The Use of Data-Mining for the Automatic Formation of Tactics

Hazel Duncan

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh
2007

Abstract

As functions which further the state of a proof in automated theorem proving, tactics are an important development in automated deduction. This thesis describes a method to tackle the problem of tactic formation. Tactics must currently be developed by hand, which can be a complicated and time-consuming process. A method is presented for the automatic production of useful tactics.

The method presented works on the principle that commonly occurring patterns within proof corpora may have some significance and could therefore be exploited to provide novel tactics. These tactics are discovered using a three step process.

Firstly a suitable corpus is chosen and processed. One example of a suitable corpus is that of the Isabelle theorem prover. A number of possible abstractions are presented for this corpus.

Secondly, machine learning techniques are used to data-mine each corpus and find sequences of commonly occurring proof steps. The specifics of a proof step are defined by the specified abstraction.

The formation of these tactics is completed using evolutionary techniques to combine these patterns into compound tactics.

These new tactics are applied using a naive prover as well as undergoing manual evalutation. The tactics show favourable results across a selection of tests, justifying the claim that this project provides a novel method of automatically producing tactics which are both viable and useful.

Acknowledgements

I would like to thank my supervisors Alan Bundy, Amos Storkey and John Levine, for all their input and advice. I would also like to thank both the Mizar group and the Ωmega group for allowing me to work with them and for making me feel welcome in their respective universities. In particular, I would like to thank Martin Pollet in the Ωmega group at Saarbrücken for his valuable insights.

Special thanks are due to my partner Jim Macdonald for helping to support me, both financially and emotionally, throughout the whole of my studies.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Hazel Duncan*)

Table of Contents

1 Introduction **1**

1.1 Technique Outline 2
1.2 Original Contribution 3
1.3 Thesis Outline 4

2 Related Work **7**

2.1 Automated Theorem Proving and Provers 7
2.1.1 Proof Tactics 8
2.1.2 COQ 8
2.1.3 NUPRL 9
2.1.4 PVS 9
2.1.5 Mizar 9
2.1.6 LEGO 10
2.1.7 Isabelle 10
2.1.8 IsaPlanner 11
2.1.9 LambdaClam 11
2.1.10 Ωmega 12
2.1.11 Summary 12
2.2 Previous Learning Methods 12
2.2.1 Pre-condition Analysis 13
2.2.2 Learning Proof Methods 13
2.2.3 Learning using Markov Models 14
2.2.4 Random Fields 14
2.2.5 Proof Reuse and the Simulation of Human Learning 15
2.2.6 Explanation-Based Learning 16
2.2.7 Learning Heuristic Control 17
2.2.8 Summary 17

2.3 Genetic Algorithms 17
2.3.1 Koza 18
2.3.2 Learn2Plan 18
2.3.3 Summary 19
2.4 Summary 19

3 Obtaining Data 21
3.1 Choosing the Theorem Prover 21
3.1.1 Requirements 22
3.1.2 Options 22
3.2 Isabelle 25
3.2.1 Proofs in Isabelle 26
3.3 Extracting and Formatting Proofs 29
3.4 Abstraction 30
3.4.1 Options 31
3.5 Summary 34

4 Pattern Discovery 37
4.1 Overview 37
4.2 Specification 38
4.3 Existing Models 41
4.3.1 Sparse Markov Transducers (SMT) 42
4.3.2 Teiresias 42
4.4 Implementation 43
4.4.1 Variable Length Markov Models 44
4.5 Finding the patterns 46
4.5.1 Preprocessed data 46
4.5.2 Linearisation Process 47
4.5.3 Finding Patterns 49
4.6 Experimental Results 52
4.6.1 Input Choices 53
4.6.2 Threshold 55
4.6.3 Some Patterns 56
4.7 Summary 58

5 Tactic Formation **61**
5.1 Introduction . . . 61
5.2 Specification . . . 61
5.3 Grammar . . . 63
5.4 Genetic Programming . . . 64
5.5 Traditional GP Method . . . 65
5.5.1 Implementation . . . 65
5.5.2 Performance . . . 70
5.6 Pairwise Combination . . . 73
5.6.1 Implementation . . . 74
5.6.2 Performance . . . 79
5.7 Summary . . . 81

6 Application **85**
6.1 Designing an Automatic Isabelle Prover . . . 85
6.1.1 Isabelle Tools . . . 86
6.1.2 Implementation . . . 88
6.2 Adapting the Prover for Different Abstractions . . . 90
6.2.1 Performance . . . 92
6.3 Summary . . . 94

7 Evaluation **95**
7.1 Patterns . . . 96
7.1.1 Manual Evaluation . . . 96
7.2 Tactics . . . 98
7.2.1 Manual Evaluation . . . 98
7.2.2 Usefulness . . . 98
7.2.3 Quality . . . 98
7.3 Some Examples . . . 99
7.3.1 A Typical Tactic . . . 99
7.3.2 A Simple Tactic . . . 100
7.3.3 A Complicated Tactic . . . 101
7.3.4 A Good Tactic . . . 103
7.3.5 A Bad Tactic . . . 104
7.3.6 Overall Evaluation . . . 104
7.4 Application . . . 105

7.4.1 Test Theorems 106
7.4.2 Choosing the Best Tactics 106
7.4.3 Different Abstractions 107
7.4.4 Tactic Application results 108
7.5 Other Abstractions 115
7.6 Summary 116

8 Conclusion **117**
8.1 Summary 117
8.2 Critique 121
8.3 Related Work 122
8.4 Further Work 123

A Glossary **125**

B Isabelle rules and theorems **129**
B.1 Rule definitions 130
B.2 Some complete proof scripts 131

Bibliography **133**

List of Figures

1.1 An abstract example of a tactic. 3

2.1 A simple theorem proving example. 8

3.1 An example proof from the Isabelle corpus 28

4.1 Two patterns combined by an or branching structure 39
4.2 The abstracted proof represented in tree structure 40
4.3 The traditional proof tree 40
4.4 Picture of example proof linearisation. 48
4.5 Threshold required to gain 20 patterns from random selections against chosen selections. Four comparisons are made. This graph describes the average of 15 runs across different domains. 54
4.6 Pattern from a set of 500 *selected* theorems. 56
4.7 Combined pattern after tactic formation.Here *impI*+ means 1 or more repetitions of *impI*. 57
4.8 Pattern from a random set of 500 theorems. 58

5.1 Two patterns which show potential for an $\vee$ introduction. 62
5.2 Patterns which show potential for a *plus* introduction. 62
5.3 Two patterns which show potential for macro introduction. 62
5.4 Two patterns which show potential for an $\wedge$ introduction. Note that here *iffI* is stored as a step which results is a branch 63
5.5 Results of a crossover when no branches are present and the patterns end with the same step 67
5.6 Results of a crossover when no branches are present and the patterns end with different steps 68
5.7 Results of a crossover when one of the candidates already contains a branch 68

5.8 Results of a crossover when both candidates already contain a branch 69
5.9 Example of a tactic found using Genetic Programming 71
5.10 This shows the percentage of proofs which could have 0, 1, 2, 3 or more tactics applied to them. 72
5.11 Measure of efficiency of Genetic Programming. The x axis shows the number of iterations and the y axis shows the decrease in population size. 73
5.12 Introduction of a macro identifier. 75
5.13 Introduction of the plus operator. 76
5.14 Introduction of the $\vee$ operator. 77
5.15 Introduction of the $\wedge$ operator. 77
5.16 How a repetition over more than one step can be found using the macro identifier 78
5.17 Example of a tactic found using Pairwise Combination 79
5.18 No. of tactics applicable within proofs. This shows the percentages of proofs which have: 0, 1, 2 or 3 or more tactics applicable to them. . . 81
5.19 Measure of efficiency of Pairwise Combination. 82

6.1 Example of a tactic. 89
6.2 Number of theorems proved with increasing complexity. X-axis shows an increasing complexity rank. Y-axis show the number of theorems proved or not proved, respectively. 94

7.1 Discovered pattern representing one full branch of a proof. 97
7.2 Example of Complicated Tactic 102
7.3 Performance of different abstractions. Rule name only: *rno*. Rule name with direction *rnwd*. Class only *co*. Class with direction *cwd*. X-axis shows increasing complexity. Y-axis shows the average number of theorems proved over 20 runs. 107
7.4 Time performance with different abstractions. Rule name only: *rno*. Rule name with direction *rnwd*. Class only *co*. Class with direction *cwd*. X-axis shows increasing complexity, Y-axis shows increasing time. 108
7.5 Average numbers of theorems proved by Isabelle prover from theorems taken from narrow groups. X-axis shows increasing complexity, Y-axis shows the average number of theorems proved 109

7.6 Average times taken to prove a theorem by Isabelle prover from theorems taken from narrow groups. 110
7.7 Average numbers of theorems proved by Isabelle prover from theorems taken from a broad spectrum of theories. 111
7.8 Average times taken to prove a theorem by Isabelle prover from theorems taken from a broad spectrum of theories. 111
7.9 Average numbers of theorems proved by Isabelle prover from theorems taken from a broad domain of theories with tactics trained on a narrow domain of theorems. 112
7.10 Average times taken to prove a theorem by Isabelle prover from theorems taken from a broad spectrum of theories with tactics trained on a narrow set of theorems. 113
7.11 Average numbers of theorems proved by Isabelle prover from theorems taken from a narrow group of theories with tactics trained on a broad spectrum of theorems. 114
7.12 Average times taken to prove a theorem by Isabelle prover from theorems taken from a narrow group of theories with tactics trained on a broad spectrum of theorems. 114

Chapter 1

Introduction

Within the field of automated deduction, the huge search spaces involved in finding correct proofs means that fully automated theorem provers are not as advanced as it was once thought they would be by this time. For example, Newell and Simon claimed that a computer would "discover and prove an important new mathematical theorem" by January 1st 1968 [Simon and Newell (1958)]. However, the vast search spaces involved in finding even a relatively simple mathematical proof means that automated theorem provers are not nearly so advanced. The majority of fully automated theorem provers can only prove relatively simple mathematical theorems and can only do this much within a specialised domain. Interactive theorem provers allow for human intervention to generate the 'eureka' steps while providing some automation for the more tedious steps of a formal proof. In order to increase the capability of automated theorem proving, many techniques to aid proof discovery have been developed but this remains a field with a long way to go to realise its potential.

An important advance in theorem proving was made by Robin Milner when he introduced the notion of tactics [Gordon et al. (1979)]. Tactics are functions from goals to subgoals which raise appropriate error messages when they fail. The use of tactics has greatly helped the field of theorem proving by guiding search. The technique described in this dissertation aims to build upon that success by implementing a method to allow tactics to be formed automatically. Robin Milner used tactics in his automatic proof assistant Edinburgh LCF [Gordon et al. (1979)], which initiated the current theorem-proving, proof-checking, proof-assisting methods. LCF has led to descendents such as HOL [Gordon (1985)], ISABELLE [Paulson (1986)], COQ [Dowek et al. (1991)], LEGO [Luo and Pollack (1992)], Nuprl [Constable et al. (1986)] and PVS [Owre et al. (1992)]. The concept of a *tactic* has become somewhat overloaded in theorem proving

with many different techniques and theorem provers using it to mean different, yet specific, things. Within this document *tactics* is used with the classic meaning of a set of instructions which will further a proof, more specifically information about rules and techniques which should be applied to the subgoal in order to advance the proof state. This phraseology encompasses simple tactics which are sequences of proof steps to compound tactics which contain more complex operators and information about how to apply proof steps.

Tactics must currently be developed by hand. The most intricate tactics (such as Rippling, developed by Bundy *et al.* [Bundy et al. (1993)]) can take many years and significant human effort to develop. Even more straightforward tactics require human intervention and inspiration.

1.1 Technique Outline

This thesis presents the NewT (New Tactic generator) system and its Isabelle-specific implementation IsaNewT. The evaluation of IsaNewT shows that it can form useful tactics automatically using a combination of techniques from probabilistic reasoning, machine learning and genetic programming.

By adapting probabilistic reasoning techniques, such as Variable Length Markov Models (VLMM) [Ron et al. (1996)], rule sequences have been identified. These techniques are used to discover commonly occurring patterns existing in proof corpora. Such patterns can be viewed as simple probabilistic tactics. What constitutes a proof step varies across different systems and the main NewT system does not require a specific form. For the ease of reading we represent all tactics in Isabelle formatting which is the form we have used for development and testing. Within the Isabelle system such proof steps generally consist of a theorem which is used as a rewrite rule.

These patterns are adapted using Koza-style genetic programming [Koza (1992)]. Using this, the simple tactics are generalised into compound ones, e.g. containing repetition, branching and other operators from the regular grammar defined in chapter 5. This process requires the development of an evaluation function for scoring the evolving tactics. A new evolutionary programming technique is compared against the traditional Koza-style Genetic Programming method in terms of efficiency and output tactics.

The tactics generated by IsaNewT have the form:

```
Tac1 = [step, repetition(step list), branch([step list],[step list])]
```

It can be more useful to imagine the tactics as a pictorial tree such as in figure 1.1, particularly when dealing with complicated tactics.

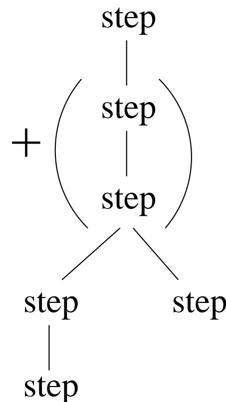

Figure 1.1: An abstract example of a tactic.

In order to evaluate these new tactics a fully automated prover within the interactive theorem proving system Isabelle has been developed (IsaAuto). The new generalised tactics are evaluated by applying them to a test set of theorems and comparing their performance within IsaAuto. The results with and without the newly discovered tactics are compared to provide a measure of usefulness.

1.2 Original Contribution

These are the main original contributions of the method described in this thesis:

1. *Pattern discovery within a proof corpus* A method for automatically scanning a proof corpus and finding commonly occurring patterns within these proofs is provided. Although many techniques exist to discover patterns a particular adaptation of these is used to produce a method specific to discovering patterns within proof structures. In particular, there is a requirement to adapt methods designed for sequences to handle trees.

2. *Evolution of patterns into tactics* A method is provided for combining sequences of proof steps together into compound tactics. At this point an adaptation of existing Genetic Programming techniques is used to combine sequences together

into more general structures which describe multiple sequences. For our purposes *sequences* are lists of abstracted proof steps.

3. *Automatically produced tactics* The two previous contributions lead to the main purpose, which is to provide a method to produce tactics automatically. As discussed in the introduction, this achievement describes a significant contribution to the field.

   The IsaNewT system describes the first attempt to provide a fully automated method for producing tactics. The analysis of these tactics yields favourable results which show that IsaNewT is capable of producing tactics which are both viable and useful.

1.3 Thesis Outline

- Chapter 2 describes previous work carried out in the fields that the method presented is concerned with. Existing work in the field of automated deduction is considered, along with previous learning methods from text processing, computer vision and bioinformatics. Also considered is related work in the field of genetic algorithms and existing automated theorem provers.

- Chapter 3 considers the available choices for a proof corpus. It explains the choice of proof corpus and the method of obtaining the necessary data. Most importantly, this chapter describes possible choices of abstraction and justifies the choices made.

- In Chapter 4 the pattern discovery process is described in detail. Some available methods are considered and the production of a new technique is presented. Some results from this stage of the technique are also presented.

- Chapter 5 presents some methods for combining these patterns into compound tactics. Two approaches are described and compared in terms of results and efficiency. Some preliminary results for this process are again provided.

- In Chapter 6 the method for applying these tactics in order that they can be better evaluated is provided. This chapter describes the implementation of a naive automated prover which is used to evaluate the new tactics.

- Chapter 7 shows an in-depth analysis and experimental results from the entire IsaNewT system. This chapter enables a judgement to be drawn regarding whether the discovered tactics can be considered useful.

- Chapter 8 presents the conclusions of the entire process.

- *Appendix A:* This contains a glossary to clarify all technical terms used, in particular, this is used to explain the overloading of certain terms.

- *Appendix B:* This contains some technical information about proofs and proof steps.

Chapter 2

Related Work

This chapter gives a broad overview of related work in various fields. It covers a number of processes which have been applied to a variety of problems.

