ASSIMILATION IN PLAN RECOGNITION VIA TRUTH MAINTENANCE WITH REDUCED REDUNDANCY

by

Bruce Spencer

TR90-060 December 1990

This is an unaltered version of the author's
Ph.D. (CS) Thesis

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B. E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Assimilation in Plan Recognition via Truth Maintenance with Reduced Redundancy

by

Bruce Spencer

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1990

Abstract

Plan recognition is the recognition of an agent's plan from the agent's actions. Recognizing the agent's plan can help predict the agent's next action; in a natural language setting it can help to form a cooperative response. Most artificial intelligence research into plan recognition relies on a complete set of pre-stored plans, a form of the closed world assumption. Faced with a novel plan, these systems simply fail. Our approach for giving up this assumption entails (1) providing new planning information on demand, and (2) incorporating the new information into the candidates that are proposed as the agent's current plan. We focus on task (2).

Most plan recognition settings require timely responses. So as new information is provided, the candidates should be be repaired rather than recalculated. We found that existing truth maintenance systems, such as de Kleer's ATMS, were unsuitable for candidate repair. They introduce extra search and redundancy to handle the disjunctions that arise in plan recognition. We provide a refinement of linear resolution that reduces redundancy in general. Based on this refinement we provide a new truth maintenance system that does not introduce extra search or redundancy. We then use this truth maintenance system as the basis for a plan recognition system which incorporates novel information through candidate repair.

Acknowledgements

I am grateful to the following people:

to Robin Cohen, my supervisor for her rare gift: the ability to manage graduate students. She gave me a free reign to explore my interests, which were often not her own. She showed patience, but enough push to drive me through the program. I am especially grateful for the moral support, the financial support and the "two hour turnaround" when reading sections of my thesis.

to Fei song and Peter van Beek, for our "Friday Research Group" and to Scott Goodwin, Eric Neufeld, Andre Trudel and Paul van Arragon. Thanks for friendship, support and the all too rare excursions to the Grad Club.

to my office mates Hosam AboElFotoh and Vickie Martin, whose warm friendship made the office a "second home."

to the members of my committee, James Allen, Stan Burris, Paul Larson and Fahiem Bacchus for making the oral defence as close to enjoyable as such an ordeal can be.

to the present and former members of the LPAIG, especially Randy Goebel and David Poole and to John Sellens for an excellent working environment.

to all of the staff, especially Dermot Harriss, Mary Chen, Kim Gingerich, Jane Pullin, Linda Norton, Ursula Theone and Sue Thompson.

to the gang in Ottawa, especially Steve MacKay and Gary Stewart.

to all our friends at MSA, especially Jennie and John Molson.

to Jack Dyment and the late Bonnie Jackson for the BNR Postgraduate Award, and to Ragui Kamel and Bill Older for helping me to get it.

Dedication

To Gina, for your love and your prayers, and for all of the times you have had to be both parents. There are easier times ahead, but the best time is any time we spend together. I love you.

To Andrea and Michelle for your *joie de vivre*, humor, trust, honesty, innocence and unconditional love; for teaching me the most important things I have learned.

To my parents, for bringing me up in an understanding and loving home, and for continuing to help in so many ways. You've shown me how parenting ought to be done.

To Don, Sheila and Susan, and your families. We have always helped each other, and I am sure you do not realize how much you have done for me.

To my parents-in-law, for coming to help us on four occasions with a newborn or a crisis, and for welcoming me into your family.

To God, for the many times there was only one set of footprints in the sand.

Contents

List of Figures

Part I

Assimilation in Plan Recognition

This thesis is divided into two parts to discuss the two areas we have been investigating, in plan recognition and automated theorem proving. But the two halves are related since we solve a specific problem in plan recognition with a specific result in automated theorem proving.

In the first half, we discuss the general area of plan recognition, what it is, why it is worth doing, and review some of the methods that have appeared in the literature. Then we discuss a problem that has largely been ignored in the literature, recognizing a novel plan. Our approach to the problem is to break it into two smaller problems, discovering the information necessary to recognize the novel plan, and assimilating it.

We focus on the assimilation problem. We have chosen to implement our assimilation solution within Kautz's plan recognition paradigm [16]. We have used ideas from de Kleer's assumption-based truth maintenance system (ATMS) [6] to perform the assimilation.

Chapter 1

Introduction

1.1 Plan Recognition : What it is

Plan recognition is the recognition of an agent's plan by observing some of the agent's actions. Knowing the agent's plan tells us what the agent intends to do and how he intends to do it.

In the general plan recognition setting (see Figure 1.1) an agent is pursuing a plan and so takes some actions. Observations of these actions are given to a plan recognition system, which has access to a plan library, a set of plans that an agent in this particular domain might be pursuing. The plan recognizer's job is to extract some candidate plans from the library, plans that include at least the observed actions. The candidates are only tentatively proposed; new observations or additions to the plan library might defeat their candidacy. Thus plan recognition is an instance of non-monotonic reasoning. Reasoning is non-monotonic if a conclusion which may be drawn from body of knowledge may no longer be drawn from a larger body of knowledge.

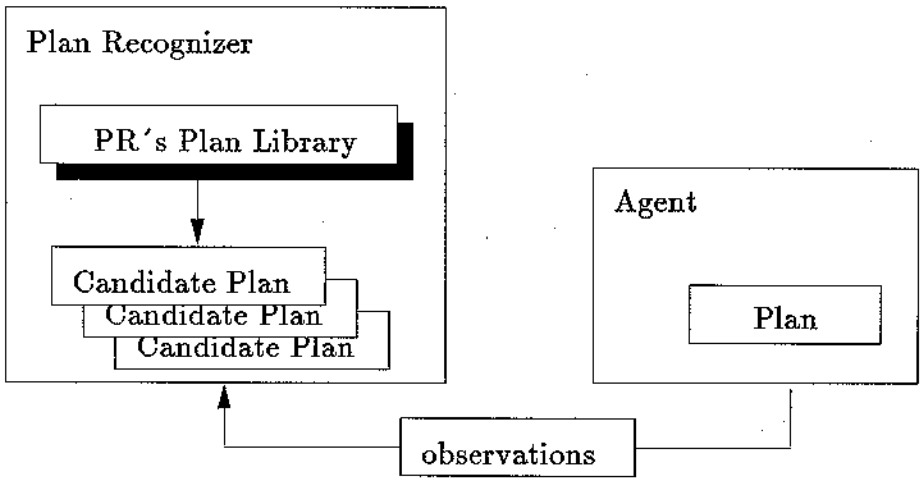


Figure 1.1: A General Plan Recognition Setting

1.2 Why do plan recognition?

Allen and Perrault [1] applied plan recognition to analyse utterances in natural language. According to Austin[2] and Searle[28] every utterance is the result of several actions or *speech acts*, with which the speaker intends to affect the hearer. For example, the speaker may intend to inform the hearer of something or to request him to do something. Allen and Perrault used plan recognition to identify these intentions, in order to understand the utterances.

There are many ways in which a computer system that has a representation of the agent's plan can be more helpful to a user of that system than it could have been without the plan. Here are four ways.

Knowing the user's plan can help the system to decide what information might be useful for the user, even though the user did not explicitly request more information. For example, a person walks up to an information booth at a train station and

says "When does the Montreal train leave?" From this observation, the attendant can reason that finding out when the train leaves is a step in the plan of boarding the train. So the attendant can presume that person wants to board the train. Since boarding the train requires also knowing from where the train will leave, the attendant decides to add that information as well, and responds "3:15 at Gate 7".

Plan recognition can also help recognize the intention behind sentence fragments. If a person asks the attendant "The 5:15 train to Windsor?", the attendant can recognize that the person intends to board this train. Boarding it requires knowing from where and when it will leave. The person already knows when, so the attendant responds "Gate 10".

Having the speaker's plan can help the system to avoid misleading responses. Consider a course advisor that helps students plan their curriculum. Suppose a student that is failing numerical analysis asks "Can I drop numerical analysis?" The student's plan is to avoid receiving a failing grade in numerical analysis. At this particular school, if a student drops a course while failing it, he still receives a failing grade. If the course advisor is not sensitive to the student's plan, he or she (or it) will respond "Yes". But this will mislead the student. It is necessary to have access to the student's plan in order to formulate a more cooperative response, such as "Yes, but you will still fail the course since your mark will be withdrew-while-failing."

A fourth reason for plan recognition is to help predict the agent's next action. This can be used to improve user interfaces. An example of this is the CHemical Engineering Computer-aided-design System (CHECS) of Litman and Goodman[18]. CHECS is a cooperative tool for designing chemical processes. Its user interface is similar to MacDraw©; icons representing equipment, such as pipes, reactors, and heaters, are placed in a schematic diagram of a chemical manufacturing plant. The

plan recognizer attempts to recognize the chemical reaction that the designer has in mind. Having access to the reaction lets the system apply its chemical domain knowledge to suggest more optimal designs, or detect errors. When enough details have been specified to uniquely identify the reaction, the system can predict the actions needed to complete the design and fill out the mundane details itself.

1.3 Plan Recognition Methods

Several different methods for computing candidate plans have been suggested in the literature.

1.3.1 Likely Inference

Allen and Perrault [1] suggested building chains of "likely inference" from the observed action to an objective. One example link is the effect/action rule:

> **if** a speaker wants some predicate P to hold
> and the action ACT achieves P
> **then** the speaker may want ACT to occur.

This system uses heuristics to assign numerical scores to the chains as they are built. It controls the computation by extending the chain with the highest score, a form of best-first search.

1.3.2 Abduction

Abduction was proposed by Charniak and McDermot [5] as a method for plan recognition. In abduction, hypotheses are proposed to build up explanations. For

example, if we observe that B is true, and we know that A implies B, we might propose A as an explanation for B. This is used in plan recognition to explain the agent's actions by making assumptions about what else the agent is doing. For example we might assume that some action has occurred although it was not mentioned. If we can construct a plan from these assumed actions and the observed actions, then the plan is the abductive explanation for the actions.

1.3.3 Kautz's Plan Recognition System

Kautz [16] proposed using circumscription to do plan recognition. He first proposed organizing the plan library into an abstraction/decomposition hierarchy of events. This hierarchy is a collection of fixed-format axioms in first order logic that encode the abstraction and decomposition relations between events. He then generalized previous work in plan recognition by showing how to reason about events in this first order logic. Kautz provides new operators for drawing logical inferences that are particular to plan recognition. These new operators correspond to applications of circumscription[20, 21]. One operator, c-entailment, sanctions conclusions based on the belief that everything about the planning domain is known to the recognizer. By encoding the assumptions that allow plan recognition to proceed, Kautz makes clear certain assumptions which were often left unstated in previous work. For instance, in the work of Allen and Perrault, rules have the form "if one observes act A then it may be that it is part of act B". But reasoning from A to B is not a sound deduction. Kautz would instead reason from A to the disjunction of all of the events B of which A is a part, and base this deduction on the assumption that all such B's are known.

The new generalizations that result from using Kautz's theory include: the

ability to deal with an observation described as a disjunction of event types, the ability to respond with a disjunction of possible plans rather than committing prematurely to one particular plan, and the ability to recognize situations where more than one plan is being pursued, to achieve more than one objective.

Kautz's work is also noteworthy because the agent's actions can be described with the time intervals over which they have occurred, and the plans in the library include temporal constraints that must be met for the plan to be recognized.

1.4 The Novel Plan Problem

In this work we ask the question: what if the agent's plan is not in the plan recognizer's library? This is an important question since it may be difficult to preconceive all of the possible plans in some large domains. Also, an agent may be pursuing an innovative plan, so it will not be in any plan library. For example, in the chemical engineering domain if every plan were already known, then there would be no reason to have a CAD tool for designing new ones.

Most plan recognition systems will simply fail if no candidate plan can be found that contains the observed actions. However, to produce robust systems plan recognition must be allowed to continue, rather than fail.

Our approach to this problem is to divide it into two smaller problems, discovering the new information and assimilating it. Discovering the new planning information that is required to recognize the novel plan is a form of the learning problem. For our purposes, we will assume that there is a source of new information, an abstract, helpful third party that we call the oracle. If the plan recognition setting allows a dialog between the agent and the plan recognizer, then the agent

himself might act as the oracle. Other possibilities are an automated learning system, such as an explanation-based learner [10], or a human expert.

1.5 Harry Grosz Example

Recognition of novel plans occurs naturally in dialogs between people. Harry Grosz is a financial expert who offers advice to callers during a radio phone-in program. This transcript excerpt was provided by the Department of Computer Science in the University of Pennsylvania.

Joe: Harry?

Harry: Yeah.

Joe: This is Joe.

Harry: Welcome Joe.

Joe: I got a simple problem for you on the uh, tax fifty problem. On schedule A, deductions.

Harry: Right.

Joe: My wife and I are both over 65. What do I put on line 40?

Harry: Oh, God. I don't know what line 40 says. What does line 40 say?

Harry has recognized Joe's plan to fill out Schedule A to calculate deductions. A step in that plan is to fill in line 40. Harry's plan library is incomplete because he does not know what line 40 says. So there is a detail to Joe's plan that is novel to Harry. In the continuation of the dialog, Joe reads the text on line 40 to Harry and then Harry is able to give Joe advice.

1.6 Plan Recognition with an Oracle

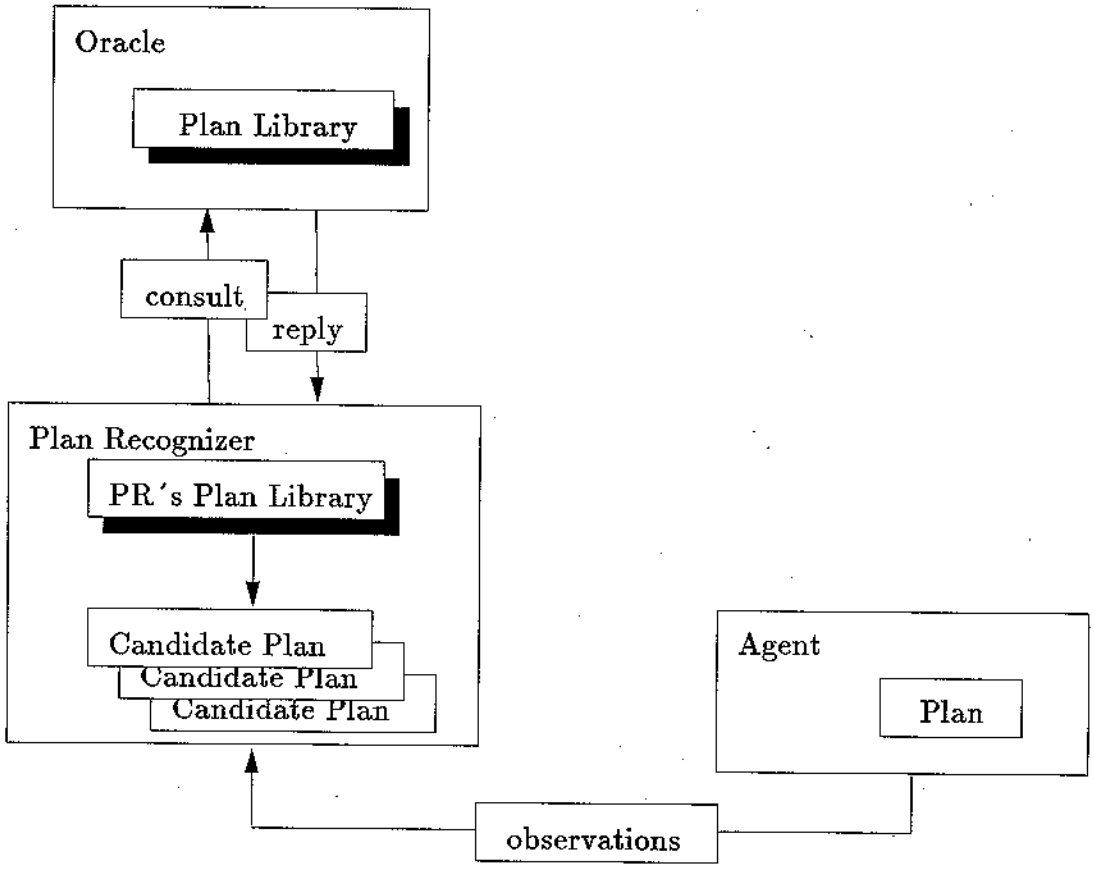


Figure 1.2: PRO : Plan Recognition with an Oracle

Consult \ Reply	gives complete knowledge	gives partial knowledge	may give lies
"I give up."	omniscient		
"Tell me more about X."			
"Is it true that P?"			random bit

Figure 1.3: Various Oracles

We propose a general architecture (See Figure 1.2) for recognizing novel plans that relies on the oracle for its new information. A plan recognizer may consult the oracle whenever it has difficulty accounting for the observations. Several issues arise in defining the interface between the plan recognizer and the oracle. When does the plan recognizer consult the oracle? In what form are the queries posed? What form do the responses take? Are the responses complete and trustworthy? Deciding these issues in various ways shifts the responsibilities between the oracle and the plan recognizer.

Consider the following extreme cases. (Figure 1.3) Let the plan recognizer consult the oracle with "I give up," whenever it cannot account for the current set of observations. Let the oracle give complete and correct information that applies to the agent's plan. Then this oracle is omniscient in that it knows both the question and the answer.

Alternately, if the plan recognizer is able to ask specific "yes-no" type questions, and the oracle is allowed to give false information, then a random bit would serve

as the oracle.

Exploring the area between these extremes is an important future activity. In this thesis we assume that the plan recognizer's plan library is correct, but not complete, and that the information from the oracle is correct. We assume that the oracle is consulted with "I give up".

1.7 Assimilation

In this thesis, we focus on the assimilation problem, namely: incorporating new information into the plan library and the candidate plans proposed by the plan recognizer. It is an instance of the truth maintenance problem, referred to by Etherington [13] as the update problem of non-monotonic logics, a problem solved by truth maintenance systems[11, 6].

Two methods of assimilation suggest themselves; we shall refer to them as assimilation by recalculation and assimilation by repair. To illustrate recalculation, imagine Harry Grosz has a tape recorder, and records the part of the dialog where Joe provides information about himself and his problem. After Harry receives the information about line 40, he adds it to his body of financial knowledge. Now, imagine that he forgets everything he knows about Joe's problem and personal situation, rewinds the tape and plays it back. Now he relates Joe's comments to the larger body of financial knowledge, so he does not need to ask about line 40. One criticism of the recalculation approach is that work will be redone unnecessarily.

Assimilation as repair requires the plan library to be organized hierarchically, so that the candidate plans will be built up from subplans. The subplans themselves are non-monotonically proposed. New information may contradict some of the

subplans, and if so, they should be discarded. The subplans that remain and the new information are combined to form the new candidates.

Although the repair method does not do the unnecessary work of the recalculation method, it does require two tasks that are not part of the recalculation method: finding and removing the contradicted subplans and combining the remaining subplans with the new information to compute the new candidate plans. One possible criticism of the repair approach is that the cost of these tasks may overwhelm the benefits.

In this thesis we provide one repair method which achieves assimilation in plan recognition. We also argue for its benefit over the recalculate method.

Chapter 2

Background

This chapter provides a more detailed description of Kautz's plan recognition theory, gives the basic definitions we will need from propositional logic, and introduces de Kleer's truth maintenance system[6].

2.1 Kautz's Plan Recognition Theory

It is customary to describe formal reasoning systems on three levels. The model theory describes how the formal statements relate to the domain (the real world), the proof theory describes what is to be computed, and the algorithms tell how to compute it. Kautz's plan recognition system [16] is described on these levels. In later chapters we will adopt Kautz's model theory and proof theory, and extend the algorithms with assimilation capabilities provided by a truth maintenance system. This constitutes providing Kautz's style of plan recognition with truth maintenance.

2.1.1 The Representation Language and its Semantics

Kautz uses first order predicate logic with equality to represent the planning domain. Statements in the logic are interpreted with respect to the domain in the usual manner, through models. A model provides the following: for each term in the logic it provides an individual in the domain, for each function it provides a mapping from tuples of individuals to individuals, and for each predicate it provides a set of tuples of individuals. Statements containing quantification and logical connectives are interpreted in the usual way.

Events in the domain are considered individuals, and so they are represented as terms in the logic. Event types are unary predicates that further describe the event. An event may be of several types. For instance, the same event may be described as "reading a book" or as "avoiding housework", depending on your point of view. In the logic, this event is represented by a term, e.g. the constant C, and described further as $ReadBook(C)$ and $AvoidHousework(C)$.

2.1.2 Representing the Plan Library

In this theory the planning domain is described by axioms in the logic. The plan library is described by two particular types of axioms: abstraction axioms that express the "is-a" relation between events, and the decomposition axioms, that express the "has-part" relation. The set of abstraction and decomposition axioms make up a hierarchy of events. There is a natural graphical representation for this abstraction/decomposition hierarchy. In Figure 2.1, from [16], thick grey arrows represent the abstraction axioms. For example, a CashCheck event is an End event. Thin black arrows represent the decomposition axioms. A RobBank event has two parts: a GetGun event is step s1 and a GoToBank event is step s2.

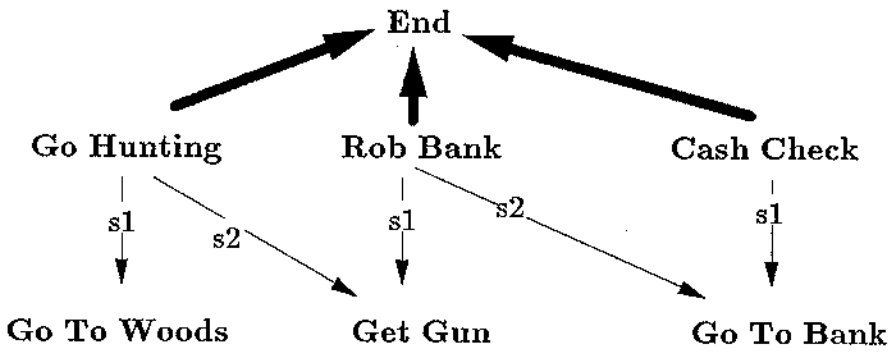


Figure 2.1: A simple example hierarchy

2.1.3 Recognizing a Plan

Although the abstraction and decomposition axioms state true things, they do not state in a general way what events should be inferred from a given observed event, and what candidate plans should be proposed. Suppose that we observe the agent getting a gun. Then the event C is described to the plan recognizer as GetGun(C). One model of the abstraction and decomposition axioms (where we describe a model by listing the positive literals assigned true by it) is {GetGun(C)}. In other words, it is not possible to conclude anything further. The problem is: if we conclude only what is true in all models, there are too many models of the observations and the axioms to conclude anything useful.

One way to remove these unwanted models is to apply McCarthy's circumscription [20, 21] operation on the axioms. Kautz provides a procedure to effectively compute circumscription by constructing new axioms. These axioms are based on the assumption that the abstraction and decomposition hierarchy is complete, so they strengthen the hierarchy. When we add these new axioms the number of models is reduced; the models that remain are called covering models. Plan recognition

is based on concluding what is true in all of these covering models, instead of all of the models. The statements that hold in all covering models are said to be *c-entailed.*

The new axioms express the assumption that hierarchy is complete. There are three sets of new axioms. EXA contains the exhaustiveness axioms which imply that every event is of some basic event type, where a basic type is one that does not abstract any other type. In other words, we can partition the set of events into a number of basic types. DJA contains the disjointedness axioms which imply that every event is of at most one basic type. The effect of both EXA and DJA is that we can partition the set of events into *disjoint* basic types. EXA and DJA deal only with the abstraction hierarchy. Given the hierarchy in the previous example, every event is classified as exactly one of GoHunting, RobBank, CashCheck, GoToWoods, GetGun or GoToBank.

CUA contains the component use axioms, which state that every event is either an End event, or a component in some End event; thus there are no useless events. There is an axiom in CUA for each event type that can be used as a component in some other event. The CUA axiom lists the possible uses for that event. In the previous example, a event of type GetGun is either a step s1 of a RobBank event, or step s2 of a GoHunting event.

The overall effect of these three sets of new axioms is to allow a reasoner to trace the possible uses of an observed event to find what part or parts it may play in an End event. Formally, Kautz's plan recognition consists of showing for every covering model of the observations and the axioms that some End event must be occurring.

Plan recognition algorithms can proceed by building a structure that contains first the observed event, then the events where it may be used, and so on through

the events where they in turn may be used, until each use is eventually shown to be impossible or leads to some End event. The structure so built represents all of the possible plans that contain the End event and the observed event. These are the candidate plans.

2.1.4 Failure to Recognize a Plan

Suppose that not every possible use for the observed event can be shown to be an End event or a component, either directly or indirectly, of some End event. Then there is a model of the axioms and the observation that is not a model of the End event. In this case plan recognition fails.

One cause of failure is that the hierarchy is actually incomplete, although it was assumed to be complete. For example, if a use of the observation could not be related to the End event and new information becomes available that now allows it to be related, then by adding this new information we would like to make plan recognition succeed.

2.1.5 Relating Different Observations

Plan recognition systems typically operate by accepting a number of observations and proposing the plan(s) of the agent.

The C-entailment operator cannot combine information from different observations. Kautz defines another operator, *mc-entailment* which sanctions the inference of a statment that is true in all covering models that contain the smallest possible number of distinct End events.

Just as c-entailment was related to circumscription, mc-entailment can be described as an application of circumscription. The proof theory of mc-entailment corresponds to accepting the first consistent member in a series of assumptions. The first assumption in the series states that all observed events correspond to a single End event. If that leads to an inconsistency the second assumption is applied which states that there are two End events, and so forth. When a consistent assumption is found, say the nth in this series, then we may conclude from the set of observations that n End events are occurring, so n different plans are being pursued. There may be a large number of ways of grouping the observations into n sets. In the worst case the number of ways is exponential in the number of observations.

Because of this combinatorial grouping problem, and because of intuitions about the way people solve incremental recognition problems, Kautz provides another operator for combining observations when more than one End event must be proposed. Incremental mc-entailment, or *imc-entailment* applies when a series of observations is made and after each observation the non-deductive beliefs are adopted as full beliefs. If a subsequent observation cannot be grouped with a set of old observations that together relate to one plan, then this last observation will be considered part of a new plan, and the old grouping of the previous observations will persist.

2.2 Propositional Logic

In this section we briefly present the terminology of propositional theorem proving. For a complete presentation we refer to reader to [3], and [19].

> Symbolic logic considers languages whose essential purpose is to symbolize reasoning encountered not only in mathematics but also in

daily life. [3]

The simplest logic, propositional logic, consists of an alphabet of symbols called *atomic formulas* or *atoms*. They are used to denote true or false propositions, declarative sentences from mathematics or daily life. Atoms are combined with *logical connectives* to build *well-formed formulas*. For instance, if g is an atom then $\neg g$ is a well formed formula. This example $\neg g$ is commonly called the complement of g. Likewise g is the complement of $\neg g$. Both an atom and its complement are known as *literals*. One common restriction of formulas is *conjunctive normal form*, or CNF. A *clause* is a disjunction of literals, e.g. $g_1 \vee \neg g_2 \vee g_3$. A formula in CNF is a conjunction of clauses, e.g. $(g_1 \vee \neg g_2 \vee g_3) \wedge (\neg h_1 \vee \neg h_2) \wedge (k_1)$. We will often use a set of clauses to represent a conjunctive normal formula, e.g. $\{g_1 \vee \neg g_2 \vee g_3, \neg h_1 \vee \neg h_2, k_1\}$. A Horn clause is a clause with at most one positive literal. The second and third clauses in our example set are Horn clauses, $\neg h_1 \vee \neg h_2$, and k_1. The first clause $g_1 \vee \neg g_2 \vee g_3$ is a non-Horn clause.

A clause may also be written with an $\leftarrow$, where all of the positive literals are placed on the left of the arrow and are called consequents, and all of the negative literals are complemented (i.e. stripped of their $\neg$), placed on the right and called conditions. There is an implicit $\vee$ between consequents and an implicit $\wedge$ between conditions. For example $g_1 \vee \neg g_2 \vee g_3$ becomes $g_1, g_3 \leftarrow g_2$. (Another convention is to use a $\supset$ symbol in place of the arrow, and switch the left and right hand sides, so that the example becomes $g_2 \supset g_1 \vee g_3$.)

Since atoms denote true or false sentences, each atom may be interpreted as either true or false. Thus a formula containing n distinct atoms has 2^n interpretations. Given an interpretation of the atoms, we can determine the truth value of the formula under this interpretation by evaluating each atom, and by using the

standard truth tables for the connectives. An interpretation of a formula is a *model* of the formula if the formula evaluates to true under the interpretation. A formula is *valid* if and only if its truth value is true under all interpretations. A formula is *consistent* if and only if its truth value is true under at least one interpretation.

In both mathematics and daily life we often have to decide if a statement follows from other statements. The corresponding notion in logic is to decide if a formula is a logical consequence of another formula. G_2 is a *logical consequence* of G_1 if and only if for every interpretation I if the truth value of G_1 under I is true then the truth value of G_2 under I is true. We write $G_1 \models G_2$. The operation of deciding if G_2 is a logical consequence of G_1 consists of building a formal proof of G_2 from G_1. If this can be done we say that G_2 is provable from G_1 and write $G_1 \vdash G_2$.

