. This comparison can also be accomplished for each pair of clauses in parallel. Thus, PIAP facilitates multiple levels of parallelism,

- when the problem is subdivided into subproblems
- at the intermediate level (at each intermediate node of the tree)
- at the primitive data, namely, literals.

One can term all these as parallelization of different granularities-coarse-grain, medium-grain and fine-grain. In the following sections, suitable architectures for each of these levels of parallelism are explored and a suitable hybrid architecture for **PIAP** is proposed to solve the prime implicants problem.

There have been some attempts earlier to handle theorem proving, logic programming and ATMS [Rothberg 89] by parallel computers. But so far, no attempt has been made

to device parallel algorithm to compute prime implicants. The present study is the first of it's kind and can also be extended for parallel RMSs. The earlier work relevant to the present study is briefly reviewed here.

Earlier work

Any theorem proving problem involves the combinatorial exploration of a solution space. One of the theorem proving procedures that appears to be particularly efficient in many cases is the Davis-Putnam procedure (DPP) [Davis 60]. Based on the DPP, Chen and Liu [Chen 87] proposed a parallel approach to decompose a formula. At each iteration, a variable is chosen arbitrarily and the formula is split into two subfonnulae with respect to the variable. Based on this approach, a divide-and-conquer strategy together with a pipeline discipline has been proposed for theorem proving in prepositional logic.

In order to fully minimize the vectorization techniques, Fang and Chen [Fang 92] generalize the rules by considering more than one variable at a time. Then, for efficient vectorization, a vectorized representation of a clause is given. Finally, vectorization techniques are utilized to deal with the generalized rules so that they can be implemented in terms of vector instructions. It is shown that the approach is effective in a sense that most operations involved in the algorithm are simple operations like AND, OR, and MERGE bit vector instructions, which are most efficient on vector computers.

5.5 Different Levels of Parallelism in PIAP

The major goal in characterizing the algorithm is to identify and exploit its inherent parallelism (i.e., potential for concurrency). The levels of resolution at which we can attempt to find this parallelism are coarse-grain, medium-grain and fine-grain. The levels of granularity in PIAP is the subject of discussion in this section.

5.5.1 Coarse-grain parallelism in PIAP

The architecture at this level of parallelism is a general purpose, shared memory parallel processing system. In such a shared memory system, the binary tree discussed in Chapter 4 can be made available to all processors. The tree is partitioned in such a way that each processor does approximately the same amount of work, in order to balance the work load. This task can be accomplished by a judicious choice of the literal r at every step of tree partitioning. However, a perfect balance of work load may not be possible in practice. In the coarse-grain-level of parallelization, PIAP works as follows:

During each iteration k, all prime paths of nodes at level *k* are computed. The *k* ranges from '*depth*'(depth of the tree) to 0. Computation of prime paths of each node at level k of the tree is independent, and is computed by each processor, assuming that there are as many number of processors as there are nodes at level *k*. In such cases, once the computation of one node is over, the processor corresponding to that node is idle. If there are only m processors available and the maximum number of nodes at a level is n, different cases arise; (1). $m \geq n$ and (2). $m < n$.

Case 1. In this case, *n* nodes at the level *k* is given to n processors and the remaining m—n processors are idle. Since the number of processors exceeds the number of nodes at the level, each of the node is given to different processors.

Case 2. In this case, since the maximum number of nodes at the particular level exceeds the number of processors, we find the level which has number of nodes less than or equal to m. At this level, the nodes are given to m processors, and the computation of paths upto this level is performed sequentially.

The entire tree is available to all the processors through the shared-memory. Each of the processors performs its computations and the result is stored in the common memory.

The processors which need the computed data collect the data from the common memory. The *m* processors are assigned to it by a wrap-mapping scheme so that each one has approximately the same amount of work. After the computation of prime paths at each level is complete, the processors must synchronize with each other before they can proceed to the next level. Since *m* is never very large, and since partitioning and synchronization may be expensive, it is proposed to switch to a one processor sequential phase where there are fewer than m nodes.

The parallel algorithm PARPIAP to compute the prime paths for Case 1 ($m \geq n$) can formally be stated as given below. The function nodepath *(r, $L_i(r)$, $R_i(r)$)* in the algorithm PARPIAP computes the prime paths of the node N_i, having label r and $L_i(r)$ and $R_i(r)$ as the left and right child, respectively.

```
Algorithm PARPIAP
(Number of processors is assumed to be m)
  begin { initialization }
    For d = depth to 0 do in parallel
    for each node at level d do in parallel
        assign the node N_i to processor P_i
        for each processor P_i do in parallel
            collect the label r, prime paths of L_i(r) and R_i (r)
            nodepath(r, L_i(r), R_i(r));
            write paths of node N_i, to the common memory
  end
```

For case 2 ($m < n$), *d* ranges from *k* to 0 instead of *depth* to 0 where *k is* the level of the tree where the number of nodes is less than or equal to m.

5.5.2 Medium-grain parallelism **in PIAP**

At the medium-grain parallelism, each of the processors *Pi* handles a set of paths. The function of these processors is to collect the label r of the node and the paths generated by both the child nodes, check for subsumption among the set of paths and concatenate the set of paths with the literal r and with another set of paths. Since the paths are independent, their concatenation with a literal, subsumption check between pair of paths, and the concatenation of paths can be carried out simultaneously by these processors. Hence, each of these processors is visualized as a set of processors *(V_1, V_2 ... V_{m1},)* so as to handle the paths simultaneously. Such a processor is depicted in Figure 5.1.

Concatenation of literal r to a path *p*

The Step 5 of PIAP is to concatenate a literal r to the prime paths in $P[S—S_r, TU \{r\}]$. Let $p_1, p_2, \ldots, p_{m1}$ be the prime paths in $P[S — S_r, T U \{r\}]$. As mentioned above, since these prime paths are independent, the concatenation of these paths with the literal r can be performed independently and simultaneously.

It is assumed that there is a host which distribute to each of these processors V_i a path p, in $P[S — S_r, T \cup \{r\}]$. It is also assumed that the number of processors is large enough to handle all the paths simultaneously with one path residing at one processor. However, necessary modification can be done trivially when the number of processors is less than the number of paths. For concatenation of literal r with the paths *Pi* (i = 1,2,... ,m_1), r is given to all the processors at a time and then each processor after checking fundamentality returns $p_i \uplus \{r\}$ to the host. This is pictorially depicted in Figure 5.1. If each processor takes unit time to perform this, then m_1 concatenations arc performed in unit time since there are m_1 processor. If the number of vector processors is less than the number of paths in $P[S — S_r, T U \{r\}]$, then one set of m_1 paths arc

passed onto the processors first, and then another set of m_1 paths and so on till the concatenation with all paths are over. If there are *n* prime paths in $P[S — S_r, TU \{r\}]$ and m_1 processors, then the concatenation of r with these paths can be performed in $[n/m_1]$ units of time.

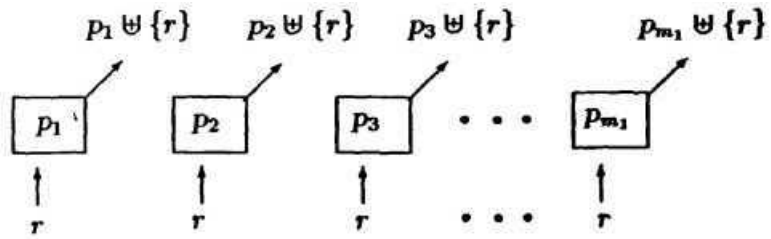


Figure 5.1: Concatenation of r with p_i, i = 1,2, ...,m_1 using mi vector processors.