Traditional reasoning methods and how they have led to this method are considered along with techniques from other disciplines and how they can be adapted to relevant uses.

As this technique crosses several disciplines, some of the most relevant work within each discipline are considered with the uses to which these techniques are normally put. A range of existing automated theorem provers are introduced followed by an examination of existing learning methods from a range of fields. This chapter concludes with some genetic programming techniques, including how they have been applied to a related problem.

2.1 Automated Theorem Proving and Provers

Automated theorem proving at its most simple is a term which describes the use of a computer program to prove a mathematical theorem. Within this field are *interactive theorem provers* which provide proof assistance to a human and *fully automated theorem provers* which require no human intervention at all.

A conjecture made up of a set of assumptions and a consequence is given to these provers which then use their knowledge base (made up of axioms and derived inference rules) to explore the search space in order to prove the conjecture. The proof is stored as a sequence of steps which can derive the consequence of a conjecture from its assumptions.

A proof step can be applied either forwards (what can be proved from A) or back-

wards (how can A be proved). As an example, the forwards proof of the conjecture '$A \wedge B \longrightarrow A \wedge (B \wedge A)$' is given in figure 2.1.

Conjecture $A \wedge B \vdash A \wedge (B \wedge A)$
Derived rule conjI $P, Q \vdash P \wedge Q$

$$\dfrac{\dfrac{\overline{B}^{asm} \quad \overline{A}^{asm}}{B \wedge A}\, conjI}{A \wedge (B \wedge A)}\, conjI$$

Figure 2.1: A simple theorem proving example.

2.1.1 Proof Tactics

A *proof tactic* is a computer program for applying the rules of inference of a mathematical theory library [Gordon et al. (1979)], a *proof tactic* guarantees correctness by only applying valid rules. Tactics are widely used in interactive proof systems for automating common patterns of proof and, hence, improving productivity. Tactic-based theorem provers have been developed both in academia (COQ, NUPRL, PVS, Mizar, LEGO, HOL, Isabelle, Nuprl) and industry (Forte, ProofPower). Until recently, this has required the manual construction of tactics. The technique presented here reduces this impediment by providing a fully automated method for producing new tactics.

2.1.2 COQ

The COQ tool is a formal proof management system [Dowek et al. (1991)] : a proof done with COQ is mechanically checked by the machine. All logical judgements in COQ are typing judgements. The core of the COQ system is the type-checking algorithm that checks the correctness of proofs. It checks that a program complies to its specification. COQ also provides an interactive proof assistant to build proofs using tactics.

COQ has an interactive mode in which commands are interpreted as the user types them in from the keyboard and a compiler mode where commands are processed from a file.

- The interactive mode may be used as a debugging mode in which the user can develop his theories and proofs step by step, backtracking if needed and so on.

- The compiler mode acts as a proof checker taking a file containing a whole development in order to ensure its correctness. Moreover, COQ's compiler provides an output file containing a compact representation of its input.

2.1.3 NUPRL

The NUPRL proof development tool [Constable et al. (1986)]- first released in 1984 - is a framework for the development of formalised mathematical knowledge as well as for the synthesis, verification and optimisation of software. It includes formalisations of the fundamental concepts of mathematics, data types and programming. The system supports interactive and tactic-based reasoning, decision procedures, evaluation of programs, language extensions through user-defined concepts, and an extendable library of verified knowledge from various domains.

2.1.4 PVS

The PVS theorem prover [Owre et al. (1992)] provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. The implementations of these primitive inferences are optimised for large proofs: for example, propositional simplification uses BDDs, and auto-rewrites are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies. Proofs yield scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs.

2.1.5 Mizar

The Mizar proof assistant [Rudnicki (1992)] is similar to the compiler mode in COQ. A user writes an entire proof and the system checks it for correctness. The source text is prepared using any ASCII editor and typically includes from 1500 to 5000 lines. The text is run through the Accommodator. The directives from the Environment Declaration guide the production of the environment specific for the article. The environment is produced from the available data base. Now the Verifier is ready to start checking.

The output contains remarks on unaccepted fragments of the source text. These three steps are repeated in a loop until no errors are flagged and the author is satisfied with the resulting text.

A finished Mizar article is submitted to the Library Committee of Association of Mizar Users for inclusion into the Mizar Mathematical Library. The contributed article is subject to a review and if needed the authors must revise their file. The contents of an accepted article is extracted by the Exporter utility and incorporated into the public data base distributed to all Mizar users.

2.1.6 LEGO

LEGO [Luo and Pollack (1992)] is an interactive theorem prover designed and implemented in Edinburgh using New Jersey ML. It implements various related type systems - the Edinburgh Logical Framework (LF), the Calculus of Constructions (CC), the Generalised Calculus of Constructions (GCC) and the Unified Theory of Dependent Types (UTT).

LEGO is a tool for interactive proof development in the natural deduction style. It supports refinement proof as a basic operation. The system design emphasises removing the more tedious aspects of interactive proofs. For example, features of the system like argument synthesis and universe polymorphism make proof checking more practical by bringing the level of formalisation closer to that of informal mathematics. The higher-order power of its underlying type theories, and the support of specifying new inductive types, provide an expressive language for formalisation of mathematical problems and program specification and development.

2.1.7 Isabelle

Isabelle is a mechanical theorem prover developed with the language ML [Paulson (1986)]. Isabelle is capable of dealing with many types of logic such as first order logic (FOL) and Zermelo-Fraenkel set theory (ZF). Most commonly, Isabelle is used with higher order logic (HOL).

Isar is an extension to traditional Isabelle which operates with HOL, it is based on the natural language representation used in the Mizar system. It improves on Isabelle in a number of ways:

- It has a new theory format supporting interactive development and unlimited

undo operations. This makes developing theories easier to edit and simpler to debug.

- A formal proof document language designed to make mathematical proofs more readable has been developed for Isar.

- It contains a simple document preparation system for typesetting formal developments together with informal text.

Most LCF systems such as Isabelle use a definitional approach. This means that everything must be proved from a very small number of initial axioms, namely those of higher order logic. This has the benefit of ensuring that each step has a well-founded base – i.e. no axioms are defined which are inconsistent with the existing theory.

Isabelle has the additional advantage for our project of having a large electronic proof corpus which has proofs from a wide variety of logics (as described above) and disciplines (from geometry to real analysis to HOL logical reasoning).

2.1.8 IsaPlanner

Lucas Dixon has recently developed IsaPlanner [Dixon and Fleuriot (2003)] as a generic framework for proof planning in the interactive theorem prover Isabelle. It facilitates the encoding of reasoning techniques, which can be used to conjecture and prove theorems automatically. IsaPlanner provides an interactive tracing tool that allows you to interact with the proof planning attempt.

IsaPlanner includes techniques to allow rippling, deductive synthesis and generation of natural language from IsaPlanner traces among other things.

2.1.9 LambdaClam

The Mathematical Reasoning Group at Edinburgh implemented the technique of proof planning in the *Clam* and λ*Clam* proof planners [Bundy et al. (1990); Richardson et al. (1998)] and applied it particularly to the kind of inductive proofs that arise in verification and synthesis of IT systems. It has extended the range of problems that can be solved without human intervention. In particular, the use of proof critics has automated the discovery of intermediate lemmas and generalisations [Ireland and Bundy (1996)] - so called ‘eureka’ steps, which were previously thought to require human intervention.

The λ*Clam* system specialises in using induction based on the rippling heuristic. An interactive theorem prover Oyster-Clam [Horn and Smaill (1990)] has been designed to work with the Clam system. It is based on the NuPrl system but is implemented in Prolog [Pereira et al. (1979)].

2.1.10 Ωmega

The Ωmega group, based in the Saarland University, Saarbrücken and the German Research Center for Artificial Intelligence (DFKI) developed the Ωmega system [Benzmüller et al. (1997)]. Ωmega is a tool with the ultimate purpose of supporting theorem proving in main-stream mathematics and mathematics education. The current system consists of a proof planner and an integrated collection of tools for formulating problems, proving subproblems, and proof presentation.

Ωmega allows each user to build up their theory from a small original set provided with Ωmega, this allows the capacity for operation in a wide variety of domains. Ωmega is currently being used with MathWeb [Kohlhase (2000)] which supplies an infrastructure for web-supported mathematics.

2.1.11 Summary

This section has described a broad range of the available automated and interactive theorem provers available. Some of these systems, such as COQ are used most commonly for checking the correctness of programs, and some such as Mizar pride themselves on having a large library of purely mathematical theorems. Many of the systems, such as the Isar extension to Isabelle strive to bring the readability of the proofs built using them closer to that of traditional mathematics. In many cases (such as IsaPlanner, Ωmega and LEGO) the emphasis is placed on removing the lower level steps traditionally required by formal mathematics. These cases use techniques such as tactics and proof methods to form a higher level proof structure closer to those developed in informal mathematics.

The method behind IsaNewT is applicable to any method which involves an element of automatic proof search so it could theoretically be applied to any of the provers described here. However, an important consideration is the availability of a suitably sized proof corpus to learn the tactics from. Although many of the provers listed above have a significant library, the format of these proofs and the ease of extracting them has played a significant role in the choice of proof corpus for NewT.

2.2 Previous Learning Methods

There have been several previous attempts to learn new proof methods or tactics from example proofs. Also of interest to us are systems which learn and predict patterns, this has been particularly common in the bioinformatics community.

2.2.1 Pre-condition Analysis

Bernard Silver applied techniques of explanation-based learning to the automated learning of proof methods for equation solving [Silver (1984)]. His Learning-Press system analysed successful solutions to equations and generalised these solutions to form methods for guiding the Press equation solving system. In this way, he was able to automatically rediscover simplified versions of many of the previously hand-coded methods of Press.

Similarly, Roberto Desimone automated the reconstruction of inductive proof plans [Desimone (1987)]. Silver and Desimone used precondition analysis which learns new inference methods by evaluating the pre- and post-conditions of each inference step used in the proof. A dependency chart between these pre- and post-conditions is created, and constitutes the pre- and post-conditions of the newly learnt inference systems. These methods are syntactically complete proof steps.

The techniques of both Silver and Desimone generalise from single successful proofs and require the system to be primed with some key meta-level concepts for expressing the preconditions and effects of the methods they learnt. This requirement for priming is a significant drawback to these techniques.

2.2.2 Learning Proof Methods

Kerber, Jamnik, Pollet and Benzmüller have applied the techniques of least general generalisation to a family of similar proofs to learn new proof methods for various domains [Jamnik et al. (2002)]. They present a framework for automated learning within mathematical reasoning systems. In particular, this framework enables proof planning systems to automatically learn new proof methods from well chosen examples of proofs that use a similar reasoning pattern to prove related theorems.

Their framework consists of a representation formalism for methods and a machine learning technique which can learn methods using this representation formalism. They present an implementation of this framework, called LearnΩMatic, which adds new

methods to the Ωmega proof planner. Methods are represented using a regular grammar over individual proof steps and previously learned methods, allowing a hierarchical collection of methods. Note that this technique requires all the proofs in the family to be examples of the learned method.

2.2.3 Learning using Markov Models

Ron, Singer and Tishby applied the probabilistic techniques to Variable memory Length Markov Models [Ron et al. (1996)]. VLMMs processes can be described as a subclass of probabilistic finite automata (PFA) which they call Probabilistic Suffix Automata (PSA). Though hardness results are known for learning distributions generated by general probabilistic automata, they prove that the algorithm they present can efficiently learn distributions generated by PSAs.

In particular, they show that for any target PSA, the divergence between the distribution generated by the target and the distribution generated by the hypothesis the learning algorithm outputs can be made small with high confidence in polynomial time. The learning algorithm is motivated by applications in human-machine interaction. In their paper, they present two applications of the algorithm.

In the first one they apply the algorithm in order to construct a model of the English language, and use that model to correct corrupted text. In the second application they construct a simple stochastic model for *E.coli* DNA. They looked at data which has a *short memory property*, i.e. consider the empirical probability distribution on the next symbol in a sequence given the preceding symbols, then there exists a length L (*memory length*) such that the conditional probability distribution does not change substantially if we condition on preceding subsequences of length greater than L. These can form Markov models of order $L > 1$, they give efficient procedures both for generating sequences and for computing their probabilities.

Markov Models have been frequently used in Bioinformatics, especially for classifying incomplete DNA strands. Some of the work on pattern matching in DNA sequences [Brazma and Cerans (1994)], as in the GENOME project, is related to the NewT learning mechanism.

2.2.4 Random Fields

Stephen Della Pietra, Vincent Della Pietra and John Lafferty presented a technique for constructing random fields from a set of training examples in their paper *Induc-*

ing Features of Random Fields [Della Pietra et al. (1997)]. Their learning paradigm builds increasingly complex fields by allowing potential functions, or features, that are supported by increasingly large subgraphs. Each feature has a weight that it trained by minimising the divergence between the model and the empirical distribution of the training data. A greedy algorithm determines how features are incrementally added to the field and an iterative scaling algorithm is used to estimate the optimal values of the weights.

The Random Field models and techniques introduced by Della Pietra, Della Pietra and Lafferty differ from those common to much of the computer vision literature in that the underlying random fields are non-Markovian and have a large number of parameters that must be estimated. Relations to other learning approaches, including decision trees, are given. As a demonstration of the method, they described its application to the problem of automatic word classification in natural language processing.

2.2.5 Proof Reuse and the Simulation of Human Learning

Kolbe, Walter and Brauburger [Kolbe and Walther (1998); Giesl et al. (1998)] in addition to Melis and Whittle [Melis and Whittle (1998)], have done related work on the use of analogy and proof reuse. Their systems require a lot of reasoning with one example to reconstruct the features which can then be used to prove a new example. The reconstruction effort needs to be spent on every new example for which the old proof is to be reused. In contrast, we learn our reasoning patterns from a large number of examples. A piece of related work in Cognitive Science is Furse's Mathematics Understander [Furse (1995)], MU, which stores mathematical domain and procedural knowledge in a contextual memory system, and tries to simulate how students learn mathematics from textbooks. MU builds up a uniform low-level data structure, and while the principle behind this approach is similar to that of this project, IsaNewT builds generalised tactics from a range of examples rather than focusing on the minutiae of a single example.

In terms of a learning mechanism, fairly recent work on learning regular expressions, grammar inference and sequence learning by Sun and Giles [Sun and Giles (2000)] is related. Learning regular expressions is equivalent to learning finite state automata, which are also recognisers for regular grammars.

Muggleton has done related work on grammatical inference methods [Muggleton (1990)] which automatically constructs finite-state structures from trace information.

His method IM1 is a general one and can describe all other existing grammatical inference methods. IM1 consists of first, generating a prefix tree from example traces, second, merging of states to get canonical acceptor states (which still describe only the example traces), and third, merging states which essentially does the generalisation of the structure. The generalisation, i.e., merging, is determined by a particular chosen heuristic measure.

The existing state automata learning techniques differ depending on the heuristic that they employ for generalisation. These techniques often require supervision or an oracle which confirms when new examples are representative of the inferred generalisation.

There have been various approaches to incorporate learning in planning. In the PRODIGY system [Minton et al. (1989)] a number of techniques for learning are available. The goal of the learning process is either to get control knowledge, that is, rules that describe which goal to tackle next and which method to prefer at the decision points of the planning algorithm, or learn planning operators from the change of planning states by observing an expert agent. The aim of NewT differs in both aspects as the goal is to learn new operators that are learnt from other operators and could be compared to learning of macro operators of chunks [Rosenbloom et al. (1993)].

Another difference is that these techniques use post-conditions that are not always readily available. Proof planning methods are complex and the post-conditions are only available when a method is applied in a concrete proof situation. The NewT method is applicable without any requirement for concrete pre- and post- conditions.

2.2.6 Explanation-Based Learning

There have been a number of projects on Explanation-Based Learning (EBL) as defined by Tom Mitchell [DeJong (1988)] within the machine learning community. An EBL takes four kinds of input:

1. What is *seen* in the world.
2. A high level description of what the program is supposed to learn.
3. A description of which concepts are usable.
4. A set of rules that describe the relationship between objects and actions in the domain.