2.3 Truth Maintenance Systems

Truth maintenance systems were introduced by Doyle [11], in order to reason about what remains believed ("true") as underlying beliefs change. In [6] de Kleer introduced the assumption-based truth maintenance system (ATMS). It has been used with diagnosis systems[9], for qualitative reasoning[14], and to interpret visual images [24]. A similar technology appears in the expert system shell ART©[33].

2.3.1 Proof Theory

The input to the ATMS is a set J of propositional axioms, expressed as clauses. Some of the literals in the clauses are designated as assumptions, which means they may be assumed true unless shown false.

For each literal the ATMS computes all of the minimal, consistent sets of assumptions such that each, together with the axioms, implies the literal. In symbols, for every literal P which is an atom or a negation of an atom in J, it computes all sets E of assumptions such that

$$E \cup J \vdash P, \text{and } E \cup J \text{ is consistent.}$$

In ATMS terminology, E is called an environment for P. (In Theorist [23] terminology, E is called an explanation.)

2.3.2 Operation

Most ATMS applications need the environments for only some of the literals. So why does the ATMS compute them for all literals? Because for most applications the environments for one literal depend on the environments for a significant number of the rest of the literals. And by computing and storing the environments for the given literals, it can amortize the cost of computing them for a new literal. (Most of the work will already have been done.)

Contrast this with the operation of most reasoning systems. They accept a description of the situation but do no work until they are given a problem to solve, a form of "lazy" evaluation. The ATMS is an "eager" system that anticipates all of the (single literal) queries from a set of axioms and precomputes the answers to each.

2.3.3 ATMS Algorithms

The following illustrates the ATMS algorithms from a high level perspective. For a detailed discussion of the algorithms, we refer the reader to [8], or to Chapter 5

of this work.

Consider the following example. The initial set of axioms is $\{Xis1 \leftarrow A, Yis2 \leftarrow B\}$ where A and B represent ATMS assumptions, and $Xis1$ and $Yis2$ are propositions.

The ATMS generates a directed graph from the clauses. Each node in the graph contains a literal. The directed arcs represent the condition / conclusion relationships between the literals. Each node in the graph also stores the set of environments for its literal. This set of environments is called a label.

In our example, the initial label set is

$$Xis1 \ : \ A$$
$$Yis2 \ : \ B$$

which essentially means that $Xis1$ depends on A, and $Yis2$ depends on B.

When new information is given to the ATMS, in the form of a new set of conditions for some conclusion, that information is added to the graph by adding new arcs from the conditions to the conclusion. The environments of the conditions are propagated along these arcs to form new environments for the conclusion. Propagation continues to the consequences of the conclusion, and so forth through their consequences until all the effects of the new information are computed.

Returning to our example, suppose we know that $Z = X + Y$. Then we might add a new proposition, $Zis3$. Adding $Zis3 \leftarrow Xis1 \wedge Yis2$ to the set of axioms causes the labels for $Xis1$ and $Yis2$ to be propagated to $Zis3$, generating a new label for $Zis3$.

$$Zis3 \ : \ A \wedge B$$

In order to keep the labels consistent, the ATMS maintains a database of minimal inconsistent environments, called *nogoods*. Two routines are used to ensure that every environment in a label is consistent. The first checks that a new environment is consistent before being added to a label, by checking that it is does not contain any nogoods. Whenever a new nogood environment is discovered, the second routine scans all of the existing labels to remove any environments that are supersets of this new nogood.

Continuing our example, if we learn that Y is not 2, we may add $\neg Yis2$ to the ATMS. This causes the ATMS to find that B is contradicted, so B is added to the set of nogoods. Any environment that is a superset of B is now inconsistent. Removing these inconsistent environments yields

$$\begin{aligned} Xis1 &: A \\ Yis2 &: \\ Zis3 &: \end{aligned}$$

There is no longer any reason to believe that Z is 3.

Besides maintaining consistency, the ATMS also maintains label minimality by removing subsumed environments from the label. Since all of the environments are readily available in the label, the ATMS can compare all pairs of environments to detect when one is subsumed by another. An environment is subsumed by another when it is a superset of the other.

2.3.4 ATMS Restrictions

The basic ATMS deals with propositional Horn clauses. These are clauses with at most one positive literal. With Horn clauses, then, it is impossible to reason by

cases through a disjunction of two positive literals, or to conclude such a disjunction. It is because the language is limited to Horn clauses and the reasoning is simplified that the ATMS has been implemented very efficiently.

Attempts to extend the ATMS to general clauses by employing hyperresolution have encountered efficiency problems [7, 8]. We return to the ATMS for general clauses in our discussion in Chapter 5.

Chapter 3

How to do Assimilation

In Kautz's style of plan recognition, the first phase is to add axioms to the hierarchy that express the assumption that the knowledge in the hierarchy is complete. Corresponding to this, the first phase of plan recognition *with assimilation* is to compute the new axioms that express the assumption that the library consisting of the new knowledge from the oracle together with the old hierarchy is complete. We call this step *library assimilation* since it deals with closing the new plan library.

In the other phases of plan recognition, observations are given and the candidate plans are computed. Initially consider the case of recognizing a plan from a single observation. (This special case will be generalized to deal with multiple observations in Section 5.5.4.) In Kautz's proof theory this corresponds to showing that the observation c-entails the fact that there exists an End event. Recall that this is performed by showing that the observation, along with the hierarchy and the closure axioms entails the existence of an End event. The demonstration is performed by building a proof, and the structure of the proof is taken to represent the candidate plans. In this phase of plan recognition *with assimilation*, the task is to show

that the observation, the hierarchy with both the old and the new information and the closure axioms from this larger hierarchy entail the existence of an End event. Since the new proof encodes the new candidate plans we call this step *candidate assimilation.*

It is necessary to present some of Kautz's plan recognition theory in more detail than it appeared in Section 2.1. In Section 3.1 we shall look at the axioms in the abstraction/decomposition hierarchy, and the axioms that express the completeness assumptions.

Sections 3.2 and 3.3 contain our proposals for library assimilation and candidate assimilation, respectively. Each section includes the proof theory, an example, and a discussion of the algorithms. The full algorithms are presented in Appendix A.

Our solution to candidate assimilation, Section 3.3, involves using a truth maintenance system. To do this we need to recast the algorithms Kautz has provided, and replace a portion of them with a truth maintenance system. In particular, we need to explicitly identify the searching and the reasoning done in Kautz's algorithm, and split them into two separate tasks. A comparison of Kautz's algorithms and our own follows.

3.1 Kautz's Plan Library in detail

3.1.1 Representing the Plan Library

The plan library is represented as an abstraction/decomposition hierarchy of events. There are five basic sets that make up this hierarchy:

- H_E is the set of events types, including the type End.

- H_A is the set of abstraction axioms, which represent the "is-a" relations between two events. The general form for an abstraction axiom is

$$\forall x \ E_1(x) \supset E_2(x)$$

where E_1 and E_2 are two event types. This states that any event of type E_1 is also an event of type E_2. We say that E_2 *directly abstracts* E_1, or equivalently E_1 *directly specializes* E_2. The transitive closure of directly abstracts (directly specializes) is written abstracts* (specializes*).

- H_{EB} is the set of basic event types, event types that do not directly abstract any other types.

- H_D is the set of decomposition axioms. Collectively, they represent the way that each event can be decomposed into a number of steps, which are themselves events. They also describe the constraints under which the event can occur. There are two general forms, one for a describing a step, and the other for describing a constraint.[1] The form for a step in an event is

$$\forall x.E(x) \supset E_1(f_1(x)).$$

where E_1 is the type of the step, and f_1 is a role function (a function from one event to another event). The form for a constraint is

$$\forall x.E(x) \supset \kappa.$$

where κ is a constraint.

- H_G is the set of general axioms required to describe the domain.

[1]Kautz places all of the steps and constraints that pertain to a particular event into a single axiom. It is more convenient for our purposes to keep the axioms in clausal form.

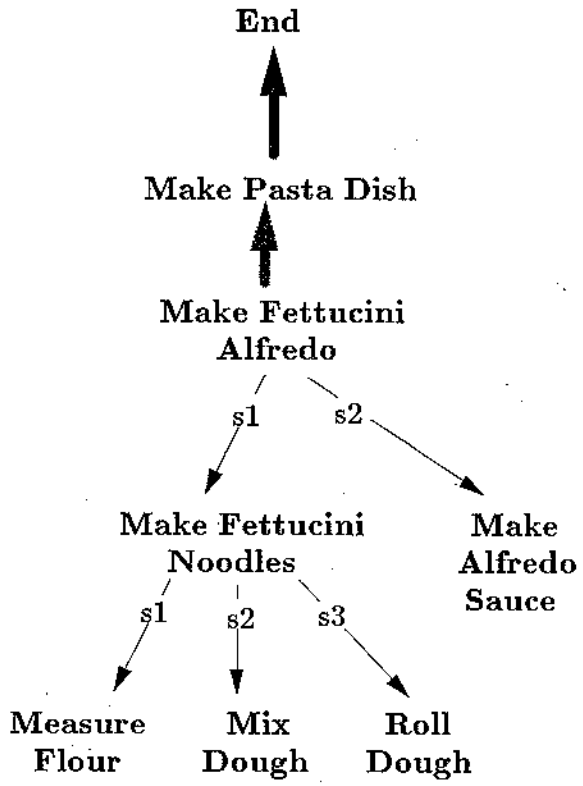


Figure 3.1: An Example Hierarchy

There is a natural graphical representation for the abstraction decomposition hierarchy. In Figure 3.1 thick, grey arrows represent the abstraction axioms, and thin arrows represent the decomposition of an event into steps. A specialized MakePastaDish event is MakeFettuciniAlfredo.

$$\forall x. MakeFettuciniAlfredo(x) \supset MakePastaDish(x)$$

MakeFettuciniAlfredo can be decomposed into two steps. Step s1 is MakeFettuciniNoodles and step s2 is MakeAlfredoSauce.

$$\forall x. MakeFettuciniAlfredo(x) \supset MakeFettuciniNoodles(s1(x))$$
$$\forall x. MakeFettuciniAlfredo(x) \supset MakeAlfredoSauce(s2(x)))$$

Note that the role functions s1 and s2 do not indicate a temporal ordering. An example constraint on the MakeFettuciniNoodles event is that the dough must be mixed before it can be rolled, to allow it to properly rise.

$$\forall x. MakeFettuciniNoodles(x) \supset MixDough(s2(x))$$
$$\forall x. MakeFettuciniNoodles(x) \supset RollDough(s3(x))$$
$$\forall x. MakeFettuciniNoodles(x) \supset Before(time(s2(x)), time(s3(x)))$$

3.1.2 Reasoning

For a single observation, Kautz's plan recognition at the model level corresponds to selecting the set of models that are minimal according to several criteria, including minimal in non-End events. Conclusions that hold in all of these minimal models are *c-entailed* by the observation. The process of model minimization corresponds to applying McCarthy's predicate circumscription. At the proof level, this is accomplished by computing an additional set of axioms that express the supposition that the hierarchy H describes the entire planning domain. These new axioms are combined with H to produce the closure $cl(H)$. Kautz shows that c-entailment from H

corresponds to ordinary entailment from $cl(H)$. So techniques for truth-preserving deduction can be applied to recognize a plan from a single observation.

Closing

Closing the hierarchy means computing a new set of axioms based on the axioms already in the hierarchy. They express the assumption that the hierarchy is complete for the planning domain. Three sets of axioms are computed: EXA, DJA and CUA.

EXA expresses the notion that all ways of specializing an event are known. There is an EXA axiom for every non-basic event type. Suppose that $\{E_1, \ldots, E_n\}$ are all of the event types that directly specialize E_0. Now a new axiom will be in EXA that says if an event of type E_0 is occurring then that event must also be of type E_i for at least one i. In symbols,

$$\forall x.\, E_0(x) \supset (E_1(x) \vee \ldots \vee E_n(x)).$$

From the example in Figure 3.1, since there is only one way to specialize *MakePastaDish* an element of EXA is

$$\forall x. MakePastaDish(x) \supset MakeFettuciniAlfredo(x)$$

To discuss DJA and CUA we first need a definition: E_1 and E_2 are *compatible event types* if there is an event type E_3 such that E_1 and E_2 each abstract* E_3.

The axioms in DJA express the assumption that event types are disjoint, unless stated otherwise. This is used to categorize every observed event into exactly one basic event type. For every pair of event types that do not abstract* some common

event type, there is an axiom in DJA that says two the event types are disjoint. In symbols,

$$\forall x. \neg E_1(x) \vee \neg E_2(x)$$

if E_1 and E_2 are not compatible.

Among the axioms in DJA from the example in Figure 3.1 is

$$\forall x. \neg End(x) \vee \neg MakeFettuciniNoodles(x).$$

The axioms in CUA depend on two ideas: every possible use for an event is in the hierarchy, and every observed event will be used to achieve some End event. There is a CUA axiom for an event type if that event type, or some compatible event type, appears as a step in some decomposition. The CUA axiom lists the possible uses for that event. This allows us to infer up the hierarchy toward the End event. To form the axiom for an event type E, a collection is made of every event E_j such that E (or some type compatible with E) appears with role function f_{ij} in the decomposition of E_j. Then the axiom is built that states: from observing an event of type E we can conclude either that it is an End event or that some E_i has also occurred and that the observed event plays the f_{ji} role in E_i. Let m be the number of times an event compatible with E appears in the decompositions. Then we write

$$\begin{aligned} \forall x.E(x) \supset\ & End(x) \vee \\ & \exists y.(E_1(y) \wedge f_{1i}(y) = x) \vee \\ & \ldots \vee \\ & (E_m(y) \wedge f_{mi}(y) = x) \end{aligned}$$

From the example in Figure 3.1, a member of CUA is

$$\forall x. MakeFettuciniNoodles(x) \supset End(x) \vee$$

$$\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x)$$

Recognizing the Plan

Kautz shows that conclusions which are c-entailed by the abstraction/decomposition hierarchy are entailed by the abstraction/decomposition hierarchy and the axioms EXA, DJA and CUA. Thus reasoning can proceed via ordinary deduction from a single observation to conclude that some End event is occurring. In this way a plan can be recognized, a plan that depends upon the belief that the hierarchy is complete.

3.1.3 The Pasta Example

Given the axioms defined in the previous section, some simple plan recognition problems can be solved. Consider the cooking domain hierarchy described in Figure 3.1. If the agent is observed measuring flour, we can recognize his plan to make fettucini, and then reasoning up the hierarchy through fettucini alfredo, we can conclude he is planning to make a pasta dish. In symbols,

1: Observation	$MeasureFlour(M)$
2: CUA axiom	$\forall x MeasureFlour(x) \supset$
	$End(x) \vee$
	$(\exists y.MakeFettuciniNoodles(y) \wedge s1(y) = x)$
3: Universal Instantiation	$End(M) \vee$
and Modus Ponens(1,2)	$(\exists y.MakeFettuciniNoodles(y) \wedge s1(y) = M)$
4: EXA axiom	$\forall x. \neg End(x) \vee \neg MeasureFlour(x)$

5: Universal Instantiation and Resolution(1,5)	$\neg End(M)$
6: Resolution(3,5)	$(\exists y.MakeFettuciniNoodles(y) \wedge s1(y) = M)$
7: Existential Instantiation (N is a new symbol)	$MakeFettuciniNoodles(N)$
8: EXA axiom	$\forall x. \neg End(x) \vee \neg MakeFettuciniNoodles(x)$
9: Universal Instantiation and Resolution(7,8)	$\neg End(N)$
10:CUA axiom	$\forall x MakeFettuciniNoodles(x) \supset$ $End(x) \vee$ $(\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x)$
11:Universal Instantiation and Modus Ponens(7,10)	$End(N) \vee$ $(\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = N)$
12:Resolution(9,11)	$\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = N)$
13:Extract Conjunction	$\exists y.MakeFettuciniAlfredo(y)$
14:Existential Instantiation (P is a new symbol)	$MakeFettuciniAlfredo(P)$
15:Abstraction axiom	$\forall x.MakeFettuciniAlfredo(x) \supset MakePastaDish(x)$
16:Universal Instantiation and Resolution(14,15)	$MakePastaDish(P)$
17:Abstraction axiom	$\forall x.MakePastaDish(x) \supset End(x)$
18:Universal Instantiation and Modus Ponens(16,17)	$End(P)$

Once we have shown that there exists an End event, in this case End(P), then plan recognition is complete.

3.2 Library Assimilation

In Kautz's style of plan recognition, the first step is to compute $cl(H)$, the closure of the hierarchy H, by computing the sets EXA, DJA and CUA. The corresponding first step in assimilation is to compute $cl(H^+)$ where H^+ is the superset of H that also contains the new plan knowledge given by the oracle.

The axioms EXA, CUA and DJA were described in the previous section. Our algorithms in Appendix A build these sets of axioms directly from the descriptions in a straightforward way.

To do library assimilation as recalculation, we would calculate $cl(H^+)$ directly from H^+, using the library closure algorithms.

Library assimilation as repair requires calculating the changes to $cl(H)$ that result from the additions to H. Since the closure operation is non-monotonic there are in general two sets that need to be computed, the deletions C^- and the additions C^+, such that

$$cl(H) \setminus C^- \cup C^+ = cl(H^+)$$

(where $\setminus$ is the set difference operator.)

3.2.1 An Illustration of Library Assimilation

In our example from Figure 3.1, suppose the oracle gives us new information that MakePastaDish can also be specialized as MakeFettuciniMarinara, and that one step, s_1, in MakeFettuciniMarinara is MakeFettuciniNoodles. (See Figure 3.2.)

To do library assimilation, first it is necessary to remove the axiom in CUA asserting that all of the previously known uses of fettucini noodles, namely in

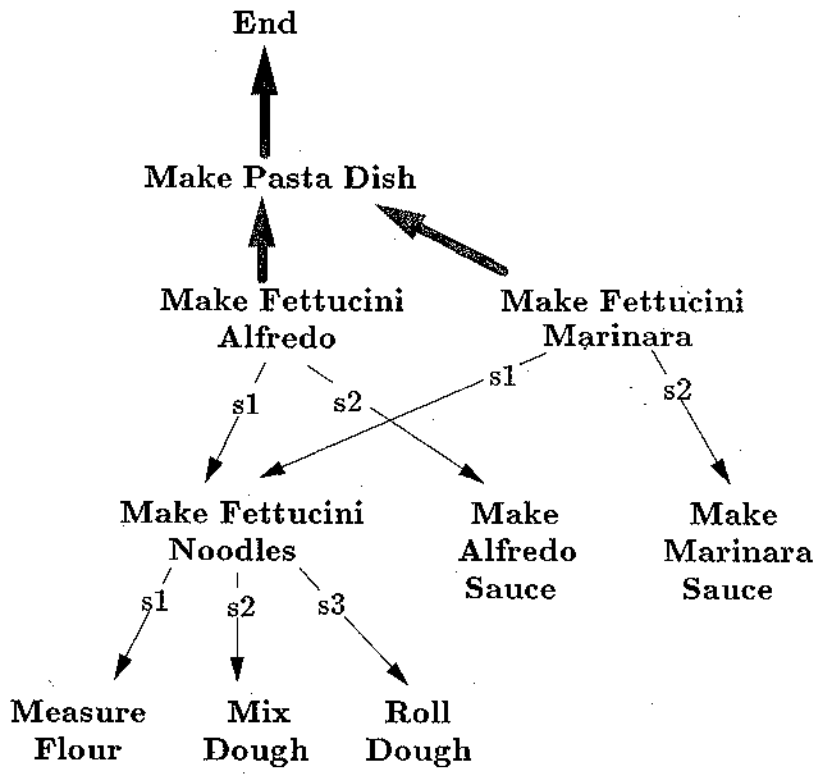


Figure 3.2: The Example Hierarchy with new information

fettucini alfredo, is a complete list. (A *use* of an event type is any event type which appears in its CUA axiom.) This axiom will be replaced by a new axiom that lists the both the new and old uses of fettucini noodles, in fettucini marinara and fettucini alfredo.

It is also necessary to remove the EXA axiom that says that the only specialization of MakePastaDish is MakeFettuciniAlfredo. It must be replaced with one that says that a MakePastaDish event is either a MakeFettuciniAlfredo event or a MakeFettuciniMarinara event.

3.2.2 The Library Assimilation Algorithms

In this section we describe at a high level the library assimilation algorithms to compute C^- and C^+. The full details are presented in Appendix A. (Numerical labels are provided in the discussion and corresponding labels are provided in the algorithms.)

The oracle could provide new information into any of the sets of axioms in the hierarchy: the abstraction axioms H_A, the decomposition axioms H_D, or the general domain axioms H_G. The general axioms do not affect the structure of the library so we will not consider them further. An addition to H_D may either be a new step event added to the decomposition of an event, or it may be a new constraint on the decomposition of an event. A new H_A axiom adds a new abstraction relationship between two event types.

New Constraint

Taking the easiest case first, suppose the oracle adds a new constraint κ on the event type E. Then the clause

$$\forall x.E(x) \supset \kappa$$

should be added to H_D.

This addition only affects the structure of the hierarchy if E is a event type that was not previously in the hierarchy. In that case, it is not compatible with any event type, so for each existing event type E_{old} a new axiom needs to be added to DJA that states that E and E_{old} are disjoint,

$$\forall x.\neg E(x) \vee \neg E_{old}(x).$$

New Step

(1) If the oracle indicates that an event E_{step} is the R step in the event type E, then a new clause should be added to H_D,

$$\forall x.E(x) \supset E_{step}(R(x)).$$

(2) Consider each event E_{com} which is compatible with E_{step}. (E_{step} itself is such an event.) Suppose the CUA axiom for E_{com} is

$$\begin{aligned} \forall x.E_{com}(x) \supset\ & End(x) \vee \\ & \exists y.(E_1(y) \wedge f_{1i}(y) = x) \vee \\ & \ldots \vee \\ & (E_m(y) \wedge f_{mi}(y) = x) \end{aligned}$$

This axiom will need a new disjunct

$$E(y) \wedge R(y) = x.$$

Therefore the existing CUA axiom must be removed, so it is put into CUA^-. A new one with the new disjunct is put into CUA^+.

For example, when the new decomposition

$$\forall x. MakeFettuciniMarinara(x) \supset MakeFettuciniNoodles(s_1(x)).$$

is added to the pasta hierarchy, giving the hierarchy in Figure 3.2, the old CUA axiom for MakeFettuciniNoodles is removed, because there is now a new use for fettucini noodles. We put into CUA^- this axiom:

$$\begin{aligned} \forall x MakeFettuciniNoodles(x) \supset\ & End(x) \vee \\ & \exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x). \end{aligned}$$

The new CUA axiom is computed by adding fettucini marinara to the possible uses of fettucini noodles. We put into CUA^+ this axiom:

$$\begin{aligned} \forall x MakeFettuciniNoodles(x) \supset\ & End(x) \vee \\ & \exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x) \vee \\ & (MakeFettuciniMarinara(y) \wedge s1(y) = x) \end{aligned}$$

(3) As in the case of adding a constraint, if either of the event types E_{step} and E are new to the hierarchy, new axioms need to be added to DJA to assert that the new event is disjoint from all of the previously existing events.

In the example, MakeFettuciniMarinara is new, so we add, for example,

$$\forall x. \neg MakeFettuciniMarinara(x) \vee \neg MeasureFlour(x).$$

New Abstraction

(1) When the oracle indicates a new abstraction, that E_{abs} abstracts E_{spec}, then the clause

$$\forall x.E_{spec}(x) \supset E_{abs}(x)$$

is added to H_A. As many as four other types of changes to the closure axioms could result.

(2) Suppose that the EXA axiom for E_{abs} before the addition was

$$\forall x.E_{abs}(x) \supset E_1(x) \vee \ldots \vee E_n(x).$$

The new abstraction also adds the disjunct $E_{spec}(x)$ to the EXA axiom for E_{abs}, yielding

$$\forall x.E_{abs}(x) \supset E_1(x) \vee \ldots \vee E_n(x) \vee E_{spec}(x).$$

From the example in Figure 3.2, when the new abstraction axiom

$$\forall x.MakeFettuciniMarinara(x) \supset MakePastaDish(x)$$

is added, the EXA axiom for MakePastaDish, which was

$$\forall x.MakePastaDish(x) \supset MakeFettuciniAlfredo(x)$$

is replaced by the axiom

$$\forall x.MakePastaDish(x) \supset \begin{array}{l} MakeFettuciniAlfredo(x) \vee \\ MakeFettuciniMarinara(x) \end{array}$$

(3) Define sets A and B such that A contains event types that abstract* E_{abs} and B contains events types that are compatible with E_{spec} through some lower bound. (See Figure 3.3.) The new abstraction has made every event type in A compatible

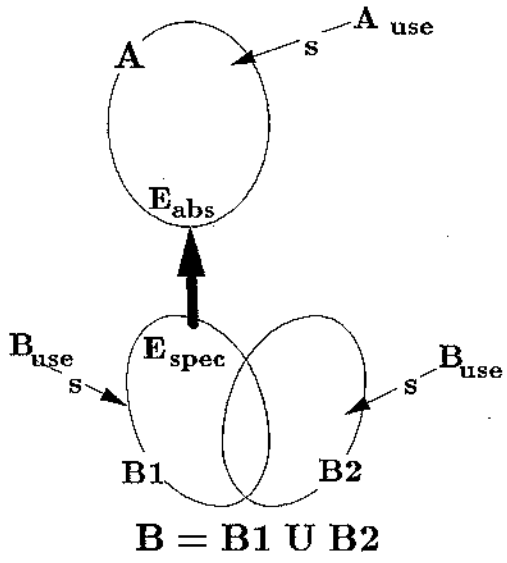


Figure 3.3: Assimilating a new abstraction

with every event type in B. Those pairs A and B that were not compatible before had DJA axioms so stating. These axioms need to be removed.

In the example in Figure 3.2, suppose the new decomposition was added before the new abstraction. Then just before the abstraction is added MakeFettuciniMarinara and MakePastaDish are disjoint so there is a DJA axiom

$$\forall x.\neg MakeFettuciniMarinara(x) \vee \neg MakePastaDish(x).$$

Once the new abstraction axiom is added, making MakeFettuciniMarinara and MakePastaDish compatible, this DJA axiom ought to be removed.

(4) Again consider the sets A and B in Figure 3.3. Since the CUA axioms for event types depends upon the compatible relation, the CUA axiom for each event type in A and in B might be affected. Here is the effect: the list of uses for each event type in A must now include the uses for every event type in B, such as

$B_{use}(x)$. Similarly, the uses for every event type in B should now include the uses for every event type in A, such as $A_{use}(x)$.

In the example in Figure 3.2 the set corresponding to A is {MakePastaDish}. There are no decompositions that include MakePastaDish so this type of change does not arise. Instead consider an example where B_{event} is an event type in B, from Figure 3.3. Let the CUA axiom for B_{event} be

$$\begin{aligned} \forall x.B_{event}(x) \supset\ & End(x) \vee \\ & \exists y.(B_{use1}(y) \wedge g_1(y) = x) \vee \\ & (B_{use2}(y) \wedge g_2(y) = x). \end{aligned}$$

Let A_{event} be an event type in A. Let the CUA axiom for A_{event} be

$$\begin{aligned} \forall x.A_{event}(x) \supset\ & End(x) \vee \\ & \exists y.(A_{use1}(y) \wedge f_1(y) = x) \vee \\ & (A_{use2}(y) \wedge f_2(y) = x). \end{aligned}$$

Then the new CUA axiom for B_{event} is

$$\begin{aligned} \forall x.B_{event}(x) \supset\ & End(x) \vee \\ & \exists y.(B_{use1}(y) \wedge g_1(y) = x) \vee \\ & (B_{use2}(y) \wedge g_2(y) = x) \vee \\ & (A_{use1}(y) \wedge f_1(y) = x) \vee \\ & (A_{use2}(y) \wedge f_2(y) = x). \end{aligned}$$

(5) Finally, as in the addition of a new constraint and a new step, we must consider if a new event type has been added to the hierarchy, and if so add an axiom to DJA for each event that is not compatible with it.

3.3 Computing Candidates and Candidate Assimilation

The second step in plan recognition is computing the candidates. The candidate plan consists of events that are c-entailed by the observation and the hierarchy H. Kautz shows that these events are entailed by the observation and the closure $cl(H)$.

In this thesis we shall directly perform candidate assimilation only into candidate plans from one observation. In section 5.5.4 we discuss how this can still contribute to assimilation into candidate plans from a number of observations. More discussion on this matter may be found in section 6.2.1.

Recall that the plan recognition task is to determine from the observations what the agent's objective is, and how he intends to achieve it. Therefore the candidate plan should at least contain the events in the hierarchy that relate the observation to the End event, since the End event is the "top level" objective in the domain. Our plan recognition algorithm, like Kautz's, builds a proof that the End event is occurring, from the observation and $cl(H)$. It constructs a disjunction of End events, at least one of which must be occurring. The proof encodes all of the candidate plans. In symbols, we compute a proof P which shows that

$$cl(H) \cup \Gamma \models \Omega$$

where Γ is an observation and Ω is a computed disjunction of End events.