Concatenation, $p_i \uplus q_j$ of two paths $p_i \in P[S - S_r, T \cup \{r\}]$ and $q_j \in P[S_r, T \cup \{r\}]$ can also be performed in the same manner. Let $q_1, q_2, \ldots, q_{m2}$ be the prime paths in $P[S_r, T \cup \{r\}]$. The host distributes to each of the processors a path pi. The q_i's are send to the processors through the input port in a pipeline fashion by the host.

The paths *q'js* enter with a time lag of one cycle between consecutive paths. A cycle is equal to the time taken by a processor to concatenate two paths p_i and q_j and to output the resulting path $p_i \uplus q_j$ at its output port to the host. This is illustrated in Figure 5.2.

The problem of subsumption is essentially to check whether $p_i \subset q_j$ or, $q_j \subset p_i$. Subsumption is handled in a the same manner as concatenation. Instead of returning $Pi \uplus q_j$ at every instance of time, every processor returns 1, if p_i subsumes q_j, —1, if p_i, is subsumed by q_j, and 0, otherwise. Once when one cycle of computation is over, the host checks the results obtained and acts accordingly. If any output (say, i^{th}) is 1, then the i^{th} path *Pi* subsumes the q_j and hence q_j is deleted from the set of paths by making all the components of q_j a positive value bigger than 2 (say, 3). On the other hand, if any

output (say, j^{ih}) is —1, then the j^{ih} path P_J is subsumed by some path q_i and hence P_j is deleted from the set of paths by making all components of P_j a positive value bigger than 2 (say, 3). A value bigger than 2 is given so as to make sure that this path does not play any role in deciding the subsumption of other paths.

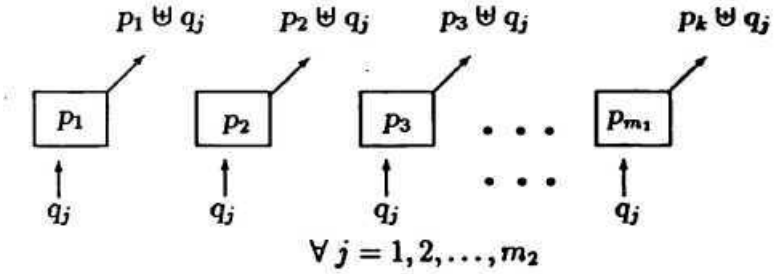


Figure 5.2: Concatenation of p_i) q_j, for = 1, 2, ..., and j = 1, 2, ..., m_2 using m_1 vector processors.

5.5.3 Fine-grain parallelism in PIAP

At the finest level of granularity, the paths are visualized as set of literals and parallel implementations of the concatenation and subsumption operation are performed. In order to accomplish these, let us assume that some ordering of literals exists. Without loss of generality we also assume that all the paths follow this ordering of literals. The paths are represented in the form, of a binary vector with 1 if the literal is present and 0, otherwise. The number of *processing elements* (PEs) in these vector computers is assumed to be equal to the number of literals. Concatenation and subsumption are handled as follows:

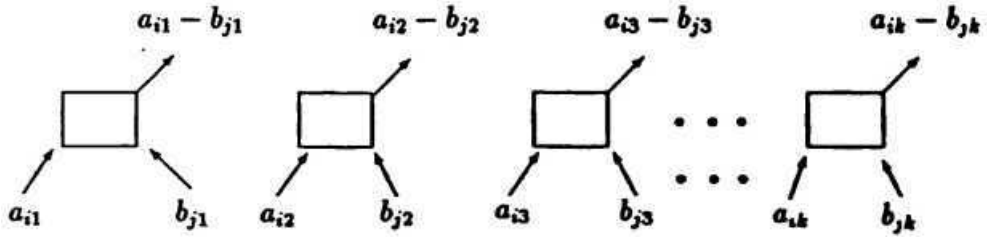
Concatenation in parallel

Let $p_i = \{a_{i1}, a_{i2}, \ldots, a_{ik}\}$ and $q_j = \{b_{j1}\ b_{j2}, \ldots, b_{jk}\}$- In order to perform concatenation of p_i and q_j, each PE computes the maximum of a_{ik} and b_{jk} for all k. The result is sent to

the host through the output port. The host on receiving this, checks for fundamentally and sends it to the common memory if it is fundamental, and discards it, otherwise.

Subsumption in parallel

In order to accomplish subsumption, each processor computes $a_{ik} - b_{jk}$ for all k. Since p_i and q_j are binary vectors, $a_{ik} - b_{jk}$ has the values 0, +1, or -1 for all k. The host of these processors checks $a_{ik} - b_{jk}$ for all k. If $a_{ik} - b_{jk}$ is either 0 or -1 for all k, then p_i subsumes q_j, and if $a_{ik} - b_{jk}$ is either 0 or 1 for all k, then q_j subsumes p_i, and if $a_{ik} - b_{jk}$ is 0 for all k, then both p_i, and q_j, are the same and either of them is considered to subsume the other. If $a_{ik} - b_{jk}$ has the value 0, +1, and -1 for some k, then p_i and q_j are entirely two different clauses which do not subsume each other at all. The host on receiving the values checks values it received. If some q_j subsumes p_i, then p_i is deleted from the set of prime paths $P[S - S_r, T\{r\}]$ and the elements in the next path are distributed to the PEs. If Pi subsumes q_j, then the q_j is given a flag and is not considered further. Based on these discussions, the working of the parallel algorithm and implementation are explained in the following section.



$\forall\, i = 1,2,\ldots,m_1$ and $\forall\, j = 1,2,\ldots,m_2$

Figure 5.3: Computing a_{ik} - b_{jk}, for i = 1,2,..., m_1 and j = 1,2,..., m_a for subsumptions using m_1 vector processors.

5.6 Parallel Algorithm and Implementation

The three levels of parallelism in PIAP are discussed in Section 5.5. As mentioned earlier, the entire binary tree is available to all processors *Pi* through shared memory. At the d^{th} level of the tree, let us assume that there are / nodes and correspondingly, there are / processors. The i^{ih} processor, *Pi* collects the data relevant to the i^{th} node in the d^{th} level, i.e., *Pi* collects the label r_i, left child prime paths $P(L_i\text{-}(r_i))$, and right child prime paths $P(R_i(r_i))$ of the i^{ih} node in the d^{th} level. It is assumed that there are as many number of vector processors as there are paths in $P(L_i(r_i))$. The j^{th} path p_j of $P(L_i(r_i))$ is allocated to the j^{th} vector processor V_j. As already mentioned, it is assumed that the length of the vector representing any path is equal to the number of literals in £ (the set of literals). In V_j, the k^{th} component in p_j is allocated to the k^{th} PE of V_j. The entire architecture is pictorially depicted in Figure 5.4.