From this, the EBL computes a generalisation of the training examples that is sufficient not only to describe the goal concept but also to satisfy the operational criteria. From this description it would be fair to describe the method used by NewT as an EBL. In fact, NewT extends this functionality to a more general case. The techniques used in this project allow for a generalised proof to be learnt and the learnt tactics to be improved with operators such as repetition and branching. These key functionalities are not part of EBLs.

2.2.7 Learning Heuristic Control

Schulz 2001 [Schulz (2001)], which is a continuation of previous work such as [Fuchs and Fuchs (1998); Denzinger and Schulz (1996)], investigates learning of heuristic control knowledge in the context of machine oriented theorem proving, more precisely, equational or superposition-based theorem proving. Knowledge gained from the analysis of the inference process is used to learn important search decisions, which are represented as abstract clause patterns. These are employed in heuristic evaluation functions to better guide the search when attacking new proof problems. The selection of heuristic evaluation functions for a new problem at hand is guided by meta-data.

Unlike the technique used by NewT, the learnt information in Schulz's work is not represented as a reasoning primitive (as are NewT's learnt tactics). It rather guides the search amongst the existing primitives at the global search layer instead of building up new, structured chunks of encapsulated search processes.

2.2.8 Summary

In this section different techniques and uses for learning methods have been examined. Techniques such as the Press system and LearnΩmatic use information from existing proofs to advance a new proof. The probabilistic methods used by Ron *et al.* and Della Pietra *et al.* are much more commonly used within the bioinformatics community than within the automated theorem proving community. However, the application of these techniques in these cases show how they can be used to learn patterns, which has a direct bearing on NewT.

Existing work which examines the potential for the reuse of proofs is extensive, however, these methods often focus on the features of specific examples. These techniques all have a need for context (pre- and post-conditions) which can be difficult to obtain.

2.3 Genetic Algorithms

In this section an examination of Genetics Algorithms (GAs) is presented and considered as a technique for generalising simple functions into more compound ones. GA techniques require a minimum of direction, with the input population being sufficient to randomly improve the functions. Although not known for their efficiency, these techniques are often put to good use when the specification of a problem is difficult to match.

2.3.1 Koza

John Koza explains the principals of Genetic Programming in his book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza (1992)]. Koza's work describes and illustrates genetic programming with 81 examples from various fields, particularly interesting is the 'Evolution of Subsumption', which is what the traditional approach tested in chapter 5 is based on.

Koza's approach genetically breeds populations of computer programs to solve problems by executing three steps:

1. Generate an initial population of random compositions of the functions and terminals.
2. Iteratively perform the following sub-steps until the termination criterion has been reached:
   (a) Execute each program in the population and assign it a fitness value
   (b) Create a new population by:
      (i) Reproduction: Copy existing programs to the new population
      (ii) Crossover: Create two new programs by genetically recombining randomly chosen parts of two existing programs
3. The best program at the time of termination is deemed to be the result of the genetic programming. This may be a complete or partial solution.

Although Koza describes his technique in terms of programs, functions and terminals, there is a direct correlation with tactics, proof steps and operations from the grammar used by NewT.

2.3.2 Learn2Plan

John Levine and David Humphreys' [Levine and Humphreys (2003)] developed L2Plan (learn to plan), a genetic programming based method for planning. Their system represents control knowledge as a *policy* and learns using Genetic Programming. The program's crossover and mutation operators are augmented by simple local search. L2Plan was able to produce policies which solved all the test problems it was given, outperforming hand-coded policies written by the authors. The genetic programming used for this is well suited to the task of generalising patterns into tactics, randomly generating an initial population and then evaluating their fitness against the test set used by IsaNewT produces results that would be difficult to find using other methods.

2.3.3 Summary

Genetic programming as defined by Koza is traditionally used for the development of software programs. However, the adaptation of this technique to planning as done with the Learn2Plan system shows its versatility. The evolution of simple patterns into compound tactics is a problem well suited to Genetic Programming.

2.4 Summary

This survey has covered works within the field of automated theorem proving, machine learning and genetic programming as the approach used by IsaNewT bridges all three fields. Within the field of automated reasoning there is much well-documented experimentation into re-using existing proofs in order to find new proofs. However, no other method has attempted to automatically learn new tactics from such a broad spectrum of existing proofs.

Pattern discovery is also well-documented in the machine learning community, particularly when applied to DNA sequencing and text processing. None of these techniques have attempted to learn from a proof-style tree structure, and there are no applications of pattern discovery techniques within the automated reasoning community.

Within the automated theorem proving community, traditional learning techniques have usually involved an examination of just a few examples (such as with Desimone and Kolbe, Walter and Brauburger). This has led to a predominance of learning techniques suited to this purpose. In contrast, the bioinformatics community traditionally gathers data from a much wider source (such as DNA data as used by Ron, Singer and

Tishby). In order to generate a model of a large proof corpus, the probabilistic methods employed by the bioinformatics community are much more suitable than the methods used by the automated theorem proving community.

Genetic programming techniques are mostly used to generate new programs, but systems such as Learn2Plan have previously utilised these techniques in planning problems. The Learn2Plan approach is not used within automated theorem proving but their application shows how viable this approach is.

Although L2Plan works well on a small search space (it was designed for a small subset of block moving and stacking problems in robotics) it would not be feasible in terms of efficiency to extend this to the more general theorem proving search space.

Chapter 3

Obtaining Data

In this chapter the proof corpus that will used to acquire new tactics is introduced. The requirements for the theorem proving system that will be used are discussed. An introduction to our chosen theorem prover along with an overview of the proof styles and techniques that are available to it are presented.

The tools used in order to extract the proofs from the chosen system are described. The format that the proofs are put into in order that they can be passed to the pattern discovery technique in the next stage is presented.

More importantly the notion of an abstraction with respect to the existing proof scripts is introduced. The choice of abstraction can have a large effect on the quality of the patterns discovered and also the amount of search required to fill in the missing information when the discovered tactics are applied. A number of possible abstractions are presented and discussed. The advantages and drawbacks associated with each abstraction are shown before introducing in detail the selection which will be used in subsequent chapters.

3.1 Choosing the Theorem Prover

The techniques used in the NewT approach place certain demands on the corpus that is chosen. In this section these requirements are described and several theorem prover which may have a suitable proof corpus are considered. The requirements on the corpus are the only requirements made by the NewT system; given a suitable corpus the techniques described can be applied to any theorem prover.

3.1.1 Requirements

A theorem prover with a suitable corpus of proofs to be used with NewT must be chosen, this corpus must meet precise requirements:

1. It must be stored in computational form, so that it is available for machine learning
2. It must be sufficiently large to contain many examples of multiply occurring patterns of proof
3. There must be an appropriate diversity of kinds of proof steps, i.e. sufficient different kinds of proof steps that patterns can be identified, but not so much diversity that patterns do not recur. Note that appropriate diversity is relative to corpus size and abstraction: the larger the diversity, the larger the corpus required for the re-occurrence of patterns.

Note first that the huge search space generated by resolution-style theorem provers are, unfortunately, mostly unsuitable because of requirement 3 above: typically only one or two rules of inference are used. It could be possible to differentiate rule applications by the formulae they manipulate, but these formulae are generated during the proof search and are often too diverse, e.g. millions of derived clauses. In addition, it has been suggested that interactive theorem provers may be more likely to yield interesting patterns due to the structure that people insert in their proofs. Conversely, it has also been suggested that a wholly automatic theorem prover may yield patterns as it searches for proofs in an algorithmic way.

3.1.2 Options

A selection of well-known interactive theorem provers are Isabelle, Mizar, COQ, LEGO and PVS.

3.1.2.1 Isabelle

Isabelle [Paulson (1986)], the interactive theorem prover developed at Cambridge [Paulson (1994)] satisfies the necessary criteria:

1. Isabelle's theory libraries are available online, and they also come with the Isabelle implementation (which is also available online).

2. Isabelle has several hundred theory files, including a large number based on its higher-order logic (HOL).

3. Isabelle HOL has a relatively small basis of theorems and higher-order logic axioms, all subsequent theorems are built upon these (and upon earlier theorems). It is possible to deconstruct any theorem to this basis, or to any lower level which provides the appropriate diversity.

Isabelle has some inbuilt commands which allow the proof of a theorem to be extracted.

3.1.2.2 Mizar

Mizar [Rudnicki (1992)], the interactive theorem prover designed in particular to produce human-readable proof scripts also satisfies the necessary criteria:

1. Mizar's theory libraries are available online.

2. Mizar has a huge number (thousands) of large theory files, it claims to have the largest number of mechanical proofs.

3. Mizar proofs are generated by a user specifying the next subgoal. The prover itself uses its small number of axioms within the verifier to assert that this is a correct step achievable using the internal proof rules.

The main drawback to Mizar is the difficulty in obtaining the steps used at each stage of verification by the internal Mizar mechanisms. This requires direct access to Mizar's *verifier* which is restricted by the Mizar group. However, NewT could be used with Mizar if the corpus was extracted.

3.1.2.3 COQ

COQ (2.1.2), the interactive theorem prover developed within the LogiCal (logique et calcul) project also compares favourably with the criteria:

1. COQ's theory libraries are available online, and they also come with the COQ implementation (which is also available online).

2. COQ has over 3000 entries in its lemma database.

3. COQ also has only a small number of basic axioms.

The COQ project has some automatic commands (similar to Isabelle), the steps implemented in these applications must be extracted. Unless COQ provides a tool to do this, this could be a complex step. Tactics could be generated over the set of proofs containing these automated steps, but this may mask patterns and can limit the diversity of the corpus.

3.1.2.4 LEGO

LEGO (2.1.6) is the interactive theorem prover developed at Edinburgh:

1. LEGO's library files are available online and as part of the source.

2. LEGO has a large library containing thousands of theorems.

3. As with most other interactive provers, LEGO is based on a small number of initial axioms.

The LEGO system is an older theorem prover which is not so commonly used now. It is preferable to use a theorem prover which is currently in frequent use where the discovered tactics will be of more use.

3.1.2.5 PVS

PVS (2.1.4) is the interactive theorem prover where the user guides the application of primitive inferences:

1. PVS has a large online library which is contained in the program source.

2. The PVS library contains thousands of theorems.

3. PVS is based on a small number of powerful primitive inferences including propositional rules, quantifier rules, induction, rewriting, and decision procedures for linear arithmetic.

The PVS libraries are far less easy to read than those of the other systems described here. They are a PVS system dump rather than a human-produced theory file. Although this provides no problem for an automated system, it could make inspecting existing theories in order to learn about PVS more difficult.

3.1.2.6 Summary

In spite of some minor difficulties extracting the corpus in some cases, all of the systems investigated are viable candidates for NewT.

Isabelle has a few advantages over the other systems for the purpose of extracting the proof corpus and learning from the proofs. The other interactive systems looked at sometimes involve machine checking of a human-written proof. This means that the internal mechanism only uses a few specific steps to check that each progression is valid. This can create problems with the specification that the corpus we use must be appropriately diverse. Although the user may write a rich proof, the proof collected by the internal system may well contain only the few rewrite rules required to check the correctness of any step in the proof. In particular, the automatic steps in the final product of the human proofs could mask the details which may describe many of the patterns. However, if the internal proofs were considered, it would be possible to miss the interesting mathematical steps which would entail the necessary diversity.

The corpora from the other systems could be extracted - for example, patterns could be learned what people type in and not from the verification steps. However, Isabelle is more conveniently organised for the stated purposes. As the Isabelle system provides a much more user-friendly approach to accessing the internal proof scripts it has been chosen as the basis for the proof corpus and hence NewT becomes IsaNewT.

3.2 Isabelle

As previously described, Isabelle is an interactive theorem prover developed with the language ML. Isabelle is capable of dealing with many types of logic such as first-order logic (FOL) and Zermelo-Fraenkel set theory (ZF), although the most commonly used is higher-order logic (HOL).

The syntax of Isabelle is given in the table in table 3.1.

Isar, an extension to Isabelle, has been designed in order to provide more human-readable proofs. However, increasing the readability of a proof does not help to extract the proof scripts. In fact, the older method - still traditionally called Isabelle - of a sequence of steps each applying a rule is much more suited to IsaNewt's purpose.

Although Isar is the newer method, and is fast becoming the more commonly used form of Isabelle, the vast majority of proofs in Isabelle's proof corpus are written in procedural Isabelle, this more traditional approach is also still fully compatible with all

Table 3.1: Syntax in Isabelle

Syntax	Description
&	$\wedge$, and
$\sim$	$\neg$, not
==>	$\Longrightarrow$, implication (meta level)
–>	$\longrightarrow$, implication (object level)
=	$\equiv$, $\Longleftrightarrow$, if and only if
! or ALL	$\forall$, for all
? or EX	$\exists$, exists
@	ε, Hilbert choice
%	λ, lambda abstraction

new releases of Isabelle. This is unlikely to change, as although the Isar style is more readable, the Isabelle approach is often considered to be more useful in developing a new proof. Both approaches may be used simultaneously, with more procedural commands being used within an Isar proof in order to aid development (although it is unusual for these to remain when a final proof is constructed). This means that the Isabelle approach of a sequential step approach can be used without fear of becoming obsolete in the near future.

There has also been some consideration to an automatic method of converting proofs from Isabelle to Isar (and vice-versa), this would allow IsaNewT to continue to use new proofs written in the Isar style.

3.2.1 Proofs in Isabelle

Isabelle uses a definitional approach meaning that everything must be proved from a very small number of initial axioms, namely those of higher order logic (or whatever logic is being adopted). This has the benefit of ensuring that each step has a well-founded base – i.e. no axioms are defined which are inconsistent with the existing theory. However, this also has the disadvantage of requiring that even ‘trivial’ and intuitive theorems must be proven from first principles at some point. Isabelle’s libraries are so large that most common trivial proofs are already represented within the system. These theorems can be used as rules within a proof.

Isabelle is user-directed. That is, although Isabelle has a number of automated

tools, the user decides which proof strategy and which rules and theorems to use at all times. This has the advantage of allowing Isabelle to provide both a forward and a backward proof system (even within one proof). However, this also means that the user must be familiar with the hand proof, and, unlike with other (automated) theorem provers, must understand how the proof works.

Isabelle has several important and powerful commands for the user to make use of. Some of the more frequently used are:

auto This is perhaps the most powerful of Isabelle's automatic tools. It attempts to apply all rules defined as simplification rules to all subgoals. This can make a huge difference in removing tedious simplifications and can also 'clean up' the proof so that the next step becomes clear. It uses Isabelle's classical reasoner as well as its simplifier – this enables it to perform natural deduction steps using introduction and elimination rules.

simp This works in a similar way to auto but is restricted to applying the simplifier only to simplifying one subgoal at a time.

blast Blast is one of Isabelle's classical reasoning tools. It is an integrated Tableau prover that can be used to prove subgoals which involving predicate logic. It is only applied to one subgoal at a time.

rule This command is used to apply a rule (these rules are often previously proven lemmas and theorems). It has variations

- *erule* for backward proofs,
- *drule* for forward proofs and
- *frule* which keeps the assumption so that it can be used again.

In a backwards proof construction, the user supplies a goal and applies existing rules to simplify it to simpler subgoals. This process is continued until all the subgoals are solved.

In a forwards proof construction, the assumptions of a rule are resolved with other rules to give new assumptions. This is continued until either the conclusion of the goal is an instance of some assumption, or the entire goal is an instance of a theorem.

```
lemma contrapos_pn:"[| Q; P ⟹ ¬Q |] ⟹ ¬P"
apply(rule notI)
apply(rule_tac P=Q in notE)
apply simp
apply(assumption)
done
```

Figure 3.1: An example proof from the Isabelle corpus

The proof shown in figure 3.1 is presented here simply to give an example of how an Isabelle proof looks. It begins with the declaration of the theorem (preceded by *lemma*). Each of the steps are preceded by the tactic applier *apply*. In the second rule step there is a user specified instantiation ($P = Q$) which instantiates the instance of P in the theorem *notE* to by Q from the subgoal, this user-specification requires that the addition *"_ tac"* be added to *rule*. In the third step, the user has utilised Isabelle's inbuilt simplifier (*simp*). The final step *assumption* completes the proof by instantiating an assumption to the conclusion. The finished proof is closed with the command *done* this allows the lemma to be used elsewhere by calling on the given name ($contrapos_{p}n$ in this case).