Candidate assimilation is the incorporation of new information, from the oracle, with the previously derived candidates that were proposed as the agent's plan. Let H^+ be H with the additional information from the oracle. We want to know for each observation the events that are c-entailed by both the new plan hierarchy,

H^+, and the observation, Γ. We are especially interested in the events that directly relate the observation to the End event, since they tell us what the agent intends to achieve and the way he intends to achieve it. In symbols, we compute a proof P^+ which shows that

$$cl(H^+) \cup \Gamma \models \Omega^+$$

where Ω^+ is a computed disjunction of End events and Ω^+ may be different from Ω.

Candidate assimilation done via recalculation computes P^+ and Ω^+ directly from $cl(H^+)$ and Γ, using the same algorithm that proposed the original candidate.

If library assimilation has been done as repair, the sets of library changes C^-, deletions from $cl(H)$, and C^+, additions to $cl(H)$, are available. Candidate assimilation as repair means using these changes to calculate the changes to P and Ω, and so derive P^+ and Ω^+.

3.3.1 An Illustration of Candidate Assimilation

Recall the pasta example from Section 3.1.3 where MeasureFlour has been observed. Now consider this example with the new information of Figure 3.2 given by the oracle and assimilated into the library. In order to assimilate the new information into candidate plans, a new proof will be needed. Continue the previous proof from line 7.

7:		$MakeFettuciniNoodles(N)$
8:	EXA axiom	$\forall x. \neg End(x) \vee \neg MakeFettuciniNoodles(x)$
9:	Universal Instantiation and Resolution(7,8)	$\neg End(N)$

10:CUA axiom	$\forall x.MakeFettuciniNoodles(x) \supset$
	$End(x) \vee$
	$(\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x) \vee$
	$(MakeFettuciniMarinara(y) \wedge s1(y) = x))$
11:Universal Instantiation	$End(N) \vee$
and Modus Ponens(7,10)	$(\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = N) \vee$
	$(MakeFettuciniMarinara(y) \wedge s1(y) = N))$
12:Resolution(9,11)	$\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = N) \vee$
	$(MakeFettuciniMarinara(y) \wedge s1(y) = N)$
13:Distribution of	$\exists y(MakeFettuciniAlfredo(y) \vee$
$\vee$ over $\wedge$	$MakeFettuciniMarinara(y)$
14:Existential Instantiation	$MakeFettuciniAlfredo(P) \vee$
(P is a new symbol)	$MakeFettuciniMarinara(P)$
15:Abstraction axiom	$\forall x.MakeFettuciniAlfredo(x) \supset MakePastaDish(x)$
16:Universal Instantiation	$MakeFettuciniMarinara(P)$
and Resolution(14,15)	$\vee MakePastaDish(P)$
17:Abstraction axiom	$\forall x.MakeFettuciniMarinara(x) \supset MakePastaDish(x)$
18:Universal Instantiation	$MakePastaDish(P)$
and Resolution(16,17)	
19:Abstraction axiom	$\forall x.MakePastaDish(x) \supset End(x)$
20:Universal Instantiation	$End(P)$
and Modus Ponens(18,19)	

By referencing lines 7, 14, and 18, we can see that the plan includes making fettucini noodles for use in either fettucini marinara or fettucini alfredo, but in either case a pasta dish.

3.3.2 Toward a solution

The proof immediately above, and the proof in Section 3.1.3 are identical in the first 9 steps. After that they follow similar lines of reasoning. Candidate assimilation, roughly, is the process of transforming the previous proof into the proof here. We say "roughly" only because these proofs are not in the format that we will use for our candidate plans. But they still serve to illustrate the problems of candidate assimilation. There are two problems. First, as the underlying assumptions are removed, it is necessary to identify which of the conclusions should be retained and which should be rejected. For example, after the original CUA axiom for $MakeFettuciniNoodles$ is rejected, $MakeFettuciniNoodles(N)$ is still c-entailed from the measure flour observation, but $MakeFettuciniAlfredo(P)$ is not, because we cannot necessarily conclude Alfredo over Marinara. Second, it is necessary to combine the new information with the conclusions that were *not* rejected, in order to derive the new resulting conclusions. The new CUA axiom and the new specialization of $MakePastaDish$ are combined with the old specialization for $MakePastaDish$, and the old conclusion $MakeFettuciniNoodles$ to again derive $MakePastaDish(M)$.

Characteristics of a solution

Based on this discussion, we need a reasoning system with four special characteristics. First, because of the non-monotonic nature of plan recognition it must be able to reason with hypotheses or assumptions. Next, it must retain at least some of its derived conclusions from stage to stage, in particular the derived candidate plans between additions of new planning information to the library. Third, it must be able to relate a conclusion to the assumptions or hypotheses on which it depends,

so that that conclusion can later be rejected if the assumptions that imply it are rejected. Finally, it must be able to combine the old information with the new information to derive the new results.

Create or remodel?

To acquire a new tool which meets a set of criteria, one has the choice of using a tool that is already available, if any is already adequate or can be rendered adequate, or devising a new, special purpose tool.

A special purpose system for candidate assimilation would transform the proof P into P^+ directly. Whatever form the proof takes, whether it is a sequence of formulas, as in the examples so far, or a graph as Kautz computes, the special purpose system would directly remove the rejected conclusions and combine the new information with the retained conclusions.

The ATMS is an existing system which can store and manipulate propositional proofs. It meets the four criteria listed in the previous section for the language of propositional Horn clauses. If the ATMS can be rendered adequate for candidate assimilation, this approach has at least two advantages over constructing a new special purpose tool. First it allows us to break the candidate assimilation problem into two simpler problems: the problem of notifying the truth maintenance system of the changes to the plan library, and the problem of computing the effects of those changes with the truth maintenance system. As a second advantage, we would have a new truth maintenance system which may apply to problems other than candidate assimilation.

Overview of the solution

We will remodel the ATMS and make it adequate for candidate assimilation. Two issues remain. The language of $cl(H)$ differs from the language of the ATMS in two ways. Statements in $cl(H)$ use first order logic but the the ATMS accepts propositional clauses. Not all of the clauses in $cl(H)$ are Horn clauses but the ATMS is effective only for Horn clauses.

We deal with the first problem in the rest of this chapter. The solution is based on recasting Kautz's algorithm for plan recognition from a single observation. The recasting is necessary for separating the searching task from the reasoning task. We use a search algorithm to identify the space of solutions. The space is described by a set of ground instances of the clauses in $cl(H)$. Linear resolution and Skolemization guarantee the instances are ground. In principle, limiting the reasoning to ground clauses does not affect completeness.

Once the search space is identified, the problem of reasoning within that space is handed over to the truth maintenance system that can deal with non-Horn clauses. In Part II we discuss what happens inside in the truth maintenance system. In Section 5.5 we describe how to interpret the results computed by the truth maintenance system, and we provide an overall description of the resulting plan recognition system.

3.3.3 A Recasting of Kautz's Explain Algorithm

Kautz provides an algorithm, called explain, that shows that $\exists x.End(x)$ is c-entailed by an observation. Using the structure of the hierarchy as a guide, it searches from the observation toward the End event and identifies disjunctions of events with their types that are also c-entailed. From these events it constructs a

graph, called an explanation graph or e-graph that explains how the observation relates to the objective. An e-graph can either be interpreted as a proof of $\exists x.End(x)$ or as the disjunction of candidate plans.

Since we have adopted Kautz's proof theory we will also compute c-entailment and we will use the sets EXA, CUA and DJA to compute it. Our task, candidate assimilation, makes it necessary for us to use a different algorithm. In Kautz's algorithm the space searched and the conclusions drawn are in lock-step. As a result it is impossible to withdraw some of the conclusions without also withdrawing a record of having searched the space where they were found. For our task we want to withdraw the parts of the candidates that depend upon assumptions that have been violated (because of the new information from the oracle) but we do not want to again search the space where they were found, since that would amount to assimilation as recalculation.

3.3.4 Plan Recognition with Ground Literals

The ATMS is a propositional truth maintenance system, that is, the literals it stores are treated as propositions. The language of $cl(H)$ is first order logic with equality. Literals from a first order logic can contain variables. When manipulating a proof containing literals with variables, some variables may become bound, for instance if a new resolution is made. It would not be correct to treat these literals as propositions, that is to ignore the bindings, since the variable bindings must be applied to all literals in a clause.

Perhaps surprisingly, we can use a propositional truth maintenance system to store the proofs that arise in plan recognition from $cl(H)$. We employ standard techniques (Skolemization and linear resolution) to ensure that the literals in the

proofs are ground. Ground literals contain no variables, so they may be treated as propositions.

Our truth maintenance system, like the ATMS, builds proofs in a particular linear resolution format.

Definition 3.1 [3, (p. 130)] A *linear deduction* of C_n from C_1, given a set S of clauses is a sequence $C_1, \ldots, C_n$ such that C_{i+1} is a resolvent of C_i and B_i, and B_i is either a clause in S or is some C_j for $j < i$.

We use Skolemization[3, (p.46)][29] and linear deduction to guarantee that a proof that contains only ground literals can be built if and only if any proof exists. Therefore restricting the reasoning to ground literals does not limit the ability to recognize any plan.

To translate a clause to Skolem standard form, replace each existentially quantified variable y in the scope of the universal quantifiers $\forall x_1, \ldots, \forall x_n$ with a new n-place function $f(x_1, \ldots, x_n)$, and delete the quantifier $\exists y$. The only clauses in $cl(H)$ with existential quantifiers are in CUA. Recall that the general form of a clause in CUA is

$$\begin{aligned} \forall x.E(x) \supset\ & End(x) \vee \\ & \exists y.(E_1(y) \wedge f_{1i}(y) = x) \vee \\ & \ldots \vee \\ & (E_m(y) \wedge f_{mi}(y) = x). \end{aligned}$$

Replacing y with the Skolem function $sk_1(x)$, the Skolemized form of this clause is

$$\forall x.E(x) \supset End(x) \vee$$

$$E_1(sk_1(x)) \wedge f_{1i}(sk_1(x)) = x) \vee$$
$$\ldots \vee$$
$$(E_m(sk_1(x)) \wedge f_{mi}(sk_1(x)) = x.$$

Theorem 3 Let $cl(H)$ be the closure of the abstraction/decomposition hierarchy H. Let Γ be a ground observation such that $cl(H) \cup \Gamma \models \exists x\, End(x)$. Suppose that $cl(H)_{sk} \not\models \exists x.End(x)$.[2]

Then for each Skolemized form $cl(H)_{sk}$ of $cl(H)$ there exists a disjunction Ω of literals of the form $End(x)$, and a linear deduction from Γ to Ω given $cl(H)_{sk}$.

A proof of this theorem and the next may be found in Section A.3.

Theorem 4 Let $cl(H)_{sk}$ be the Skolemized closure of the abstraction / decomposition hierarchy H. Let Γ be an ground literal that represents an observation and let Ω be a disjunction of End events. If there is a linear deduction of Ω from Γ given $cl(H)_{sk}$ then every clause in the deduction is ground.

Theorem 3 shows that a proof must exist and by Theorem 4 we know that it will be a ground proof. Therefore this restriction to ground proofs does not affect our ability to recognize any plans.

3.3.5 Candidate and Candidate Assimilation Algorithms

In this section we give a high level description of the candidate assimilation algorithm. See Appendix A for more detailed descriptions. Recall that the ATMS computes for every literal P all environments E (sets of assumptions) such that

$$E \cup J \vdash P, \text{and } E \cup J \text{ is consistent.}$$

[2]The condition that $\exists x.End(x)$ is not already entailed eliminates the case where plan recognition may succeed without having any observations.

For our task, J will be the observation Γ and ground instances of clauses in $cl(H)_{sk}$. Our truth maintenance system will compute a disjunction Ω of *End* events, in addition to a proof for and the environments for that disjunction. (Note that there may be an environment for a disjunction when there is no environment for each of the disjuncts considered separately, because a disjunction represents a weaker conclusion. We deal with disjunctions in a TMS in Chapter 5.)

Handling non-monotonic clauses

Assumptions in our truth maintenance system will arise from the non-monotonic clauses in $cl(H)_{sk}$. The non-monotonic clauses in $cl(H)_{sk}$ are all and only the clauses in $EXA \cup CUA_{sk} \cup DJA$. (Only CUA needs to be Skolemized since only it contains existential variables.) Let C be in a clause in $EXA \cup CUA_{sk} \cup DJA$, and let x be the universally quantified variable in C. Replace C by the clause $A_C(x) \supset C$ where A_C is a new, unique predicate symbol. That is, each of the non-monotonic clauses are given an extra condition. When the clauses are ground, the ground instances of these extra conditions will become assumptions to our truth maintenance system.

These new assumptions allow us to assimilate the deletion of clauses, those clauses that appear in C^- from the library assimilation phase. Let C be a clause in C^-. To remove the effect of C, for each instance $C(N)$ of C that was added to the TMS we now add the clause

$$\neg A_C(N).$$

In the case that C is from CUA_{sk} there will be a Skolem constant in C. After the assumption is denied, that constant can be "freed up", available for use in a

new clause. Thus when a new CUA axiom for this literal arises, it can reuse this constant.

Identifying the Search Space

Although linear resolution tells us that ground literals suffice, it does not tell us *which* ground literals to use. There is an infinite set of ground literals that could be used, since the Herbrand universe is infinite. In this section we give a high level description of two algorithms, Search and AssimSearch, that identify a finite space that contain the candidate plans.

The intuition behind our algorithms is the same intuition that is behind Kautz's explain algorithm. They explore paths in the hierarchy from the observation toward the objective. (This corresponds to tracing the observation to an End event, see Section 2.1.3.)

There are in fact two searching algorithms; they perform similar tasks but Search is applied when the event is first observed, and AssimSearch is applied after additions are made to the plan library. Consider Search first. It takes as input a disjunction of event types that describe an observed event, and identifies instances of the axioms in $cl(H)$.

When Search first encounters a literal, the constraints on it, stored in H_D, are checked to see if the event that it represents can possibly occur. If it cannot then no further searching from this literal occurs, and the reasoner is advised that the event is impossible. Otherwise paths are pursued from this literal according to the arrows in the hierarchy associated with the event type. As each hierarchy arrow is traversed, the reasoner is given the ground instance of the clause that represents both the arrow and the direction of traversal. The following table shows what

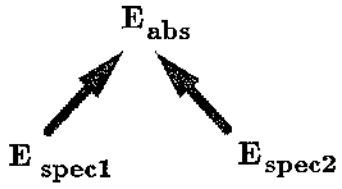


Figure 3.4: An "is-a" plateau

clauses are given for the arrow and the direction.

(The current event is represented by the ground term x_0. Initially the current event is the observed event. As searching progresses, it is a event in which the observed event is a component.)

Arrow	Type	Axiom
$E_a \Rightarrow E_b$	H_A	$E_a(x_0) \supset E_b(x_0)$
$E_a \Leftarrow E_b$	EXA	$E_a(x_0) \supset E_b(x_0) \vee \ldots$
$E_a \leftarrow R - E_b$	CUA	$E_a(x) \supset End(x) \vee E_b(sk_i(x_0) \vee \ldots$

All possible paths are pursued from the observation to the End event, with two exceptions (as in [16]). First, decomposition links are not pursued, since paths from a decomposition lead to deeper levels of detail in the hierarchy. Plan recognition is usually concerned with relating the observation to the objectives, so searches in this direction are usually unnecessary.[3] Second, no path is followed that includes an abstraction link upward immediately followed by another abstraction link downward. (See Figure 3.4.) This pair of links, known as an "is-a" plateau, does not represent any necessary relation between the two specialized events.

[3]However this is one reason why Kautz's system and our system are incomplete.

AssimSearch is applied after the oracle has made additions to the library, and after library assimilation has occurred. It takes as input the changes C^- and C^+ to $cl(H)$. It identifies the instances of assumptions made by Search (and previous invocations of AssimSearch, if any, e.g. if more than one piece of new information comes from the oracle) that have been invalidated. It also identifies the additions to the search space. In fact, it pursues exactly the paths that would have been pursued by Search during the plan recognition phase, if the new axioms had been available then. However, it does *not* pursue paths that were already pursued. (This helps reduce the input to the reasoning system.)

Search and AssimSearch maintain a set Visited of visited literals (as in [16]). When a path leads to a literal that is not already in Visited, the literal is added to Visited, and then paths from that literal are considered. If some path leads to a literal already in Visited, then we know that this path has merged with some previous path. Therefore paths from this literal are already being considered, so there is no need to pursue them again. The graph search from this point can be stopped.

AssimSearch uses the set of visited literals to identify the boundaries of the search space. If a member of Visited is an instance of the condition (left-hand side) of some new axiom, then new paths from the visited literal that start from this path need to be added to the search space.

Equality

In this simple implementation, as in Kautz's, explicit equational reasoning is not required. Therefore we ignore equality literals where they arise.

3.3.6 Revisiting the example

Reconsider the example in Section 3.1.3 where we have observed the agent measuring flour,

$$MeasureFlour(M),$$

and we have the simple hierarchy in Figure 3.1.

Search

Follow the decomposition arrow in a reverse direction to arrive at

$$MakeFettuciniNoodles.$$

This signals Search to generate this Skolemized instance of a clause in CUA_{sk}:

$$CUA_1 \wedge MeasureFlour(M) \supset$$
$$End(M) \vee MakeFettuciniNoodles(sk_1(M))$$

where sk_1 is the Skolem function selected for the existential variable in the clause (discussed in Section 3.3.4), and CUA_1 is the assumption assigned to this non-monotonic clause (discussed in Section 3.3.5).

Pursuing the reversed decomposition link, this time from $MakeFettuciniNoodles$, Search arrives at $MakeFettuciniAlfredo$, and generates the corresponding clause instance to the TMS,

$$CUA_2 \wedge MakeFettuciniNoodles(sk_1(M))$$
$$\supset \; End(sk_2(sk_1(M))) \vee MakeFettuciniAlfredo(sk_2(sk_1(M)))$$

where sk_2 is the Skolem function chosen for the existential variable in this clause, and CUA_2 is the assumption.

Search pursues successive abstraction links to arrive at

$$MakePastaDish(sk_2(sk_1(M)))$$

and then at

$$End(sk_2(sk_1(M)).$$

The clauses generated are

$$MakeFettuciniAlfredo(sk_2(sk_1(M))) \supset MakePastaDish(sk_2(sk_1(M)))$$

and

$$MakePastaDish(sk_2(sk_1(M))) \supset End(sk_2(sk_1(M)))$$

.

The complete set of visited literals is

$\{MeasureFlour(M), End(M), MakeFettuciniNoodles(sk_1(M)), End(sk_1(M)),$
$MakeFettuciniAlfredo(sk_2(sk_1(M))), MakePastaDish(sk_2(sk_1(M))),$
$End(sk_2(sk_1(M)))\}$

AssimSearch

Since the CUA clause for $MakeFettuciniNoodles$ is in C^-, it is to be eliminated. This means every assumption associated with some instance of this clause is no longer consistent. There is only one instance, so we add

$$\neg CUA_2.$$

There is a new axiom in CUA$^+$

$\forall x MakeFettuciniNoodles(x) \supset End(x) \vee$

$$\exists y.(MakeFettuciniAlfredo(y) \wedge s1(y) = x) \vee$$
$$(MakeFettuciniMarinara(y) \wedge s1(y) = x)$$

which, when Skolemized, stripped of the equality literals, given a unique assumption and made into the appropriate instance, becomes

$$CUA_3 \wedge MakeFettuciniNoodles(sk_1(M)) \supset$$
$$End(sk_1(M)) \vee$$
$$MakeFettuciniAlfredo(sk_2(sk_1(M))) \vee$$
$$MakeFettuciniMarinara(sk_2(sk_1(M)))$$

Since $MakeFettuciniNoodles(sk_2(sk_1(M)))$ was visited, new paths are explored that use this axiom. $MakeFettuciniAlfredo(sk_2(sk_1(M)))$ was already visited, so searching in this direction halts. AssimSearch also considers

$$MakeFettuciniMarinara(sk_2(sk_1(M)))$$

so this literal is put into Visited, and paths from it are considered. There are none at this point.

Next the new abstraction

$$\forall x.MakeFettuciniMarinara(x) \supset MakePasta(x).$$

is assimilated. Since $MakeFettuciniMarinara(sk_2(sk_1(M)))$ is now in Visited the instance

$$MakeFettuciniMarinara(sk_2(sk_1(M))) \supset MakePasta(sk_2(sk_1(M)))$$

of the new abstraction axiom is generated.

The literal $MakePasta(sk_2(sk_1(M))$ is in Visited, so searching stops. The final result of searching is a set of ground clauses that will be given to a reasoning system so that a proof of a disjunction of End events can be constructed.

In this case, the disjunction is

$$End(M) \vee End(sk_1(M)) \vee End(sk_2(sk_1(M))),$$

which can be reduced to $End(sk_2(sk_1(M)))$ due to DJA axioms. We discuss the reasoner in Chapter 5.

3.3.7 Properties of the algorithms

Disjunctive Observations

Although Kautz's c-entailment is defined for an observation which is described with a disjunction of possible event types, the graph algorithm requires the event be described by a single type.

Our system does handle disjunctive observations. Our search algorithm identifies the search space including each of the event types in the disjunction. Our truth maintenance system accepts general clauses so disjunctive observations are acceptable. For example in [17] there is an example where MakeFettucini or MakeSpaghetti is observed so we would accept the input $MakeFettucini(C) \vee MakeSpaghetti(C)$.

Soundness and Completeness

Since our algorithms generate instances of the axioms, what follows from the instances must follow from the axioms. If the reasoner is sound (in our case the truth maintenance system, which is sound) then the overall system must be sound.

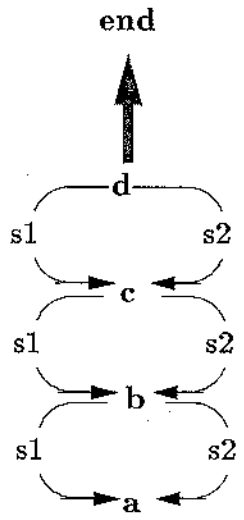


Figure 3.5: Kautz's combinatorially explosive hierarchy

Neither our system nor Kautz's system is complete. Both sacrifice completeness for efficiency, by only admitting a part of the total search space. If we find that some desired inferences are not in the space identified by the search algorithm, then we have the option of extending the search algorithm to enlarge the space. The truth maintenance system is complete, so the desired inferences will be made.

Complexity

Both Kautz's algorithm and our own may search a space that is exponentially large in the size of the abstraction/decomposition hierarchy. However Kautz's example, taken from [16] and reproduced in Figure 3.5, is handled by our system in time which is a polynomial in the number of layers in the hierarchy. In Kautz's system this example requires time which is exponential in the number of layers.

Kautz's explain algorithm merges graphs at abstraction points. Abstraction

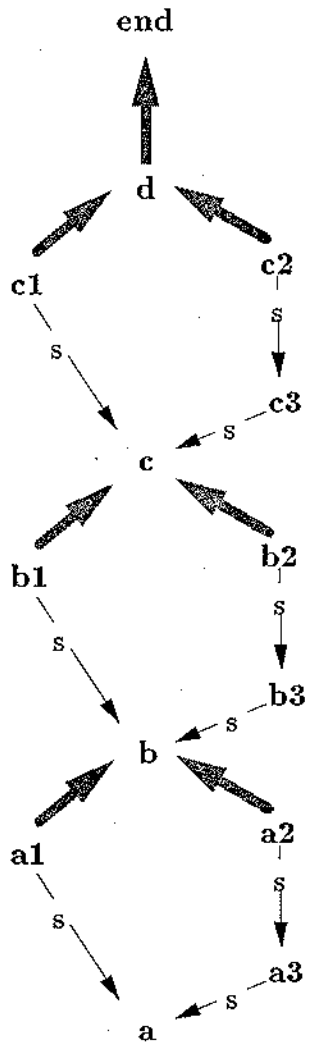


Figure 3.6: Our combinatorially explosive hierarchy

points in the abstraction/decomposition hierarchy are event types that have many specialized types. This means, for example in Figure 3.6 if a path from a1 to b has previously been pursued so that some instance of b has been visited, and now a path from a2 to b is being pursued, the algorithm will not continue to pursue this second path, even though the second instance of b is different from the first. It is "good enough" to know that an event of type b is occurring.

Kautz suggests one way to control the combinatorial explosion: use an algorithm that merges graphs at abstraction points and design the hierarchy so that there are many abstraction points. As a result, Kautz's explain algorithm does not require exponential time to pursue paths through the graph in Figure 3.6. Our algorithm does not merge paths at abstraction points, so it does require exponential time for this example. We have implemented and tested a version of Search and AssimSearch that also merges search paths at abstraction points. We have found that this algorithm requires only polynomial time on these examples. (We have not used this version because of two extra complications that arise as a side effect of separating the searcher from the reasoner, which was required for assimilation. In the version that merges search paths, the searcher must advise the reasoner of the merged paths, and the reasoner must keep track of these merges so that when it reports candidate plans it also shows the merges.)

Part II

Reasoning with non-Horn Clauses

In Part I we deferred the problem of reasoning with non-Horn clauses within a truth maintenance system. The method proposed by de Kleer appears in the extended ATMS[7], and the extension is a routine that reasons by cases with a method based on hyperresolution. Reasoning by cases is required when non-Horn clauses are present. de Kleer notes that the extended ATMS exhibits extreme inefficiency in some cases, and that hyperresolution does not integrate well into the ATMS algorithms[8]. One cause for the inefficiency is that the extended ATMS, unlike the basic ATMS, searches much of its database at many steps to perform hyperresolution. The basic ATMS propagates information from where it is derived to all of the places where it will be needed, so searching is not necessary. But the basic ATMS is not capable of reasoning by cases.

We provide a truth maintenance system that reasons by cases and propagates information from where it is derived to all of the places where it is needed. Our solution is based on linear resolution, rather than hyperresolution. The search at propagation time is eliminated. Our system uses the same algorithm as the basic ATMS, with a single additional propagation step, so it integrates well into the ATMS algorithms. However, our system depends on building a new and/or graph to propagate the results from one case to another when reasoning by cases. This causes a new problem: if there are many instances of reasoning by cases, then a large and/or graph will be needed to represent the different ways of propagating results from one case to another. Only one of these ways is necessary; the rest are redundant.

Peeling the next layer from the onion, we investigate the problem of redundancy in linear resolution. We describe a variant of linear resolution called the foothold proof format. It is a refinement of the well-known the MESON proof format[19], but it admits fewer duplicate proofs than MESON. The duplicates arise when reasoning

by cases.

The foothold format can be used within an implementation of a first order theorem prover. The refinement depends on a simple condition that can be checked quickly, and can detect redundancy before the proof is completely generated. This is important from a practical point of view, since the earlier redundancy is detected, the more unnecessary work can be avoided.

The foothold result can be applied to remove the redundancy that arises when reasoning by cases in our truth maintenance system. Our resulting system exhibits a performance improvement over de Kleer's extended ATMS for non-Horn clauses. We then apply this new truth maintenance system with reduced redundancy to the problem of assimilation in Kautz's style of plan recognition.

Chapter 4

Avoiding Duplicate Proofs

Wos [34] asks

> What strategy can be employed to deter a reasoning program from deducing a clause already retained, or from deducing a clause that is a proper instance of a clause already retained?

Although our procedures in this chapter are computing proofs rather than clauses, the question still stands. It takes time to compute a redundant proof, and more time to decide if the proof is redundant.

We consider the setting of first order logic without equality where we are given a goal literal and a set of clauses and we wish to know which instances of the literal follow from the clauses. To do this we use a proof procedure to compute substitutions for the variables in the literal. Prolog is a special case of this setting. A redundant substitution duplicates or is an instance of another substitution.

In the next chapter we consider the default logic setting described in [6, 23]. We are given a set of clauses that represent the known facts, and a distinguished

set of literals called assumptions that represent the hypotheses we are prepared to believe, in the absence of contrary evidence. An explanation for the goal is a set of these assumptions that (1) is consistent with the clauses and (2) that together with the clauses implies the goal. For example, de Kleer's assumption-based Truth Maintenance system[6] (ATMS) computes propositional explanations. A redundant explanation duplicates or is more specific than some other explanation.

These two settings have one feature in common: the results computed are properties attached to a proof of the goal. Redundancy arises in each setting when two proofs give us the same result, or when one result is more specific than the other. In this chapter we consider the first variety of redundancy, duplicated results.

We introduce the foothold format[30] for linear resolution proofs that admits fewer duplicate proofs than MESON[19], a well-known proof format. The duplication we avoid arises in MESON when reasoning is done by cases. Reasoning by cases is required to handle clauses with more than one positive disjunct, clauses that do not fall into the Horn subset. Thus this type of duplication does not arise in Prolog, nor in the basic, Horn ATMS. It does arise in non-Horn extensions to Prolog [31], and in non-Horn extensions to the ATMS [27, 7], and therefore in our implementation of Kautz's plan recognition system with truth maintenance. It also arises in Theorist[23], a first order default reasoning system.

4.1 Background: The MESON Proof Format

In this section we introduce negated ancestor proof graphs, which is a new definition for Loveland's MESON proof format. The new definition makes it convenient to add a new condition which eliminates certain redundant proof graphs.

4.1.1 Negated Ancestor Proof Graphs

Linear resolution [3, 19] can be used in our first order logic setting to generate the instances for a goal literal by building a proof of the goal from the clauses, and applying the resulting substitutions to the goal. We consider one form of linear resolution, based on Loveland's MESON format, and we introduce a new definition of this format. Given a goal and a set of clauses we show that the goal follows from the clauses if and only if there is a proof graph of the goal that is built from (some of) the clauses.