The parallel algorithm using this architecture for the example with tree structure given in Figure 4.5 is explained here. At level 2 of the tree there are four nodes, of which the first node is NULL. The remaining three nodes (nodes 5, 6 and 7) are assigned to three different processors P_1 ,P_2 and P_3, respectively. PI collects the label **c** and the prime paths of its left child and right child. Similarly, processors P_2 and P_3 also collects the data required for nodes 6 and 7, respectively. Left child of node 5 is NULL and hence P_1 writes the prime paths for the node 5 as **{c̃}** (the label), and **{ã}, {f,g}** (the right child prime paths of node 5 into the shared memory). The processor P_2 has two paths 6 = (0001000000000) and / = (0000000000100) in $P(L_2(g))$ and paths *d* = (0000001000000) and / = (0000000000010) in $P(R_2(g))$. The two paths in $P(L_2(g))$ are distributed to two processors; b to V_1 and / to V_2. The path *d* in $P(R_2(g))$ is sent to V_1

and V_2 simultaneously. Thus, V_1 has the vectors (0001000000000) and (0000001000000).

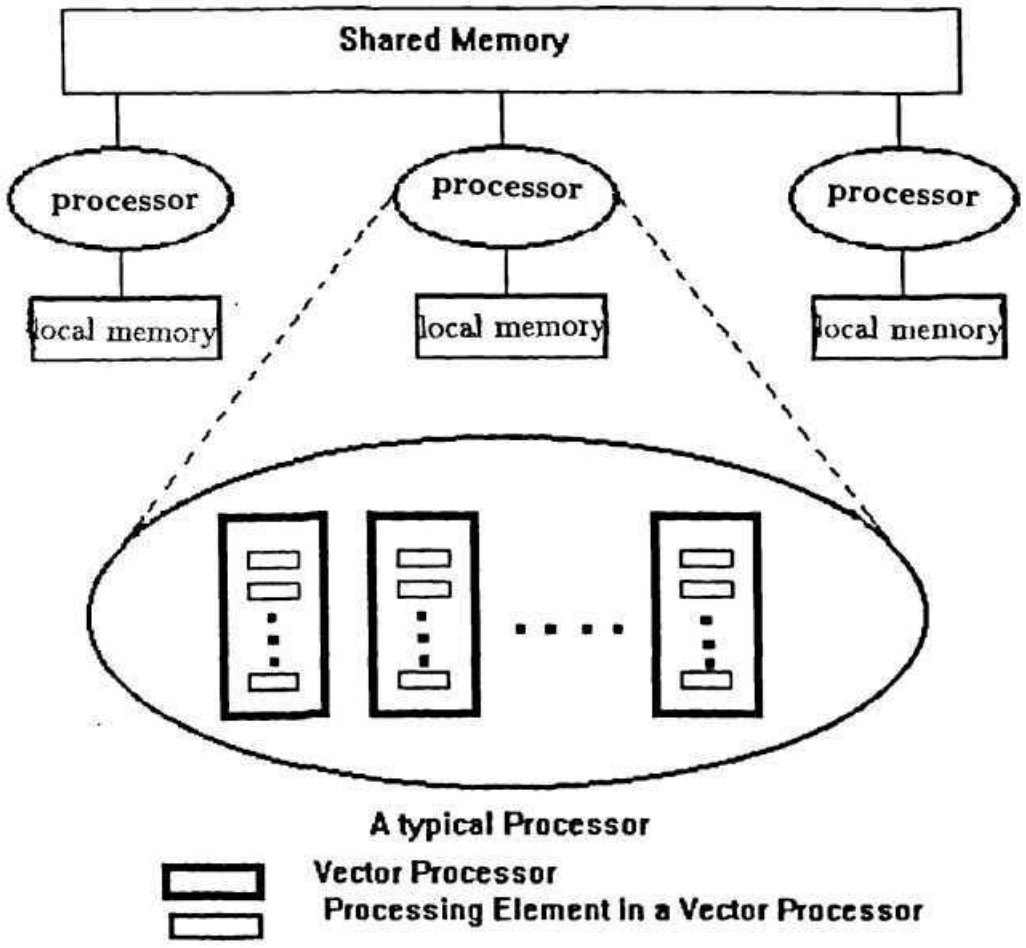


Figure 5.4: The hybrid architecture for PARPIAP

In order to perform the subsumption operation on these data the components of these vectors are given to each of the PEs of V_1. The first *PE* PE_1 has a_{i1} = 0 and b_{j1} = 0; PE_2 has 0 and 0, and so on. Each of the processors compute a_{tk} — b_{jk} and

V_1 obtains the vector (000100-1000000). This vector has 0, 1 and -1, and hence neither of them subsumes nor is subsumed. Similarly the paths in V_2 also neither subsumes nor is subsumed by any path. Hence, the host of the processor P_2 sent g to V_1 and V_2, simultaneously so as to perform the concatenation operation. In order to perform the concatenation, V_1 has vectors (0001000000000) and (0000000000001). The concatenated path obtained ({b,g} = (0001000000001)) is sent to the common memory. Similarly {f,g} is sent to the common memory by V_2.

In order to perform the concatenation operation of the form plsl q where p € $P(L_2(\tilde{g}))$ and q € $P(R_2(\tilde{g})), \{d\}$ and $\{f\}$ are sent to both V_1 and V_2 one after the other and the result obtained is stored in the local memory of P_2. Subsumption operation is performed on the paths $(\{b,d\}$, {d,f} and $\{b,f\})$ thus obtained. For this, the first path (say, $\{b,d\}$) is passed to all the vector processors in P_2. The other paths are sent to these processors one after the other for Subsumption check. The results sent by these vector processors are stored in the local memory of the processor. The unsubsumed paths are sent to the common memory. The processor P_3 also works in a similar fashion for the node 7. Once all the three processors have written the results in the common memory, the algorithm proceeds to the next lower level.

5.7 Conclusion

An attempt is made to explore the inherent parallelism in PIAP, and a parallel algorithm PARPIAP, the parallel version of PIAP, to compute the prime implicants of a formula is designed. The different levels of granularities—coarse-grain, medium-grain and fine-grain are explained. Coarse-grain parallelism is trivial due to the divide-and-conquer paradigm. The prime paths of two subsets of clauses are computed at this level of granularity. The prime paths thus obtained are concatenated followed by subsumption

check. The concatenation of paths as well as the concatenation of paths with a literal with respect to which the set of clauses is partitioned are performed simultaneously. The subsamption check is also performed simultaneously. These computations exhibit medium-grain parallelism, Finally, fine-grain parallelism is achieved when each of the path is considered a vector, and the components of the path obtained as a result of concatenation are computed simultaneously using vector processors.

The hybrid architecture suitable for the parallel computation of prime paths is proposed. The proposed algorithm will help in designing a parallel knowledge compilation technique which in turn can be an efficient tool for RMS.

This algorithm has the synchronization cost of one synchronization per level of the tree. Judicious choice of the literal r would help in obtaining a balanced tree and hence proper load balancing can take place. However, even for a balanced tree, the number of prime paths at each node need not be the same and hence, proper load balancing is difficult to achieve. Moreover, if the number of vector processors in the processors is fixed, there may not be that many prime paths in the left child of the node. Hence there may be idle vector processors.

Chapter 6

APPLICATION TO COMPUTER VISION

6.1 Introduction

Computer vision is an area of AI that deals with the automatic analysis of the visual information. Shape recovery, especially, recovering 3D shape from 2D images is one of the prime areas of interest in computer vision research. Several cues like texture, shading, and stereo help the process of shape recovery. Many techniques have been developed to tackle this problem. *Shape from Silhouettes* is the method of recovering 3D shape using silhouettes as cue. It makes use of multiple 2D outlines as binary images to determine the three-dimensional details of an (otherwise unknown) object.