This lemma demonstrates some of the problems that we have in designing our system. Although the steps used within the simplifier can be extracted, including the instantiations made by the user would over-specify the proof as there are an infinite number of potential instantiations. By default, Isabelle instantiates any rule to the first possible situation in the assumption of a subgoal. Instead, using a specification, a user can ensure that a rule is applied to the correct part of a subgoal in order to find a proof. It would be possible to treat instances of *simp*, *auto* etc. as atomic, but as these are often overlapping commands and users invoke them at different times, it would be possible that many significant patterns would be missed. For example, one user may utilise the command *auto* to resolve a subgoal, while another may apply another 2 or three steps by hand in order to be able to find the solution using the less powerful command *simp*.

Unfortunately, we will always have the problem with our technique that we may have the correct sequence of rule steps to find a proof but be lacking the relevant instantiation information. This does not preclude our discovered tactics from being used as a guide to recommend future steps to a user, one such application of this has already been implemented by Alison Mercer in PGTips [Mercer (1996)]. The user could then examine the proof to see if extra information of this kind should be applied.

3.3 Extracting and Formatting Proofs

In almost all proofs available in Isabelle, tools such as *auto* have been used. For this reason the proof must be extracted from the system in order to include the steps which happened at these points, or *auto* must be adopted as a primitive step in the IsaNewT abstraction.

However, if *auto*, *simp*, *blast* etc. are adopted as primitives there is a risk of losing important steps which would be part of a commonly occurring sequence as many users will invoke these tools at different points. In addition, the steps covered by these tools are likely to be the simpler step that it would expect would be found as part of patterns which would be commonly used across many kinds of theorem. Isabelle has a number of tools to allow the steps performed during a call of these tools to be extracted.

Firstly, during installation of Isabelle, it is critical that the full proof derivations are kept. In more recent versions of Isabelle, these are precompiled and so full derivations are kept. In older versions compilation was done during installation, so the possibility to keep only a minimal derivation was included in order to save memory during installation. This minimal derivation includes some information on types necessary to allow each proof to be used as a rule in future, but does not include any information on the steps used to find each proof.
Extraction of the proof is then made possible by the proof syntax tool

`ProofSyntax.print_proof_of` *bool thm*

The theorem names required can be obtained by

`thms_of` *theory*

This prints out all the lemmas and theorems defined within the theory file *theory.thy*.

The output from the proof syntax command is a large tree (even for small proofs) containing some λ variables, some instantiation information, the rule names applied and the specifications of these rules along with the direction and a large amount of white noise. However, a straightforward parser can be implemented to remove any unnecessary or unwanted information and represent the proof information in a neater (and potentially much smaller) tree structure. This parser can be designed to keep as much or as little information as required.

3.4 Abstraction

The notion of an *abstraction* is defined to be the proof remaining after the formatting mentioned above has been performed. In particular, this abstraction can be varied by the amount and type of information thrown away. The abstraction used has a direct effect on how much search remains to be done in order to apply the discovered tactics as well as how much space will be needed to store the information throughout the whole tactic-formation process.

In addition, and more importantly, a bad choice of abstraction could remove any chance of finding any suitable patterns. Too vague or too precise an abstraction and there is a risk of leaving the boundaries of 'appropriate diversity' as defined in section 3.1.1.

To demonstrate some possible abstractions we look at the proof of:

```
exI: P x ⟹ ∃ x. P x
```

The proof of this theorem in Isabelle is:

```
apply (unfold Ex_def)
apply (rule allI)
apply (rule impI)
apply (erule allE)
apply (erule mp)
apply assumption
done
```

The details of each step are as formed as follows:

1. apply (unfold Ex_def)
   This *unfolds* the definition of $\exists$
   Ex_def: $\exists P \equiv \forall Q. (\forall x. P x \longrightarrow Q) \longrightarrow Q$
   to give
   $P x \Longrightarrow \forall Q. (\forall x. P x \longrightarrow Q) \longrightarrow Q$

2. apply (rule allI)
   This applies
   allI: $P x \Longrightarrow \forall x. P x$
   to give
   $P x \Longrightarrow (\forall x. P x \longrightarrow Q) \longrightarrow Q$

3. apply (rule impI)
   This applies
   impI: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q)$
   to give
   $[| \; P\; x; \forall\, x.\; P\; x \longrightarrow Q \; |] \Longrightarrow Q$

4. apply (erule allE)
   This applies
   allE: $[| \; \forall\, x.\; P\; x; P\; x \longrightarrow R \; |] \Longrightarrow R$
   to give
   $[| \; P\; x; P\; (x_2\; Q) \longrightarrow Q \; |] \Longrightarrow Q$

5. apply (erule mp) This applies *modus ponens*
   mp: $[| \; P \longrightarrow Q; P \; |] \Longrightarrow Q$
   to give
   $P\; x \Longrightarrow P\; (x_2\; Q)$

   Here the conclusion can be instantiated to match the assumption so the theorem is solved by

6. apply assumption

This shows the complete proof of the theorem *exI*. This example has no user specified instantiation or use of Isabelle tools. It was chosen to demonstrate the correlation between the steps that could be used and the exact proof as held in the Isabelle libraries.

3.4.1 Options

This section introduces a selection of possible abstractions. In each case the advantages and disadvantages of the selection are considered:

1. **Rule name only:** This potential abstraction is a list of rule names. In Isabelle, each rule name is a previously proven theorem or definition.

   Example [Ex_def, allI, impI, allE, mp, assumption]

   Advantages This abstraction allows for a wide diversity (as discussed before in our description of Isabelle, rules can be deconstructed into their component parts). It also describes an integral part of the transition from one subgoal to the next in a proof script.

Disadvantages As this abstraction leaves out much of the information needed to perform a proof step, later application of discovered tactics will require more search than other abstractions might require.

2. **Rule name with direction:** In this potential abstraction each proof step is a combination of the rule name as in abstraction 1 and the direction in which it is applied. In Isabelle, the direction is given by the tacticals *rule, drule, erule* and *frule*. Definitions are unfolded using *unfold*.

   Example [unfold Ex_def, rule allI, rule impI, erule allE, erule mp, assumption]

   Advantages This option has similar advantages to that above with the addition of a mild decrease of the search required at the application stage.

   Disadvantages As with the previous example, the amount of search left is the main drawback (in spite of a slight decrease as compared to abstraction 1). This abstraction will also require additional space (and hence processing time at other stages) in order to store this extra information. Comparison between this abstraction and the previous one relies on the improvement on search time against the extra storage space needed.

3. **Class of rule only:** In this potential abstraction, rule names are classified into groups and the proof step contains only this classification.

   Example [definition, Quantifier_elim, rewrite, Quantifier_elim, rewrite, assumption]

   Advantages This type of abstraction would allow patterns involving classes to be determined, in particular, this would help spot tendencies such as applying rewrite rules together or stripping off all outside quantifiers as soon as possible.

   Disadvantages By limiting the number of different proof steps there is a risk of reducing the set until there is not the appropriate diversity as required in the specification. There will also undoubtedly be an unreasonable amount of search to do before any tactics can be applied.

4. **Class of rule with direction:** This potential abstraction combines two of the features from previous suggestions. Each proof step consists of the direction a rule was applied and the class that the rule has been assigned.

Example [unfold definition, rule Quantifier_elim, rule rewrite, erule Quantifier_elim, erule rewrite, assumption]

Advantages Again, this abstraction will help spot tendencies of the type described previously.

Disadvantages This will increase the diversity over 3, but the excessive search space will remain.

5. **Rule name with subgoal information:**

There are a number of different options for including subgoal information. We could include operator information from the part of the subgoal the rule is applied to. It would be possible to include some instantiation information or information of which assumption a rule is applied to. However, as each rule can be applied in such a wide variety of situations the situation where no patterns at all would be found could easily arise.

6. **Main proof operator:** In this potential abstraction each rule is reduced to its most significant operator. A proof step contains the most significant operator of the applied rule.

Example [def, $\wedge$, $\longrightarrow$, $\wedge$, $\Longrightarrow$, assumption]

Advantages This technique provides a side method for including subgoal information. For example, if a rule has been applied where the subgoal includes an $\wedge$ operator it is known that this step can only be applied to a subgoal containing such an operator.

Disadvantages There are a limited number of operators available and there are many rules associated with each operator. There are also many occasions where there are more than one significant operator and the *main* operator is not clear. In most cases, the significant operator would become clear through the context of a rule application, however, context is not examined so this refinement would be impossible without a change to the stated approach. It would be possible to examine the context of a proof, however, this would also involve examining the subgoals at each step of the proof in order to understand the application of a rule. Such a context-sensitive method would require a radically different approach to IsaNewT.

7. *Main proof operator with direction:* This potential abstraction combines the direction a rule was applied with the rules main operator.

   Example [unfold def, rule $\wedge$, rule $\longrightarrow$, erule $\wedge$, erule $\Longrightarrow$, assumption]

   Advantages This abstraction has the same advantages as above with the addition of having reduced the amount of search needed

   Disadvantages As with the previous abstraction.

8. **Rule name with position in proof:** This potential abstraction contains the rule name that was applied along with a general indication of the position it was applied in the proof.

   Example [beginning Ex_def, beginning allI, middle impI, middle allE, end mp, end assumption]

   Advantages Rules which are likely to start (or end) a proof can be examined. At application time rules marked beginning would only have to be tested once for each theorem.

   Disadvantages Some patterns may appear at the start most of the time, but not exclusively. Most rules would be marked "middle" which would be wasteful.

A full test and discussion of these abstractions is performed later in the evaluation. For now, discussion of a proof step is with respect to abstraction 1, 'rule names only'. This is for clarity purposes only, abstraction 2 is at least as good a candidate.

3.5 Summary

In this chapter, the requirements for a suitable proof corpus have been defined. The alternatives have been considered the Isabelle theorem prover selected as the most suitable to NewT's needs, it satisfies the necessary criteria of availability, size and diversity. The formatting required to transcribe the Isabelle corpus into a suitable format has been described, and an introduction to the structure of an Isabelle proof has been presented.

Most importantly, the notion of an abstraction has been presented. A number of available abstractions have been presented and their pros and cons discussed. For

IsaNewT's purposes an abstraction containing either the name of the rule applied at each step on its own or this rule name with the direction of application is best suited. For simplicity, the *rule name only* abstraction will be used in the rest of this dissertation unless otherwise stated.

Chapter 4

Pattern Discovery

This chapter describes the technique for discovering commonly occurring sequences of proof steps from the chosen proofs corpus. The definition of these sequences depends on the choice of abstraction as described in the previous chapter. The process for discovering patterns is:

1. Proofs given in tree structure format are explored using machine learning techniques.
2. These proofs are modelled and all occurring sequences are read from this model.
3. The list of all occurring sequences is limited to commonly occurring sequences using a threshold value.

Any information which is contained in the abstraction of the proof corpus can be learned. However, the simplicity of the abstraction (i.e. the number of proof steps given per proof) has a direct correlation on the efficiency of any learning algorithm.

The chapter begins with an overview of the goals involved in this part of IsaNewT, followed by specific requirements for the software used. A survey of some existing methods which were tested along with some of the typical problems encountered is given. The outline to the method for pattern discovery used by IsaNewT is presented followed by a detailed description of the pattern discovery process. In conclusion, some experimental results obtained from the pattern discovery process are given.

4.1 Overview

This stage of the IsaNewT involves the search for commonly occurring patterns within proofs. In particular IsaNewT is looking for sequences of proof steps (rule names in

the given abstraction) which occur with a specific level of significance. The level of significance is defined by a pattern attaining a frequency of occurring which exceeds a specified threshold. This threshold is variable - a greater significance threshold will lead to fewer patterns being discovered but each of these having a higher frequency of occurrence.

In some cases a falsely low score may be assigned to a pattern due to small fluctuations. For example, the theorems:

le_min_iff_conj:

which contains the branch:

[conjE,notE,impI,iffI,disjCI,conjI,swap]

and:

if_bool_eq_disj:

which contains the branch:

[conjE,notE,impI,iffI,conjI,ccontr,swap]

Both have similar theorems in many places but the small differences shown here would be enough to ensure that they would not be counted as two occurrences of the same pattern. However, both patterns are common enough that if the threshold was set low enough to ensure they were found to be significant then the tactic formation stage would combine them using an $\vee$ operator:

$[[conjE, notE, impI, iffI, \vee([conjI, ccontr], [disjCI, conjI]), swap]$

In situations such as this it would not be desirable to lose these patterns. After all, any low-significance patterns which are not improved by the tactic formation stage can be discarded before application. For cases like this, it can be desirable to have a threshold set to disregard insignificant patterns rather than catch significant ones.

The patterns are discovered automatically from a wide variety of proofs taken from the Isabelle theory libraries. The intention was to provide a system that allowed a transition from a corpus of proofs to a group of commonly-occurring patterns without any human intervention. This represents a significant difference from existing work as the input proofs do not need to be hand chosen.

4.2 Specification

The requirements specified by the IsaNewT approach are as follows:

- The approach must find commonly occurring patterns within proof trees (rather than sequences).

Pre-processing forms the proof structures into $\wedge$ trees where each node on the tree is a step in the proof which takes the proof goal from one state to another. For example, figure 4.2 represents a proof which would solve the simple logic theorem 'box equals' ($[|a = b, a = c, b = d|] \longrightarrow a = d$).

A more traditional proof tree would have the goal state at the nodes and the proof step as labels on the branches which transform one goal state to another, however, because no information from the goal state is used the trees can be arranged as described. For the purposes of clarity, the full proof tree (i.e. including all the information that has been removed to form the abstraction) is shown in figure 4.3.

In this proof:

each application of *trans* represents the rewrite rule

$[|r = s, s = t|] \longrightarrow r = t$
and the step

sym represents the rule of symmetry

$r = s \equiv s = r$

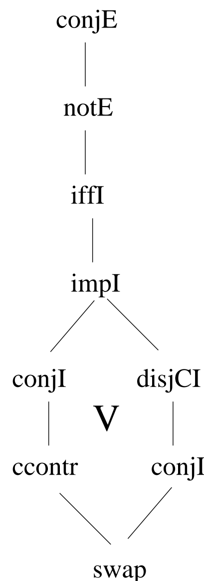

Figure 4.1: Two patterns combined by an or branching structure

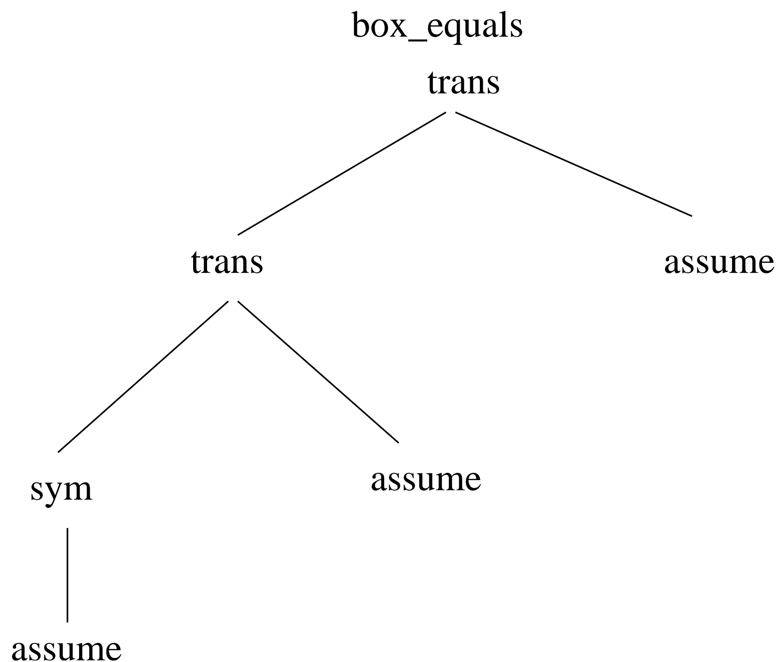

Figure 4.2: The abstracted proof represented in tree structure

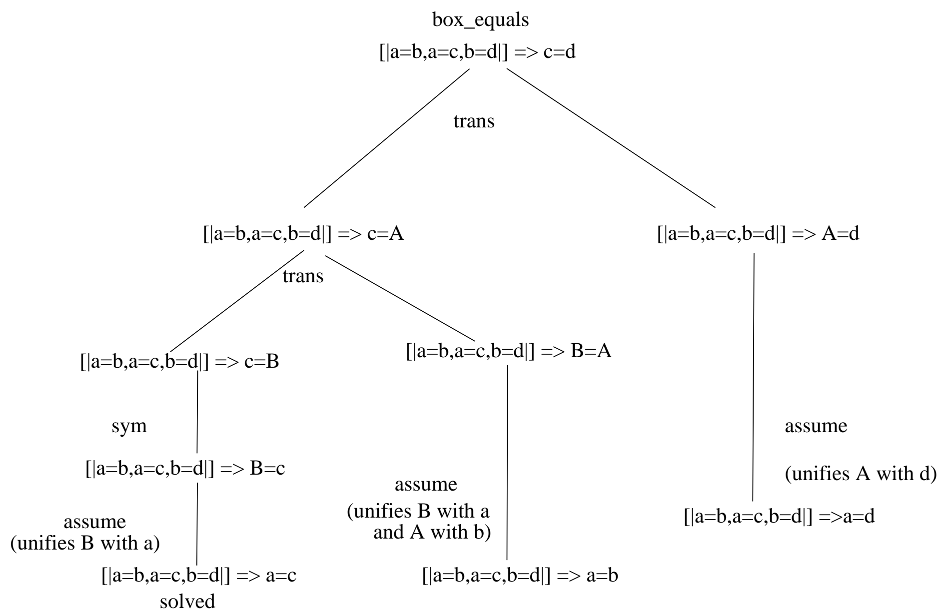

Figure 4.3: The traditional proof tree

assume simply attempts to solve the subgoal by unifying one of the assumptions with the goal. In this notation, a capital letter (such as A) represents a variable.