We begin with two preliminary definitions.

Definition 4.1 If g is a literal then let $\overline{g}$ be the complement of g. That is, the overbar function adds the $\neg$ connective if it is not present in g, and removes it if it is.

Definition 4.2 For a clause $g_1 \vee \ldots \vee g_n$ there are n **contrapositive rules** as follows:

$$\forall i = 1, \ldots, n$$

$$g_i \leftarrow \overline{g}_1 \cdots \overline{g}_{i-1} \ \overline{g}_{i+1} \cdots \overline{g}_n$$

Informally, to construct a negated ancestor proof graph of a goal g, begin by constructing a node that contains g. This node will be the root of the graph. Given a partial negated ancestor proof graph, and a goal node containing the literal h (initially h is g) there are two ways to complete the graph. Either (case a) find a node already in the graph containing $\overline{h}$ which is connected to this node by tree edges, and introduce a back edge from the goal node to this ancestor. Or (case b) find a contrapositive rule with h on the left hand side, make a new node for each

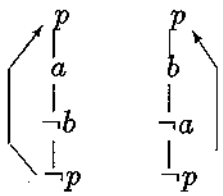


Figure 4.1: Negated Ancestor Proof Graphs

member of the right hand side, and introduce a tree edge from the goal node to each new node. Recursively prove each new node.

For example, consider the clauses

$$p \vee \neg a$$
$$p \vee \neg b$$
$$a \vee b$$

and the goal p. The contrapositive rules are

$$p \leftarrow a \quad \neg a \leftarrow \neg p$$
$$p \leftarrow b \quad \neg b \leftarrow \neg p$$
$$a \leftarrow \neg b \quad b \leftarrow \neg a$$

There are two negated ancestor proof graphs of p, as shown in Figure 4.1. As in the conventional presentation of trees, the root is at the top, child nodes are lower than parent nodes, and tree edges have no arrows. Our back edges point upward.

Now we can formally present negated ancestor proof graphs.

Definition 4.3 Let P be a set of propositional clauses, and g a literal.

4.3.1 A **negated ancestor proof graph** of g is a directed graph (V, E). Each node in V contains a single literal. There is exactly one node G called the root,

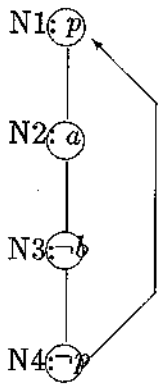
and it contains g. (V, E) is an NA1 proof graph of G and the ancestor path it contains is empty.

4.3.2 An NA1 proof graph of a node G containing the ancestor path A is a directed graph (V, E). $E = T \cup B$, where T is the set of tree edges, B is the set of back edges, T and B are disjoint and (V, T) is a tree. Each node in V contains a single literal. A is a sequence of nodes that represents a path of tree edges, which are the ancestors of G.[1] Let g be the literal in G. One of the two following conditions must be met:

(case a) There exists a node N in A such that N contains the literal $\overline{g}$ and there exists a back edge from G to N in the NA1 proof graph.

(case b) There exists a contrapositive rule $g \leftarrow g_1 \ldots g_n$ from some clause in P and for each $i = 1, \ldots, n$ there exists a node G_i containing g_i, a tree edge (G, G_i) and an NA1 proof graph of G_i containing the ancestor path (A, G).

To illustrate the definition we start with one of the proof graphs in Figure 4.1. We show it again, with labels on the nodes.



[1] G is *not* the root of the NA1 graph; rather, the first member of A is.

We will illustrate the definition by showing how this graph satisfies the conditions. It is a negative ancestor proof graph of p if it is an NA1 proof graph of $N1$ with the empty ancestor path. So we check $N1$ against Definition 4.3.2 with A, the ancestor path, set to (). (Case a) cannot hold since there is no member of A, so we try (case b). Choosing the rule $p \leftarrow a$, we note that there is a node $N2$ containing a, and a tree edge to from $N1$ to $N2$. If we can satisfy ourselves that there in an NA1 proof of $N2$ containing the ancestor path $(N1)$ we will be done. Apply Definition 4.3.2 again. (Case b) applies, with the rule $a \leftarrow \neg b$, and there is a node $N3$ containing $\neg b$ and a tree edge from $N2$ to $N3$. We now must check that there in an NA1 proof of $N3$ with the ancestor path $(N1, N2)$. Again (case b) of the definition applies and, using the clause $\neg b \leftarrow \neg p$ we see that there is a node $N4$, which contains $\neg p$ and a tree edge from $N3$ to $N4$. Finally we ask if there is an NA1 proof graph of $N4$ containing the ancestor path $(N1, N2, N3)$.This time (case a) applies since there the node $N1$ in A contains p which is the complement of $\neg p$. Therefore we are satisfied that there is an NA1 proof graph of $N4$, which allows us to conclude there is an NA1 proof graph of $N3$, and so forth for $N2$ and $N1$. We conclude that this is an negated ancestor proof graph of p.

The part of the MESON format we have extracted preserves the important properties of soundness and completeness.

Theorem 5 Let P be a consistent set of propositional clauses, and g a literal. $P \models g$ if and only if there exists a negative ancestor proof graph of g using clauses in P.

Proofs for the theorems in this chapter are found in Appendix B.1.

4.1.2 First Order Proofs

We will assume that the reader is familiar with first order logic, as presented in, for example [3].

Suppose we are given a set P of clauses and a literal g and we want to find the instances of g that follow from P. Procedures that build first order MESON proofs can also build substitutions for the variables in g. For example if $P = \{g(a)\}$ then the proof of $g(x)$ returns the substitution $x := a$. There may also exist indefinite instances of the goal, as defined by a disjunctive set of substitutions. For example if $P = \{g(a) \vee g(b)\}$ then the proof of $g(x)$ should return $x := a \vee x := b$. Green [15] provides a method for computing disjunctive answers. Stickel [31] describes how disjunctive answers can be computed within the MESON format:

> Indefinite answers can be obtained by solving the query with its negation included among the axioms, and examining the proof to find the query's instantiations.

Completeness of the MESON format extends from the propositional case to the first order case [19]. To see why, recall Herbrand's theorem which states: a set of clauses is unsatisfiable if and only if there is a finite unsatisfiable set of ground instances of these clauses. If $P \models g$ then $P \cup \{\overline{g}\}$ is unsatisfiable, so there is an unsatisfiable set $P' \cup \{\overline{g}_1, \ldots, \overline{g}_n\}$ where P' is made up of ground instances of clauses of P, and each g_i is a ground instance of g. Therefore $P \cup \{\overline{g}_2, \ldots, \overline{g}_n\} \models g_1$. Since ground literals may be treated as propositions, and since the propositional MESON format is complete there is a propositional MESON proof of g_1 from $P' \cup \{\overline{g}_2, \ldots, \overline{g}_n\}$. A lifting argument can now be made to show that from the propositional proof, the appropriate first order proof can be constructed which produces the disjunction of substitutions.

4.2 Foothold Proof Graphs

In this section we present foothold proof graphs, a refinement of negated ancestor proof graphs that does not generate as many redundant proofs, but retains completeness. The basic idea is to break up the symmetry in duplicate negated ancestor proofs. In Figure 4.1, the duplication is due to the symmetry of the two contrapositive rules of $a \vee b$, namely $a \leftarrow \neg b$ and $b \leftarrow \neg a$. In a foothold proof we assign labels from the set {-1,0,+1} to the literals on the right hand side of the contrapositive rules. For this example the symmetry is broken by assigning the label $(+1)$ to $\neg b$ and (-1) to $\neg a$.

To build the labeled contrapositive rules for a general clause, first separate the positive literals from the negative literals. Put all of the literals of each type into an ordered sequence. Any ordering will do. For each p_i from the positive literals, build a contrapositive rule as follows. Put p_i on the left hand side. On the right hand side put the complements of the other literals in the clause. To each positive literal that appears before p_i in the ordered sequence assign the label $(+1)$, and to each literal that appears after p_i assign (-1). To each negative literal assign (0). Also, build a rule for each negative literal n_i as follows. Assign $(+1)$ to each negative literal that appears before n_i, assign (-1) to each negative literal that appears after n_i and assign (0) to each positive literal.

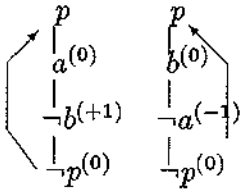
In the example the labeled contrapositive rules are

$$\begin{array}{ll} p \leftarrow a^{(0)} & \neg a \leftarrow \neg p^{(0)} \\ p \leftarrow b^{(0)} & \neg b \leftarrow \neg p^{(0)} \\ a \leftarrow \neg b^{(+1)} & b \leftarrow \neg a^{(-1)} \end{array}$$

This labelling is based on the ordering b, a for the literals in the clause $a \vee b$. Now, build the negated ancestor proof graph using these labeled contrapositives. During

the search for a negated ancestor, sum up the labels of all of the literals encountered. If the sum is positive when we reach the negated ancestor, then accept the proof. Otherwise reject it.

In the example the foothold proofs graphs are



The first of them is accepted and the second is rejected.

The name "foothold" refers to an analogy between reaching the negated ancestor and climbing a tree. If there are more positive labels than negative labels, there is a foothold on which we can climb. If there are more negative labels than positive ones, there is no foothold, so it is impossible to climb.

This simple idea can now be presented more formally.

4.2.1 Propositional Foothold Format

Definition 4.4 Let P be a set of propositional clauses.

4.4.1 The **foothold rule set** of P is the union over clauses C in P of the set of foothold rules of C.

4.4.2 Let C be a propositional clause. Let $(l_1, \ldots l_n)$ be an ordered sequence of the negative literals in C and $(r_1, \ldots r_m)$ be an ordered sequence of the positive

literals. The **foothold rules of** C with respect to these orderings are

$$\forall i = 1, \ldots, n$$
$$l_i \leftarrow \overline{l}_1^{(+1)} \ \ldots \ \overline{l}_{i-1}^{(+1)} \ \ \overline{l}_{i+1}^{(-1)} \ \ldots \ \overline{l}_n^{(-1)} \ \ \overline{r}_1^{(0)} \ \ldots \ \overline{r}_m^{(0)}$$

$$\forall i = 1, \ldots, m$$
$$r_i \leftarrow \overline{r}_1^{(+1)} \ \ldots \ \overline{r}_{i-1}^{(+1)} \ \ \overline{r}_{i+1}^{(-1)} \ \ldots \ \overline{r}_m^{(-1)} \ \ \overline{l}_1^{(0)} \ \ldots \ \overline{l}_n^{(0)}$$

4.4.3 The superscript (s) on each literal on the right hand side of a foothold rule is called the **foothold label.** A foothold label written $(*)$ is one whose value we disregard (a "don't care").

Definition 4.5 Let P be a set of clauses, g a literal, and F_P be a foothold rule set of P.

4.5.1 A **labeled negated ancestor proof graph** containing F_P is a negated ancestor proof graph with an integer label assigned to each node and to each back edge as follows:

(1) to the root, assign an unspecified label, $(*)$.

(2) to each child node G_i, containing g_i assign (l_i) where the parent node contains g, the contrapositive rule chosen for g was $g \leftarrow g_1, \ldots, g_i, \ldots, g_n$, and the corresponding foothold rule is $g \leftarrow g_1^{(l_1)}, \ldots, g_i^{(l_i)}, \ldots, g_n^{(l_n)}$.

(3) to each back edge (N_n, N_1) assign $\Sigma_{i=2}^{n} l_i$ where $N_1, \ldots, N_n$ is the path of tree edges such that N_n and N_1 contain complementary literals, and N_i has the label (l_i).

4.5.2 A **foothold proof graph** is a labeled negated ancestor graph such that every back edge has a positive label.

Theorem 7 Let P be a consistent set of propositional clauses, and g a literal. Let F_P be a foothold rule set of P. $P \models g$ if and only if there exists a foothold proof graph of g from F_P.

4.2.2 First Order Foothold Proofs

Foothold labels can be assigned to first order clauses in exactly the same way that they are assigned to propositional clauses. First order foothold proofs are defined as first order negated ancestor proofs, with the additional restriction that each backedge from a goal to its negated ancestor has a positive foothold sum.

Completeness of first order foothold proofs is also preserved. The argument in Section 4.1.2 for the completeness of first order negated ancestor proofs also applies to first order foothold proofs. The only property of propositional negated ancestor proofs that was used in that proof was propositional completeness, a property shared by propositional foothold proofs. Since foothold proofs are a special case of negated ancestor proofs, the same lifting lemma may be applied to generate first order foothold proofs.

4.3 Evaluation

4.3.1 How easily can foothold proofs be computed?

Like the MESON format, the foothold format leads to proof procedures that are simple to implement. Appendix B.2 contains a Prolog procedure for computing foothold proofs. Any theorem prover that is based on the MESON format, such as PTTP [31] can easily be converted to the foothold format.

Procedures that compute MESON proofs can take advantage of an important pruning rule. If a goal is encountered that is the exact duplicate of a goal in the ancestor sequence, then this branch of the tree can be ignored and no success is reported. There is no reason to pursue a proof of this goal, because this condition arises when the procedure has entered a loop. Procedures that compute foothold proofs can take advantage of the same pruning rule, for the same reason. The Prolog procedure in the appendix implements this pruning rule.

4.3.2 When should the foothold format be used?

The foothold format should be used whenever one is reasoning with clauses that extend beyond the Horn subset and the objective of the search is to find more than one result. If the objective of the search is to find a single solution, or to show that a set of clauses is unsatisfiable, then foothold proof procedures are not necessarily better or worse than MESON proof procedures.

If the objective is to find a proof with minimum height, where the height of a proof is the maximum size of the ancestor set, then the foothold refinement should not be used. The shortest negated ancestor proof may be rejected by the foothold condition, while a longer proof, composed of the same clauses in a different arrangement, meets the condition.

Proof height is an important consideration for "iterative deepening" proof methods. By restricting the search to a preset limit, and iteratively increasing this limit, depth first search strategies can be prevented from running down infinite paths. They will find each proof up to that limit. Thus depth first search achieves the completeness of breadth-first search, without also requiring exponential space.

The iterative deepening strategy can be used with the foothold refinement.

Because the minimum height proof may be eliminated, the searcher may be forced to look deeper to generate the same number of distinct proofs. The reduction in search space helps offset the cost of the additional depth. (See Figure 4.3.)

4.3.3 How many proofs are avoided?

The amount of redundancy reduced depends on the clauses, but we are guaranteed that the foothold procedure will generate no more redundancy than the negated ancestor procedure, since the foothold format is strictly more specific than the negated ancestor format.

There are some examples where the foothold format still admits an exponential amount of redundancy, in the sense that more than one proof can be built from the same set of clauses. Redundancy in these examples is due to interaction between features of the clauses, including the non-Horn feature. In the following example different parts of the proof must use different subproofs to prove the same literal. Redundancy arises because in yet another part of the proof the same literal can be proved by any of the subproofs. There are $\Pi_{i=1}^{n} i!$ foothold proofs of p from the following:

$$
\begin{array}{ll}
p \vee \neg p_1 \vee \ldots \vee \neg p_n & \\
p_i \vee \neg p_{i+1} \vee \neg a & \forall i = 1 \ldots n-1 \\
p_n \vee \neg a & \\
p_i \vee a & \forall i = 1 \ldots n
\end{array}
$$

There are some examples where the foothold procedure eliminates all redun-

dancy. Consider the following:

$$
\begin{array}{ll}
p \vee \neg p_1 \vee \ldots \vee \neg p_n & \\
p_i \vee \neg a_i & \forall i = 1 \ldots n \\
p_i \vee \neg b_i & \forall i = 1 \ldots n \\
a_i \vee b_i & \forall i = 1 \ldots n
\end{array}
$$

For each n there is exactly one foothold proof of p from the above, but there are 2^n negated ancestor proofs. In the following table we compare the time to calculate all proofs of p from this program for different values of n. (All programs in this chapter were written in Quintus Prolog, and run on a VAX 8600. Times are reported in milliseconds.)

Value of n	Foothold	Negated Ancestor
1	16	17
2	33	33
3	33	100
4	67	216
5	83	417
6	150	1033
7	184	2583
8	200	5483
9	300	12100
10	367	26400
⋮	⋮	⋮
30	3850	$\approx 10^{10}$

As expected, the time to build the 2^n negated ancestor proofs increases exponentially with n, but the time to build the only foothold proof, which has n literals, increases linearly with n.

Figure 4.2: Footholds on Pelletier's Problems

	#	Time	Count
FH	1	16	1
NA	1	17	1
FH	2	33	1
NA	2	33	1
FH	3	17	1
NA	3	0	1
FH	4	33	1
NA	4	17	1
FH	5	33	1
NA	5	33	1
FH	6	34	1
NA	6	17	1
FH	7	17	1
NA	7	17	1
FH	8	17	1
NA	8	0	1
FH	9	50	1
NA	9	50	4
FH	11	33	1
NA	11	33	1
FH	12	766	1
NA	12	222450	4096
FH	13	17	1
NA	13	33	1
FH	14	50	1
NA	14	50	4
FH	15	17	1
NA	15	33	1
FH	16	0	1
NA	16	17	1
FH	17	67	1
NA	17	67	2
FH	18	33	1
NA	18	16	1

	#	Time	Count
FH	19	17	1
NA	19	17	1
FH	20	50	1
NA	20	33	1
FH	21	67	2
NA	21	167	12
FH	22	4800	207
NA	22	50167	3782
FH	23	167517	4620
NA	23	1437117	97200
FH	24	100	1
NA	24	200	8
FH	25	2583	5
NA	25	4533	36
FH	27	67	1
NA	27	83	2
FH	28	34	1
NA	28	50	1
FH	30	50	2
NA	30	67	4
FH	31	50	1
NA	31	17	1
FH	32	50	1
NA	32	50	1
FH	35	0	1
NA	35	16	1
FH	36	50	4
NA	36	67	4
FH	37	483	29
NA	37	550	36
FH	39	33	1
NA	39	67	8
FH	44	50	1
NA	44	66	2
FH	45	16683	16
NA	45	1524550	9589

	#	Count at each height							
		3	4	5	6	7	8	9	10
FH	22	1	14	17	23	29	35	41	47
NA	22	4	73	165	298	467	672	913	1190
FH	23	2	10	108	276	516	828	1212	1668
NA	23	8	82	1350	4716	9990	17208	26370	37476
FH	25				2	2	1		
NA	25			5	9	10	6	4	2
FH	37	1	4	4	4	4	4	4	4
NA	37	1	5	5	5	5	5	5	5
FH	45						9		7
NA	45				5	7	147	708	8722

Figure 4.3: Proof Heights for Selected Problems

4.3.4 How much time is saved?

In Figure 4.2 we compare two procedures for producing proofs. We report on the time to build each proof and the number of proofs built. The procedure NA builds all negated ancestor proofs; FH builds all negated ancestor proofs that conform to the foothold condition.[2] If FH finds that a part of the proof does not conform to the foothold condition then it does not continue expanding that proof. The procedures are otherwise identical. They prune branches with identical ancestors, and they prune the search after a height of 10. We have run these programs on the tests suggested by Pelletier[22] that do not include equality axioms.

Pelletier's tests are unsatisfiable sets of clauses ; we converted each set of clauses into a problem of showing that a literal is entailed by a set of clauses by introducing

[2]Checking this condition adds an extra numerical operation each time we search up the ancestor path and an extra logical check if a complementary ancestor is found. The extra costs for these operations are negligible.

a new literal into the one clause in the set, and asking if the negation of that literal followed.

We report on problems for which a proof can be found in the reduced search space in a reasonable amount of time. Some problems required sound unification, with the occur-check. Because we used Prolog's unification, these could not be run.

In these tests the foothold refinement is never more expensive than the negated ancestor procedure, except in simple problems, such as 3 and 8, where the times are so small they have little significance. The time saved by the foothold restriction varies from a negligible amount to better than two orders of magnitude. More complex problems exhibit more savings. This suggests that for still larger problems the savings will continue to be significant.

Figure 4.2 indicates that the foothold format is never slower than the negated ancestor format, and can be much faster.

Figure 4.3 reports the number of proofs of each height for some of the more complex problems. In Problems 25 and 45 the minimum height proof is rejected. Proofs are found after 1 and 2 more levels, respectively. In these two examples the search spaces are especially reduced. This result suggests the the search space reduction will compensate for the loss of the minimum height proof.

4.4 Conclusion

In settings where results are computed from proofs, such as first order logic and default logic, to compute more than one result requires computing more than one proof. Duplicate proofs produce the same result. Time is wasted computing the duplicate proof and checking the result for redundancy. We have presented the

foothold format, a refinement of the MESON format that admits fewer duplicate proofs. Procedures that compute these proofs can detect redundancy before the entire proof is constructed. Empirical evidence shows a simple foothold procedure is never slower and sometimes much faster then the same procedure without the foothold refinement. The savings are greater for more complex examples.

Chapter 5

Our Truth Maintenance System

While the assumption-based Truth Maintenance System is uniquely well-suited for some reasoning problems where the information contains no disjunctions, it cannot handle situations where it is known that one of a number of facts holds, but it is not known which one. Plan recognition is such a situation. On observing some event that could be part of several plans, we conclude that one of the plans is occurring, but we cannot tell which one. In this chapter we discuss our efforts toward extending the ATMS to reason with disjunctions,[1] i.e. with non-Horn clauses.

Our goal has been to integrate our new procedures with the well-designed, existing ATMS procedures. Therefore we will discuss the existing procedures in more detail first. Then the new work is described in relation to the old.

As we indicated at the beginning of the previous chapter, we will apply the foothold restriction, that eliminates redundant proofs, to our truth maintenance system.

[1] By disjunctions we mean disjunctions of positive literals.

5.1 de Kleer's Basic ATMS

A high-level description of de Kleer's ATMS was given in Chapter 2.

In this presentation we give a more formal description, and a motivation for truth maintenance systems by relating them to default logic.

5.1.1 Default Logic

Commonly in artificial intelligence, conclusions must be drawn despite the absence of complete knowledge about the world. In these cases it may be possible to use knowledge of typical cases that is often true, but may admit exceptions. For example, if we are asked if a specific bird flies, since we know that most birds fly, it may be reasonable to assume that this is one that does. This type of reasoning is called *non-monotonic reasoning* because the set of conclusions we can reach may decrease or increase as we add more knowledge.

A default logic, such as Reiter's[26], is one that allows us to express that a statement it typically true. The following default rule expresses that birds typically fly:

$$\frac{Bird(x) : \mathrm{M}\, Fly(x)}{Fly(x)}.$$

Here M is to be read as "it is consistent to assume".

Defaults of the form

$$\frac{\beta(x) : \mathrm{M}\, \alpha(x)}{\alpha(x)}$$

are called *normal* defaults.

A normal default can be translated to a set of ordinary first order formulas and

a default of the form

$$\frac{M\ a(x)}{a(x)}$$

where a is a single literal. This translation has two steps.

- If $\alpha(x)$ is not a single literal then replace $\alpha(x)$ with a new literal $a'(x)$.

$$\frac{\beta(x) : \mathrm{M}\ \alpha(x)}{\alpha(x)} \Rightarrow \frac{\beta(x) : \mathrm{M}\ a'(x)}{a'(x)} \text{ and } a'(x) \leftarrow \alpha(x)$$

- Replace the default conditioned on $\beta(x)$ with an unconditional one, where $a(x)$ is a new predicate.

$$\frac{\beta(x) : \mathrm{M}\ a'(x)}{a'(x)} \Rightarrow \frac{: \mathrm{M}\ a(x)}{a(x)}, \text{ and } a'(x) \leftarrow \beta(x), a(x)$$

The bird example would be translated to

$$Fly(x) \leftarrow Bird(x), Birdsfly(x), \text{ and } \frac{: \mathrm{M}\ Birdsfly(x)}{Birdsfly(x)}.$$

In some systems for default reasoning, such as Theorist[23] and the ATMS, single literal defaults are represented by designating the literal as an assumption.

5.1.2 The Explanation Problem

In systems that represent defaults by designating literals as assumptions the explanation problem is

> Given a formula J, where some literals of J are designated as assumptions, compute a set E of assumptions such that for some literal l
>
> $$E \cup J \models l, \text{ and } E \cup J \text{ is consistent.}$$
>
> E is called an explanation of l.

5.1.3 A definition of the ATMS

The ATMS can be defined as a system that solves a special case of the explanation problem, with the following syntactic restrictions:

- J is a conjunction of Horn clauses, i.e. each clause is a disjunction containing at most one positive literal.
- All literals in J are propositions.
- g is a positive literal.
- Only minimal explanations are computed.
- All minimal explanations are computed for all positive literals.

(This definition is equivalent to Reiter's and de Kleer's characterization of an ATMS [27].)

Recall that the clause

$$a \vee \neg b_1 \vee \ldots \vee \neg b_n$$

is equivalent to

$$a \leftarrow b_1 \wedge \ldots \wedge b_n.$$

Since the ATMS computes explanations for only positive literals, and since support for a positive literal can be expressed with positive literals, there is no need for the ATMS to explain any negative literals.

The ATMS is a storage system. The minimal explanations for a literal p can be stored as a set of sets of assumptions. These explanations are called environments in the standard ATMS terminology. The data structure that contains this set of sets is called a label. The label for a literal is stored with the literal. (The set

of sets represents a disjunction of conjunctions, which is also called a disjunctive normal formula.)

The ATMS is also a maintenance system. It accepts new clauses into J, and allows new literals to be designated as assumptions. Given these changes, it computes the resulting changes to each of the explanations. This feature makes the ATMS applicable to situations where the set of clauses is given incrementally, but explanations are required after each addition to the clauses. The ATMS is designed to keep the explanations up-to-date by applying changes to them directly, rather than by resorting to recomputing them from the original clauses. Section 5.1.5 describes how this is done.

For situations where explanations are needed for only some of the literals, it may appear wasteful that the ATMS computes explanations for all of the literals. The ATMS is intended to be used with a problem solver that distills pertinent information from a problem domain, and presents the ATMS with a tightly-coupled set of clauses. Then the ATMS strategy of explaining all literals makes sense, since the explanations for a literal will depend directly on the explanations for other literals, and indirectly on a significant fraction of the set of all literals.

5.1.4 And/Or Graphs

The ATMS operates by constructing an and/or graph from the clauses in J, and propagating information through this graph.

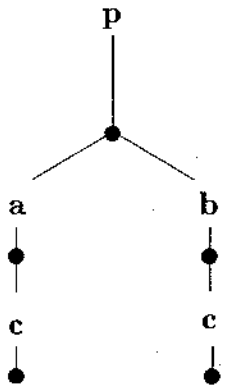
An and/or tree is a set of and-nodes, a set of or-nodes and a set of directed edges that connect either an and-node to an or-node, or an or-node to an and-node.

The ATMS will associate a positive propositional literal with each or-node. A deduction of a positive literal from a set of Horn clauses can be represented by an

and/or tree where all of the literals are positive and there is only one and-child for each or-node. For example, given the clauses

$$\begin{aligned} p &\leftarrow a \wedge b \\ a &\leftarrow c \\ b &\leftarrow c \\ c & \end{aligned}$$

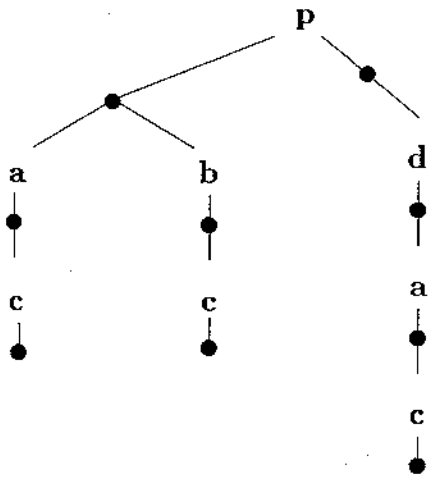
a deduction of p is represented by this and/or tree. (And-nodes are shown as circles. An or-node is represented, non-uniquely, by the literal it contains. An and-node with no children represents "true", e.g. the and-node child of the or-node containing c says that c is true.)



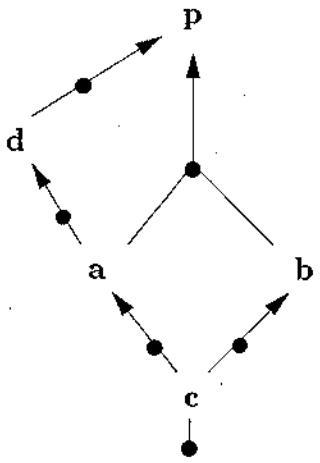
To represent more than one deduction with a single structure, we can allow more than one son for each or-node. For example, add these two clauses to the previous example.

$$\begin{aligned} p &\leftarrow d \\ d &\leftarrow a \end{aligned}$$

Then we can show the other deduction of p.



By collecting all or-nodes that contain the same literal into one node we reduce the number of nodes, but we change the tree to a graph.



Here there is a one-to-one correspondence between the and-nodes and the Horn clauses. If we add in all of the clauses, then the graph represents all of the proofs of all of the positive literals in the clauses. The ATMS builds these and/or graphs to represent all of the proofs. Since explanations are built by building proofs, all of

the explanations can be built from this graph. As they are built, the explanations are stored in the labels, in the or-nodes.