Indeed, reasoning is the corner stone of intelligence and hence forms the basis of AI applications. Though most of the high level computer vision problems are essentially concerned with reasoning, very few attempts have been made to formally address this problem in computer vision. The traditionally used algorithmic approach, surely pro\ides a fast and reliable method of interpretation and analysis of scenes. But, it is inefficient in handling qualitative information and related inferencing. Moreover, when the result cannot be anticipated a *priori*, the algorithmic approach is not suitable for the purpose. One of the first attempts to use logic programming for image interpretation is the pioneering work of Reiter and Mckworth [Reiter 90] where it is proposed to interpret linedravung as a mapping among the set of logical axioms.

A logical framework for *Shape from Silhouette* problem is proposed here. The problem is rephrased as a problem of computing prime implicants of a propositional formula. It is shown that all possible reconstructions can be described as a consequence of the set of prime implicants of formula. Volume Intersection method [Chien 86] has been the computational paradigm for recovery of shape from multiple silhouettes. This method becomes a particular case of the proposed approach discussed in this chapter. Hence, this framework would prove to be more powerful and elegant than the computational method. Further, this method is suitable for incorporating additional knowledge for better reconstruction.

In the following section, the basic concept of shape from silhouettes is introduced. In Section 6.3, the problem is-formulated as a problem of prime implicants and shown that all possible minimal reconstructions can be obtained by PIAP. In the following section, it is established that the object reconstructed using the conventional technique of Volume Intersection is a particular case of the set of objects that can be generated by the proposed method. Thus, the proposed technique is more general in handling concavities of the object which cannot otherwise be achieved.

6.2 Shape from Silhouettes

A *silhouette* of an object is a binary image corresponding to the occluding contour of the object uniformly filled with black and the background filled with white. The silhouettes of an object are usually different for different viewing directions. On the other hand, two distinct objects may give rise to same silhouettes. A silhouette is obtained by bi-level thresholding of the input grey level image. As a result of this bi-level thresholding, a lot of useful information, which otherwise is important and can aid in proper reconstntctkm, is lost. Inspite of this, silhouettes contain a large amount of information necessary for

a reasonable shape recovery. *Shape from silhouettes* takes the silhouettes as input to reconstruct the three-dimensional object. Volume Intersection [Chien 86] is a method which employs this shape from silhouettes paradigm to recover the shape of a 3D object from multiple silhouettes. The underlying principle is simple as unlike the case of shading, it does not make use of any a *priori* assumptions regarding the surface characteristics, illumination, reflectance, etc. The silhouettes are swept along the viewing directions to form cylinders. These cylinders are intersected to reconstruct the object.

It is assumed that the 3D object is represented as a three-dimensional binary array of voxels of the universe cube and the object reconstruction is concerned with determining the binary values of these voxels based on the binary values of the silhouettes-pixels. It is to be noted that any pixel of a 2D image corresponds to a set of voxels of the universe cube. The problem is to resolve this many-one mapping based on the following principles:

1. *If a pixel is white in a silhouette, then all the voxels that project onto it* are *white.*
2. If *a pixel is black, then at least one of the voxels that project onto it is black.*

6.3 Logical Framework

The problem of recovering 3D shape from silhouettes can be framed as a problem in prepositional logic and the reconstruction process can be essentially the determination of prime implicants. Let FRONT, TOP and SIDE be the binary arrays representing the silhouettes along front, top and side views, respectively. We assume that there is a logical variable associated with every silhouettes-pixel and with the universe cube-voxel. For example, x_{ij} is a literal corresponding to the ij^{th} pixel of FRONT, y_{ij} *i*s a literal corresponding to the ij^{th} pixel of SIDE, z_{ij} is a literal corresponding to the if^{th} pixel of TOP. Similarly, let v_{ijk} be the literal associated with the ijk^{th} voxel of the universe cube.

We say that the associated logical variable is true if the pixel (or voxel) is black, and false, otherwise. We have the following rules according to the foregoing discussion.

1. If x_{ij} is true, then atleast one of v_{ijk} $\forall k$ is true.
2. If y_{ij} is true, then atleast one of v_{ijk} $\forall j$ is true.
3. If z_{ij} is true, then atleast one of v_{ijk} $\forall i$ is true.

The truth or falsity of x_{ij}, y_{ij}, and z_{ij} can be observed from the three silhouettes. Based on these observations, the above set of rules can be simplified to a set of prepositional clauses in CNF. The shape recovery problem is essentially to determine the prime implicants of a set of clauses using the above rules and the observation made about the silhouettes. The reconstruction process aims at determining the truth assignments of the voxels satisfying the axioms of prepositional clauses. The semantics of the axioms is essentially all possible reconstructions.

Example 6.3.1:-

Let us consider a simple case when the universe cube contains eight voxels (say, $v_1, v_2, \ldots, v_8$) and each of the three silhouettes contains four pixels. Hence, based on the above discussion, we have the following rules:

$x_{11} \Rightarrow v_1 \vee v_5$

$x_{12} \Rightarrow v_2 \vee v_6$

$x_{21} \Rightarrow v_3 \vee v_7$

$x_{22} \Rightarrow v_4 \vee v_8$

$y_{11} \Rightarrow v_1 \vee v_2$

$y_{12} \Rightarrow v_5 \vee v_6$

$y_{21} \Rightarrow v_3 \vee v_4$

$y_{22} \Rightarrow v_7 \vee v_8$

$z_{11} \Rightarrow v_1 \vee v_3$

$z_{12} \Rightarrow v_2 \vee v_4$

$z_{21} \Rightarrow v_5 \vee v_7$

$z_{22} \Rightarrow v_6 \vee v_8$

Let us assume that the silhouettes FRONT, SIDE and TOP are completely black. Then we have $(x_{11} \wedge x_{12} \wedge x_{21} \wedge x_{22}) \wedge (y_{11} \wedge y_{12} \wedge y_{21} \wedge y_{22}) \wedge (z_{11} \wedge z_{12} \wedge z_{21} \wedge z_{22})$ representing the observed image. Thus, using the above rules, we have

$$(v_1 \vee v_5) \wedge (v_2 \vee v_6) \wedge (v_3 \vee v_7) \wedge (v_4 \vee v_8) \wedge$$
$$(v_1 \vee v_2) \wedge (v_3 \vee v_4) \wedge (v_5 \vee v_6) \wedge (v_7 \vee v_8) \wedge$$
$$(v_1 \vee v_3) \wedge (v_2 \vee v_4) \wedge (v_5 \vee v_7) \wedge (v_6 \vee v_8)$$

Since the formula is now in CNF, using **PIAP**, one can compute the prime implicants. The prime paths are $\{v_2 v_3 v_5 v_8\}$, $\{v_1 v_4 v_6 v_7\}$, $\{v_1 v_2 v_4 v_5 v_7 v_8\}$, $\{v_1 v_3 v_4 v_5 v_6 v_8\}$, $\{v_1 v_2 v_3 v_6 v_7 v_8\}$, and $\{v_2 v_3 v_4 v_5 v_6 v_7\}$,

Thus it can be seen that each of the prime paths corresponds to a possible minimal object that can be reconstructed for a given set of silhouettes. Thus an object whose silhouettes are totally black along FRONT, SIDE and TOP give rise to 6 minimal objects in the reconstruction. If at this stage, a silhouette of an additional view is obtained, using the incremental method, the reconstruction can be refined.

6.3.1 Volume Intersection Vs. Logical Framework

As discussed in the previous section, the object reconstruction can be achieved using the **PIAP** algorithm. In this section, this method is compared with the well-known method of Volume Intersection.