The approach must mine trees of this type to find common patterns.

- The approach should avoid prejudices against length

A two-step combination has a good chance of occurring often simply due to chance, whereas a long string is unlikely to appear together a high number of times without some specific reason. This requirement implies that simple 'counting' solutions would be unsatisfactory. Probabilistic methods which would allow a measure significance based on how often a string of steps occur together as a proportion of how often the rules are used within the corpus is more appropriate.

- The approach should find subsets of patterns which are themselves patterns

It is possible for a string of (say 4) proof steps which are a common pattern to be contained within a longer string of (say 9) proof steps. If the smaller pattern also occurs independently of the larger pattern, it should be considered to be a significant pattern in its own right. For example, if the discovered sequence:

$[a, b, c, a, b, c]$

has a frequency of 0.09 but the subsequence:

$[a, b, c]$

has a frequency of 0.13. Then it is desirable for both sequences to be carried forward, as the subsequence is also a pattern in its own right.

4.3 Existing Models

The initial hope was that it would be possible to find some existing 'off the shelf' methods which could be adapted to IsaNewT's purpose. Although there is an abundance of pattern matching software available, most pattern discovery software appears to be linked with the bioinformatics community for DNA string completion. However, some examples were found for text; in particular for looking for commonly occurring words within text documents. The restrictive nature of each sequence within DNA meant that the techniques designed for this were often restrictive in the variety of input data they could handle - many different rule names would not be accepted. The diversity of language meant that text tools seemed to present a more likely solution.

A description of two pieces of software which initially appeared to be good candidates is given, along with some of the problems and incompatibilities encountered.

4.3.1 Sparse Markov Transducers (SMT)

The SMT [Eleazar Eskin and Singer (2000)] algorithm works by forming a tree based on Markov Models given by training data. This specification seemed to be ideal for IsaNewT's purpose. In IsaNewT's case this training data consists of the abstracted proofs. This prediction tree has been used to provide the likelihood of certain strings appearing together as patterns.

Applied to the corpus, the SMT algorithm gives a number of patterns. However, inspection immediately shows that all the patterns have the same first step and also are significantly less varied than could reasonably be expected. This is due to the way that the SMT algorithm works. The SMT program was developed specifically for completing amino acid sequences in DNA. This means that the training data (abstracted proofs in IsaNewT's case) are assumed to be always in a specific position. For IsaNewT's purpose the position that the step names appear within the proof is unimportant with only the positions relative to other steps being relevant. For this reason, some of the patterns may in fact not be patterns at all but a trait of a certain step appearing at a certain point in the proof (such as the *assumption* step always being used at the end of a proof).

These problems were not all foreseen before testing was carried out, but the difference in intention was expected to cause problems. Originally, it was hoped that adapting the method would be a feasible alternative. However, after reviewing SMT, this no longer appeared to be a viable option.

4.3.2 Teiresias

The Teiresias [Rigoutsos and Floratos (1998)] algorithm does not make use of probabilistic methods as intended, instead it finds patterns using a scanning algorithm which counts the occurrences of each pattern. Although this does not agree with the original specifications, it was felt that examining this software could be useful. A user-defined parameter allows the number of occurrences required to define a pattern to be specified. The program then combines the patterns allowing 'wild-card' characters to find more general sequences. This looked useful as a method of finding sequences which only differ in one or two steps. The algorithm returns the most specific sequences which still have the same number of occurrences. The results show that even when a high significance level is chosen, a high (and varied) number of varied patterns are found. When a smaller significance level is given to define a pattern a very large number of

occurrences are found in just a few seconds (online).

Full documentation for the Teiresias text-word pattern discovery tool along with free (for non-commercial use) downloads are available online [Rigoutsos and Floratos (1998)].

In spite of the fact that the SMT program has a specification much closer to that desired, the Teiresias program seems to provide a better range of patterns. This was particularly useful in the initial stages for giving an indication of the types of results that could be expected later. However, as with the SMT technique, this algorithm also has significant drawbacks.

1. It does not notice the significance of two tactics always occurring together if they only appear a small number of times. For example, if the step $trans$ only occurs (say) 19 times, but 18 of these are followed by $disjI$, the Teiresias algorithm would not notice this to be a pattern.

2. There is no significance given to longer patterns, for IsaNewT's purposes it could be desirable for long patterns to have to occur fewer times to constitute a pattern, but the Teiresias algorithm treats all strings the same no matter the length.

Adapting any of the methods investigated in order that it would be applicable for IsaNewT's specific purposes would be prohibitively complex due to the overheads involved in understanding software written by someone else in enough detail to change it. It is more sensible to design a complete, specific approach from scratch, where it could be certain that it would perform exactly to the specifications.

4.4 Implementation

After examining existing methods it became clear that the best solution would be to design a specific approach for IsaNewT. As previously stated, the requirements imply that it would be desirable to look at probabilistic methods for a solution to the pattern discovery problem.

One way of quantifying the significance of a pattern of a specific length would be to fit the patterns with a generative probabilistic model that captures their statistical correlations with the occurrences. Then, whenever the trained model M is presented with a pattern $Pat = [a, b, c...n]$, it assigns to it a score (S), the normalised probability that M would emit Pat out of all possible patterns of the same length. So the score S

would be the frequency of a pattern of length n being Pat. Subsequently, a threshold above which a pattern is considered to be significant can be set. Markov Models are trained in this way and then used as predictors to compute the normalised probability of one step appearing given (a set number of) previous steps.

As patterns of varying lengths are desired, the training stage of a variation on Markov Models was inspected.

4.4.1 Variable Length Markov Models

Variable length Markov models deal with a class of random processes in which the memory length - i.e. the length of *Pat* described above - varies. Their advantage over a fixed memory Markov Model is the ability to locally optimise the length of memory required for prediction. This results in a more flexible and efficient representation which is particularly attractive in the case where it is desirable to model patterns of varying lengths.

VLMMs offer the ability to capture statistical correlations of different length scales in a single probabilistic model. Rather than estimating all possible sequences of length d that could exist in the state space, the VLMM models a selected set of sequences of different lengths. The chosen sequence set S is determined by the training data, and includes longer sequences where these appear in the data and shorter sequences when the longer ones are not required. This sequence selection scheme avoids the exponential explosion of higher order Markov Models altogether.

However, Markov Chain Models cannot be used as required on the $\wedge$-branching tree structures that describe the proofs extracted. Although trees could theoretically be learned in this fashion, the large numbers of branches within proof structures would cause a massive increase in time and space required for the model. This would make any substantial set of proofs impossible to mine for patterns.

An ideal solution has not been produced as linearisation down branches assumes the independence of branches which is not necessarily true. Other methods of dealing with such tree structures have similar problems, each is a compromise which will lose some of the detail. A decision was made to linearise the proofs by splitting down the branches. The subsequent step of genetic programming will have the capacity to partially reconstruct the link between branches, this is described in chapter 5. Some suggestions have been made for future work which involves modelling directly over tree structures.

The approach designed for IsaNewT takes the preprocessed data as a sample of proofs, then searches this data for information regarding proof step names and the number of times each step occurs in the proof sample. Each of the proofs within the sample is linearised to remove branches and corresponding weights are attached to ensure that the occurrences are not given false emphasis due to the linearisation process, this is explained in detail in section 4.5.2.

The approach examines each step of each proof and updates a database of normalised probabilities of each combination occurring. That is, as a sequence of (two or more) steps is found, it is added to the database with a normalised probability based on this occurrence in relation to the number of times the lead proof step occurs within the proof sample. The frequency is based on the lead proof step because it describes the normalised probability of the sequence occurring given the first step, i.e. 'if we have A, what is the probability that B, C, D come next'. The examination continues through the proof steps, when a sequence is found that is already present in the database (i.e. this pattern occurs elsewhere) the frequency attached to this sequence is increased.

This examination process covers every combination of every length and at every stage present in the proof sample. By examining each proof not just from the initial proof step, but also from the second, then the third and so on, it is ensured that to be found, a pattern does not need to begin at the start of a proof.

By using a probabilistic technique which refers to the overall occurrence of a step, the bias towards patterns involving more frequently used steps is counteracted.

Also, by examining different lengths of potential patterns as well as different starting points, significant subsets along with any larger patterns they are contained within are found. By definition, this will mean that any pattern of the form *abcde* which is found to be significant will automatically generate *ab*, *abc*, *abcd*, *bc*, *bcd* etc. as significant patterns. However, unless these sub-patterns also occur elsewhere in the sample (in which case it would be desirable for them to be considered as significant patterns in their own right) they will end up with exactly the same normalise probability as the larger pattern. Therefore, it is easy to weed out any patterns which are contained within any other pattern *and* have the same final frequency.

The probabilities associated with the modelled sequences of a specific rule only sum to 1 if the recursive set of subsequences are not considered. In the case where a repetition occurs, such as $[a, b, c, a, b, c]$ repeated parts of the pattern such as $[a, b, c]$ would be updated twice. This would count as two occurrences of such a pattern. Also, $[a, b]$ would also be recorded which would cause the sum of the frequencies (for a) to

exceed 1.

Ultimately, after weeding out any ‘insignificant’ sub-patterns and removing any patterns which are linked to a proof step which is only used once within the proof sample (a fairly rare occurrence which usually crops up when specific mathematical results become theorems and therefore can be used as proof steps in Isabelle), any patterns which have a frequency attached to them which is above a user-determined threshold can be weeded out. This threshold is the *significance value*.

The methodology behind the approach used with IsaNewT is similar to that used by probabilistic parsing [Eisner (1996)] which examines examples with reference to a grammar to identify which transitions are most likely to happen next. However, the approach used by IsaNewT focuses on gathering complete common sequences so patterns which are found are, in effect, a collection of steps which often occur together. Probabilistic parsing typically follows one branch and uses the information is has gleaned to predict the next step.

4.5 Finding the patterns

Here a detailed description of the approach outlined above is presented, including examples to demonstrate how each process works. A detailed description of the preprocessed data is given first, followed by an illustrated description of how the linearisation process works. A detailed technical description of how these linearised sets are mined to find significant patterns concludes the section.

4.5.1 Preprocessed data

The abstracted data is preprocessed from the proof scripts to give a simple list of lists representation for a proof tree. The end of each proof is denoted by ‘;’ which marks the end of a command in ML, and so allows the program to read each proof separately. These proofs are written in a file called ‘data’ (this filename is the default for the program, but can be easily changed by the user).

An example proof, as given in tree representation in figure 4.2, is given in IsaNewT form as:

[trans,[[trans,[[sym,assume],[assume]]],[assume]]].

Isabelle is built upon a very small number of basic axioms. These axioms form the basis of the proof system and are used as rules to prove subsequent lemmas and

theorems. Each such lemma and theorem is given a name and can henceforth be used as a rule to prove other theorems. These names of rules which are in fact the names of axioms, lemmas and theorems, are what is meant by a reference to a 'proof step'. However, this distinction is based entirely on the chosen abstraction (where the proof steps consist only of rule names). A different abstraction (higher, lower or intermediate) can be chosen fairly simply and can be easily integrated into IsaNewT. For example, by including the direction a rule is used the information could describe another abstraction. This information could easily be included in the description given below by adding the direction directly to the name of the theorem, such an adaptation would look like:

[rule_trans,[[rule_trans,[[rule_sym,assume],[assume]]],[assume]]]

So it is clear that only the preprocessing step has to be adapted in order to allow different levels of abstraction. Indeed, a selection of different abstractions has been shown and will later be examined and compared with the main choice.

4.5.2 Linearisation Process

One of the major problems encountered with the pattern discovery is the branching in the proofs. Although many pieces of software exist for identifying patterns in sequences which could be adapted for use with proof structures, no existing methods or unimplemented theorised techniques which identify patterns within tree structures have been identified. It was suggested that branches could be ignored or simply treated as a special case i.e. a 'split token'. However, the frequency of occurrence of some sort of branching structure within a proof means that in this case many interesting patterns may well be lost.

The technique decided upon was to split the proofs into separate sequences and give weights accordingly i.e. for all the steps before each split the weights are given as:

$1/(b*w)$

where b represents the number of branches resulting from the split and w represents the weight immediately after the split. All the steps at the end of each branch have weight 1 – so a tree which has 2 two-way splits would have a weight of 1 at the end of each branch, 0.5 on every branch between the last two splits and 0.33 before there is ever a split point.

The list of lists representation is converted to weighted lists using a recursive program which:

$$[[0.33, trans], [0.5, trans], [1, sym], [1, assume]]$$

The original tree is shown by figure 4.4.

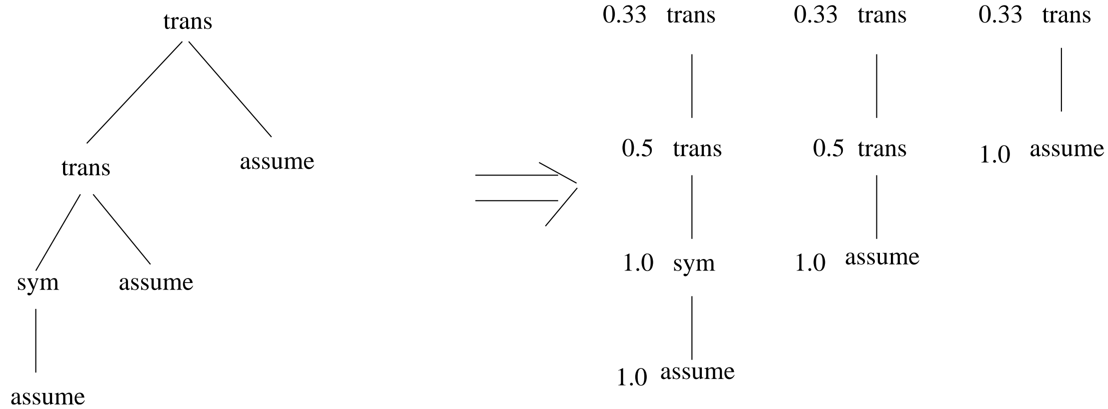

Figure 4.4: Picture of example proof linearisation.

- The process cycles until the end of the nested list is found and then assigns the weight 1 to each node of each branch at this level of the nesting.
- Adds together the inverse of the weights given to each branch and assigns the weight $1/\textit{total weight}$ to the next highest nest.
- Continues this process until the top of the list has been reached.