An or-node containing the literal $\perp$ is always present in an ATMS graph. $\perp$ represents "false". A Horn clause with no positive literal

$$\neg b_1 \vee \ldots \vee \neg b_n$$

is equivalent to

$$\perp \leftarrow b_1 \wedge \ldots \wedge b_n.$$

These negative-only clauses correspond to the and-children of the $\perp$ node. The minimal explanations of $\perp$ are the minimal inconsistent sets of assumptions, and are called *nogoods*. Since the labels are complete, every inconsistent explanation is a superset of some nogood.

5.1.5 Two Operations

The ATMS routines operate on a graph where each label is assumed to contain all of the minimal explanations of its literal. When the ATMS accepts new information either a new clause is added to the graph, or a literal is designated as an assumption. After the routines complete, the labels again contain all of the minimal explanations.

A new clause

A new clause

$$a \leftarrow b_1 \wedge \ldots \wedge b_n$$

is to be added. (See Figure 5.1.) A new and-node is added to the graph, corresponding to the new clause. An edge is constructed from the and-node to the

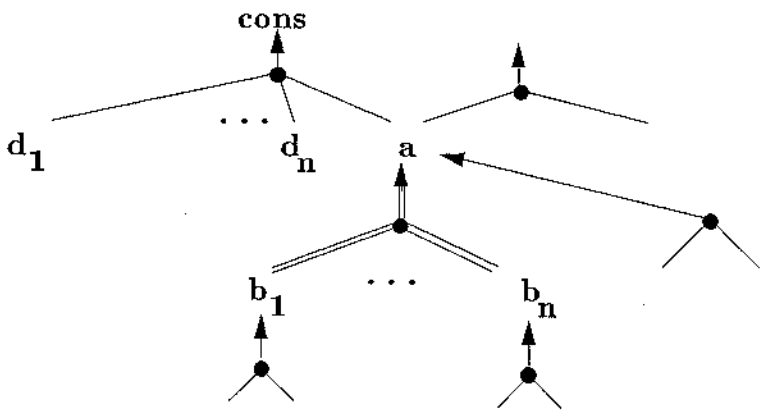


Figure 5.1: Adding a justification to the and/or graph

or-node containing a. For each i an edge is constructed from the or-node containing b_i to the new and-node. The new edges are shown as double lines.

New explanations for a may result from the new clause. To construct them, the ATMS builds the conjunction of the b_i labels. This conjunction of disjunctive normal formulas is converted to one disjunctive normal formula. Each conjunction in the new disjunctive normal formula that is consistent is an explanation for a.

For example, suppose there are just two b's, b_1 and b_2, and their labels are $(c \wedge d) \vee (e \wedge f \wedge g)$ and $(c \wedge e) \vee (d)$ respectively. Conjoining these and converting is done by considering each pair of conjunctions. The result is $(c \wedge d \wedge e) \vee (c \wedge e \wedge f \wedge g) \vee (c \wedge d) \vee (d \wedge e \wedge f \wedge g)$. If elsewhere we have derived that $(d \wedge g)$ is a nogood, then the fourth conjunction is discarded because it is inconsistent.

These new explanations for a are combined with the existing explanations. To ensure that only minimal explanations are computed, any explanation that is a superset of another explanation is discarded. In the example, the first explanation $(c \wedge d \wedge e)$ is discarded because of $(c \wedge d)$.

If the label for a changes as a result of the new clause, the new explanations will contribute to new explanations for the consequences of a. Suppose there is an existing clause for *cons*, a consequent of a,

$$cons \leftarrow d_1 \wedge \ldots \wedge d_n \wedge a.$$

Then the new explanations for a will be combined with the explanations for the d_i to form the new explanations for *cons*. We say that the effect of the new clause has been propagated from a to *cons*. Likewise it is propagated to all consequences of *cons*, and to other consequences of a.

If a new explanation is found for $\perp$ then a new nogood has been discovered. Existing explanations that are supersets of this nogood are now known to be inconsistent. Every label is checked to see if it contains an explanation made inconsistent by this nogood, and if so the explanation is removed.

If the label for a literal is not changed by the new explanations propagated to it, then it is not necessary to propagate to the consequences of the literal. In this way propagation eventually halts.

There may be loops in the graph, so a label may be updated several times. But propagation is guaranteed to terminate since there are only a finite number of explanations and each update to a label adds to the number of consistent explanations in the graph.

A new assumption

A new assumption may be added to the ATMS, or an existing literal may be designated as an assumption. An explanation for an assumption is itself, assuming it is consistent. Adding a new assumption to the graph, then, is simply a matter of

adding a new explanation to its label, consisting of itself, and using the propagation algorithm described in the previous section to propagate the effect of that new explanation through the graph.

The basic ATMS algorithms proposed by de Kleer [8] are presented in more detail in Appendix C.1.

5.1.6 Effectiveness of the ATMS

Since the ATMS is providing a service to a problem solver to help it become more efficient, it is important that the ATMS do its own work, building and propagating explanations, as efficiently as possible. Two features of the algorithms streamline propagation.

- Calculating the new explanations does not require searching. All of the information needed to calculate the new explanations is propagated to the or-node.
- Redundant and inconsistent information is not propagated. Since the label contains all of the explanations, any duplicate or subsumed explanation can easily be detected. Inconsistent explanations are supersets of a nogood, so they can be detected.

5.2 Non-Horn Clauses and Truth Maintenance

There are some domains that cannot be adequately described by Horn clauses, yet would benefit from a truth maintenance system. Plan recognition is an example of one. The problem is to find a truth maintenance system that maintains complete support for its literals, yet deals with non-Horn clauses.

A desirable feature of a solution is that it integrates well with the existing ATMS algorithms, since that will help ensure its effectiveness.

5.3 Our Solution

Negated ancestor proof graphs from Chapter 4 bear a close resemblance to the ATMS and/or graphs. Both represent the dependency of a consequent on its conditions using edges in the graph. Since negated ancestor proof graphs can represent a deduction from non-Horn clauses it is natural to ask if they can become the basis for a truth maintenance system for non-Horn reasoning.

The and/or graphs of the basic ATMS are insufficient for representing negated ancestor proof graphs in two ways. The negated ancestor proof graphs may need to use different contrapositive forms of the same clause, and and/or graphs do not contain anything corresponding to backedges.

In Section 5.3.1 we will show that and/or graphs can be built that contain all possible contrapositive forms. We call that the *main* graph.

To handle the backedges, in Section 5.3.2 we introduce a companion graph, called an *backedge* graph, to be used in conjunction with the and/or graph. Like the main graph, the backedge graph is also an and/or graph, with nodes that contain literals and edges defined by contrapositive forms of the clauses.

Following that we construct a new truth maintenance system based on and/or graphs and backedge graphs. Like the ATMS each or-node contains a label where the explanations for the literal are stored. We show that the propagation algorithms for keeping the labels up-to-date are the same as the original ATMS algorithms, except that under certain conditions propagation occurs from the backedge graph

and the main and/or graph. Therefore the system is well integrated, and exhibits the same properties that account for the ATMS's efficiency.

5.3.1 And/or graphs from all contrapositives

In Section 5.1.4 we showed how all of the proofs of positive literals from a set of Horn clauses could be represented by an and/or graph that only contained positive literals. There was only one and-node for each clause in that graph.

But if there are non-Horn clauses, such as $a \vee b$, then to represent all proofs the graph must contain some negative literals since negative literals can now provide support for positive ones, as in $a \leftarrow \neg b$. To guarantee we have all of the possible proofs represented, first we must build the and/or graph with *all* contrapositive forms of the clauses. Thus for a clause of n literals there are n and-nodes present in the graph.

And/or graphs from all contrapositive forms have some interesting symmetries. See Figure 5.2.

Extracting proof graphs from and/or graphs

We can extract some negated ancestor proofs from an and/or graph, by first selecting one and-node for each or-node, producing an and/or subgraph, and then replicating or-nodes that have more than one and-parent, so that each or-node has at most one and-parent. This produces an and/or tree. (As a byproduct of the extraction two different nodes may contain the same literal – as in the tree on page 89.) Trivially an and/or tree can be converted to a negated ancestor proof graph by replacing each and-node such as

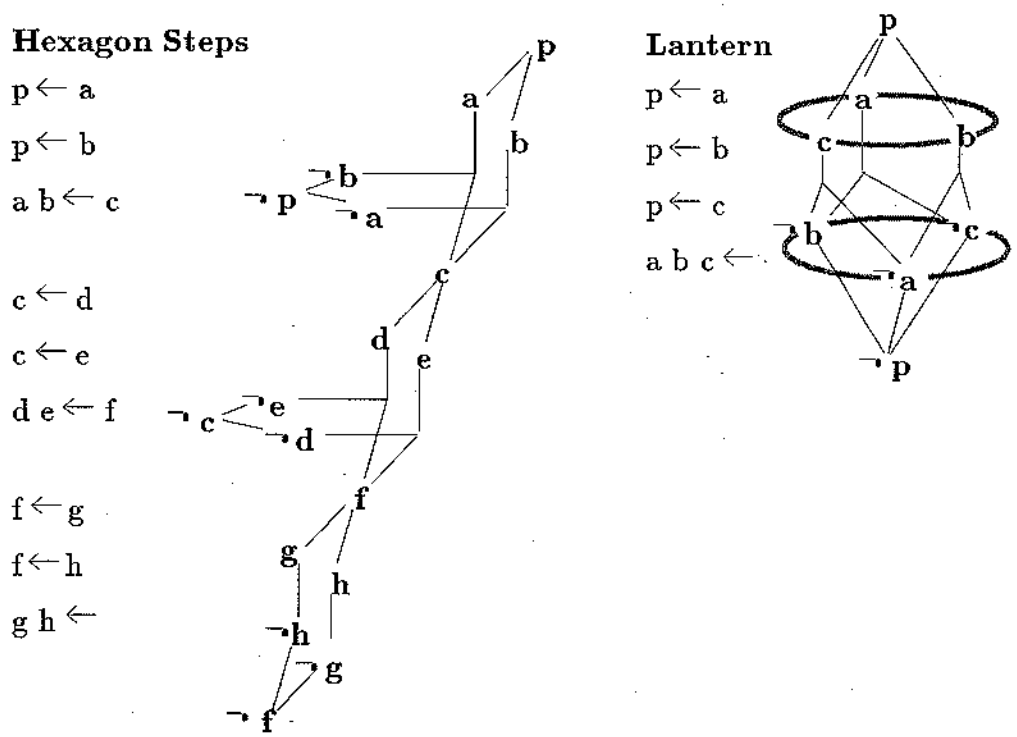
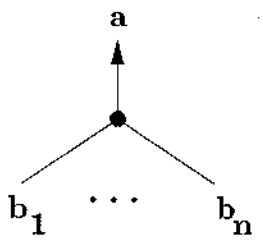
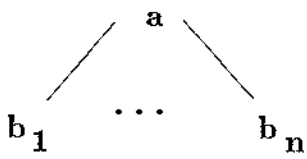


Figure 5.2: And/Or graphs from all contrapositive forms



with direct edges from its or-parent to its or-children, resulting in the following.



But negative ancestor proof graphs that contain backedges are not represented in the and/or graph with all of the contrapositive forms. To represent them we also need to add something that corresponds to the backedges in the negated ancestor proofs.

5.3.2 Backedge Graphs

In the previous section when we converted an and/or graph into a negated ancestor proof graph we did not have any backedges to deal with. As a result these graphs cannot represent a deduction that arises from reasoning by cases, the deduction that is represented by a backedge. We propose to allow this type of reasoning by building a companion and/or graph, which we call a backedge graph.

Recall that a backedge in a negated ancestor proof graph connected a node containing a literal, say l, to an ancestor node containing the complement literal,

$\bar{l}$. The edges on the path from the node to the ancestor came from specific contrapositive forms. We construct the companion graph from the nodes and edges necessary to include these contrapositive forms. This companion graph represents the parts of negated ancestor proof graphs that give rise to backedges from l to $\bar{l}$. We call this companion and/or graph the backedge graph for l.

For example, given the clauses

$$\begin{aligned} goal &\leftarrow p \\ p &\leftarrow q \\ p \vee q &\leftarrow Asn_1 \\ p &\leftarrow r \\ r &\leftarrow Asn_2 \end{aligned}$$

(where Asn_1 and Asn_2 are literals designated as assumptions.) Figure 5.3 is a portion of the and/or graph from these clauses.

Because there is a path from $\neg p$ to p, we can construct a backedge graph for $\neg p$, as shown in Figure 5.4.

5.3.3 A new truth maintenance system

Our truth maintenance system builds the two structures we have discussed (1) an and/or graph from all contrapositives, and (2) from some of the paths in this graph from a literal l to $\bar{l}$, a backedge graph for l.

Recall that for the and/or graph in the basic ATMS, each or-node contains both a literal and a label that is interpreted as the explanations of the literal. Likewise in our and/or graph, each or-node contains a label. Furthermore or-nodes in the backedge graph for l contain labels; they are interpreted slightly differently. These labels contain the explanations that would hold *if l were true.*

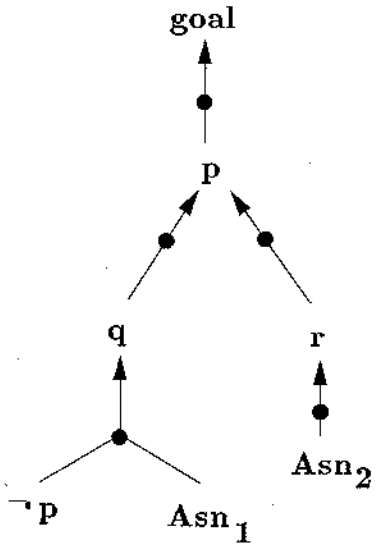


Figure 5.3: An example And/Or graph

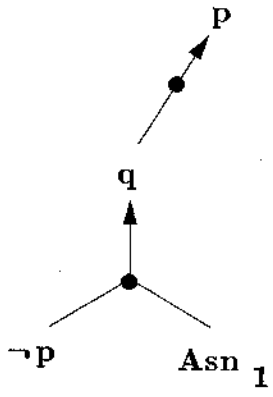


Figure 5.4: An example backedge graph

In the main and/or graph, reasoning is accomplished by propagating labels from conditions to consequences, along the edges in the graph. In the backedge graph for l, reasoning is done in exactly the same way, so the original ATMS algorithms are used for propagating within the backedge graphs.

To do reasoning by cases we need to propagate information from the backedge graph for l to the main and/or graph. In the backedge graph for l one of the nodes is $\bar{l}$ and its label has a special property. Unlike other labels in this graph, it does not depend on assuming l is true. Why? If E is an explanation of $\bar{l}$ assuming l is true then $l \rightarrow (E \rightarrow \bar{l})$, so $l \wedge E \rightarrow \bar{l}$. This means $E \wedge l$ is inconsistent, so $E \rightarrow \bar{l}$. In other words, E is an explanation of $\bar{l}$.

Based on this observation, the propagation algorithm performs one new function: the label for $\bar{l}$ in the backedge graph for l should be propagated to the $\bar{l}$ in the main graph. The new function gives our system the ability to reason by cases.

Return to the example of Figure 5.3. Since Asn_1 and Asn_2 were assumptions, each is its own explanation. Propagating these through the and/or graph in Figure 5.3 yields the following labels:

$$\begin{aligned} goal &: \{\{Asn_2\}\} \\ p &: \{\{Asn_2\}\} \\ r &: \{\{Asn_2\}\} \\ q &: \{\} \\ \neg p &: \{\} \end{aligned}$$

Since $\neg p$ and q have no support each of them has the label $\{\}$ which contains no environments, and represents false.

Note that Asn_1 is also an explanation for *goal*, but that the and/or graph cannot create it.

In the backedge graph for $\neg p$ (Figure 5.4) the label for $\neg p$ is set to $\{\{\}\}$ which contains the empty environment and represents true. (This is done because the labels in this backedge graph are interpreted as if $\neg p$ were true.) This label is conjoined with the label for Asn_1, which is just $\{\{Asn_1\}\}$, and propagated to q in the backedge graph. Subsequent propagation of the labels through this backedge graph yields

$$\begin{aligned} p &: \{\{Asn_1\}\} \\ q &: \{\{Asn_1\}\} \\ \neg p &: \{\{\}\} \end{aligned}$$

Since p has an environment Asn_1 in the backedge graph for $\neg p$, it is propagated to p in the main and/or graph. As a result it is propagated to *goal*, so the label for *goal* becomes $\{\{Asn_1\}\{Asn_2\}\}$, and this is the complete set of explanations for *goal*.

5.3.4 Our TMS Algorithms

The propagation algorithms for our TMS are adapted from the algorithms for the basic TMS, in Section 5.1.5, by three simple changes.

Recall that we are using two types of and/or graphs, the *main* and/or graph that contains all of the contrapositive rules of all of the clauses, and the backedge graph that contains only the contrapositive rules needed to identify backedge paths. We must propagate information through both of these graphs. We can use the same algorithms to do it if we add to each TMS routine a parameter that specifies which graph is being affected.

As new information is added to the system, the effects of that information must be propagated through the backedge graphs as well as the and/or graph. To do

this we add a new statement to the end of Propagate (the routine which performs propagation) so that after an incremental update is applied to a literal in the main graph, it is also applied to backedge graphs that contain this literal.

The new environments that the backedges allow us to find, and that the backedge graph computes must be propagated into the main and/or graph. If there is an update to the node containing $\bar{l}$ in the backedge graph for l then, according to the discussion on page 101 there should also be an update to the node containing $\bar{l}$ in the main and/or graph. A line at the end of Propagate accomplishes this.

See Appendix C.2 for a full description of our TMS algorithms.

5.3.5 Restricting Backedge Graphs

In this subsection and the next we describe how backedge graphs can be constructed with backedge path algorithms. The TMS algorithms in Appendix C.2 together with the backedge path algorithms, is our solution to TMS with non-Horn clauses, to be compared to the extended ATMS of de Kleer's[7].

Every deduction that can be represented by a negated ancestor proof graph can also be represented by our truth maintenance system. Therefore if the backedge graphs contain every possible backedge path our truth maintenance system will be complete.

There may be an exponential number of paths through the graph from a literal to its complement. Can we avoid adding all possible paths into the backedge graphs? There are two types of paths that we never need to add. A non-minimal path is a path from a literal to its complement that contains a complementary pair of literals (that are not at the ends of the path.) A path that does not conform to the foothold

condition never needs to be added because, as we saw in Chapter 4, such backedges are never needed.

Two other conditions may arise in a specific domain that eliminate the need to find all backedge paths. If a complete set of explanations is required for one literal, or a small number, but not for all, then searching can be restricted.

Non-minimal paths

A non-minimal path connects a literal to its complement but contains another pair of complementary literals. For example, from the clauses

$$\begin{aligned} q &\leftarrow p \\ p &\leftarrow a \\ p &\leftarrow b \\ a &\vee b \end{aligned}$$

There is a path from $\neg q$ to q

$$\neg q \rightarrow \neg p \rightarrow \neg b \rightarrow a \rightarrow p \rightarrow q$$

that contains a path from $\neg p$ to p. If we add to the backedge graph only the path from $\neg p$ to p, then the backedge graph will provide the deduction of p. Knowing p allows us to deduce q. The other deduction of q, from the non-minimal path, is redundant.

The Foothold Condition

The foothold condition in the previous chapter can be used to eliminate a large, possibly exponential, number of the paths. Foothold labels can be assigned to the

contrapositives and paths that do not conform to the foothold condition should not be added to the backedge graphs.

For example, from the clauses in the previous section, there is are two paths from $\neg p$ to p:

$$\neg p \rightarrow \neg b \rightarrow a \rightarrow p$$
$$\neg p \rightarrow \neg a \rightarrow b \rightarrow p$$

If we assign foothold labels to the contrapositives of $a \vee b$ as follows,

$$a \leftarrow \neg b^{(-1)}$$
$$b \leftarrow \neg a^{(+1)}$$

then the paths become

$$\neg p^{(0)} \rightarrow \neg b^{(-1)} \rightarrow a^{(0)} \rightarrow p$$
$$\neg p^{(0)} \rightarrow \neg a^{(+1)} \rightarrow b^{(0)} \rightarrow p.$$

We only accept the second path since it has a positive foothold sum. We do not need to add the second path to the backedge graph for $\neg p$.

Selective Completeness

In some applications that use truth maintenance systems the problem may be to produce a complete set of explanations for only some literals, and not all literals. This is called selective completeness, and can be viewed as a further restriction to the minimal path and foothold conditions. With selective completeness, the TMS finds fewer backedge graphs, i.e. only those that contribute support to one of the literals of interest. Suppose l is one of these literals. To ensure that its label is complete, a search is done from l, similar to the search required to build a foothold proof for l, described in Section 4.2.

For example, if we are given the clauses in the example on page 99 with the additional clause $\leftarrow p \wedge q$, and we are only interested in the literal *goal*, then only the backedge shown in Figure 5.4 is constructed. Another backedge graph exists from the full set of clauses, namely

$$\neg q \leftarrow \neg p \leftarrow q.$$

5.3.6 Backedge Path Algorithms

The algorithm that finds backedge paths should be guided by the restrictions in the preceding section. In Section 5.5, for the case of assimilation in plan recognition, we will use selective completeness incorporating the foothold condition. We describe here the algorithm FindPaths for finding the backedge paths (inherent in the contrapositive rules) according to the selective completeness and foothold restrictions, and the algorithm AddPath for adding these paths to the backedge graphs. These algorithms are presented fully in Appendix C.2.1.

The FindPaths algorithm assumes that the set of contrapositive rules has been labelled with some foothold labelling so that a foothold label is associated with each condition in each rule. This assignment needs to be done only once for each clause. A convenient place to do it is near the beginning of AddClause (the routine which serves to add clauses to the TMS) where the contrapositives are formed.

FindPaths is given a start literal, say l, from which it begins the search. It uses an ancestor sequence to keep track of the partial backedge paths. This corresponds to the ancestor path A used in the definition of NA1 proofs, Definition 4.3.2. Initially the ancestor sequence is empty. Given a literal and an ancestor sequence FindPaths checks that this literal is not already in the ancestor path (as that would be mean a loop has been encountered, and paths with loops are one type

of non-minimal path, so they are not allowed.) If the complement $\bar{l}$ of the literal is found among the ancestors, (case a) of Definition 4.3.2, then a backedge path has been found. If this backedge path satisfies the foothold condition then this backedge path can be added to the backedge graph for $\bar{l}$, used by the TMS. This addition is done by AddPath, below. Regardless of whether the foothold condition holds, no further searching is needed from this literal. If the complement of the literal is not among the ancestors, then FindPaths steps through (case b) of Definition 4.3.2, non-deterministically choosing a contrapositive rule and a condition.[2] The algorithm eventually backtracks so that all of these non-deterministic choices are considered.

The set of backedge paths found are given to the algorithm AddPath which adds them to a backedge graph. For the example on page 69, one of the negated ancestor proof graphs is shown on page 70. There is a backedge path from $\neg p$ to p. Here is the corresponding backedge graph with both the and-nodes and or-nodes shown.

AddPaths would generate this graph as the backedge graph for $\neg p$.

[2]When choosing a contrapositive, it is not necessary to choose one taken from a clause in CUA^- or EXA^- since those clauses are defunct.

AddPaths works as follows. For each rule in the path, a new and-node is put into the graph. An or-node must be put into the graph for each literal in the rule. (If a node does not already exist for some literal then a new one is created and it is given the same label as the label for that literal in the main graph. In this way the labels in the backedge graph are always more specific than those in the main graph. This is important since the backedge graph labels are interpreted with the extra condition of assuming $\neg l$ is true.) Once these or-nodes are in place, the or-node for the conclusion of the rule becomes the parent of the new and-node. The or-nodes for each condition become the children of the and-node. Any propagation that results from these additions is done next. As its final step AddPath ensures that the label for $\neg l$ is $\{\{\}\}$, and any propagation that arises is done.

5.4 Comparison with de Kleer's Extended ATMS

5.4.1 de Kleer's Solution

The extended ATMS [7] allows the problem solver to express disjunctions only in a restricted form

$$\text{choose } \{A_1, \ldots, A_n\}$$

where each A_i is a positive assumption. It is possible to encode any clause as a set of Horn clauses and a set of disjunctions of positive assumptions. Four encoding methods are given in [7]; each method involves introducing new assumptions which are not relevant to the problem at hand. These so-called encoding assumptions ought not to be revealed to the problem solver, so any explanation that involves an encoding assumption should be ignored.

In order to reason with disjunctions, two new procedures are needed, positive

hyperresolution and negative hyperresolution. The negative hyperresolution procedure finds nogoods that arise as a consequence of disjunctions. It implements the following rule:

$$\begin{array}{l}\text{choose } \{A_1, \ldots, A_n\} \\ \underline{\text{nogood } \alpha_i \text{ where } A_i \in \alpha_i \text{ and } A_{j \neq i} \notin \alpha_i \text{ for all } i} \\ \text{nogood } \bigcup[\{\alpha_i - \{A_i\}]\end{array}$$

In order to perform this step we must find a choose such that for each assumption there is a nogood with a singleton intersection with the choose. Then from the remainders of these nogoods we build a new nogood.

To discover all nogoods with the negative hyperresolution rule, this procedure must be applied whenever a new choose or a new nogood is given or discovered. Whenever the procedure succeeds, a new nogood is discovered, and the search must begin anew.

The positive hyperresolution procedure finds the new environments for propositions that arise from the chooses. Let $\langle \beta, \lambda \rangle$ be the node containing the literal β and the label λ. Positive hyperresolution implements the following rule:

$$\begin{array}{l}\text{choose } \{A_1, \ldots, A_n\} \\ \langle \beta, \lambda \rangle \\ \underline{\text{nogood}[\{A_i\} \cup \alpha_i] \text{ or } \{A_i\} \cup \alpha_i \in \lambda, \text{ and } A_{j \neq i} \notin \alpha_i \text{ for all } i} \\ \langle \beta, \{\cup \alpha_i\} \cup \lambda^* \rangle\end{array}$$

where λ^* is λ with all supersets of $\cup \alpha_i$ removed.

Whenever a new choose is given, a new nogood is found, or a new explanation is found for a literal β, the positive hyperresolution procedure should be applied to find any new explanations for β. The procedure searches for a choose such that

for each assumption in it, there is a nogood or an explanation of β that has a singleton intersection with the choose. From the remainders of these nogoods and explanations we build a new explanation for β. After such a new explanation is found, the positive hyperresolution procedure must be invoked again if completeness is to be assured.

Various special cases of each of these rules can be efficiently implemented. For example when a choose is a singleton, it can be removed from each nogood. The general cases, however, are still necessary.

5.4.2 Comparison

The hyperresolution procedures in the extended ATMS are considered expensive [7, 8]. Let us point out some of the reasons for the high cost.

- Encoding adds work. Besides the cost of automatically translating non-Horn clauses into ATMS inputs, each assumption introduced for encoding adds some work to the ATMS. In the worst case, each assumption could double the amount of work, since the amount of work is proportional to the number of explanations, which is exponential in the number of assumptions.

- Hyperresolution must be applied frequently, to guarantee consistency and completeness. Both positive hyperresolution and negative hyperresolution will be invoked each time a new choose is declared and each time a new nogood is discovered. Positive hyperresolution will also be invoked as a new environment is added to a label, which is the most common operation.

- New information is not propagated to where it is needed. Hyperresolution must search for relevant information. For instance, a new choose requires

positive hyperresolution to search the entire database, and parts of it many times. This is contrary to the design philosophy of the basic ATMS, where discovered information is propagated to all of the places it is needed.

- Hyperresolution might produce uninteresting information. Positive hyperresolution may add environments containing encoding assumptions, that will be ignored on output. Negative hyperresolution may discover nogoods with encoding assumptions, so explanations with them would not have been reported anyway. In these cases the results are not relevant to the problem solver.

Our solution, with backedge graphs, does not exhibit these causes of inefficiency. Since the input form of the clauses is not restricted, there are no encoding assumptions. This covers the first and last causes. The second does not apply. As for the third, our system does propagate discovered information to where it is needed, and integrates well with the basic ATMS.

Three criticisms can be raised against our solution. Our and/or graph contains all of the contrapositives of the clauses, while de Kleer's extended ATMS just contains one form of each clause. Next, our system requires a search for backedge paths. In Section 5.3.5 we pointed out how and in what situations this search can be controlled. Finally our solution finds explanations for all positive and negative literals, whereas de Kleer's only explains positive literals. (This is actually an an advantage of our system, when there is interest in explanations for negative literals.)

How can we decide which truth maintenance method is more effective? Comparing by performing experiments is subject to questions about the effectiveness of the particular implementations. Deciding based on worst case analysis is problematic since it has been shown that the truth maintenance problem is NP-complete [25, 12]. We conclude by pointing out that for our problem domain, plan recognition, our

truth maintenance system has lead to an effective solution to the assimilation problem. It is to this solution we now turn our attention.

5.5 Our TMS applied to Plan Recognition

In this section we will put together the parts of our solution to assimilation in plan recognition into an overall algorithm. Initially we consider only the case of a single observation; multiple observations are handled later.