The reconstruction by Volume Intersection is attained by

1, Sweeping the silhouettes along the viewing direction

2. Intersecting the volume so obtained.

It can be shown that the reconstructed object is a particular case of the proposed logical framework. This particular case is obtained by adding the following default rule:

Unless contrary, assume a voxel to be black.

Appending this default rule to the above rule we get a construction as $\{v_1 \text{ A } v_2 \text{ A } v_3 \text{ A } v_4 \text{ A } v_5 \text{ A } v_6 \text{ A } v_7 \text{ A } v_8\}$ which means that the object is a complete cube of size 2.

The Volume Intersection method observes the following rules:

1. If a pixel is white in a silhouette, then all the voxels that project onto it are white.

   By the logical formulation proposed above, this can be rewritten as $x_{ij} \Rightarrow v_{ijk}\ \forall k$.

2. A voxel is black if all the three silhouettes are black.

   Similarly, it can be written as $x_{ij} \text{ A } y_{ik} \text{ A } z_{kj} \Rightarrow v_{ijk}$.

The above rules can be used to conclude that $v_{ijk} = 1\ \forall i, j, k$. Hence, the Volume Intersection method reconstructs the object in which all the eight voxels are black.

The six minimal objects that can be reconstructed finally give rise to 35 different objects [Nagaraju 91] out of which the maximal object is the one that is obtained by the Volume Intersection technique. The six minimal reconstructions possible are given in Figure 6.1.

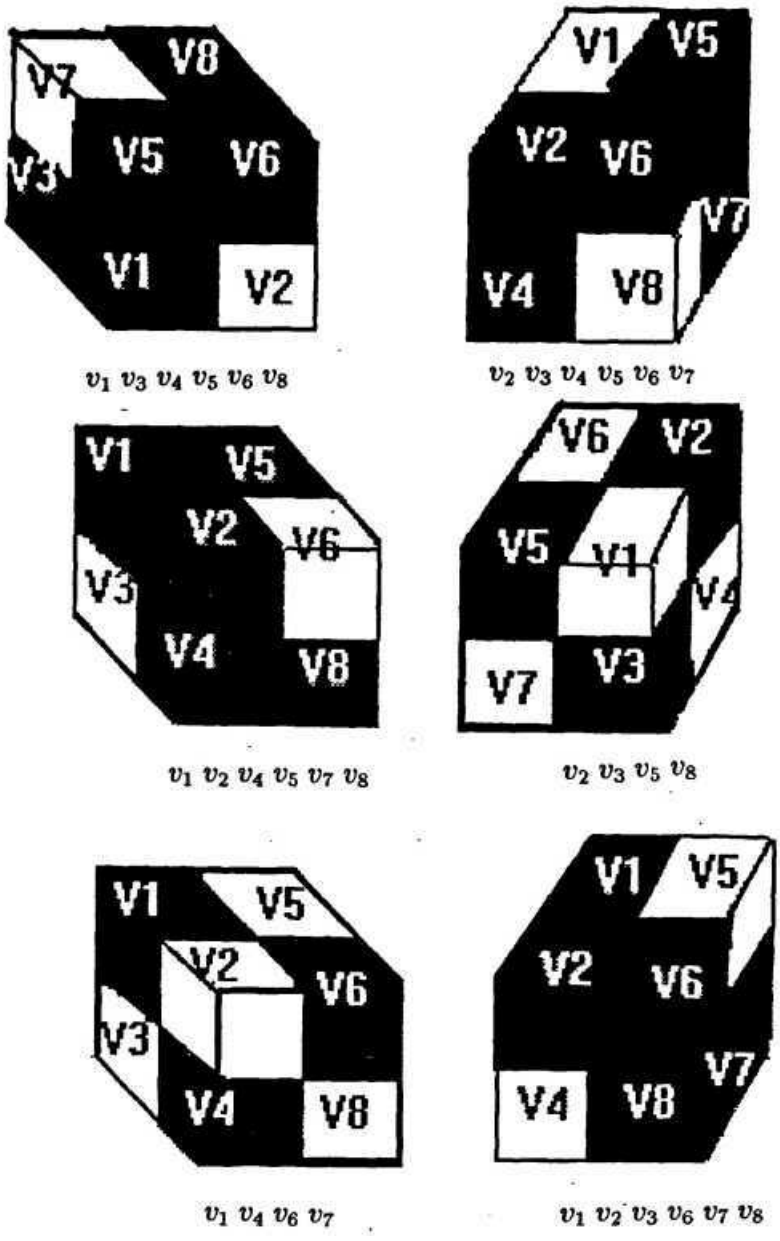


Figure 6.1: The minimal reconstructions of the 3D object

The notation $V_i = \boldsymbol{v_i}$ V i

6.4 Conclusion

An attempt is made to demonstrate the use of prime implicants in the area of Computer Vision. The problem of recovering shape from silhouettes is rephrased as a problem of computing the prime implicants of a formula. The logical framework discussed here provides a method better than the Volume Intersection method to reconstruct 3D objects from the silhouettes. This framework is more general in the sense that it gives all possible minimal reconstructions of 3D objects whereas the Volume Intersection method gives just one reconstruction. Since the prime implicants algorithm **(PIAP)** presented in this dissertation is suitable for incremental computation of prime implicants, the information about additional silhouettes can be incorporated to achieve better reconstruction. Moreover, additional constraints such as convexity, rectilinearity, etc. can be used following a hypothetical reasoning paradigm.

There are many approaches in the areas of Computer Vision, VLSI design, Knowledge-based Systems, Combinatorial Optimization, real-time Systems where the proposed algorithm **PIAP** can be made use of. However, only one application of **PIAP** in the area of Computer Vision is illustrated here.

Chapter 7

CONCLUSIONS

An attempt is made to design a new efficient algorithm to compute the prime implicants of a formula. This dissertation reports theoretical arguments and experimental results of the proposed algorithm. Chapter 3 reports the theoretical basis of the algorithm. The implementation details as well as experimental results are presented in Chapter 4. The advantage of the algorithm being suitable for the incremental computation of prime implicants is explored. The proposed algorithm is suitable for parallel computation of prime implicants. The parallel algorithm is given in Chapter 5. Though the prime implicants are of great importance in AI applications, its application in computer vision, for the problem of shape from silhouettes is presented in Chapter 6. A detailed summary of these achievements is presented in this chapter. The limitations and further research plans are also summarized.

Summary

In this dissertation, a study of the relevance of prime implicants in the context of RMS and hypothetical reasoning is carried out. Hence a brief review on RMS and algorithms to compute the prime implicants of a formula is reported. Realizing the need of prime implicants in different areas of AI, and the drawbacks of the existing methods to compute the prime implicants of a formula, a new algorithm is designed. In Chapter 2, the matrix representation of a formula, and the concept of path in a matrix are presented. The drawbacks of three well-known algorithms are also explained using examples.