Example:
[trans,[[trans,[[sym,assume],[assume]]],[assume]]]

[trans,[[sym,assume],[assume]]] $\wedge$ [assume]

[sym,assume] $\wedge$ [assume] $\wedge$ [] assign weight 1

[assume] $\wedge$ [] assign weight 1 $\wedge$ [] assign weight 1

[] assign weight 1 $\wedge$ [] assign weight 1 $\wedge$ [] assign weight 1

[[1,assume]] ∧ [[1,assume]] ∧ [[1,assume]]

[[1,sym],[1,assume]] ∧ [[1,assume]] ∧ [[1,assume]]

new weight = $1/(1/1+1/1) = 0.5$
[[0.5,trans],[1,sym],[1,assume]] ∧ [[0.5,trans],[1,assume]] ∧ [[1,assume]]

new weight = $1/(1/0.5+1/1)$ = 1/3 = 0.33 (for each amalgamation)
[[0.33, trans],[0.5, trans],[1,sym],[1,assume]] ∧
[[0.33,trans],[0.5,trans],[1,assume]] ∧ [[0.33,trans],[1,assume]]

These weights are incorporated simply at the point where the Markov Model is updated. It would be much more elegant to have software which learned Markov Models directly from the tree structures but as most proofs contain (often multiple) branches, the increase in complexity made this not feasible. Some suggestions have been made to implement a method for learning directly from trees in future.

For the purposes of the tactic formation step later, each time a split is found the step preceding it (i.e. the step whose application resulted in split) is noted. These steps are kept in a 'split token' file so that later it will be known that they appeared at a branching point.

In the example given, the step *trans* is labelled as a split token and kept in the split token file. In this way, if both branches represent patterns the tactic formation step will note that *trans* should be reformed as a branching point and can reform the branches. If the connection over a branch is indeed significant, one branch representing a commonly occurring pattern should result in the other branch also being represented. This means that any truly significant correlation between branches should be rediscovered at the tactic formation stage.

4.5.3 Finding Patterns

1. The training data is searched with a simple sweep to find out the names of all the different tactics (T). The occurrence (O(T)) of each of these tactics is found.This occurrence incorporates the weights so that each tactic is counted a correct number of times. The final figure for each tactic will always be a whole number.

2. The model is trained on the data to give probabilities for each combination oc-

currence O(T).

3. The results are returned as all the patterns with a frequency above a user-specified threshold.

Step 1 is straightforward.

Step 2 involves assigning a frequency to every possible combination of two or more consecutive steps within a proof. For example a proof $[a,b,c,d]$ would generate probabilities for $[a,b],[a,b,c],[a,b,c,d],[b,c],[b,c,d]$ and $[c,d]$.

If no probability for a combination of steps ($[a,b,c]$) already exists in the database then the normalised probability given is:

$P = (1/o) * w$ where o is the number of times the tactic a occurs and w is the weight assigned to the tactic a at that particular point. The weight comes from the first tactic in the sequence as this weight represents the sequence beginning from this point, It can be thought of as a weight associated with the branch, not with the step. This P is the normalised probability that a given sequence is used together within the proof corpus.

If a normalised probability is already contained in the database ($OldP$) then P is added to this value. $OldP$ accounts for the occurrences of this sequence before this point and P accounts only for this occurrence so adding P to $OldP$ keeps the frequencies up to date.

Example:

For the purposes of this example the following occurrences to each step are assigned.

O(trans) = 67

O(sym) = 14

O(assume) = 157

Note that O(assume) will never be used as *assume* by its very nature can only be used at the end of a proof. The sequences are measured from their first step and *assume* can never be a **first** step.

For demonstration, it can be assumed 3 occurrences of *sym* being followed by *assume* (each obviously with weight 1 as assume is always at the end of a branch) have previously been found, therefore:

$P([sym, assume]) = 3 * (1/14) * 1 = 0.214$

is already contained in the database. No other relevant entries exist in the database.

The previous example continues to be used, but for the sake of brevity only the first list is looked at:

$$[[0.33, trans], [0.5 trans], [1, sym], [1, assume]] \tag{4.1}$$

First $[[0.33, trans], [0.5 trans]]$ is considered. The frequency associated with this is computed as $P = (1/67) * 0.5 = 0.0007$. This is added directly to the database as there are no previous entries for this sequence. The following probabilities are then calculated and added to the database:

$P([[0.33, trans], [0.5 trans], [1, sym]]) = (1/67) * 1 = 0.0149$

$P([[0.33, trans], [0.5 trans], [1, sym], [1, assume]]) = (1/67) * 1 = 0.0149$

$P([[0.5 trans], [1, sym]]) = (1/67) * 1 = 0.0149$

$P([[0.5 trans], [1, sym], [1, assume]]) = (1/67) * 1 = 0.0149$

Then $P([[1, sym], [1, assume]]) = (1/14) * 1 = 0.0714$ is calculated, but as the sequence already has an entry in the database that entry must be updated to be:

$P([[1, sym], [1, assume]]) = 0.214 + 0.0714 = 0.2854$

This makes sense as there are now 4 occurrences of *sym* being followed by *assume*, this is indeed a little under $1/3$ of all occurrences of *sym* (which was initialised at 14).

This *update* step of the process forms a Markov Model which contains the probabilistic information of every possible combination of steps that occurs in the corpus.

Step 3 is also quite straightforward. All sequences which do not constitute a pattern or are otherwise not useful are removed. All potential patterns with a final frequency less than the threshold are discarded. In addition, any patterns whose first element has O(T) = 1 are also discarded as discussed earlier.

It could be argued that any step which is used less than (say) 5 times should be discarded as a pattern leader. However, even if a step is used only twice, if both occurrence are followed by the same sequence of steps, the possibility that this is significant must be considered. For this reason, only patterns associated with single-use proof steps are removed. In addition, these rarely-used steps are also unlikely to occur in future proofs, so they will not cause unnecessary search at the implementation stage.

It should be noted that steps which occur only a few times are rare and constitute only the smallest number of the discovered patterns (only two or three even in a large set).

From the (*extremely* small) example above, the threshold could be chosen to be around 0.2 which would eliminate most of the patterns with small probabilities.

Similarly, as discussed earlier, any sequence which is a direct subset of another *with the same frequency* is also discarded as it only occurred because of its inclusion as part of the larger set.

In the example, it can be imagined that this is the last sequence to be considered and no more updates are carried out. All patterns except $[trans, trans]$ (0.007),

$[sym, assume]$ (0.2854), and $[trans, trans, sym, assume]$ (0.014) can be discarded without considering the threshold. It can now be seen that the only pattern to survive the threshold would be $[sym, assume]$. If figure 4.4 which shows the whole proof is again examined, it can be deduced that including the other branch as part of the sequences and updating the probabilities associated with the other branches of the tree would cause $[trans, trans]$ to be discarded also. This is because the frequency associated with it would be updated from 0.007 to 0.014, this would make it a subset of $[trans, trans, sym, assume]$ which holds the same probability meaning that it is unnecessary.

Keeping the final probabilities associated with each pattern which survives can be useful in later stages. In particular, if a low significance threshold is used, many patterns which are related to each other but are not exactly matched as the process demands can be caught.

The tactic formation step fuses many of these together, but some patterns will still be present which truly are below a good threshold. If a tactic has a low score (for example 1) after the Genetic Programming stage, then it has not been combined with any other pattern to form a more compound tactic. If such a tactic also had a near-threshold probability at the pattern discovery stage, then it can be deduced that it was not truly significant and it can be discarded. Earlier the wisdom of using a threshold which is just high enough to rule out insignificant patterns (in order to catch patterns with borderline frequencies) was mentioned. At this stage in the IsaNewT process it is possible to rule out false positive patterns which have not been improved by genetic programming. Testing has shown that with the chosen proof corpus a new threshold 0.02 higher than the original threshold is usually appropriate for this pruning.

4.6 Experimental Results

Finding the patterns is the most significant step in discovering new tactics. These discovered patterns form the basis for the new tactics, no later stage adds any new information. However, at this point it is difficult to accurately evaluate any of the discovered patterns. Indeed, until the final evaluation stage where the completed tactics are tested using a fully automated Isabelle prover, it is difficult to gain any true measure of success.

Therefore, this results section is concerned mainly with explaining the choices in the changeable metrics of the threshold value and the number and choice of input

proof scripts. Also presented are some examples of discovered patterns along with explanations of where and why they might have originally arisen.

4.6.1 Input Choices

The Isabelle theorems are split into two sets - *training* and *test* with a 1:1 ratio. This allows each set to have a large component of theorems from a variety of domains. Within Isabelle, each session of proof created by a user is kept in a *theory* file. Each of these files represent a selection of proofs mathematically related to each other. In order to keep these sets as even as possible in terms of information, the proofs from each *theory* file were randomly split between the two sets. This was to ensure that the training set and the test set would be evenly matched in terms of theorem length, type, topic and complexity.

As input, any theorems from the training set can be chosen. Some interesting questions arose when considering how much of an impact the choice of theorems would make. It would appear likely that tactics formed from a certain kind of theorem would perform best when applied to the same kind of theorem (this will be discussed more fully and tested later in the application chapter). Following this reasoning, the differences in the probabilities of the patterns discovered was examined both when a random set of input data was chosen and when a set of input data was chosen that came from the same theories or types of theories. As the training and test set were split evenly across Isabelle theories it was simple to use a test set originating in a theory (or set of theories) and to choose a training set from the same set of theories.

Using a 'mathematically similar' set of theorems gives a better probability rate for patterns - this is expected as the process behind proving mathematical theorems is not independent of domain or human influence (so two independent theories written by the same person may have more in common than two theories written by different people). In this context, 'mathematically similar' means theorems within the same domain (higher-order logic (HOL) theorems, natural number theorems etc). This is shown by comparing the results of sets of 100 input theorems. By comparing the thresholds required to get 20 patterns from a set of 100 theorems the different results gained from a random selection, and from a directed selection can be compared. The results are given in figure 4.5, it is clear that a chosen selection gives much better results in most cases. In the situations where no improvement has been found it can be postulated that these proofs are of a more general kind in nature. Every set of

results given were run multiple times on a range of inputs, the results given describe the average output of these runs.

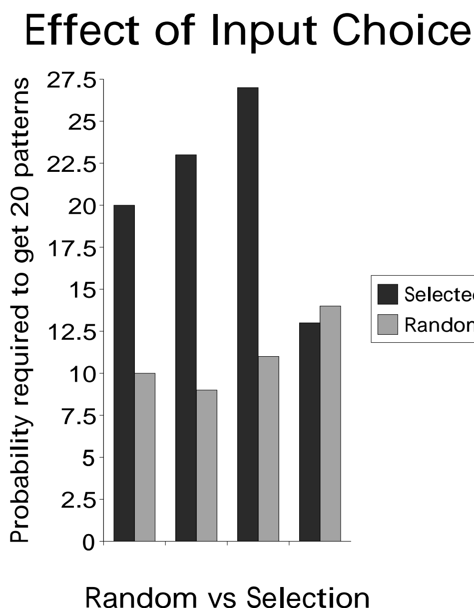

Figure 4.5: Threshold required to gain 20 patterns from random selections against chosen selections. Four comparisons are made. This graph describes the average of 15 runs across different domains.

As the domain is broadened, the frequency given to the patterns decreases in general. This does not happen in every case, as some steps represent basic (e.g. simplification) rules which will often be used in proofs of any kind. However, the main claim for IsaNewT is that tactics can be discovered using a process which requires no human intervention at all. In fact, the domain selection procedure can be done automatically using the Isabelle hierarchy, the method for which is described at the end of the next section.

4.6.2 Threshold

The variable which can make the most difference is the threshold value. However altering the threshold can in some ways artificially alter the result. It is easy to claim that (say) 90 patterns from 100 theorems can be found, but if this occurs because the threshold has been set too low, then these do not truly represent what is wanted in terms

of patterns.

The threshold can not be arbitrarily decided as can be seen from the previous discussion on input choices. The size of the input, and the relation each input theorem has to each other has a significant effect on the required threshold. Although the threshold has previously been manipulated in order to obtain a specific number of patterns, this is only really useful as a diagnostic and examinatory tool (as for discussing the effect of inputs above). Using this as a measure to guide the threshold would artificially alter the results, which should be avoided.

Therefore, it has been most useful to leave the threshold level to the discretion of the user. In a fully automated tactic formation system however, an automated threshold selector is required. For this purpose a set of rules have been designed, these reflect the average optimal threshold values from a large set of test runs:

- Begin with two default values for 100 theorems (which were chosen after extensive testing):

  1. if the theorems are chosen randomly, with no consideration given to their mathematical domain, then the threshold is set at 0.1;
  2. if the theorems are *selected* from a mathematically similar set then the threshold is set at 0.2.

- The threshold is reduced proportionally to the set size being increased:

  $Set\ Size/100 = Proportional\ Set\ Size$

  $Threshold = Original\ Threshold/Proportional\ Set\ Size$

In order to determine whether a set is from a mathematically similar set, theorems from theory files deemed to be similar have been combined into subsets so that any selection of theorems chosen from such a set can be said to be *selected*. In the Isabelle theory structure, this hierarchy is already in place so this procedure can be fully automated. When a new theory is begun, the user heads the file with the theory dependencies. Theory dependencies in this set indicate which domain the theorems in the theory belong to, this allows the IsaNewT process to continue to be automated even if the patterns are being used to prove (or recommend a step) for a brand new theory.

4.6.3 Some Patterns

To understand the patterns found, it is necessary to examine some in order to see where they came from and how they originally applied to theorems in the proof corpus. Here two examples are presented and their origins discussed. This examination also allows the appearance of patterns to be understood when it may not be clear exactly why such patterns occur.

4.6.3.1 Example 1

The first example pattern appeared in a *selected* set of 500 theorems with a frequency of 0.134. This domain contained the basic higher-order logic theorems such as propositional rules and rewrite rules. The pattern is given in figure 4.6.

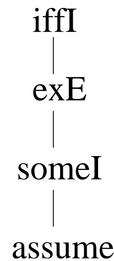

Figure 4.6: Pattern from a set of 500 *selected* theorems.

This pattern can be found contained in the proof of:

$conjI = [|\ P; Q\ |] \Longrightarrow P \wedge Q$

It completes one branch of the proof. Unfolding the definition of *and* then applying the rule *allI* leaves the subgoal:

$!!R.\ [|\ P; Q\ |] \Longrightarrow (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

Here the $!!x.$ represents an unknown constant. So the pattern is followed by applying:

$impI\ \ (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$

to this subgoal, which leaves:

$!!R.\ [|\ P; Q; P \longrightarrow Q \longrightarrow R\ |] \Longrightarrow R$

As with the discovered pattern:

$mp\ \ [|\ P \longrightarrow Q; P|] \Longrightarrow Q$

is applied to get two subgoals, only the first is shown as this is the branch that the discovered pattern comes from:

$!!R.\ [|\ P; Q; |]\ P$

This is a simple instantiation, accomplished using *assume*.

This pattern is also used to complete a branch in the proof of:

$disjI1\ \ P \Longrightarrow P \vee Q$

This proof begins by unfolding the definition of $\vee$, followed by an application of *allI* and an application of *impI*. These steps leave the subgoal:

$!!R.\ [|\ P;\ P \longrightarrow R\ |] \Longrightarrow (Q \longrightarrow R) \longrightarrow R$

To this *impI* is applied to get:

$!!R.\ [|\ P;\ P \longrightarrow R;\ Q \longrightarrow R|] \Longrightarrow R$

Again *mp* is applied to get two subgoals, the first of which is:

$!!R.\ [|\ P;\ Q \longrightarrow R|] \Longrightarrow P$

Which is solved using an instance of assume.

In fact, a second examination of the proofs shows that the tactics formation stage may well combine these two patterns to provide a compound tactic that subsumes both. This is shown in figure 4.7.

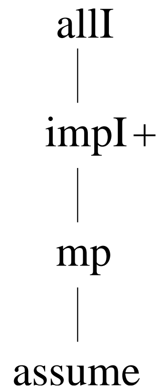

Figure 4.7: Combined pattern after tactic formation.Here *impI*+ means 1 or more repetitions of *impI*.

4.6.3.2 Example 2

The second example pattern (4.8) appeared in a random set of 500 theorems with a frequency of 0.049.