5.5.1 Interfacing the Plan Recognizer with the TMS

Before the entire solution can be described, three points must be made about interfacing the plan recognition system with the truth maintenance system. First, the TMS generates proofs, but stores them in its internal and/or graphs. We need access to these proofs because they represent the candidate plans. Next, the TMS only considers single-literal queries, but for plan recognition we need to generate a disjunction of End events. Finally there are a large number of backedge paths that *could* be added to the backedge graph, but only some of these are necessary. How do we find the ones necessary for plan recognition?

For the first question, we can reconstruct a negated ancestor proof graph from the TMS's and/or graph for a literal by the algorithm Extract, included in Appendix C.3.1. This algorithm traverses the and/or graph and produces one and-node for each or-node. The and-node chosen is the one that contributes support to the or-node. Since support in the TMS is sometimes in the main and/or graph, and sometimes in one of the backedge graphs, this algorithm must check both.

For the next question, to render our TMS able to compute candidate plans add the clause

$$end \leftarrow End(x)$$

for each instance of $End(x)$ that is in the search space identified by the search algorithm, (in Visited). Then the TMS generates the explanations for *end*, by building an and/or graph, and a backedge graph for $\neg end$. We use the Extract algorithm to extract a negated ancestor proof graph for *end* from the TMS. We can examine this proof graph for occurrences of $\neg end$, and the associated instances of End give us the disjunction of End events. (This operation is analogous to Stickel's method for generating disjunctive answers, in Section 4.1.2.)

For the final question, we can use the selective completeness restriction and the foothold restriction of Section 5.3.5 to limit the search for backedges. We are only interested in a proof of the literal *end*, so only the backedges paths produced by FindPaths(end) are added to the TMS. The algorithm appears in Appendix C.2.1, and is described in Section 5.3.6. After the paths are found, the algorithm AddPath is called for each path, to add it to the TMS and initiate propagation along these paths.

5.5.2 Plan Recognition Algorithm (Single Observation)

Given an abstraction/decomposition hierarchy representing a plan library, and a single observation which is described indefinitely as a disjunction of possible event types, our aim is to generate a candidate plan, which is represented by a disjunction of End events and a proof of that disjunction. The first step is to call CloseLibrary (this could have been done before the observation was given) to generate the sets of first order axioms, CUA, EXA and DJA. The CUA axioms are Skolemized, and

both the CUA and EXA axioms are given new, unique assumptions. Then Search is called to generate ground instances of the first order axioms, and store them in the set Axioms. The ground instances of any corresponding assumptions are put into a set called Assumptions. Search also generates Visited, the set of literals visited in the course of searching. The Reasoning task then begins. The TMS routines to add new clauses and new assumptions are used. Each ground axiom in Axioms is now added to the TMS with AddClause. Assumptions in the set NewAssumptions are added with AddAssumption. For every literal $End(M)$ in Visited, the clause $end \leftarrow End(M)$ is also added with AddClause. FindPaths finds all backedge paths that contribute support to *end* that conform to the foothold and minimal path restrictions, and AddPath adds each path found to the one of the backedge graphs in the TMS. TMS propagation is invoked as each contrapositive form of each clause is added, as each assumption is added and as each backedge path is added. After all of the additions are done, this propagation guarantees that the labels in main and the backedges graphs are up to date. Extract is called to generate the negated ancestor proof graph for *end*, and the disjunction of End events can be read directly from this graph.

Now suppose new information is given by the oracle.[3] This means there are new abstraction relationships for H_A, new steps for H_D, new constraints for H_D, or any combination of these. Consider each new clause in turn. We call LibraryAssimilate with the clause and so generate C^- and C^+. AssimSearch is called and makes used of C^- and C^+. During its operation AssimSearch identifies new clauses to be added to the TMS, and puts them into the set NewAxioms. The ground instances

[3]We have deferred the issue of interfacing the Plan Recognizer with the Oracle. One scenario is to have the Plan Recognizer consult the oracle with "I give up" if there is no candidate plan possible.

of the assumptions in CUA^+ and EXA^+ are put into the set NewAssumptions. Ground instances of assumptions in CUA^- and EXA^- (identified by literals in Visited) are put into the set OldAssumptions. Then the Reasoning stage begins. The operations in this reasoning stage affect the same main and backedge graphs used by the TMS after the initial observation. AddClause adds each new clause in NewAxioms to the TMS. The new assumptions in NewAssumptions are added with AddAssumption. Each old assumption is removed, by adding a clause of the form $OldAssn \supset false$. For every literal $End(M)$ put into Visited by AssimSearch, the clause $end \leftarrow End(M)$ is also added with AddClause. FindPath(*end*) and AddPath are called to add the new backedge paths to the TMS. Each addition to the TMS initiates propagation. Propagation through the and/or graphs and the backedge graphs might affect the proof for *end*. If there is no effect then Extract will extract the same proof. Otherwise there may be no proof, so Extract will indicate that, or there may be a new proof, and Extract will return that new proof.

This description is summarized by the following algorithm. A more detailed version of this algorithm appears in Appendix C.3.2

procedure PlanRecognition-SingleObs
%
Let H be the initial hierarchy
Let $E_1(Obs) \vee \ldots \vee E_n(Obs)$ be the single observation
call CloseLibrary(H)
call Search($E_1(Obs) \vee \ldots \vee E_n(Obs)$)

% The Reasoner
Add each ground clause in Axioms to the TMS with AddClause
Add each new assumption in NewAssumptions with AddAssumption
For each $End(M)$ in Visited **call** AddClause $end \leftarrow End(M)$
% Deal with the Backedges
call FindPaths(end)
For each Path found **call** AddPath(Path)

call Extract(end) to get the candidate plan

loop

Accept NewInfo from the Oracle
call LibraryAssimilate(NewInfo)
call AssimSearch to get the additional ground clauses, the additional assumptions, and the old assumptions which were violated

% The Reasoner
Add each new clause to the TMS with AddClause
Add each new assumption to the TMS with AddAssumption
For each old assumption OldA **call** AddClause(OldA $\supset$ false)
% Deal with the Backedges
FindPaths(end)
for each Path found **call** AddPath(Path)

call Extract(end) to get the candidate plan
go to loop

end procedure PlanRecognition-SingleObs

5.5.3 Returning to the example

Recall the example from in Figure 3.1. From observing the agent measuring flour the plan recognizer generates the ground clauses and assumptions described in Section 3.3 and they are given to the TMS. There are no backedge paths generated for this example. Extract gives us the proof graph pictured in Figure 5.5. (For the sake of simplicity we do not show the End disjunct in the CUA axioms. In all of our cases they are eliminated by some DJA axiom.)

Now the oracle provides the new information about fettucini marinara, shown in Figure 3.1. After library assimilation, AssimSearch generates new clauses and assumptions, and the old CUA axiom for MakeFettuciniNoodles is removed. The

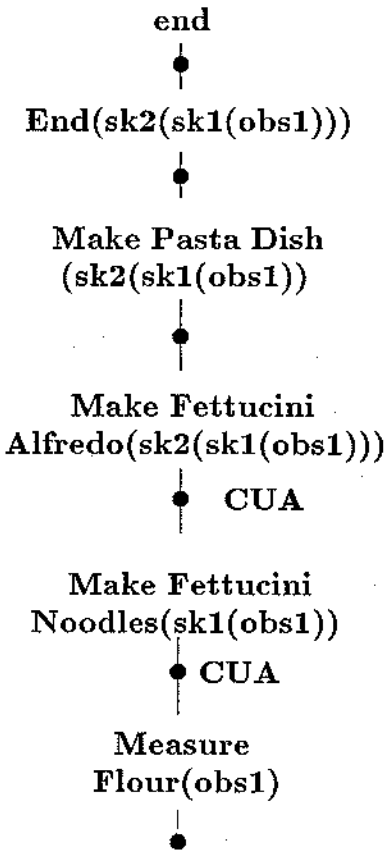


Figure 5.5: Proof Graph from the Initial Library

final proof graph extracted is shown in Figure 5.6.

In this example FindPaths generates the following backedge path

$$MakePastaDish(sk2(sk1(obs1))) \leftarrow$$
$$MakeFettuciniAlfredo(sk2(sk1(obs1))$$
$$MakeFettuciniAlfredo(sk2(sk1(obs1)) \leftarrow$$
$$MakeFettuciniNoodles(sk1(obs1)) \wedge$$
$$\neg MakeFettuciniMarinara(sk2(sk1(obs1))$$

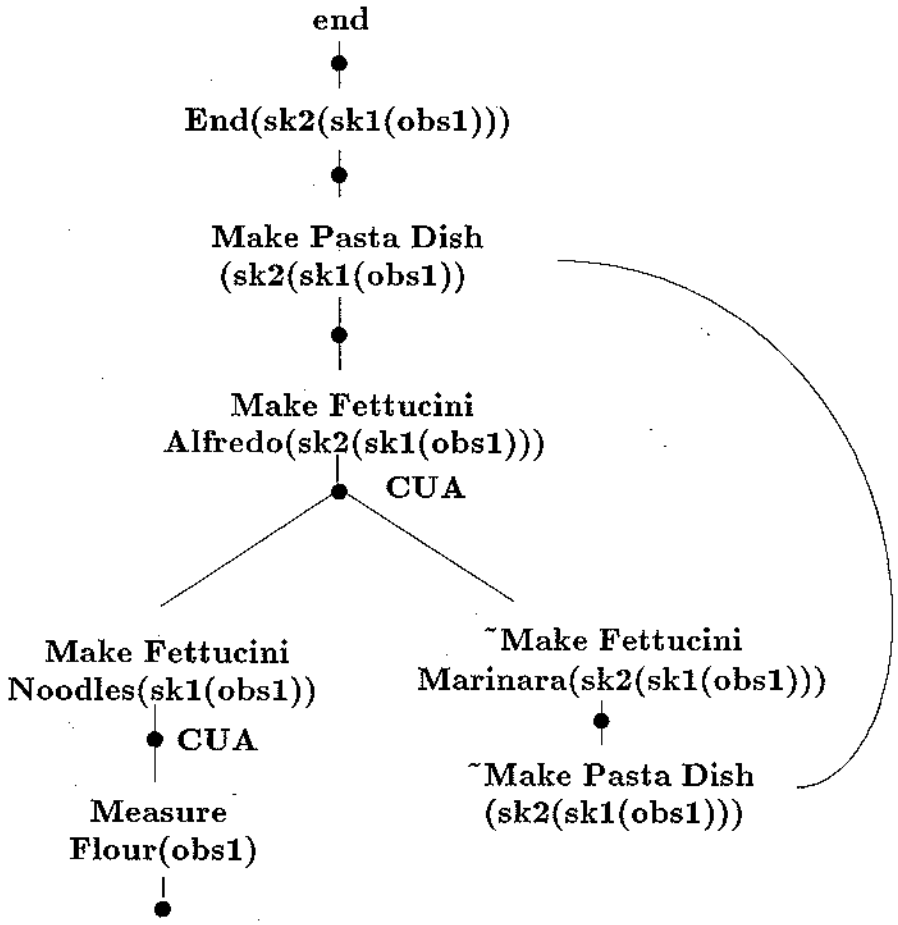


Figure 5.6: Proof Graph from the Library after Assimilation

$\neg MakeFettuciniMarinara(sk2(sk1(obs1)) \leftarrow$
$\neg MakePastaDish(sk2(sk1(obs1)))$

If it were not for the foothold condition, another backedge path would have been added. It differs only in the MakeFettuciniAlfredo is swapped with MakeFettucini-Marinara. Footholds remove this symmetry.

$MakePastaDish(sk2(sk1(obs1))) \leftarrow$
$MakeFettuciniMarinarMarinara(sk2(sk1(obs1))$
$MakeFettuciniMarinara(sk2(sk1(obs1)) \leftarrow$
$MakeFettuciniNoodles(sk1(obs1)) \wedge$
$\neg MakeFettuciniAlfredo(sk2(sk1(obs1))$
$\neg MakeFettuciniAlfredo(sk2(sk1(obs1)) \leftarrow$
$\neg MakePastaDish(sk2(sk1(obs1)))$

Illustrating assimilation of constraints

Our algorithms for plan recognition using truth maintenance can be applied to a variety of assimilation problems. We illustrate the case of assimilating constraints here.

Continuing with our example, suppose the oracle provides a new constraint on the making of fettucini, that the agent must have a strong hand to mix the fettucini noodles and the alfredo sauce, because the sauce becomes thick as it cools. (This does not occur with marinara sauce.) When this new knowledge is assimilated, a check of the constraint is made. If it is known to the plan recognizer that the agent is not strong then it is reasonable to reject the fettucini alfredo plan and propose that the agent is making fettucini marinara.

The general form of the constraint is

$$\forall x.MakeFettuciniAlfredo(x) \supset strong(agent(x))$$

When AssimSearch notices that $MakeFettuciniAlfredo(sk2(sk1(obs1)))$ is in Visited it generates the appropriate instance

$$MakeFettuciniAlfredo(sk2(sk1(obs1))) \supset strong(agent(sk2(sk1(obs1))))$$

An attempt is made to prove that $\neg strong(agent(sk2(sk1(obs1)))$, and this attempt succeeds. As a result

$$MakeFettuciniAlfredo(sk2(sk1(obs1))) \supset false$$

is added to the TMS. This causes a new proof for end to arise in the main graph, which previously could not be generated, because there was no support for

$$\neg MakeFettuciniAlfredo(sk2(sk1(obs1))).$$

Extract generates the proof graph in Figure 5.7.

5.5.4 Candidate Assimilation and Multiple Observations

Until now we have only discussed candidate assimilation in the case of a single observation. In order to handle multiple observations, Kautz defines two logical operations, mc-entailment and imc-entailment. (We introduced these at the proof level in Section 2.1.5.)

At the algorithm level, Kautz provides two routines, match-graphs and group-observations, to build up candidate plans for multiple observations. They do so by combining explanation graphs, or e-graphs for singleton observations. E-graphs were introduced in Section 3.3.3. In this section we describe how to interface what we have done with these routines for handling multiple observations.

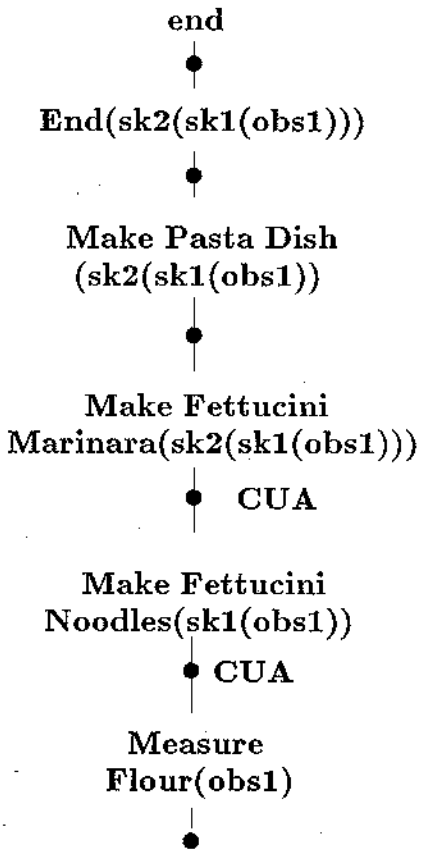


Figure 5.7: Proof Graph after a New Constraint is Added

Match-Graphs and Group-Observations

Kautz's e-graphs may include more than one observed event. E-graphs in general encode a disjunction of possible End events. In order to be included in the disjunction, an End event must include all of the observed events as components.

The first routine, called match-graphs, accepts as input two different e-graphs. Each e-graph contains a set of observed actions. Match-graphs tries to equate the End events in the first with the End events in the second. The equalities are propagated downwards through graphs, reducing disjunctions. Match-graphs produces as output an e-graph that contains the union of the given sets of observations, and the End events that, taken singly, can include all of these observations, if there are any. In this way the number of possibilities is reduced and the level of detail for each possibility is increased. If no End event can contain all of the observations, match-graphs fails.

There are several versions of the second algorithm, group-observations. Each applies a different strategy to find groupings of the observations. It uses match-graphs to decide what groupings make sense. For instance, the mc-entailment version would attempt to propose the minimal number of End events. Thus the best situation is one where all of the observed events can be seen as components of exactly one End event. In plan recognition this corresponds to proposing one plan that includes all of the observations. Failing that, it is best to propose two End events such that each observation is a component in at least one of these. Let n be the smallest number of End events required to include all of the observations. The theory of mc-entailment sanctions the (very weak) conclusion that the observations are grouped into n sets in one of the many ways that this is possible.

An mc-entailment version[4] of group-observations would apply match-graphs to build up each of these groupings. Consider the example of Figure 2.1. Suppose there are three observations *GetGun*, *GoToWoods* and *GoToBank*. In this case there is no e-graph containing all three, although there is an e-graph for each single observation. Then this version of group-observations would call match-graphs to attempt to match the e-graphs for *GetGun* and *GoToWoods*, so it could partition the observations into the sets $\{GetGun, GoToWoods\}$ and $\{GoToBank\}$. It would also try to match the e-graphs for *GoToWoods* and *GoToBank* to build the partitions $\{GetGun\}$ and $\{GoToWoods, GoToBank\}$. Finally it would try to match the e-graphs for *GetGun* and *GoToBank*, to build the partition for $\{GoToWoods\}$ and $\{GetGun, GoToBank\}$. For each of these calls to match-graphs that succeeded, the corresponding e-graphs are proposed to explain the observations. In this case the first call succeeds, as the *GoHunting* plan includes *GetGun* and *GoToWoods*. The last call also succeeds since *RobBank* plan includes *GetGun* and *GoToBank*. The second call fails since no single plan includes *GoToBank* and *GoToWoods*. The conclusion would be either (1) the e-graph for *GetGun* and *GoToWoods* and the e-graph for *GoToBank* or (2) the e-graph for *GetGun* and *GoToBank* and the e-graph for *GoToWoods*.

Multiple Observations with Assimilation

We propose to handle multiple observations in cases where new information may need to be assimilated. We want to make use of Kautz's algorithm for multiple

[4]In fact, Kautz does not provide an mc-entailment version of group-observations. This discussion was provided as a illustration of how group-observations would work in this general case. Instead Kautz provides more psychologically plausible and less expensive versions based on imc-entailment.

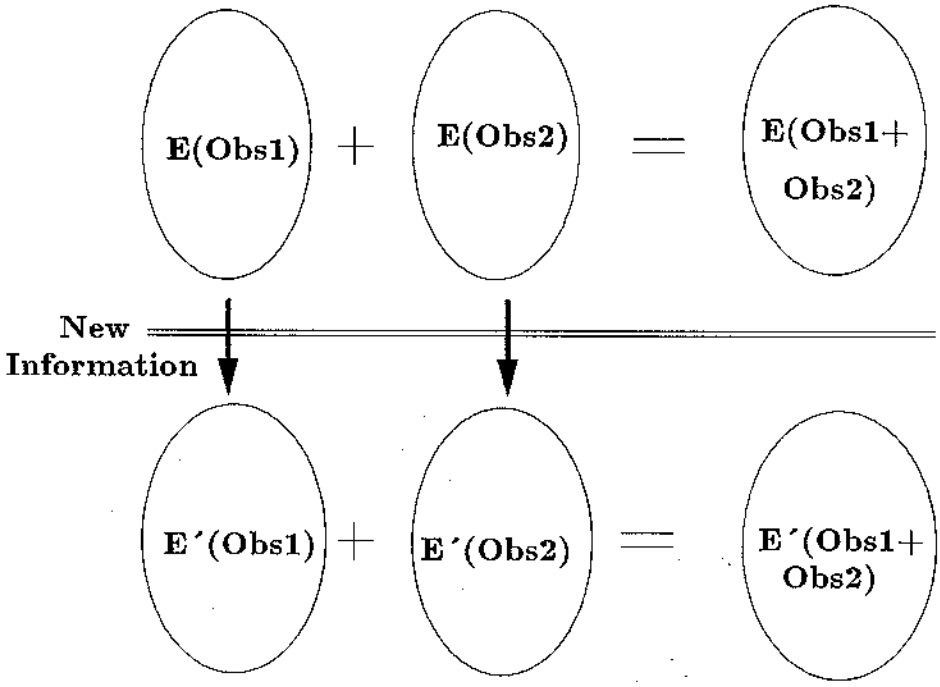


Figure 5.8: Assimilation with multiple observations

observations. The algorithm Extract (Appendix C.3.1) provides us with negated ancestor proof graphs. These are proofs of $\exists x.End(x)$, so there is a natural conversion to the e-graphs of Kautz, used in his multiple observation algorithms.

The solution we sketch below is for the case where Obs_1 and Obs_2 have both been processed, using our single observation algorithm. Now, new information from the oracle arrives. When it is necessary to assimilate new information into the e-graph for the combination of observations, we assimilate the new information to the e-graphs for all of the singleton observations separately. Then we use group-observations to apply match-graphs to build up the e-graphs for the new combinations.

For example, let $E(Obs_1)$ and $E(Obs_2)$ represent the e-graphs for Obs_1 and Obs_2, respectively. Then Figure 5.8 shows how new information would be assimilated into the e-graph $E(Obs_1 + Obs_2)$ for Obs_1 and Obs_2 taken together. The grey arrows represent assimilation steps. The +,= represent a call to match-graphs. The figure shows that we assimilate into the singleton e-graphs first and then attempt to match the new e-graphs.

Consider the three-observation example on page 123. Suppose new information has not affected the e-graph for *GetGun* and *GoToWoods*, but it has affected the e-graph for *GoToBank*. For example, the new information could be a constraint that the bank must be open. We want the conclusion to reflect the new information, so we make whatever calls to match-graphs are required to repair the conclusion. One new call is needed for conclusion(2), to match the graphs for *GetGun* and *GoToBank*. The e-graph combining *GetGun* and *GoToWoods* (part of conclusion (1)) is not affected. Therefore we have saved the work of calling match-graphs for this case. (In Section 6.2.1 we speculate on whether the truth maintenance system should be extended to assimilate directly into the combinations of observations.)

Here is a general description of the illustration in the previous paragraph. Our truth maintenance method does not allow us to assimilate new information directly into the e-graphs for more than one observation. Even so, we can identify by inspection the combinations of End events that are not affected by the new information. An e-graph explaining a combination of observations is not affected by the new information if the e-graphs for each of the singleton observations is not affected. A e-graph for a singleton observation is affected only if either of two conditions are met: (1) Have any closure axioms that were used in that e-graph been invalidated by the new information? (2) Have any of the new axioms the result from the new information resulted in additions to the search space for the e-graph? If both of

these are answered no for all of the e-graphs of the singleton observations then the e-graph for the combination of observations is unaffected. Unaffected e-graphs of combinations need not be recomputed, so the work that was done by match-graphs to combine these observations is saved.

We can now produce a plan recognition algorithm which accepts multiple observations and new plan information from the oracle. The complete algorithm is given in Appendix C.3.3.

Chapter 6

Conclusion

6.1 Contributions

We claim contributions to three open problems in the literature.

6.1.1 Recognizing Novel Plans in Plan Recognition

Kautz remarks

> Every day new kinds of events occur, yet they do not baffle us. An intelligent agent cannot rely only on a recognition system; it must contain a learning component as well.[16]

We proposed an architecture for dealing with novel kinds of events that appeals to an external source of information about those events. The role of the external source, or oracle, may in fact be provided by a third-party expert, an automated learning system, or the agent himself, if the plan recognition setting allows a dialog between the plan recognizer and the agent.

Our architecture is capable of candidate assimilation, which is defined as incorporating the new information with the candidates that are proposed to represent the agent's plan. The most straightforward way of ensuring that the candidates are up-to-date with the new information is to eradicate the old candidate plans and recompute the new ones based on both the old and new information. Our approach is less drastic; we assimilate the new information with the candidates, and repair them to conform to the new information.

Information from the oracle is assimilated into the candidate plans by way of the plan library. So, as new information is provided by the oracle a permanent record of it is kept in the library. Kautz's theory of plan recognition depends on a set of closure axioms that are based on assuming that the plan library is complete. We have applied our assimilation methods to Kautz's theory. This new information contradicts some of these completeness assumptions, so we must remove the invalidated closure axiom and all of the effects it has had. Also, to allow plan recognition to continue, some new closure axioms must be computed.

One problem that must be addressed when building large plan recognition systems is building and managing the large plan libraries that will be needed. This architecture can help build a plan library incrementally from actual situations. If an expert third-party is acting as the oracle, by using our architecture we can transfer information from the expert to the plan library and drive transfer by a number of actual situations. As the situations arise, the expert tells our system what is needed to recognize the situation, our system recognizes it and collects the information in the library for future similar situations.

6.1.2 Redundancy in Automated Theorem Proving

Problem #6 in "Automated Reasoning : 33 Basic Research Problems"[34] states

> What strategy can be employed to deter a reasoning program from deducing a clause already retained, or from deducing a clause that is a proper instance of a clause already retained?

Although our procedures have not computed clauses as such, the question still stands. We have provided a solution to the first problem, avoiding deducing a result already retained, in the setting of linear resolution. The redundancy we avoid arises in the MESON proof format when proving a literal by arguing by cases. MESON considers in turn each case with the exclusion of the others. Thus for n cases there are n deductions of this type that can be computed.

We showed that it is necessary to build only one of these deductions; the others are redundant. We introduced the foothold format that adds a condition to the MESON format. The new condition eliminates all but one of these cases.

When a proof requires several different instances of reasoning by cases, the number of possible proofs grows with the product of the number of cases in each instance. This leads to an exponential growth in the number of proofs. Again, the foothold format eliminates all but one of the cases in each instance, so only one proof is admitted.

We reduced redundancy by disallowing part of the search space. Because the search space is smaller it can be searched in less time. For some examples the speedup is exponential in the size of the problem. Our experimental results are encouraging since they suggest that more complex examples benefit more from our restriction.

Our procedure is based on a condition that adds a negligible cost at runtime to a proof procedure that uses the MESON proof format. The clauses are preprocessed and literals in their contrapositive forms are given labels from the set $\{-1, 0, +1\}$. The time to preprocess is on the same order as the time to compute the contrapositives.

Finally, the foothold condition is simple to compute. A procedure that performs ancestor search to compute proofs in the MESON format can take advantage of the reduced redundancy of the foothold format with a minimal change to the procedure.

6.1.3 Non-Horn Clauses and Truth Maintenance

de Kleer's basic ATMS has become a widely used tool in artificial intelligence problem solvers, where Horn clauses are sufficient to state the problem. de Kleer's extended ATMS provides the truth maintenance capabilities to problem solvers that need to use non-Horn clauses. The technique it employs is based on hyperresolution. de Kleer claims that in some circumstances the hyperresolution procedures are too expensive to employ[7], and they are difficult to integrate[8]. In addition they apply only when the clauses are in a restricted form; extra encoding is required to put them in this form, and the encoding adds assumptions, which adds work to the truth maintenance processing.

To address these problems we provide a truth maintenance system that integrates well with the basic ATMS algorithms, that does not require special encoding, that allows the problem solver to specify for which literals complete explanations are needed and for which they are not. Most importantly, it can take advantage of the foothold refinement, described in the previous section, to reduce much of the redundant work.

6.2 Future Work

6.2.1 Work in Plan Recognition

Direct Assimilation and Multiple Observations

We have shown how to implement c-entailment from one observation with a truth maintenance system, and how this is used for novel plan recognition from a single observation, as well as multiple observations. However, it may be possible to do what is represented in Figure 6.1, to propose candidate plans based on a combination of observations, and then assimilate the new information directly into those candidate plans.

A difficult issue arises. Suppose we have a number of observations $\{Obs_1, \ldots, Obs_n\}$ and we want to assimilate new information into an e-graph for all of these observations. Suppose the new information results in an inconsistency – i.e. there is no candidate plan which contains all of $Obs_1, \ldots, Obs_n$. Then we will have to re-group the observations, according to the particular strategy that GroupObservations encodes. If the TMS is used to do this regrouping, then it will need to encode this specific strategy. It is not clear that this should be the responsibility of the TMS. It is contrary to the original intention of truth maintenance systems to encode application-specific strategies within them. In addition, this solution would probably increase the complexity of the TMS.