In Chapter 3, a special scheme to partition a matrix is proposed, and it is theoretically

established that the prime paths of a formula can be obtained by the concatenation of prime paths of subformulae. The new algorithm based on this concept uses a tree-structure for representation of a formula as well as it's prime paths. Hence the system is called a Tree-structured *Reason Maintenance System.*

The implementation details as well as experimental results are presented in Chapter 4. Experiments were performed to compare the number of subsumptions required, the execution time taken, and the number of paths generated by the algorithms (Socher's algorithm and **PIAP).** Since subsumption check is the crucial part in any prime implicants algorithm, the number of subsumptions required by both the algorithms are computed and compared. It is observed that the proposed algorithm, **PIAP** requires considerably less number of subsumption operations compared to that of Socher's algorithm. Though the execution time is machine dependent, the results obtained substantiate the fact that execution time for **PIAP** is much less compared to that of Socher's algorithm. The number of subsumptions as well as the execution time depends on the paths generated by the algorithms. Hence, the number of paths generated by both the algorithms are also compared. In this comparison, **PIAP** is found to generate fewer paths. Thus, all three aspects of the experimental results affirm that the proposed algorithm is better than the Socher's algorithm.

The implementation is shown to be suitable for a complete RMS^ Hence, certain aspects of RMS are shown in terms of updating the knowledge-base and compiling the knowledge-base in incremental mode. The proposed method is well suited for incremental computation of prime implicants. Apart from being efficient in sequential mode, **PIAP** is parallelizable since the algorithm hinges on divide-and-conquer paradigm. Three different levels of granularities—coarse-grain, medium-grain and fine-grain are explored and a hybrid architecture suitable for the parallel algorithm **PARPIAP** is proposed.

Finally, an application of prime implicants is shown in the context of shape from

silhouettes. The problem of shape from silhouettes is rephrased as a problem of computing prime implicants of a formula obtained from different silhouettes of an object. It is shown that all possible minimal reconstructions of the 3D object can be obtained from the prime implicants of this formula.

Shortcomings

Though the present study proposes a design of tree-structured RMS based on a very efficient algorithm for prime implicants, the implementation of RMS is not done completely as it is not appended to a problem-solver. The computational experiments were restricted to the prime implicants algorithm. However, the same conclusion can be derived for RMS also. The proposal for parallel algorithm for prime implicants is not implemented.

The areas in which RMS has been used include qualitative process theory [de Kleer 80a], circuit analysis [de Kleer 87], analog circuit design [de Kleef 80], temporal reasoning [Williams 86] and vision [Herman 86], deductive databases [Ku 94], to name a few. Different ways in which a RMS can be used in the process of solving constraint satisfaction problem are discussed in [Bodington 88]. The ATMS is used in conjunction with forward checking algorithms to reduce search in a constraint satisfaction problem [Smith 88]. Though there are many areas in which the proposed algorithm can be used, its use in only one particular problem in the area of Computer vision is demonstrated.

Future work

More study on RMS can be carried out and a full-fledged RMS can be designed. Parallel implementation of RMS can also be carried out as a future work. The application of the proposed algorithm in different areas such as qualitative process theory, circuit analysis, analog circuit design, temporal reasoning, vision, deductive databases, constraint satisfaction problem, etc. can be investigated.

Finally, not withstanding the existing shortcomings, the proposed system is an efficient and elegant RMS, based on a new algorithm to compute the prime implicants. The algorithm is established both theoretically and experimentally to be better than the presently known algorithm for the same problem. This can be utilized by the researchers of knowledge based systems in several techniques of nonmonotonic reasoning such as hypothetical reasoning, and ATMS.

BIBLIOGRAPHY

[Bartee 62] Bartee, T. C., Lebow, I. L. & Reed, I. S. (1962). *Theory and design of digital machines.* McGraw-Hill.

[Beek 92] Beek, P.V. (1992). Reasoning about qualitative temporal information. *Artificial Intelligence,* 58, 297-326.

[Biswas 75] Biswas, N. N. (1975). *Introduction to logic and switching theory*. Gordon and Breach Science.

[Bodington 88] Bodington, R., &; Elleby, P. (1988). Justification and assumption-based truth maintenance systems: when and how to use them for constraint satisfaction. In Smith, B. &; Kelleher, G. (Eds), *Reason maintenance systems and their applications* (pp. 114-133). England: Ellis Horwood Limited.

[Bylander 89] Bylander, T., Allenmang, D., Taner, M.C. & Josphson, J.R. (1989). Some results concerning the computational complexity of abduction, *Proceedings of Knowledge Representation and Reasoning,* 44-54.

[Carney 74] Carney, J. D. (1974). *Fundamentals of Logic.* New York: Macmillan.

[Chen 87] Chen, W.T. &; Liu, L.L. (1987). Parallel approach for theorem proving in propositional logic. *Information Sciences,* 41(1), 61-76.

[Chen 91] Chen, W.T. & Fang, M.Y. (1991). An efficient procedure in propositional logic on vector computers. *Parallel Computing,* 983-995.

[Chien 86] Chien, C.H. &; Aggarwal, J.K. (1986). A volume/surface octrees for the representation of three dimensional objects. *CVGIP,* 36, 110-113.

[Copi 73] Copi. I.M. (1983). *Symbolic Logic.* New York: MacmiUan.

[Copi 86] Copi, I.M. (1986). *Introduction to Logic.* New York: Macmillan.

[Davis 60] Davis, M. & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of Associated Comput.* Macfa., 7(3), 201-215.

[de Kleer 80a] de Kleer, J. &: Brown, J. (1980). A qualitative physics based on confluences. *Artificial Intelligence,* 24, 7-83.

[de Kleer 80b] de Kleer, J. & Sussman, G, (1980). Propagation of constraints applied to circuit synthesis. *Circuit Theory and Application,* 8, 127-144.

[de Kleer 86a] de Kleer J. (1986). An assumption-based TMS. *Artificial Intelligence,* 28, 127-162.

[de Kleer 86b] de Kleer J. (1986). Extending the ATMS. *Artificial Intelligence,* 28, 163-196.

[de Kleer 86c] de Kleer J. (1986). Problem solving with the ATMS. *Artificial Intelligence,* 28, 197-224.

[de Kleer 86d] de Kleer, J. &; Williams, B. (1986). Back to backtracing: controlling the ATMS. In *AAAI-86 Proceedings of Fourth National Conference on Artificial Intelligence,* 910-917.

[de Kleer 86e] de Kleer, J. (1986). *Qualitative and quantitative reasoning in classical mechanics.* In Winston, P.H. & Brown, R.H. (Eds), *Artificial Intelligence: an MIT Perspective, Vol. 1,* (pp. 13-29). England: MIT Press.

[de Kleer 86f] de Kleer, J., Doyle, J., Steele, G.L., & Sussman, G. J. (1986). Explicit control of reasoning. In Winston, P.H. &; Brown, R.H. (Eds), *Artificial Intelligence: an MIT Perspective, Vol. 1,* (pp. 95-115). England: MIT Press.

[de Kleer 87] de Kleer, J. & Williams, B. (1987). Diagnostic multiple faults. *Artificial Intelligence,* 32, 97-130.

[de Kleer 93] de Kleer, J. (1993). A view on qualititive physics. *Artificial Intelligence* 59, 105-114.

[de Kleer 93] de Kleer, J. (1993). A perspective on assumption-based truth, maintenance. *Artificial Intelligence,* 59, 63-67.

[de Kleer 94] de Kleer, J.(1994). An improved incremental algorithm for generating prime implicates. *AAAI94*.

[Doyle 86] Doyle, J. (1986). A glimpse of truth maintenance. In Winston, P.H. &; Brown, R.H. (Eds), *Artificial Intelligence: an MIT Perspective, Vol. 1,* (pp. 116-135). England: MIT Press. 116-135.

[Doyle 78] Doyle, J. (1978). Truth maintenace system for problem solving. AI-TR-419, *Artificial Intelligence Laboratory,* Massachusetts Institute of technology.