This pattern solves the subgoal:

$!!xx_2.\ [|Pa;Px;Px_2;\forall y.\ Py \longrightarrow y = x_2|] \Longrightarrow x = x_2$

which appears in the proof of the theorem:

$some1_equality\ \ [|\exists!x.\ Px;Pa|] \Longrightarrow \varepsilon P = a$

The steps in the pattern are as follows:

$allE \; [|\forall x.\ Px; Px \Longrightarrow R|] \Longrightarrow R$

$mp \; [|P \longrightarrow Q; P|] \Longrightarrow Q$

The full proofs of $disjI1$, $conjI$ and $some1_equality$ are contained in the appendix A.

These are just examples of places these two patterns originally appeared, for them to have been designated as patterns, they must appear in many other places.

4.7 Summary

A description of the pattern discovery technique has been presented. After exploring the options available from existing methods, a new process was developed specific to the requirements. The parameters contained in this approach have been explored and some examination of the results so far has been provided.

Examination of techniques for solving problems of this type (pattern discovery) threw up a number of issues. For the most part, existing solutions deal with DNA modelling, and those which could be adapted to deal with rule names, do so in a less than ideal way. None of the software examined would be easily adaptable to the purpose of this thesis. Also, none of the software discovered patterns over trees.

When developing the new technique a compromise in dealings with the type of tree structures described by the proofs had to be made. No ideal solution to this problem was found in existing software and theory. The decision to linearise the proofs down the branches meant losing any important connections between different branches. However, it was decided that it was more important to keep the connection between successive steps.

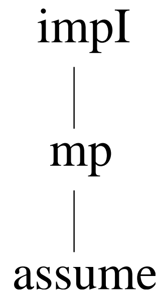

Figure 4.8: Pattern from a random set of 500 theorems.

The tactic formation step will recover much of the lost information, particularly if it is significant. If one branch is significant, and there is a direct link between it and another branch of the same proof, then the second branch will also be found to be significant. This ensures that both branches will occur as patterns in the tactic formation stage.

In order that patterns associated with common rule names are not reported as significant when they arise simply due to chance it was decided to use probabilistic methods to form the basis for the pattern discovery approach. The specification led us to the training stage of Markov Models. In particular, Variable Length Markov Models which allow modelling of patterns of any length proved an ideal solution.

The parameters within the approach have been examined and tested. By selecting the input (through an automated process) and tailoring the threshold to the input given, it has been possible to optimise the patterns returned.

Many patterns have been extracted using the techniques described and it has been shown that examining them can give some insight into how they may have come about. This information is included to allow a check that the patterns discovered are reasonable, this was particularly useful during the developmental stage. This information has no real value to the final tactics. However, it can give an indication of how well they may be expected to perform, but this cannot truly be tested until after the patterns have been combined in the tactic formation stage.

Chapter 5

Tactic Formation

5.1 Introduction

This chapter describes the technique for combining the results of the pattern discovery into compound tactics. This chapter begins with an overview of the goals, followed by a specification for solving the problem. The traditional Genetic Programming approach is described and an explanation given of why it is suited to IsaNewT's purposes. A description of the adaptation of the traditional GP methods is given followed by a detailed description of the newly developed technique.

At this stage there are some commonly occurring patterns arising from the previous stages. However, these patterns are linear and bear little relation to the original proof structures. Also, in many cases the patterns that have been found differ by only a single step. An example of this is given in figure 5.1, these examples can be seen in the proofs of $conjI$ and $conjunct1$ respectively. It is clear also that some steps may be repeated a number of times. However, the pattern discovery software will find separate instances of them as separate patterns, as can be seen in figure 5.2. In addition, where possible it is desirable to reconstruct the branching structure that was lost during the linearisation process.

5.2 Specification

As discussed, an approach for combining the patterns that have been found into compound tactics is desired. There are four items to consider:

Figure 5.1: Two patterns which show potential for an $\vee$ introduction.

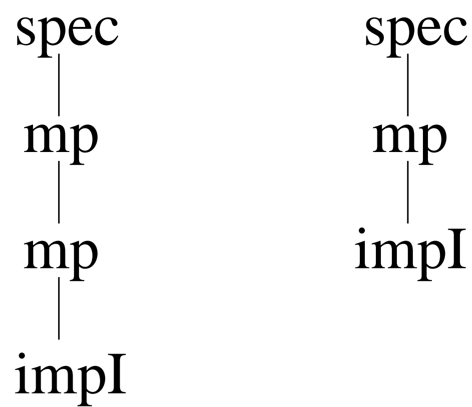

Figure 5.2: Patterns which show potential for a *plus* introduction.

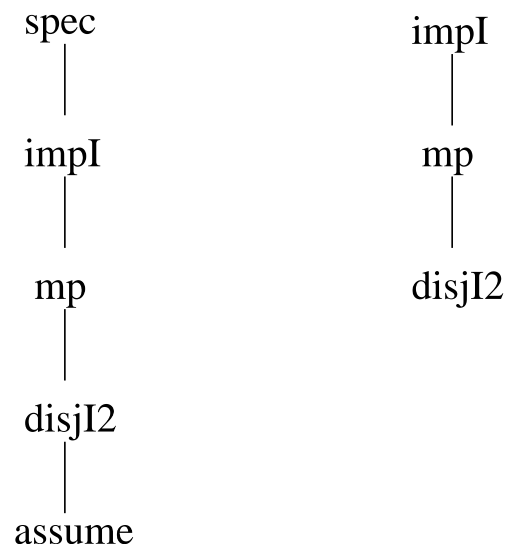

Figure 5.3: Two patterns which show potential for macro introduction.

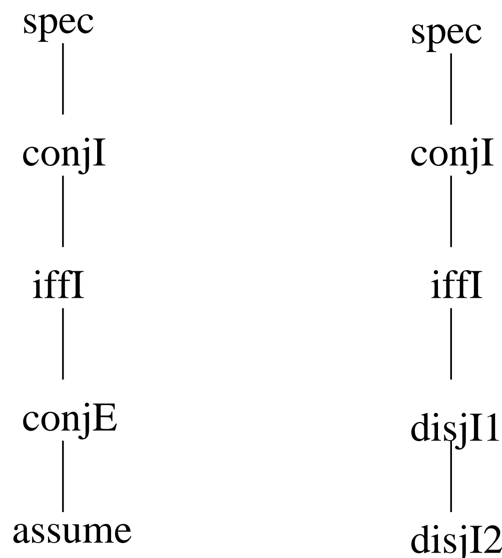

Figure 5.4: Two patterns which show potential for an $\wedge$ introduction. Note that here *iffI* is stored as a step which results is a branch

1. **Macro Formation** - Macros which represent internal parts of tactics which occur commonly are desirable. An example of two candidates is given in figure 5.3

2. $\vee$ **introduction** - Combinations of possible patterns (as in 5.1) together with an $\vee$ operator is desirable.

3. $\wedge$ **(re)introduction** - The reintroduction of the branching information lost during the linearisation procedure earlier is desirable. An example of two candidates is given in figure 5.4

4. $+$ **introduction** - A representation of repeated steps with a *plus* operator to denote 1-or-more repetitions is desirable.

5.3 Grammar

Some care is required over the choice of the tactic language. The choice ranges from regular grammars, via a limited set of tacticals to a general programming language, such as ML (as used in LCF [Gordon et al. (1979)]). A parsimonious language will be better suited to genetic programming, e.g. a limited set of tacticals. Moreover, the language must not require information that cannot be obtained by analysis of the proof corpus. For instance, it is no use including while-loops or if-then-else, if their

conditions cannot be constructed.

Non-conditional forms of repetition and non-determinism must be used instead. It has therefore been decided to represent generalised patterns in the same way as Kerber *et al.* [Jamnik et al. (2002)]. using the following language $\mathcal{L}$ which is defined as:

$r \in \mathcal{L}$ for rule name identifiers r

$m \in \mathcal{L}$ for macro identifiers m

$$\left.\begin{array}{l}[L_1, L_2] \in \mathcal{L} \\ (L_1 \vee L_2) \in \mathcal{L} \\ (L_1 \wedge L_2) \in \mathcal{L}\end{array}\right\} \text{ for } L_1, L_2 \in \mathcal{L}$$

$L+ \in \mathcal{L}$ for $L \in \mathcal{L}$

The rule name identifiers denote the rule names (or proof step associated with the abstraction) which appear in the extracted proof sequences. Macro identifiers are used as abbreviations for a pattern $L \in \mathcal{L}$.

The operators have the semantics of tacticals. The term $[L_1, L_2]$ is interpreted as sequencing (L_2 is applied after L_1), $\vee(L_1, L_2)$ stands for a disjunction (either L_1 or L_2 is applied), an $\wedge(L_1, L_2)$ has the semantics that L_1 is applied to one subgoal and L_2 to the other subgoal. The term $+L$ denotes an arbitrary number (equal to or greater than one) of repetitions of L. In order that this operator can be used with the information given, its use is defined to be '*as few as necessary*'. In other words, this step is repeatedly applied only until the next step in the tactic can be used. In the situation that it occurs at the last step of a tactic, its use is defined to be '*as often as possible*'.

An example of a generalised tactic would be:

```
[step, ∨([step,step],[step]), +[step, step], macro(m), ∧([step],[step])]
```

where $macro(m)$ is itself an identifier for a tactic.

5.4 Genetic Programming

If the discovered patterns are considered as simple tactics, then this stage of the approach can be though of as an evolution of these simple tactics to compound ones. More specifically, these simple tactics are being evolved into ones which are more suited to the task (describing the proof corpus). Evolutionary Programming is ideal for this purpose, providing an approach which will allow a generation of increasingly better tactics with each increment. As there is no exact measure of when the optimum tactic set has been found, it is desirable to use a technique such as EP where the initial

population converges on a solution. Therefore stopping the process at any time will still provide a useful result.

Genetic Programming (GP), a specific instance of EP, is ideal for IsaNewT as the tactics being generated can be though of (in some sense) as programs to solve a problem.

5.5 Traditional GP Method

Koza's GP approach genetically breeds populations of computer programs to solve problems by executing three steps:

1. Generate an initial population of random compositions of the functions and terminals.
2. Iteratively perform the following sub-steps until the termination criterion has been reached:
   (a) Execute each program in the population and assign it a fitness value
   (b) Create a new population by:
      (i) Reproduction: Copy existing programs to the new population
      (ii) Crossover: Create two new programs by genetically combining randomly chosen parts of two existing programs
      (iii) Mutation: Choose one program and randomly mutate a point on it by adding or removing an operator or command.
3. The most fit program at the time of termination is deemed to be the result of the genetic programming. This may be a complete or partial solution to the specified problem - i.e. a partial or complete program.

Although Koza describes his technique in terms of programs, functions and terminals, there is a direct correlation between this and tactics, proof steps and operations from the grammar.

5.5.1 Implementation

A seeded Genetic Programming implementation is used where the discovered patterns are the initial population set. Much of the change comes from combinations of two patterns with the approach ending after a time-out. The fitness function scores tactics over

the initial population not over the entire corpus. This is because the patterns included in the initial population are representative of the most commonly used parts of the corpus. Scoring over the entire corpus would not only be incredibly time-consuming but can result in over-generalisation of tactics which would require increase the search space at the application stage. This algorithm is covered by the following steps:

1. All the patterns are scored according to how many other patterns within the initial population (the set of discovered patterns) they subsume. Initially many of the scores will be 0. However, all patterns are ordered by rank (where the first pattern scored highest) according to their score. Patterns with the same score are ranked arbitrarily.

2. One of three procedures below are applied. The choice of procedure is randomly chosen, although with weights which provide a bias towards certain procedures. Crossover (preferred - 50%), mutation (25%) and reproduction (25%). These weights were chosen after extensive testing. The higher than normal mutation weight is required in the early stages to ensure the reintroduction of branches and operators. This allows later crossovers to work as expected and speeds up the initial process.

    - Crossover works generally by randomly choosing a branch of each tree (P1 and P2) and swapping them. However, the initial population consists solely of patterns not trees so crossover works by choosing a point on each pattern (P1 and P2) and adding the remainder to the other by inserting an $\vee$ if the last step matches (shown in figure 5.5) and an $\wedge$ if it doesn't (shown in figure 5.6). For the purposes of the figures, a tree branch represents and $\wedge$ branch. An $\vee$ branch results in a graph as in figure 5.5, in this case an $\vee$ is included for clarity. Although allowing a small random possibility for an $\wedge$ to occur even with matching ends allows more variety. This permits the situation where a ($\wedge$) branch occurs but coincidentally both branches end the same. A random number generator chooses whether a portion of the pattern will just be "cut-and-paste", whether branches will be introduced or (in the case that branches already exist) that traditional branch-swapping occurs (shown in figures 5.7 and figure 5.8).

      In each case a small random probability is allowed for one of the other feasible options to occur. For example, even if both patterns already contain

a branch there is a chance that another random branch will be inserted instead of the traditional subtree swap.

- For mutation, a random point in the pattern is chosen (via random-number generator (RNG)) and the pattern is mutated at that point. If that point is an operation ($\wedge$, $\vee$ or $+$), the positioning is changed slightly. One branch of an $\wedge$ can be moved (so effectively the split starts one step earlier or later) and the length of an $\vee$ or a *plus* can be lengthened or shortened by one step. For example, if the pattern P_1 chosen is:

  $[a,b,c,\vee([d,e],[f,g]),h]$

  and the point chosen to mutate is the $\vee$, then the boundaries of the $\vee$ can be moved to get:

  $[a,b,c,\vee([d],[f,g]),e,h]$

  which effectively adds an element of the $\vee$ to the main sequence, or it could be expanded to get:

  $[a,b,\vee([c,d,e],[c,f,g]),h]$

  If the pattern *Pat* chosen is:

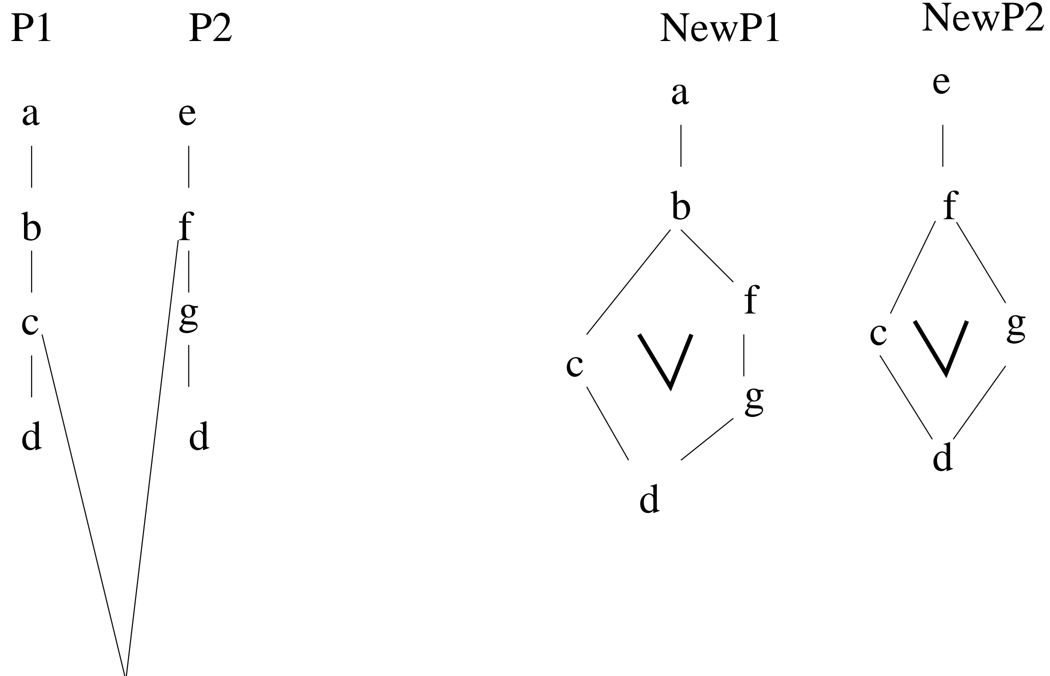

Figure 5.5: Results of a crossover when no branches are present and the patterns end with the same step