Interfacing with the Oracle

In Section 1.6 we outlined a few of the issues that arise when responsibilities are assigned to the plan recognizer and the oracle for the tasks of deciding what new

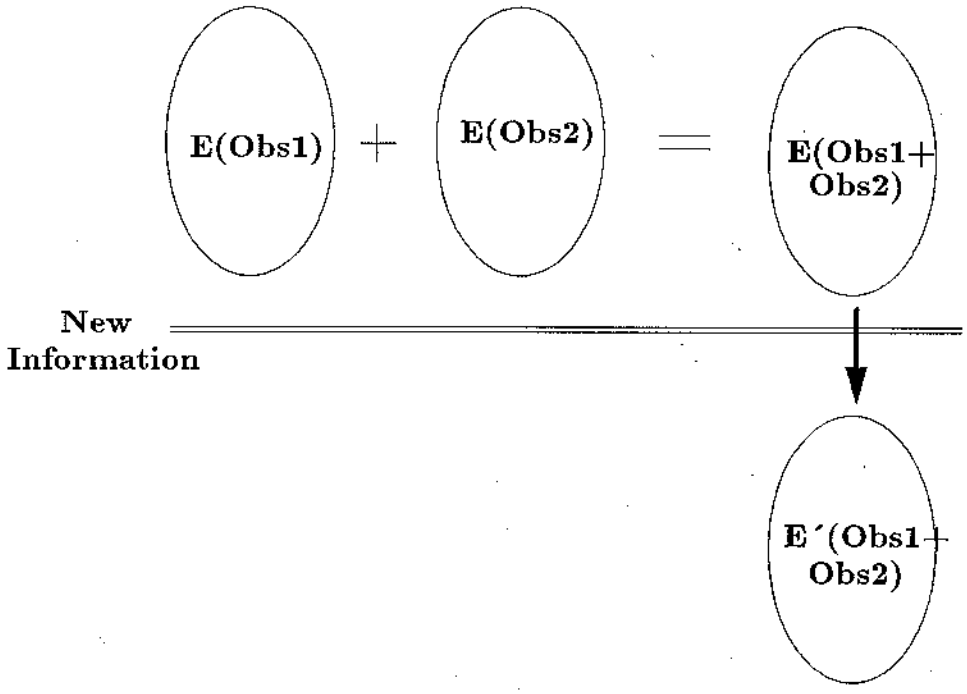


Figure 6.1: Direct Assimilation with multiple observations

information is needed and whether it is correct. Another issue is deciding when the new information is needed. One scenario has the plan recognizer accepting new information only when it is unable to recognize a plan. This may be too restrictive since it is possible that new information could be available that would improve the plan recognizer's conclusion. So it is important for the plan recognizer to be open to new information. It would be useful to investigate an oracle that takes the initiative and provides new information. This oracle is able to interrupt at any time with new information. As a result the plan recognizer could be prevented from ever computing a plan, so restrictions on the dialog between the oracle and the plan recognizer would be needed.

Domain Knowledge in the TMS

Although we used the TMS only to store axioms from the plan library, in principle one could employ the TMS to reason about the domain as well. This would apply if the plan recognizer's knowledge about the agent's world is increasing, and should be assimilated into the recognized plan.

Some preliminary investigation has been done into assimilating new temporal knowledge into a temporal network [32]. In that work it is hypothesized that a constraint can be removed from a temporal network such that the remaining network can be repaired, and that this operation can be done faster than recalculating the network.

Abductive Plan Recognition and Truth Maintenance

Truth Maintenance systems have been applied to plan recognition systems in the abductive style by Charniak and Goldman[4]. The purpose filled by the truth

maintenance system in that work was to furnish the plan recognition system with a reasoning system that supported multiple contexts, so that the disjunction of possible plans could be represented. A possible avenue for future work is to use truth maintenance to do assimilation within an abductive plan recognition system. Algorithms in this case would be different from those presented in this thesis. For example, our searcher is tied to Kautz's style of plan recognition; a different searcher would be needed for the abductive case.

Assimilation and Constructive Plan Recognition

It would be useful to explore the use of our assimilation procedure in the work of Litman and Goodman[18]. The CHECS system, mentioned in Section1.2 may be described as a novel plan recognition system where the user is the oracle. CHECS incorporates a "constructive" plan recognition component. In constructive plan recognition, if the actions are not recognized, the system still allows the user to continue. The system enters a phase of constructing and logging the new plan library information from the user's input. During this phase, plan recognition is not done, so the CHECS system cannot apply its domain knowledge to the situation at hand. This means it does not provide the assistance to the user that it normally provides. If CHECS were to apply a direct assimilation method (such as the one we provide in this thesis) to bring this new information to bear on the candidate plan so far, CHECS could continue to provide feedback and assistance to the user in the constructive phase.

Repair versus Recalculation

One issue to explore in greater detail is the effectiveness of the repair method of assimilation compared with the recalculation method. Exploring this issue warrants experimental results. Some of the factors that should be considered are: the size of the candidate plans, the amount of change in the candidate plans, and the number of times assimilation occurs.

For small candidates that are inexpensive to compute, it may be that the overheads of the various parts of our solution, building the and/or graphs, building the backedge graphs, extracting the proof graph, etc., will be more expensive than just recalculating the small plan from scratch. But for additions to a large plan library, used under conditions where the candidates are large or contain many disjuncts, the candidates are expensive to compute. In this case candidate repair should be worthwhile. If the new information has had no effect on the candidate, a condition we can detect using our assimilation method (i.e. when C^- and C^+ are empty after AssimSearch) then no recalculation is required. We expect that detecting this condition, which is quite simple in our system, will not be as expensive as recomputing the candidate. If the new information affects a relatively small portion of the candidate, then we hypothesize that replacing the small portion will be faster than recomputing the entire candidate.

6.2.2 Work in Automated Reasoning

Redundancy in other proof formats

We have removed redundancy in the MESON proof format. Can the same idea be used to remove redundancy in other proof formats? If reasoning by cases arises in

any form in these other formats, it may be possible to provide a method that breaks up the symmetry of considering the cases in different orders. Linear resolution with framed literals[3], a proof format closely related to the MESON format, would be a good format to investigate next.

Absolutely Irredundant Proof Procedures

For some examples of theorem proving problems, we have removed all of the redundancy, in the following sense: Given a set P of clauses and a literal g, suppose we have constructed a foothold proof of g. Let Q be the subset of P that contains all and only the clauses used in this proof. Then having removed all of the redundancy means there is exactly one foothold proof of g from Q.

For other problems, such as the example on page 78 there are still an exponential number of foothold proofs of g from Q. An ideal solution to the redundancy problem would be a proof format that guarantees there is exactly one. It should also depend on a condition that can be checked quickly, and can be applied to partial proofs so that redundancy can be detected and avoided before the complete proof is generated.

Appendix A

Algorithms and Proofs for Chapter 3

A.1 Library Algorithms

A.1.1 Library Closure Algorithms

procedure CloseLibrary($H_E, H_{EB}, H_a, H_D, H_G$)
$EXA := \{\}$
$CUA := \{\}$
$DJA := \{\}$
$Com := \{\}$
% Construct Com
$Com := \{(E1, E2) : \exists E3.E1$ abstracts$^* E3 \wedge E3$ abstracts$^* E3\}$
% Construct DJA
for each $E \in H_E$ **do**
    if there is an axiom $(\forall x.E(x) \supset E_{spec}(x)) \in H_A$ **then**
        $Spec(x) := \{E_{spec}(x)$: there is an axiom
        $(\forall x.E_{spec}(x) \supset E(x)) \in H_a\}$
        if $Spec(x) \neq \{\}$ **then**

Construct a disjunction $D(x)$ of all of the elements of $Spec(x)$
$EXA := EXA \cup \{\forall x.E(x) \supset D(x)\}$
end if
end if
end for each
% Construct DJA
for each $E1 \in H_E$ **do**
for each $E2 \in H_E$ **do**
if $(E1, E2) \notin Com$ **then**
$DJA := DJA \cup \{\forall x.\neg E1(x) \vee \neg E2(x)$
end if
end for all
end for all
% Construct CUA
for each $E \in H_E$ **do**
$Com_E := \{E_1 : (E, E_1) \in Com\}$
$Uses(x) := \{(\exists y.E_{use}(y) \wedge f(y) = x) : E_{step} \in Com_E$ and there is an axiom $(\forall z.E_{use}(z) \supset E_{step}(z))$ in $H_D\}$
if $Uses(x) \neq \{\}$ **then**
Construct a disjunction $D(x)$ of all of the elements of $Uses(x)$
$CUA := CUA \cup \{\forall x.E(x) \vee End(x) \vee D(x)\}$
end if
end procedure

A.1.2 Library Assimilation Algorithms

The event hierarchy H is a tuple $(H_E, H_{EB}, H_A, H_D, H_G)$. Com is a binary relation between events (members of H_E).

LibraryAssimilate

procedure LibraryAssimilate(Info)
% New information from the Oracle is to be assimilated into $cl(H)$.
% The new information, $Info$, is either a new constraint, a new
% step in a decomposition or a new abstraction.

% LibraryAssimilate calculates C^- and C^+ such that
% $cl(H^+) = cl(H) \setminus C^- \cup C^+$.
% Initialization
set $H_A^+, H_D^+, CUA^-, CUA^+, DJA^-, DJA^+, EXA^-, EXA^+$ to {}
case Info **of:**
Constraint: **call** NewConstraint(Info)
Step: **call** NewStep(Info)
Abstraction: **call** NewAbstraction(Info)
end case
set H_A to $H_A \cup H_A^+$
set H_D to $H_D \cup H_D^+$
set C^- to $CUA^- \cup DJA^- \cup EXA^-$
set C^+ to $H_A^+ \cup H_D^+ \cup CUA^+ \cup DJA^+ \cup EXA^+$
end procedure

NewEvent

procedure NewEvent(E)
% A new event is disjoint from every other event.
for each $E_{old} \in H_E$ **do**
$DJA^+ := DJA^+ \cup \{\forall x.\neg E_{old}(x) \supset \neg E(x)\}$
% It is a basic event type, and it is compatible only with itself.
$H_E := H_E \cup \{E\}$
$H_{EB} := H_{EB} \cup \{E\}$
$Com := Com \cup \{(E, E)\}$
end procedure

NewConstraint

procedure NewConstraint($\forall x.E(x) \supset \kappa$)
% A new constraint is added to the decompositions, but it does not
% affect the structure of the hierarchy unless the event is new.
$H_D^+ := H_D^+ \cup \{\forall x.E(x) \supset \kappa\}$
if $E \notin H_E$ **then call** NewEvent(E)
end procedure

NewStep

procedure NewStep($\forall x.E(x) \supset E_{step}(R(x))$)

% A new step is added to the decompositions.
(1) $H_D^+ := H_D^+ \cup \{\forall x.E(x) \supset E_{step}(R(x))\ \}$
% A new step needs to be added to the uses for every compatible event.
(2) **for each** E_{com} compatible with E_{step} **do**
% add $E(y) \wedge R(y) = x$ to the uses for E_{com}
if there is a CUA axiom with $E_{com}(x)$ on the left-hand-side **then**
Let the $Uses(x)$ be the set of disjuncts on the right-hand side of this axiom
Let $NewUses(x) = Uses(x) \cup \{\exists y.(E(y) \wedge R(y) = x)\}$
Construct a new axiom with $E_{com}(x)$ on the left hand side and all the members of $NewUses(x)$ as disjuncts on the right-hand-side
$CUA^- := CUA^- \cup$ {the old axiom}
$CUA^+ := CUA^+ \cup$ {the new axiom}
end if
(3) **if** $E_{step} \notin H_E$ **then call** NewEvent(E_{step})
if $E \notin H_E$ **then call** NewEvent(E)
end procedure

NewAbstraction

procedure NewAbstraction($\forall x.E_{spec}(x) \supset E_{abs}(x)$)
% A new abstraction is added to the set of abstractions.
(1) $H_A^+ := H_A^+ \cup \{\forall x.E_{spec}(x) \supset E_{abs}(x)\}$
% A E_{spec} is added to the list of specialization for E_{abs}
(2) **if** there is an EXA axiom with $E_{abs}(x)$ on the left-hand-side **then**
Let the axiom be $\forall x.E_{abs}(x) \supset E_1(x) \vee \ldots \vee E_n(x)$
$EXA^- := EXA^- \cup \{\forall x.E_{abs}(x) \supset E_1(x) \vee \ldots \vee E_n(x)\}$
$EXA^+ := EXA^+ \cup \{\forall x.E_{abs}(x) \supset E_1(x) \vee \ldots \vee E_n(x) \vee E_{spec}(x)\}$
else
% there was no EXA axiom for E_{abs}, so one is added
$EXA^+ := EXA^+ \cup \{\forall x.E_{abs}(x) \supset E_{spec}(x)\}$
end if
(3) $A := \{E \in H_E : E$ abstracts* $E_{abs}\}$
% The events in A transitively abstract E_{abs}.
$B := \{E \in H_E : (E, E_{spec}) \in Com\}$
% The events in B are compatible with E_{spec}.
for each $E_A \in A$ **do**
for each $E_B \in B$ **do**
% E_A is has become compatible with E_B

if $(E_A, E_B) \notin Com$ **then**
% They were not compatible, so they were disjoint
$DJA^- := DJA^- \cup \{\forall x.\neg E_A(X) \supset \neg E_B(x)\}$
$Com := Com \cup \{(E_A, E_B), (E_B, E_A)\}$
end if
end for each
end for each
(4) **if** there is an axiom in CUA with $E_A(x)$ on the left-hand-side
and there is an axiom in CUA with $E_B(x)$ on the left-hand-side **then**
Let $A_{uses}(x)$ be the set of disjuncts that appear on the right-hand-side of the CUA axiom for $E_A(x)$
Let $B_{uses}(x)$ be the set of disjuncts that appear on the right-hand-side of the CUA axiom for $E_B(x)$
$AB_{uses}(x) := A_{uses}(x) \cup B_{uses}(x)$
Construct a new disjunction $D(x)$ from the all of the elements of $AB_{uses}(x)$
for each $E_A \in A$ **do**
$CUA^- := CUA^- \cup$ {the old CUA axiom for $E_A(x)$}
$CUA^+ := CUA^+ \cup \{\forall x.E_A(x) \supset D(x)\}$
end for each
for each $E_B \in B$ **do**
$CUA^- := CUA^- \cup$ {the old CUA axiom for $E_B(x)$}
$CUA^+ := CUA^+ \cup \{\forall x.E_B(x) \supset D(x)\}$
end for each
end if
(5) **if** $E_{spec} \notin H_E$ **then call** NewEvent(E_{spec})
if $E_{abs} \notin H_E$ **then call** NewEvent(E_{abs})
end procedure

A.2 Candidate Algorithms

A.2.1 Searching Algorithm

Search

procedure Search $(E_1(Obs) \vee \ldots \vee E_n(Obs))$
% Input : an observed event, Obs in a disjunction that describes
% all of the possible types that this event might be.

% Output : *Visited*, a set of literal that have been reached in the search
% and *Axioms*, a set of ground clauses that identify the space searched
% in which a proof for the *End* event may exist.
% Initialize
$Visited := \{\}$
$Axioms := \{E_1(Obs) \vee \ldots \vee E_n(Obs)\}$
for each $i = 1, \ldots n$ **do**
    Search1($E_i(Obs)$, yes)
end for each
end procedure

Search1: Search from one event type

procedure Search1($E(M)$, ConsiderSpec)
% Input : an event M and its type E from which searching is to proceed
% and ConsiderSpec, a yes/no value used to eliminate "is-a" plateaus
% Search1 adds to *Visited* and *Axioms*
if $E(M) \notin Visited$ **then**
    for each Constraint κ on $E(M)$ **do**
        Try to prove $\neg\kappa$ (with a general purpose or special purpose theorem prover)
        if the proof succeeds **then**
            $Axioms := Axioms \cup \{E(M) \supset false\}$
            Exit from Search1
        end if
    end for each
    $Visited := Visited \cup \{E(M)\}$
    if $E \neq End$ **then**
        % Consider the Uses of $E(M)$
        if there is an axiom $(\forall x.E(x) \supset End(x) \vee Uses(x)) \in CUA$ **then**
            Let $NewAssn$ be a new assumption and associate it with this CUA axiom.
            Let $NewSk$ be a new Skolem function and associate it with this axiom.
            $Uses(x)$ is a formula of the form $\exists y.(E_1(y) \wedge f_1(y) = x) \vee \ldots \vee (E_n(y) \wedge f_n(y) = x)$
            Construct a new formula, $Uses1$, which is $E_1(NewSk(x)) \vee \ldots \vee E_n(NewSk(x))$
            $Axioms := Axioms \cup \{NewAssn \wedge E(M) \supset Uses1(M)\}$

```
            for each disjunct E_i(NewSk(M)) ∈ Uses1(M) do
                Search1(E_i(NewSk(obs)), yes)
            end for each
        end if
        % Consider the Direct Abstractions of E(M)
        for each axiom of the form (∀x.E(x) ∪ E_abs(x)) ∈ H_E do
            Axioms := Axioms ∪ {E(M) ⊃ E_abs(M)}
            Search1(E_abs(M), no)
        end for each
        % Consider the Specializations of E(M)
        if ConsiderSpec = yes then
            if there is an axiom (∀x.E(x) ⊃ E_1(x) ∨ ... ∨ E_n(x)) ∈ EXA
            then
                Let NewAssn be a new assumption and associate it with
                this EXA axiom
                Axioms := Axioms ∪ {NewAssn ∧ E(M) ⊃ E_1(M) ∨ ... ∨
                E_n(M)}
                for each i = 1,...n do
                    Search1(E_i(M), yes)
                end for each
            end if
        end if
    end if
end if
end procedure
```

A.2.2 Candidate Assimilation Algorithm

```
procedure AssimSearch(EXA⁻, CUA⁻, EXA⁺, CUA⁺, H_A^+, H_D^+)
% Assimilate EXA⁻ and CUA⁻
for each axiom C in EXA⁻ ∪ CUA⁻ do
    for each assumption A associated with C do
        Axioms := Axioms ∪ {¬A}
    end for each
end for each

% Assimilate EXA⁺
for each axiom (∀x.E(x) ⊃ E_1(x) ∨ ... ∨ E_n(x)) in EXA⁺ do
    for each member E(M) of Visited do
```

for each $i = 1, \ldots, n$ **do**
Search1$(E_i(M), yes)$
end for each
end for each
end for each

% Assimilate CUA^+
for each axiom $\forall x.E(x) \supset End(x) \vee Uses(x)$ in CUA^+ **do**
Let $OldSk$ be the old Skolem function associated with the old version of this CUA axiom

end for each

% Assimilate H_A^+
for each axiom $(\forall x.E_{spec}(x) \supset E_{abs}(x))$ in H_A^+ **do**
for each member $E_{spec}(M)$ of $Visited$ **do**
Search1$(E_{abs}(M), no)$
end for each
end for each

% Assimilate the constraints in H_D^+
for each axiom $\forall x.E(x) \supset \kappa$ in H_D^+ **do**
Try to prove κ
if the proof fails **then**
for each member $E(M)$ of Visited **do**
$Axioms := Axioms \cup \{E(M) \supset false\}$
end for each
end if
end for each
end procedure

A.3 Proofs

A.3.1 Completeness with Ground Clauses

The following results appear in [3]

Theorem 1 [appears as Theorem 4.1, p.48] Let S be a set of clauses that represents a (Skolem) standard form of a formula F. Then F is inconsistent if an only if S is inconsistent.

The symbol $\square$ is the literal that is assigned false in all interpretations.

Theorem 2 [appears as Theorem 7.2, p. 144] If C is an ordered clause in an unsatisfiable set S of ordered clauses and if S - {C} is satisfiable, then there is an OL-refutation from S with top ordered clause C.

Since an ordered clause is merely a clause upon which an arbitrary ordering of the literals has been imposed, and since an OL-refutation is a specific type of linear deduction of $\square$, we have the following corollary.

Corollary 2.1 If S is an unsatisfiable set of clauses, and S - {C} is satisfiable then then there is a linear deduction from C to $\square$ given S.

Theorem 3 Let $cl(H)$ be the closure of the abstraction/decomposition hierarchy H. Let Γ be a ground observation such that $cl(H) \cup \Gamma \models \exists x\, End(x)$. Suppose that $cl(H)_{sk} \not\models \exists x.End(x)$.

Then for each Skolemized form $cl(H)_{sk}$ of $cl(H)$ there exists a disjunction Ω of literals of the form $End(x)$, and a linear deduction from Γ to Ω given $cl(H)_{sk}$.

Proof

$cl(H) \cup \Gamma \models \exists x\, End(x)$

$\iff$ $cl(H) \cup \Gamma \cup \{\neg(\exists x\, End(x))\}$ is unsatisfiable.

$\iff$ $cl(H)_{sk} \cup \Gamma \cup \{\forall x\, \neg End(x)\}$ is unsatisfiable

by Theorem 1 where $cl(H)_{sk}$ is a Skolemized form of $cl(H)$

Since we assume that $cl(H)_{sk} \cup \{\forall x \neg End(x)\}$ is satisfiable, there exists a linear deduction from Γ to $\square$ given $cl(H)_{sk} \cup \{\forall x \neg End(x)\}$ by Corollary 2.1. Given this deduction, adjust it so that no resolution against $\forall x.\neg End(x)$ is performed, but instead the End literals are left unresolved. Then the result is a linear deduction from Γ to some disjunction of End literals given $cl(H)_{sk}$. Let Ω be this disjunction of End literals. ∎

Theorem 4 Let $cl(H)_{sk}$ be the Skolemized closure of the abstraction / decomposition hierarchy H. Let Γ be an ground literal that represents an observation and let Ω be a disjunction of End events. If there is a linear deduction of Ω from Γ given $cl(H)_{sk}$ then every clause in the deduction is ground.

Proof The first formula of the deduction, Γ, is the observation which is ground. Let $C_1 = \Gamma$. To show that $C_2, \ldots, C_n$ are ground, proceed by induction. Suppose C_k is ground for all $k = 0, \ldots, i-1$. Then C_i, the resolvent of C_{i-1} with B_i is clearly ground if $B_i = C_j$ for $j < i$. By inspection each axiom in S is a disjunction of literals that have exactly one variable, and that variable is shared by all literals in the clause. Therefore by resolving one of these literals against a ground literal, all of the other literals also become ground. So C_i is ground. ∎

Appendix B

Proofs and Algorithms for Chapter 4

B.1 Proofs

Theorem 5 Let P be a consistent set of propositional clauses, and g a literal. $P \models g$ if and only if there exists a negative ancestor proof graph of g using clauses in P.

Proof ($\Rightarrow$ by induction on the number of clauses in P)
We claim if A is a sequence of nodes representing a path of tree edges, B is the set of all literals whose complements appear in the nodes of A, $P \cup B \models g$, $P \cup B$ is consistent, and G is a node containing g then there is an NA1 proof graph of G with respect to the ancestor path A. If this claim is true then by setting A to the empty sequence we have a negative ancestor proof graph of g, and the result is obtained.

If $P = \{\}$ then $P \cup B \models g$ means $g \in B$, so there is a node in A containing $\overline{g}$, which means there is an NA1 proof, using (case a) of Definition 4.3.2. Let P have

$k+1$ clauses. If $g \in B$ then there is an NA1 proof, also by (case a). Otherwise we construct P', a subset of P such that $P' \cup B \cup \{\overline{g}\}$ is inconsistent, and for all proper subsets P'' of P', $P'' \cup B \cup \{\overline{g}\}$ is consistent. (P' can be obtained by exhaustively considering all possible subsets of P.) Assume P' does not contain a clause C which mentions g. Then since $P' \cup B$ is consistent and $g \notin B$, $P' \cup B \cup \{\overline{g}\}$ is consistent, which is false. Thus there is a clause $C \in P$ and without loss of generality assume $C = g \vee g_1 \vee \ldots \vee g_n$. Since $P' \setminus \{C\}$ is a proper subset of P, $P' \setminus \{C\} \cup B \cup \{\overline{g}\}$ is consistent. If there is a g_j in C such that $P' \setminus \{C\} \cup B \cup \{\overline{g}\} \cup \{g_j\}$ is consistent, then $P' \cup B \cup \{\overline{g}\}$ is consistent, which is false. Thus $P' \setminus \{C\} \cup B \cup \{\overline{g}\} \models \overline{g}_j$ for all j. Then by induction, setting P in the claim to $P' \setminus \{C\}$, setting B to $B \cup \{\overline{g}\}$, and setting g to $\overline{g}_j$ if we construct nodes G_j containing g_j then there is an NA1 proof graph of each G_j with ancestor path (A, G). These graphs can be combined by constructing tree edges from G to each G_j to form an NA1 proof of G with respect to the ancestor path A.

($\Leftarrow$ by induction on the height of the tree (V,T))
We claim that if there is an NA1 proof graph of some node G with respect to the ancestor path A, then $P \cup B \models g$, where B is the set of literals whose complements appear in A and g is the literal in G. If the claim is true then by setting A to the empty path, B becomes $\{\}$ and $P \models g$.

Let the tree (V,T) in the NA1 graph have height 0. Then either there is a back edge from G to a node in A so $g \in B$, or there is a contrapositive rule g with no right hand side so $g \in P$. In either case $P \cup B \models g$. Suppose the tree has height $k+1$. If there is a back edge from G to a node in A then again $P \cup B \models g$. Otherwise there is a contrapositive rule $g \leftarrow g_1 \ldots g_n$, and for all $i = 1, \ldots, n$ there is a node G_i containing g_i and an NA1 proof graph of G_i with respect to ancestor path (A, G). These smaller graphs contain trees with height at most k, so by induction

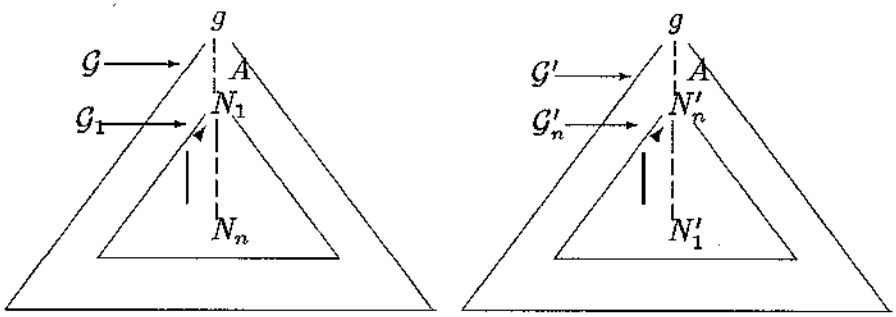


Figure B.1: From $\mathcal{G}$ with a backedge (N_n, N_1) to $\mathcal{G}'$ with a back edge (N'_1, N'_n)

$P \cup B \cup \{\overline{g}\} \models g_i$. Since $P \models g \leftarrow g_1 \ldots g_n$ it follows that $P \cup B \cup \{\overline{g}\} \models g$. Thus $P \cup B \cup \{\overline{g}\}$ is inconsistent so $P \cup B \models g$.∎

The following lemma is used in the proof of Theorem 7.

Lemma 6 Let $\mathcal{G}$ be a negative ancestor proof graph using clauses from P that contains a path of tree edges represented by the sequence of nodes $N_1, \ldots, N_n$, and a back edge (N_n, N_1). For $i = 1, \ldots, n$, let N_i contain the literal h_i. Then there exists a negative ancestor proof graph $\mathcal{G}'$ using clauses from P, that contains a path of tree edges $N'_n, \ldots, N'_1$, where N'_i contains $\overline{h}_i$, and a back edge (N'_1, N'_n).

Proof Let $\mathcal{G}_1$ be the NA1 proof graph of N_1, the node containing h_1. We shall construct $\mathcal{G}'$ by replacing $\mathcal{G}_1$ in $\mathcal{G}$ with $\mathcal{G}'_n$, an NA1 proof graph of N'_n, the node containing $\overline{h}_n$. (See Figure B.1.) Note that $h_1 = \overline{h}_n$ since there is a back edge (N_n, N_1). Let A be the ancestor path from the root of $\mathcal{G}$ to N_1. We construct $\mathcal{G}'_n$ by inductively constructing an NA1 proof graph of N'_i with ancestor path $(A, N'_n, \ldots, N'_i)$,

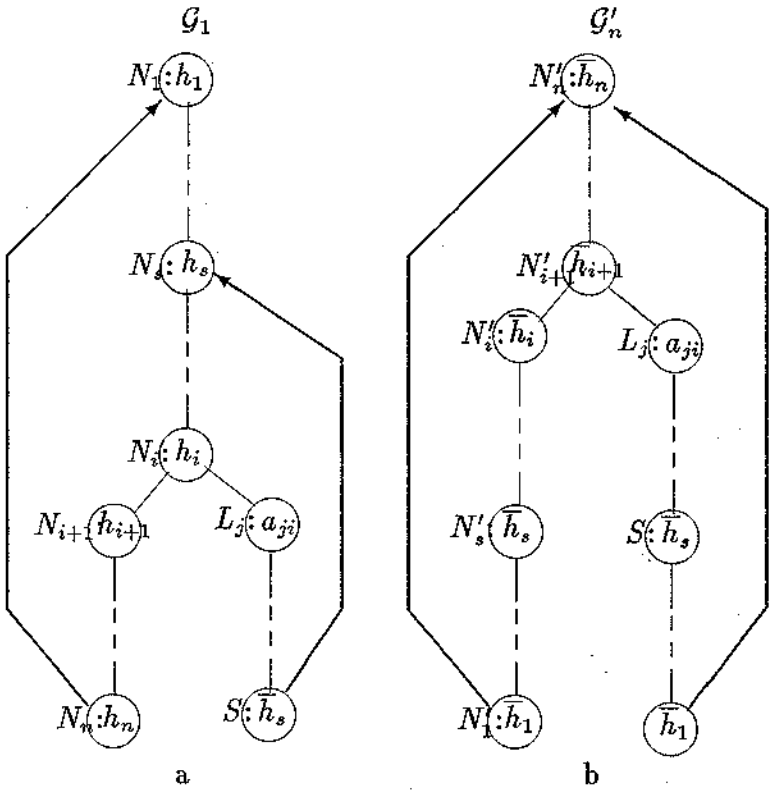


Figure B.2: Constructing $\mathcal{G}'_n$ from $\mathcal{G}_1$

such that there is no back edge leading to any one of $N'_{n-1}, \ldots, N'_i$. (See Figure B.2.)