[Doyle 79] Doyle, J. (1979). A TYuth Maintenance System. *Artificial Intelligence,* **12,** 231-272.

[Drakos 88] Drakos, N. Reaon maintenance in horn-clause logic programs. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance Systems and their applications* (pp. 77-97). England: Ellis Horwood Limited.

[Dressier 88] Dressier, 0. (1988). Extending the basic ATMS. *European Conference on AI* (pp. 535-540). Munich: Pitman Publishing, London.

[Dressier 90] Dressier, 0. &; Farquhar, A. (1990). Putting the problem solver back in the driver's seat: contextual control of the ATMS. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp.1-16). Berlin: Springer-Verlag.

[Dubois 90] Dubios, D., Lang, J., & Prade, H. (1990). A possibilistic assumption-based truth maintenance system with uncertain justifications, and its application to belief revision. In J. P. Martins & M. Reinfrank (Eds), *Truth Maintenance Systems* (pp. 87-106). Berlin: Springer-Verlag.

[Eisinger 91] Eisinger, N., Ohlbach, H. J., & Pracklein, A. (1991). Reduction rules for resolution-based systems. *Artificial Intelligence,* 50, 141-181.

[Eiter 92] Eiter, T. &; Gottlob, G. (1992). On the complexity of prepositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence,* 57, 227-270.

[Fang 92] Fang, M.Y. & Chen, W.T. (1992). Vectorization of a generalized procedure for theorem proving in prepositional logic on vector computers. *IEEE Transactions on Knowledge and Data Engineering,* 4(5), 475-486.

[Flynn66] Flynn, M.J. (1966). Very high-speed computing systems. *Proceedings of IEEE,* 54, 1901-1909.

[Freitag 88] Freitag, H. & Reinfrank, M. (1988). A non-monotonic deduction system based on (A)TMS. In Rudig, B., Kodratoff, Y., Ueberreiter, B. & Wimmer, K.P. (Eds). *European Conference on AI* (pp. 601-606). London: Pitman Publishing.

[Frost 86] Frost, R. (1986). Introduction to Knowledge base Systems. New York: Macmillan Publishing Company.

[Forbus 93] Forbus, K.D. (1993). Qualitative process theory: twelve years after. *Artificial Intelligence,* 59, 115-123.

[Fujiwara 90] Fujiwara, Y. & Honiden, S. (1990). On logical foundations of the ATMS. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 125-135). Berlin: Springer-Verlag.

[Ghosh 95] Ghosh, R.K., Moona, R. & Gupta, P. (1995). *Foundations ofparallel processing.* New Delhi: Narosa Publishing House.

[Giordano 90a] Giordano, L. & Martelli, A. (1990). An abductive characterization of the TMS. In Luigia Carlucci Aiello (Eds), *9th European Conference on AI* (pp. 308- 313). London: Pitman Publishing.

[Giordano 90b] Giordano, L. & Martelli, A. (1990). Truth maintenance systems and belief revision. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 71-86). Berlin: Springer-Verlag.

[Gottlob 93] Gottlob, G. & Fermuller, C.G. (1993). Removing redundancy from a clause. *Artificial Intelligence,* 61, 263-289.

[Hara 92] Hara, H., Yugami, N. & Yoshida, H. (1992). An assumption-based combinatorial optimization system. In Werner, R. (Eds). *Proceedings of 8th Conference on AI for Applications, IEEE Computer Society Press, USA* (pp. 120-126).

[Hayes 75] Hayes, P. (1975). A representation for robot plans, *Proceedings IJCAI-4.*

[Herman 86] Herman, M. & Kanade, T. (1986). Incremental reconstruction of 3D scenes from multiple complex images. *Artificaial Intelligence,* 30, 289-341.

[Hill 81] Hill, F.J. & Peterson, G.R. (1981). *Introduction to Switching theory and logical design.* New York: John Wiley & Sons Inc.

[Horowitz 83] Horowitz, E. & Sahni, S., *Fundamentals of Data Structures.* New Delhi: Globe Offset Printers.

[Hwang 89] Hwang, K. & Briggs, F.A. (1989). *Computer Architecture and Parallel Processing.* Singapore: McGraw-Hill Book Company.

[Inoue 90] Inoue, K. (1990). An abductive procedure for the CMS/ATMS. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 34-53). Berlin: Springer-Verlag.

[Intosh 91] Intosh, D.J. Mac, Contry, Susan, E. & Meyer, Robert A. (1991). Distributed automated reasoning: issues in coordination, cooperation, and performance. *IEEE transactions on systems, man, and cybernetics,* 21(6), 1307-1316.

[Ishizuka 90] Ishizuka, M. & Matsuda (1990). Knowledge acquisition mechanisms for a logical knowledge base including hypothesis. *Knowledge-based Systems,* 3(2), 77-86.

[Ishizuka 91] Ishizuka, M. & Ito, F. (1991). Fast hypothetical reasoning system using inference-path network. *Tools for Artificial Intelligence,* San Jose.

[Ishizuka 93] Ishizuka, M. &; Abe, A. (1993). Fast hypothetical reasoning using analogy on inference-path networks. *Tools for Artificial Intelligence,* Boston.

[Jackson 90] Jackson, P. & Pais, J. (1990). Semantic accounts of belief revision. In J. P. Martins & M. Reinfrank (Eds), *Truth Maintenance Systems* (pp. 155-177). Berlin: Springer-Verlag.

[Jackson 92] Jackson, P. (1992). Computing prime implicants incrementally.In Kapur,D(EDS).Automated Deduction -CADE—llth Conference on Automated Deduction Saratoga Springs,NY,USA. *LNCS.* pp. (253-265)..

[Jamieson Jamieson, L.H. (1988). Characterizing parallel algorithms. In Jamieson, L.H., Gannon, D.B. & Douglass, R.J. *The characteristics of Parallel algorithms* (pp. 65-100). Scientific Computation Series, MIT Press.

[Jones 88] Jones, J. (1988). Modelling unix users with an assumption-based truth maintenance system: some preliminary findings. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance. Systems and their applications* (pp. 134-154). England: Ellis Horwood Limited.

[Junker 90] Junker, U. (1990). Variations on backtracking for TMS. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 17-33). Berlin: Springer-Verlag.

[Kakas 90] Kakas, A.C. & Mancarella, P. (1990). Knowledge assimilation and abduction. In J- P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 54-70). Berlin: Springer-Verlag.

[Kaminski 90] Kamiriski, J.W & Kamiriska, A.W. (1990). Explicit ordering of defaults in ATMS. In Luigia Carlucci Aiello (Eds). *9th European Conference on AI* (pp. 714- 719). London: Pitman Publishing.

[Karnaugh 53] Karnaugh, G.(1953). The map method for synthesis of combinational logic circuits. *AIEE transactions on Communication and Electronics*, pt 1, 72, 593-599.

[Kean 90] Kean A. & Tsiknis G. (1990). An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9, pp. 185-206.

[Kelleher 88] Kelleher, G. & Smith, B. M. (1988). A brief introduction to reason maintenance systems. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance Systems and their applications* (pp. 4-18). England: Ellis Horwood Limited.

[Kohavi 78] Kohavi, Z. (1978). *Switching and finite automata theory*, McGraw-Hill.