The NA1 proof graph of N'_1 contains the ancestor path $(A, N'_n, \ldots, N'_2)$, and by (case a) a back edge (N'_1, N'_n), since N'_1 contains $\overline{h}_1$ and N'_n contains h_1. There is no back edge to N_1 since N_1 has no descendants. To construct the NA1 proof graph of N'_{i+1} with ancestor path $(A, N'_n, \ldots, N'_{i+2})$, note that there is a clause in P from which the contrapositive rule

$$h_i \leftarrow h_{i+1} a_{1i} \ldots a_{mi}$$

can be constructed, without loss of generality. Therefore the contrapositive rule

$$\overline{h}_{i+1} \leftarrow \overline{h}_i a_{1i} \ldots a_{mi}$$

can also be constructed. So we apply (case b) and construct for each $j = 1, \ldots, m$ a node L_j containing a_{ji}, a tree edge (N'_{i+1}, L_j) and an NA1 proof graph of L_j with ancestor path $(A, N'_n, \ldots, N'_{i+1})$. We have already constructed by induction an NA1 proof graph of N'_i with ancestor path $(A, N'_n, \ldots, N'_{i+1})$.

To construct the NA1 proof graph of L_j, note that part of $\mathcal{G}_\infty$ is an NA1 proof graph of L_j with ancestor path $(A, N_1, \ldots, N_i)$. (See Figure B.2a.) Make a copy of that graph, replacing the ancestor path $(A, N_1, \ldots, N_i)$ with the ancestor path $(A, N'_n, \ldots, N'_{i+1})$. Any back edge (S, N_s) for $1 \leq s < i$ in the proof graph for L_j must also be removed since its destination has been removed. To construct a new proof graph for S note that by induction we have already constructed a proof graph for N_s, a node which also contains $\overline{h}_s$. This graph has an ancestor path $(A, N'_n, \ldots, N'_{s+1})$, but no back edges to any of $N'_{n-1}, \ldots, N'_s$. Make a copy of this graph, and in the copy give the name S to N_s. Since there were no back edges to $N'_{n-1}, \ldots, N'_s$, we can replace the ancestor path in the new graph with $(A, N'_n, \ldots, N'_{i+1}, L_j, \ldots, S)$. (See Figure B.2b.) This completes the construction of the proof graph for L_j. To complete the induction we point out that there are no back edges to N'_{i+1}.∎

Theorem 7 Let P be a consistent set of propositional clauses, and g a literal. Let F_P be a foothold rule set of P. $P \models g$ if and only if there exists a foothold proof graph of g from F_P.

Proof ($\Leftarrow$) Since the foothold proof graph is a negative ancestor proof graph, the result follows by Theorem 5.

($\Rightarrow$) Let $\mathcal{G}$ be a negative ancestor proof graph of g from P, provided by Theorem 5. According to Definition 4.5 label $\mathcal{G}$ with respect to F_P. If there are no negative back edges then this is a foothold proof graph, and we are done. Otherwise select a lowest node N that is the destination of some negative back edge. We shall construct a labelled NA1 proof graph of N that has no negative back edges leading to N or to any descendant of N. By selecting a new lowest node that is the destination of some negative back edge and repeating the construction of an NA1 proof graph of it with no negative back edges we can eliminate all negative back edges. This completes the construction.

To construct the NA1 proof graph of N as claimed, let $N_1 = N$ and select a back edge (N_n, N_1) with label $k < 0$. We know $k = \Sigma_{i=2}^{n} l_i$ where l_i is assigned by the foothold rule

$$h_i \leftarrow h_{i+1}^{(l_i)} a_{1i}^{(*)} \ldots a_{mi}^{(*)}$$

Apply Lemma 6 to (N_n, N_1) to replace that back edge with (N_1', N_n') with label $k' = \Sigma_{i=2}^{n} l_i'$ where l_i' is assigned by the foothold rule

$$\overline{h}_{i+1} \leftarrow \overline{h}_i^{(l_i')} a_{1i}^{(*)} \ldots a_{mi}^{(*)}$$

If h_i and h_{i+1} are both positive or both negative literals then in the ordering chosen in F_P either h_i comes before h_{i+1}, in which case $l_i = +1$ and $l_i' = -1$, or h_i comes after h_{i+1}, in which case $l_i = -1$ and $l_i' = +1$. If one of h_i and h_{i+1} is positive and the other is negative then $l_i = l_i' = 0$. In either case $l_i = -l_i'$ so $k = -k'$. Thus $k' > 0$. We have replaced a negative back edge with a positive one. We claim that we have not negated any positive backedge so that the number of positive backedges has strictly increased. By continuing to select a negative back edge and apply Lemma 6, we strictly increase the number of positive back edges. Since there is a finite maximum number of back edges we can eliminate all negative back edges

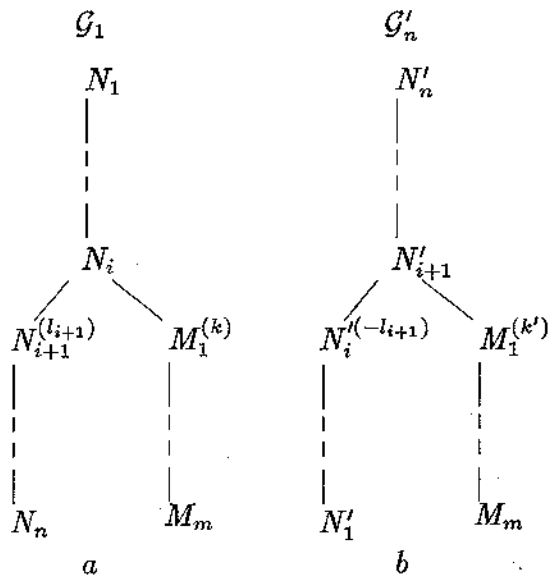


Figure B.3: The effect Lemma 6 on other back edges

in a finite number of steps.

It remains to show our claim that an application of Lemma 6 to a negative back edge does not negate any positive back edge to N_1. Let (M_m, N_1) be a positive back edge, and let the path of tree edges from N_1 to M_m be $N_1, \ldots, N_i, M_1, \ldots, M_m$, as in Figure B.3a. Let $X = \Sigma_{j=2}^{i} l_j$, let $X_N = \Sigma_{j=i+2}^{n} l_j$, let X_M be the sum of the labels of M_2 to M_m, let k be the label of M_1. After the application of Lemma 6 to

(N_1, N_n) we have the graph shown in B.3b, where k' is the new label for M_1.

Given	$X + l_{i+1} + X_N$	$<$	0
it follows	$X + l_{i+1} + X_N$	$\leq$	-1
so	X	$\leq$	$-l_{i+1} - X_N - 1$
Given	$X + k + X_M$	$>$	0
it follows	X	$>$	$-k - X_M$
Transitively	$-l_{i+1} - X_N - 1$	$>$	$-k - X_M$
so	$X_M - X_N$	$>$	$l_{i+1} + 1 - k$
Suppose	$X_M - X_N + k'$	$<$	0
Then	$X_M - X_N$	$\leq$	$-1 - k'$
Transitively	$-1 - k'$	$>$	$l_{i+1} + 1 - k$
so	$k - k' - l_{i+1}$	$>$	2

Since k, k' and l_{i+1} are restricted to values in $-1, 0, +1$ it follows that $k = 1, k' = -1$ and $l_{i+1} = -1$. Therefore in the clause in question the ordering chosen to build the foothold rules put the literal in N_i before the literal in N_{i+1} since $l_{i+1} = -1$. It put the literal in N_{i+1} before the literal in M_1 since $k' = -1$. And it put the literal in M_1 before the literal in N_i since $k = 1$. But this is absurd so the supposition that $X_M - X_N + k' < 0$ is false.∎

B.2 Computing Foothold Proofs in Prolog

```
% Propositional Foothold Proof Procedure

:- op(250, fx, (¬)).
```

```
prove(G) :- fh_prove(labelled_literal(G, _), [ ]).

fh_prove(labelled_literal(G, _), Anc) :-
        negate(G, Neg_G),
        member( labelled_literal(Neg_G, _), Anc),
        !, fail. % Cut is allowed when literals are propositions
fh_prove(labelled_literal(G, Label), Anc) :-
        ancestor_search(G, Label, Total, Anc),
        !, %Cut is allowed when literals are propositions
        Total > 0.
fh_prove(labelled_literal(G, Label), Anc) :-
        foothold_contrapositive_rule(G, Body),
        negate(G, Neg_G),
        fh_prove_all(Body, [labelled_literal(Neg_G, Label) | Anc]).

fh_prove_all([ ], _).
fh_prove_all([ G1 | G ], Anc) :-
        fh_prove(G1, Anc),
        fh_prove_all(G, Anc).

ancestor_search(Goal, S0, S0, [labelled_literal(Goal, Label) | Anc]).
ancestor_search(Goal, S0, S2, [labelled_literal(G, Label) | Anc]) :-
        S1 is S0 + Label,
        ancestor_search(Goal, S1, S2, Anc).

negate(X, ¬X) :- \+ X = ¬_.
negate(¬X, X).
```

Appendix C

Algorithms from Chapter 5

C.1 de Kleer's ATMS Algorithms

These algorithms perform the operations described in Section 5.1.5. The algorithms Propagate, Update, Weave and NewNogood were proposed by de Kleer [8]. These algorithms are based on the supposition that at the outset all of the labels are up-to-date with the information available before this new clause, and so they only compute the effect of the new clause. For example, Weave, instead of using the existing label for a (the literal with new support) uses only the additions L to a's label.

Let $a \leftarrow b_1 \wedge \ldots \wedge b_n$ be a new Horn clause to be added to the ATMS.

algorithm AddAssumption (a)
if there is no or-node for a **then**
    create a new or-node containing a and the label {}
end if
call Update($\{\{a\}\}, a$)
end AddAssumption

algorithm AddClause $(a \leftarrow b_1 \wedge \ldots \wedge b_n)$
/* The additions shown in Figure 5.1 are made to the and/or graph. */
for each $l \in \{a, b_1, \ldots, b_n\}$ **do**
    if there is no or-node for l **then**
        create a new or-node containing l and the empty label $\{\}$
    end if
end for each
Let π be a new and-node. Make π a child of the or-node containing a. Make π's children be the or-nodes containing b_i.
call Propagate($a \leftarrow b_1 \wedge \ldots \wedge b_n, \phi, \{\{\}\}$)
end algorithm AddClause

algorithm Propagate $(a \leftarrow b_1 \wedge \ldots \wedge b_n, b, I)$
/* Compute L, the incremental update for the label of a.*/
$L :=$ Weave$(b, I, [b_1, \ldots, b_n])$
/* If there is an update, perform it. (This will, in turn, generate more propagation to the parents of a.) */
if $L \neq \{\}$ **then call** Update(L, a)
end algorithm Propagate

algorithm Update (L, a)
/* Detect Nogoods
if $a = \perp$ **then**
    for each $E \in L$ **do**
        call NewNogood(E)
    end for each
else
    /* Ensure the update is minimal. */
    Delete every member of L which is a superset of some member of a's label.
    /* Ensure the new label for a is minimal. */

Delete every member of a's label which is a superset of some member of L.
/* Update a's label. */
Add L to a's label.
/* Propagate the incremental change to a's ancestors. */
for each and-parent π of a **do**
/* See Figure 5.1 */
Let *cons* be the or-parent of π.
Let $d_1, \ldots, d_n$ be the or-children of π. (a occurs among the d_i)
call Propagate($cons \leftarrow d_1 \wedge \ldots \wedge d_n, a, L$)
end for each
end if
end algorithm Update

function Weave $(b, I, [b_1, \ldots, b_n])$
/* There are two modes: either b occurs among the b_i, in which case I represents the changes to b's label, and weave should compute the effect of this change on the conjunction of the labels of the b_i, or $b = \phi$, in which case we simply want the conjunction of the labels of the b_i. Weave returns a disjunctive normal formula. */
if $[b_1, \ldots, b_n]$ is the empty sequence **then**
return I
else if $b_1 = b$ **then**
return weave$(\phi, I, [b_2, \ldots, b_n])$
else
Let I' be the set of all environments formed by computing the union of an environment of I and an environment of b_1's label.
Remove from I' all environments which are subsumed by (are supersets of) some nogood or some other environment in I'.
return weave$(b, I', [b_2, \ldots, b_n])$
end if
end function Weave

algorithm NewNogood (E)

Add E to the set of nogoods
Remove any superset of E from every node label.
end algorithm NewNogood

C.2 Our TMS Algorithms

Let $b_1 \vee \ldots \vee b_n$ be a new clause to be added to the TMS. Main refers to the main and/or graph.

algorithm AddClause $(b_1 \vee \ldots \vee b_n)$
for each $l \in \{b_1, \ldots, b_n, \overline{b}_1, \ldots \overline{b}_n\}$ **do**
    if there is no or-node for l in main **then**
        create a new or-node in main containing l and the empty label {}
    end if
/* Add each contrapositive rule */
for each $i = 1, \ldots, n$ **do**
    /* Add the contrapositive form to the main graph */
    Let π be a new and-node in main. Make π a child of the or-node in main containing b_i. Make π's children be the or-nodes in main containing $\overline{b}_j$ for $j \neq i$.
    foothold labels to the b_j according to Definition 4.4.
    call Propagate($b_i \leftarrow \overline{b}_1 \wedge \ldots \wedge \overline{b}_{i-1} \wedge \overline{b}_{i+1} \wedge \ldots \wedge \overline{b}_n, \phi, \{\{\}\}$, main)
end for each
end algorithm AddClause

algorithm AddAssumption (a)
if there is no or-node in main for a **then**
    create a new or-node in main containing l and the label {}
end if
call Update($\{\{a\}\}, a$, main)
end AddAssumption
call Propagate($a \leftarrow b_1 \wedge \ldots \wedge b_n, \phi, \{\{\}\}$, main)
end algorithm AddAssumption

algorithm Propagate ($a \leftarrow b_1 \wedge \ldots \wedge b_n, b, I$, Graph)
/* Compute L, the incremental update for the label of a.*/
$L :=$ Weave($b, I, [b_1, \ldots, b_n], Graph$)
/* If there is an update, perform it. (This will, in turn, generate more propagation to the parents of a.) */
if $L \neq \{\}$ **then**
    call Update(L, a, Graph)
    if Graph = main **then**
        call Update(L, a, BackedgeGraph) for all BackedgeGraphs where a occurs.
    end if
    if Graph = BackedgeGraph($\overline{a}$) **then**
        call Update(L, a, Main)
    end if
end if
end algorithm Propagate

algorithm Update (L, a, Graph)
/* Detect Nogoods
if $a = \perp$ **then**
    for each $E \in L$ **do**
        call NewNogood(E)
    end for each
else
    /* Ensure the update is minimal. */
    Delete every member of L which is a superset of some member of a's label in Graph.
    /* Ensure the new label for a is minimal. */
    Delete every member of a's label in Graph which is a superset of some member of L.
    /* Update a's label. */
    Add L to a's label in Graph.
    /* Propagate the incremental change to a's ancestors. */
    for each and-parent π of a in Graph **do**

/* See Figure 5.1 */
Let *cons* be the or-parent of π in Graph.
Let $d_1, \ldots, d_n$ be the or-children of π in Graph. (a occurs among the d_i)
call Propagate($cons \leftarrow d_1 \wedge \ldots \wedge d_n, a, L, Graph$)
end for each
end if
end algorithm Update

function Weave ($b, I, [b_1, \ldots, b_n]$, Graph)
/* There are two modes: either b occurs among the b_i, in which case I represents the changes to b's label, and Weave should compute the effect of this change on the conjunction of the labels of the b_i, or $b = \phi$, in which case we simply want the conjunction of the labels of the b_i. Weave returns a disjunctive normal formula. */
if $[b_1, \ldots, b_n]$ is the empty sequence **then**
return I
else if $b_1 = b$ **then**
return Weave($\phi, I, [b_2, \ldots, b_n]$)
else
Let I' be the set of all environments formed by computing the union of an environment of I and an environment of b_1's label in Graph.
Remove from I' all environments which are subsumed by (are supersets of) some nogood or some other environment in I'.
return Weave($a, I', [b_2, \ldots, b_n]$, Graph)
end if
end function Weave

algorithm NewNogood (E)
Add E to the set of nogoods
Remove any superset of E from every node label.
end algorithm NewNogood

C.2.1 Backedge Algorithms

Finding Backedge Paths

The routines to find backedge paths are presented in Prolog to facilitate the presentation of the non-determinism in the routine.

```
\* FindPaths(Lit) is a failure driven loop that finds backedge paths to add to
the TMS.*\
find_paths(Lit) :- find_paths1(Lit, _, [ ], [ ]), fail.
find_paths(Lit).


\* Find_paths1 is given Lit, a literal, Ancestors, a list of the ancestors of Lit
in the path considered so far, and SoFar, a list of the contrapositive rules on
the path so far. Find_paths1 calls AddPath whenever a path is found to Lit
from the complement of Lit. FHLabel is Lit's foothold label. Ancestors also
contains the foothold labels of the ancestors. *\
find_paths1(Lit, FHLabel, Ancestors, SoFar) :-
     ancestor_search(Lit, Ancestors, SoFar, [ ], Path, FHLabel, Total),
     !,
     \* If ancestor search succeeds once, it need not succeed again, even if the
     foothold condition is not met. *\
     Total > 0,
     add_path(Path).
 \* Eliminate non-minimal paths. *\
find_paths1(Lit, FHLabel, Ancestors, SoFar) :-
     member(lit_fh(Lit, _), Ancestors),
     !,
     fail.

find_paths1(Lit, FHLabel, Ancestors, SoFar) :-
     contrapositive_rule( Lit ← RHS),
     member(L, RHS),
     % look up the foothold label for L in this contrapositive
     foothold_label(L, Lit ← RHS, FHLabel),
```

```
find_paths1(L, [lit_fh(L, FHLabel) | Ancestors], [Lit ← RHS | SoFar]).


\* Ancestor_search searches for Lit among Ancestors, and at the same time
keeps a running total of the foothold labels. Path accumulates the contraposi-
tives found on the path to the ancestor. *\
ancestor_search(Lit, [lit_fh(L1, FHLabel) | Ancestor], SoFar, Path, Path, Total,
Total) :-
    negate(Lit, L1).
ancestor_search(Lit, [lit_fh(L1, FHLabel) | Ancestor], [CP | CPs], PathIn,
PathOut, TotalIn, TotalOut) :-
    TotalTemp is Total + FHLabel,
    ancestor_search(Lit, Ancestor, CPs, [CP | PathIn], PathOut, TotalTemp,
    TotalOut).
```

Adding Backedge Paths

algorithm AddPath(*Path*)
* AddPath adds the contrapositive rules in Path to the appropriate backedge graph, and calls Propagate to initiate Truth Maintenance *\
* *Path* is a sequence of contrapositive rules. *\
Let R be the first rule in *Path*.
R has the form $Lit \leftarrow Cond_1 \wedge \ldots \wedge Cond_n$
$Graph := BackedgeGraph(\overline{Lit})$
for each rule R in *Paths* **do**
    R has the form $L \leftarrow Cond_1 \wedge \ldots \wedge Cond_n$
    /* Add the contrapositive form to the backedge graph */
    Let π be a new and-node in *Graph*
    if there is an no or-node in *Graph* containing L **then** create such an or-node and give it the same label as the or-node in main containing L
    Make π a child of the or-node in *Graph* containing L.
    for each $Cond_i$ in R **do**
        if there is an no or-node in *Graph* containing $Cond_i$ **then** create such an or-node and give it the same label as the or-node in main containing $Cond_i$.
        Make the or-node containing $Cond_i$ a child of π in *Graph*
```
```

end for each
call Propagate($L \leftarrow Cond_1 \wedge \ldots \wedge Cond_n, \phi, \{\{\}\}, Graph$)
end for each
% Make sure that the label for $\overline{Lit}$ is $\{\{\}\}$
call Update($\{\{\}\}, \overline{Lit}, Graph$)
end algorithm

C.3 Plan Recognition Algorithms

C.3.1 Extracting Candidate Plans

Note that this algorithm extracts a graph which does not precisely match the definition of a negated ancestor proof graph, Definition 4.3. If converting to a negated ancestor proof graph would require replicating the same literal into two or more nodes then the algorithm avoids this extra step.

function ExtractGraph (Literal)
* ExtractGraph builds a proof graph from the nodes in TMS. For each Or node it selects one And node and attempts to extract the proof graph for each Or son. There must be support either in the main graph or in one of the negated ancestor paths. it returns its result in three sets, Nodes, TreeEdges and Backedges. These three sets are built up by ExtractGraph1.*\
Let N be the node in the main graph containing Literal
if N has a non-empty label **then**
call ExtractGraph1(N, main)
return (Nodes, TreeEdges $\cup$ Backedges)
end function

procedure ExtractGraph1 ($N, Graph$)
Let L be the literal in N
% Find the and/node that supports the literal

if L is an assumption **then**
    % We do not need to see the assumptions
    return
else if there is an and/node N in *Graph* such that each or-son of N has a non-empty label **then**
    Nodes := Nodes $\cup\{N\}$
    for each or-son S of N **do**
        call ExtractGraph1(S, *Graph*)
        TreeEdges := TreeEdges $\cup(N, S)$
    end for each
else if *Graph* = main **then**
    % There is no support for the L in the main graph.
    % Try the backedge graph for the complement of L
    call ExtractGraph1(N, backedge-graph($\overline{L}$))
else if *Graph* = BackedgeGraph(L) **then**
    * This is the backedge graph for the L, so there is already a node in Nodes containing $\overline{L}$ which is an ancestor of L. *\
    Nodes := Nodes $\cup\{N\}$
    Let *Anc* be the node containing $\overline{L}$
    Backedges := Backedges $\cup\{(N, Anc)\}$
else
    % This is the backedge graph for a different literal, so try the main graph
    call ExtractGraph1(L, main)
end if
end procedure

C.3.2 Plan Recognition with Assimilation (Single Observation)

procedure PlanRecognition-with-Assimilation-SingleObs
%
Let H be the initial hierarchy
Let $E_1(Obs) \vee \ldots \vee E_n(Obs)$ be the single observation
call CloseLibrary(H)
call Search($E_1(Obs) \vee \ldots \vee E_n(Obs)$)
Initialize the truth maintenance system TMS

% The Reasoner
for each $A \in Axioms$ **do**
AddClause(A)
end for each
for each assumption Asn associated with an axiom in CUA^+ or an axiom in EXA^+ **do**
AddAssumption(A)
end for each
call FindPaths(end)
% FindPaths calls AddPath for each path found
call ExtractGraph(end) to get the proof graph

loop
Accept NewInfo from the Oracle
call LibraryAssimilate(NewInfo)
call AssimSearch($EXA^-, CUA^-, EXA^+, CUA^+, H_A^+, H_D^+$)

% The Reasoner
for each $A \in NewAxioms$ **do**
AddClause(A)
end for each
for each assumption Asn associated with an axiom in CUA^+ or an axiom in EXA^+ **do**
AddAssumption(Asn)
end for each
for each assumption Asn associated with an axiom in CUA^- or an axiom in EXA^- **do**
AddClause($Asn \supset false$)
end for each
call FindPaths(end)
% FindPaths call AddPath for each path found
call ExtractGraph(end) to get the new proof graph
go to loop
end procedure PlanRecognition-with-Assimilation-SingleObs

C.3.3 Plan Recognition with Assimilation (Multiple Observations)

This algorithm shows how to do plan recognition with multiple observations while accepting new plan information from the oracle.

The algorithm uses and maintains two sets: Hypoths and SingletonStates. Hypoths is a set of sets of e-graphs that is maintained by GroupObservations. Each set of e-graphs it contains is a complete explanation of all of the observations that shows how they are related. SingletonStates is a set of 4-tuples (ObsTypes, SearchState, TMS-State, and E-graph) that is maintained by our algorithms. There is a 4-tuple for each observation, and the disjunction of types that describes the observation is ObsTypes. SearchState is the state of the Search algorithm that includes the set of Axioms that have been generated and the set of literals visited. TMS-State is the and/or graph, backedge graph and labels. E-Graph is the extracted explanation graph that contains the disjunction of candidate plans for this observation.

procedure PlanRecognition
%
Let H be the initial hierarchy
Hypoths := {}
SingletonStates := {}
call CloseLibrary(H)
loop
Input NewInfo
if NewInfo is a disjunctive observation $E_1(Obs) \vee \ldots \vee E_n(Obs)$ **then**
call Search($E_1(Obs) \vee \ldots \vee E_n(Obs)$)

% The Reasoner
Initialize a new truth maintenance system TMS in TMS-state
Add each ground clause to the TMS with AddClause
Add each new assumption to the TMS with AddAssumption
call FindPaths(end)

```
        % FindPaths calls AddPath for each path found
        call ExtractGraph(end) to extract the proof graph from this TMS
        Convert the ProofGraph to an E-graph
        Create a new member of SingletonStates, which is ( E1(Obs) ∨ ... ∨
        En(Obs), (Visited,Axioms), TMS-state, E-graph)
        call GroupObservations(E-graph) to get the new Hypoths
    else if NewInfo is an addition to the hierarchy then
        call LibraryAssimilate(NewInfo)
        for each (Obs, SearchState, TMS-state, E-graph) ∈ SingletonStates
        do
            call AssimSearch to get the additional ground clauses required, and
            the assumptions that have been violated

            % The Reasoner
            Add each ground clause to the TMS with AddClause
            Add each new assumption to the TMS with AddAssumption
            call FindPaths(end)
            % FindPaths calls AddPath for each path found
            call ExtractGraph(end) to get the proof graph from this TMS
            Convert the proof graph to an e-graph and store it in E-graph
        end for each
        Hypoths := {}
        for each (Obs, SearchState, TMS-state, E-graph) ∈ SingletonStates
        do
            call GroupObservations(E-graph)
        end for each
    end if
    go to loop
    end procedure PlanRecognition
```

Bibliography

[1] James F. Allen and C. Raymond Perrault. Analysing intention in utterances. *Artificial Intelligence*, 15:143–178, 1980.

[2] J. L. Austin. *How To Do Things With Words.* Oxford University Press, New York, New York, 1962.

[3] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York and London, 1973.

[4] Eugene Charniak and Robert Goldman. A logic for semantic interpretation. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 87–94, 1988.

[5] Eugene Charniak and Drew McDermot. *Introduction to Artificial Intelligence.* Addison Wesley, 1985.

[6] Johan de Kleer. An assumption-based Truth Maintenance System. *Artificial Intelligence*, 28:127–162, 1986.

[7] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[8] Johan de Kleer. A general labeling algorithm for assumption-based truth maintenance. In *Proceedings AAAI-88 Seventh National Conference on Artificial Intelligence*, pages 188–192, 1988.

[9] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[10] G. F. DeJong and Raymond J. Mooney. Explanation-based learning: An alternate view. *Machine Learning*, 1:145–176, 1986.

[11] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[12] Charles Elkan. A rational reconstruction of truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[13] David Etherington. *Reasoning with Incomplete Information.* Morgan Kaufmann, 1988.

[14] Kenneth D. Forbus. QPE: A study in assumption-based truth maintenance. *International Journal of AI and Engineering*, 3:197, 1988.

[15] C. Cordell Green. Theorem-proving by resolution as a basis for question-answering systems. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier Publishing Company, Inc., 1969.

[16] Henry A. Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, 1987. Available as Technical Report 215.

[17] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *Proceedings AAAI-86 Fifth National Conference on Artificial Intelligence*, 1986.

[18] Diane J. Litman and Bradley A. Goodman. A knowledge-based interface for process design. In *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, 1989. Hilton Head Island, South Carolina, USA.

[19] Donald Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, Amsterdam, 1978.

[20] John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[21] John McCarthy. Applications of circumscription to formalizing common-sense reasoning. *Artificial Intelligence*, 28:89–116, 1986.

[22] Francis Jeffry Pelletier. Seventy-five problems for testing automated theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[23] David Poole, Randy Goebel, and Romas Aleliunas. Theorist: a logical reasoning system for defaults and diagnosis. In Nick Cercone and Gord McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer Verlag, New York, 1987.

[24] Gregory Provan. Efficiency analysis of multiple-context tmss in scene representation. In *Proceedings AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 173–177, 1987.

[25] Gregory M. Provan. The computational complexity of truth maintenance systems. Technical Report 88-11, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada, 1988.

[26] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[27] Raymond Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 183–188, 1987.

[28] J. R. Searle. Indirect speech acts. In P. Cole and J. L. Morgan, editors, *Syntax and Semantics Vol.3: Speech Acts.* Academic Press, New York, 1975.

[29] T. Skolem. Logisch-kombinaterische untersuchungen über die erfüllbarkeit oder beweibarkeit mathematischer sätze. *Skrifter utgit av Videnskapsselskapet i Kristiania*, 4:4–36, 1920.

[30] Bruce Spencer. Avoiding duplicate proofs. In Saumya K. Debray and Manuel Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, page to appear. MIT Press, 1990.

[31] Mark E. Stickel. A prolog technology theorem prover. *Journal of Automated Reasoning*, 4:353–380, 1989.

[32] Peter van Beek and Bruce Spencer. Applying truth maintenance to a network of constraints. In *Proceedings of the Workshop on Constraint Processing Algorithms and their Applications*, August 1989. a workshop held in conjunction with IJCAI-89, Detroit, Michigan, USA.

[33] C. Williams. *ART, the advanced reasoning tool, Conceptual Overview.* Inference Corporation, 1984.

[34] Larry Wos. *Automated Reasoning : 33 Basic Research Problems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.