[Kondo 91] Kondo, A., Makino, T. & Ishizuka, M. (1991). An efficient hypothetical reasoning system for predicate-logic knowledge-base. *Proceedings of IEEE 3rd International Conferrence on Tools for Artificial Intelligence*, Sanjose

[Ku 94] Ku, C.S., Kim, H.D.,& Henschen, L.J. (1994). An efficient indefiniteness ineference scheme in indefinite deductive databases. IEEE Knowledge and Data Engineering, 6(5), 713-722.

[Kurfe/3 89] Kurfe/3, F. (1989). Logic and Reasoning with Neural Models. Report FKI-99-89, Technische Universitat Munchen.

[Martins 83] Martins, J.P. & Shapiro, S. (1983). Reasoning in multiple belief spaces. Tech. Rept. No. 203 Department of Computer Science, State University of Bufalo, New-York.

[Martins 88] Martins, J. &; Shapiro, S. (1988). A model for belief revision. *Artificial Intelligence*, 35, 25-79.

[McAllester 80] McAllester, D. (1980). An outlook on truth maintenance. *Artificial Intelligence Laboratory*, AIM-551, MIT, Cambridge, MA.

[McCluskey 56] McCluskey Jr. E.L. (1956). Minimization of boolean functions. *Bell Systems Technical Journal*, 35, 1417-1444.

[McDermott83] McDermott, D. (1983). Contexts and data dependencies: a synthesis. *IEEE Transactions in Pattern Analysis and Machine Intelligence.* 5(3), 237-246.

[Moore84] Moore, R.C. (1984). The role of logic in Artificial Intelligence. *Technical Report, Advanced Information Technology: Applications, Achievements, and prospects.* Cambridge, England.

[Nagaraju 91] Nagaraju, Padmaja, Pavankumar, & Pujari, A.K. (1991). A logical framework for recovering shape from silhouettes. In Balaguruswamy, E. & Sushila, B. (Eds), *Information technology the tool for productivity* (pp. 172-177). New Delhi: Tata McGraw-Hill Publishing Company Limited.

[Poole 87] Poole, D., Aleliunas, R. & Goebel, R. (1987). Theorist: A logical resonig system for defaults and diagnosis. In Cercone, N.J. &; McCalla, G. (Eds). *The knowledge frontier: essays in the knowledge representation.* New York: Springer-Verlag.

[Poole 88] Poole, D. (1988). A logical framework for default reasoning. *Artificial Intelligence,* 36,(27-47).

[Popchev 90] Popchev, I. & Zlatareva, N. (1990). A logic for truth maintenance reasoning. In Ph. Jorrand &: V. Sgurev (Eds), *Artificial Intelligence IV- methodology, systems, applications.* North-Holland: Elsevier Science Publishers B. V.

[Provan 88] Provan, G.M. (1988). A complexity analysis of assumption-based truth maintenance.systems. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance Systems and their applications* (pp. 98-113). England: Ellis Horwood Limited.

[Provan Provan, G.M. (1988). Solving diagnostic problems using truth maintenance systems. *European Conference on AI,* (pp. 547-552). London: Pitman Publishing.

[Provan 90] Provan, G.M. (1990). The computational complexity of multiple-context truth maintenance systems. In Aiello, L.C. (Ed). *Proceedings of 9th European Conference on AI,* (pp. 522-528). London: Pitman Publishing.

[Quine 52] Quine, W.V.O. (1952). The problem of simplifying truth functions. *American Mathematical Monthly,* **59,** 521-531.

[Quine 55] Quine, W.V.O. (1955). A way to simplify truth functions. *American Mathematical Monthly,* 62, 627-631.

[Quine 59] Quine, W.V.O. (1959). On cores and prime implicants of truth functions. *American Mathematical Monthly,* 66, 755-760.

[Ramsay 88] Ramsay, A. (1988). *Formal methods in AL* Cmbridge University press, Cambridge.

[Rao 75] Rao, C.R., Mitra, S.K., Matthai, A. & Ramamurthy, K.G. (1975). *Formulae and tables for statistical work.* Calcutta: Statistical Publishing Society.

[Reichgelt 88] Reichgelt, H. (1988). The palce of defaults in a reasoning system. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance Systems and their applications* (pp. 35-55). England: Ellis Horwood Limited.

[Reiter 87] Reiter, R. & de Kleer, J. (1987). Foundations of assumption-based truth maintenance systems: preliminary report. *Proc. AAAI-87,* 183-188.

[Reiter 90] Reiter, R. & Mackworth, A.K. (1990). A logical framework for depiction and image interpretation. *Artificial Intelligence,* **41,** 125-155.

[Rothberg 89] Rothberg, E. & Gupta, A. (1989). Experiences implementing a parallel ATMS on a shared-memory multiprocessor. In *Proceedings of 11-th IJCAI,* (pp. 199-205).

[Shanahan 88] Shanahan, M. (1988). Incrementality and logic programming. In Smith, B. & Kelleher, G. (Eds), *Reason maintenance systems and their applications* (pp. 21-34). England: Ellis Horwood Limited.

[Slagle 70] Slagle, J.R., Chang, C. L. & Lee, R.C.T. (1970). A new algorithm for generating prime implicants. *IEEE Transactions on Computers,* C-19 (4), 304-310.

[Smith 86] Smith, D. E., Genesereth, M.R. & Ginsbug, M.L. (1986). Controlling recursive inference. *Artificial Intelligence,* **30,** 343-389.

[Smith 88] Smith, B.M. (1988). Forward checking, the ATMS and search reduction. In Smith, B. & Kelleher, G. (Eds), *Reason Maintenance Systems and their applications* (pp. 155-168). England: Ellis Horwood Limited.

[Socher 91] Socher, R. (1991). Optimizing the clausal normal form transformation. *Journal of Automated Reasoning,* 7, 325-336.

[Stallman 77] Stallman, R. & Sussman, G.J. (1977). Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence,* 9, 135-196.

[Tayrac 90] Tayrac, P. (1990). ARC: an extended ATMS based on directed CAT-correct resolution. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 107-124). Berlin: Springer - Verlag.

[Tison 67] Tison, P. (1967). Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Transactions on Electronic Computers,* EC-16 (4), 446-456.

[Tremblay 84] Tremblay, J.P. & Sorenson, P.G. (1984). *An introduction to data structures with applications.* Japan: McGraw-Hill Inc.

[Tsiknis 88] Tsiknis, G. & Kean, A. (1988). Clause maintenance systems (CMS), **TR.** 88-21, Department of Computer Science, University of British Columbia.

[Tsuruta 92] Tsuruta, S. & Ishizuka, M. (1992). Efficient compiling methods of logic knowledge-base for abductive hypothesis synthesis. *Technical report,* ISL-92-1.

[Weiss 94] Weiss, M.A. (1994). An improved algorithm for implication testing involving arithmetic inequalities. *IEEE Transactions on Knowledge And Data Engineering,* 997-1001.

[Weld 90] Weld, D. S., & de Kleer, J. (1990). *Readings in qualitative reasoning about physical systems.* San Mateo: Morgan Kaufmann Publishers Inc.

[Williams 86] Williams, B. (1986). Doing time: putting qualitative reasoning on firmer ground. *Proceedings of AAAI-86,* 105-112.

[Witteveen 90] Witteveen, C. (1990). A skeptical semantics for truth maintenance. In J. P. Martins & M. Reinfrank (Eds), *Truth maintenance systems* (pp. 136-154). Berlin: Springer - Verlag.

[Witteveen 93] Witteveen, C.& Brewka, G. (1993). Skeptical reason maintenance and belief revision. *Artificial Intelligence,* 61, 1-36.