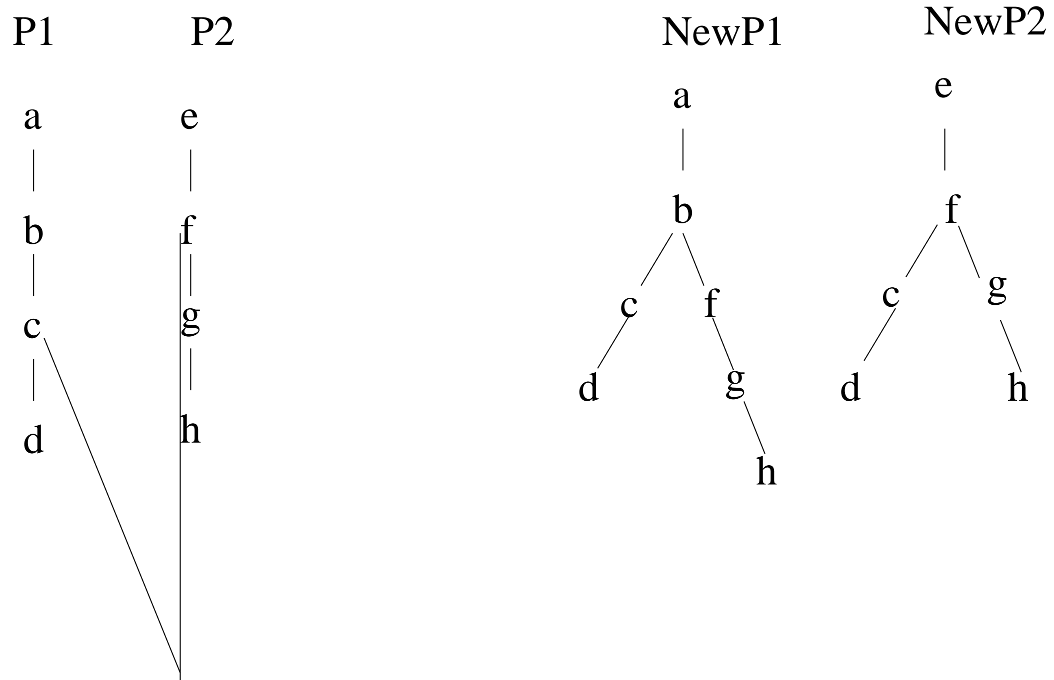

Figure 5.6: Results of a crossover when no branches are present and the patterns end with different steps

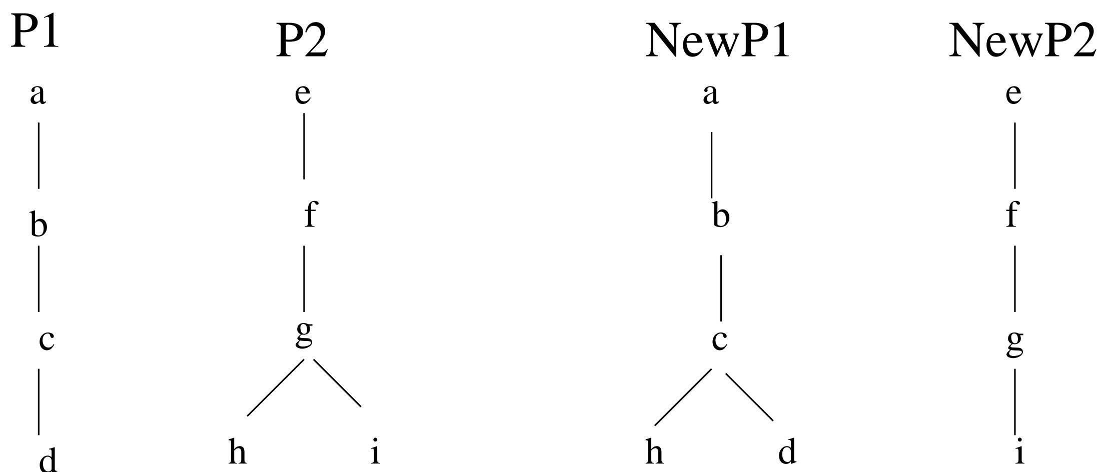

Figure 5.7: Results of a crossover when one of the candidates already contains a branch

$[a, b, c, +(d, e), f, g]$

and the point chosen is the *plus*, then:

$[a, b, +(c, d, e), f, g]$

could be obtained or the *plus* could be shrunk to get:

$[a, b, c, +(d), e, f, g]$

If the pattern *Pat* chosen is:

$[a, b, c, \wedge([d, e, f], [g, h, i])]$

and the point chosen is the $\wedge$ then the result could be any one (randomly) of:

$[a, b, \wedge([c, d, e, f], [g, h, i])]$

$[a, b, \wedge([d, e, f], [c, g, h, i])]$

$[a, b, c, d, \wedge([e, f], [g, h, i])]$

or:

$[a, b, c, g, \wedge([d, e, f], [h, i])]$

If the pattern chosen (or the point in the pattern chosen) has no operator then the step at that point can be randomly swapped (for another step), removed or an arbitrary new step or operator can be added.

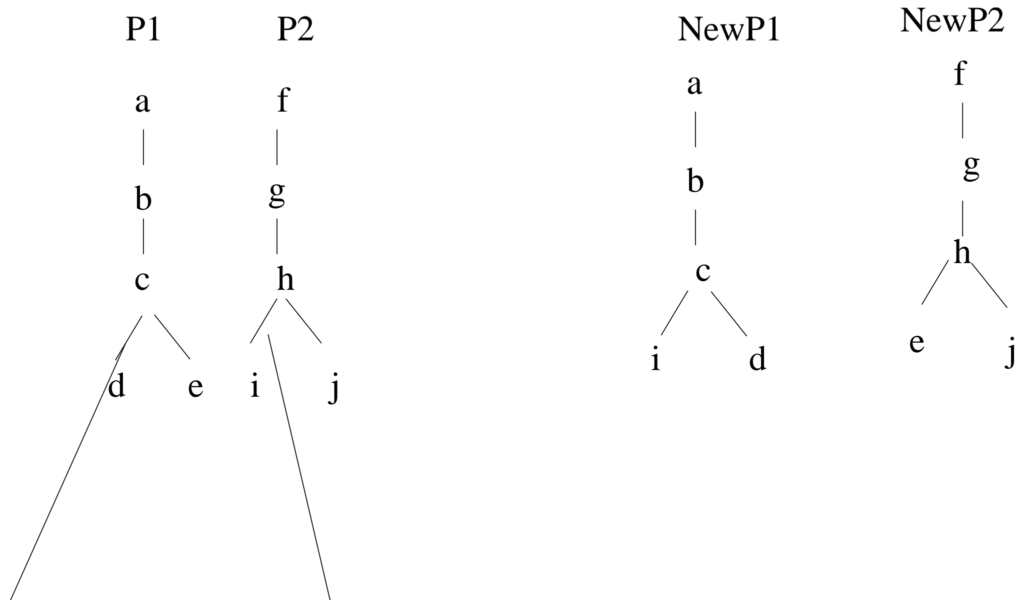

Figure 5.8: Results of a crossover when both candidates already contain a branch

- Reproduction simply involves the chosen pattern being copied directly into the new population.

3. Depending on the technique chosen, one or two patterns are selected. A bias is allowed to sway the random selection towards good patterns but bad patterns can still be chosen. The original ranking of the patterns is based on their frequencies in the pattern discovery stage, but later rankings reflect their improvement in this section (i.e. how many other patterns they subsume).

4. When a new pattern is formed, it is scored and the score is compared against its parent's score. If the offspring scores best then it is 90% more likely to be chosen for a new population than its parent. If the parent scores best then it is 90% more likely to be chosen for the new population than its offspring.

5. When the current population is empty. the procedure is repeated using the new population. However, a copy of the initial set of discovered patterns is kept so that new candidates can be scored using this.

It would be possible to evaluate each new candidate against a test set of theorems instead of the discovered patterns. However, this would lead to mutations based on this step and not on tactics formed from the discovered patterns. Although this could be useful, this approach is not used to demonstrate that it is possible to form tactics from one training set which can be repeatedly used in the future.

In fact, genetic programming of this type could be used (with a random start or otherwise) as a stand-alone approach in order to find the best set of tactics to model the whole corpus. The main drawback to this approach would be the length of time that this would take. Even when modelling a (relatively) small set like the set of patterns with a seeded input, the GP technique can be very inefficient.

5.5.2 Performance

Genetic Programming was not expected to be particularly efficient in terms of time. It was hoped that this technique would provide good tactics which would compensate for this. GP does in fact produce some interesting tactics.

In this section, some of the tactics which have arisen from this stage, along with some evaluation of these are described. Until the application of these tactics are described and a more specific appraisal of their worth can be given, they must be evaluated using the test set obtained from the proof corpus.

The efficiency of this approach is also discussed, including an examination of the differences made by adjusting the time-out value and the size of the initial population.

5.5.2.1 Results

Genetic Programming has yielded a number of new tactics, a few of these may be completely inapplicable as they have progressed into the new population due to the 'random factor'. Of those kept, most are only partially applicable. The wide range of tactics from Genetic Programming encompass; some with only two steps and no operators - unchanged from the pattern discovery stage because all changes have resulted in a lower score than the initial score, some very long tactics which include large branches ($\vee$ or $\wedge$) which are inapplicable but which have been kept because the rest of the tactic scores well.

A fairly typical example is shown in figure 5.9. This tactic represents a sequence of $ex1E$ followed by *either all_dupE* followed by *ssubst or someI* followed by exE, then 1 or more repetitions of $allE$ followed by mp.

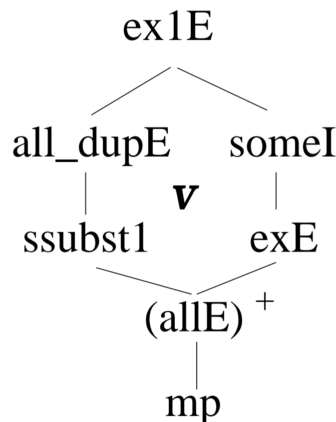

Figure 5.9: Example of a tactic found using Genetic Programming

Using one branch of the $\vee$ (left-hand), this tactic could be applied to part of the proof of:

$some1_equality = [|\exists!\, x.P\, x;\ P\, a|] \longrightarrow\ (SOME\ x.\ P\, x)\ =\ a$

However, there does not appear to be any occurrence which contains the right-hand side. This does not prove that the right-hand side can never be used, but it is not applicable within the set and would represent unnecessary search during application.

Full details of all the steps in this tactic and their application to this proof can be found in appendix B.

As an example, from one input sample of 100 patterns and 3,200,000 iterations, 53 tactics were kept. The tactics kept have a score of at least 1 from the scoring system detailed previously. This set of 53 patterns does not completely describe the initial population as some patterns have mutated until they are no longer useful. In this particular set, the highest score was only 7. This means that the best tactic only completely described 7 of the original patterns, from an initial population of 53, it would not be unreasonable to expect better. This slightly disappointing result provides the main motivation for the construction of the 'Pairwise Combination' approach, which is described in section 5.6.

A measure of the usefulness of these tactics can be gained by comparing them against the test set to see how often they would be applicable. This is not limited to once in a proof, so it is possible to measure how many proofs have tactics applicable to them and how many tactics are applicable. This measure is shown in figure 5.10.

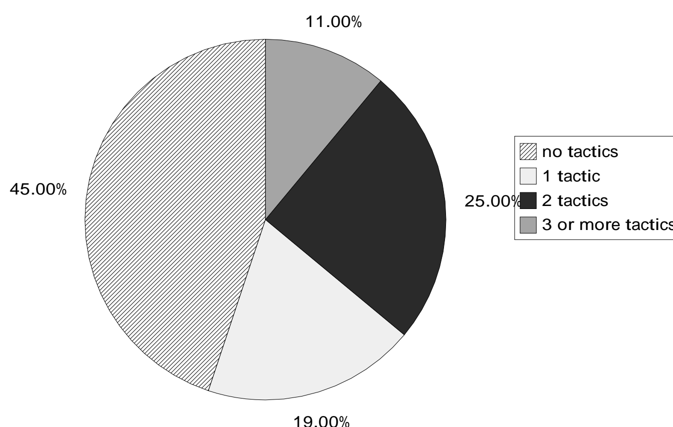

Figure 5.10: This shows the percentage of proofs which could have 0, 1, 2, 3 or more tactics applied to them.

5.5.2.2 Efficiency

As the initial population consists of patterns (containing no branches), a certain reliance must be placed on chance to provide the useful (or any) branches. This is one of the reasons that genetic programming starts off so slowly in the case of IsaNewT. By measuring the average score of the tactics against the number of iterations the effi-

ciency of the Genetic Programming technique can be examined, this is shown in figure 5.11.

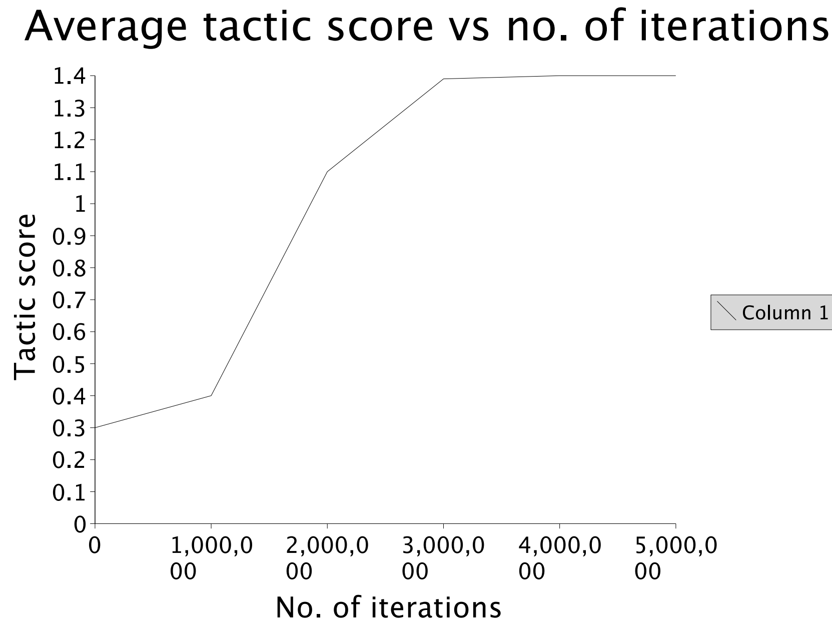

Figure 5.11: Measure of efficiency of Genetic Programming. The x axis shows the number of iterations and the y axis shows the decrease in population size.

As can be seen from figure 5.11, above 3,000,000 iterations, there is very little improvement. The initial population for this sample was 37. By comparing a number of these graphs the optimal number of iterations has been found to be roughly proportional to 100,000 for each member of the initial population. Of course, this is a rough guide, but it goes safely past the convergence point of the graph in all test cases.

5.6 Pairwise Combination

Although Genetic Programming apparently offers a good fit to the requirements of IsaNewT, it was felt that a more directed approach might be more effective. In particular, one of the great strengths of GP is the potential for mutation to allow a previously unexpected solution to arise. However, the intention with the IsaNewT method is to find tactics specifically based on commonly occurring patterns, so it is not necessary to find new possibilities in this way. For this reason, in addition to the implementation of the traditional Genetic Programming approach, there is also an implementation of a novel method.

The new approach is inspired by Koza's GP algorithm. This approach, which has been named 'Pairwise Combination', uses the discovered patterns as an initial population, but focuses solely on the crossover step of the GP algorithm.

5.6.1 Implementation

The method involves repeated iterations in which two members of the population are chosen and an attempt made to combine them. The number of iterations is decided by the user. The default is 1,000,000 iterations, which is not unreasonable even for a small set in genetic programming. The default is usually enough to generate a 'stable' set of tactics from a pattern base of 50. A 'stable' set is defined to be one which will not change significantly even when run for another 500,000 iterations. Each iteration consists of the following steps:

1. The patterns are each assigned a number at random (this assignment is done simply using the order that the patterns came out from the pattern discovery stage which is random).

2. The number of patterns are counted (N) and 2 (distinct) random numbers between 1 and N are chosen. The patterns assigned to these numbers are chosen to be compared, say (P1,P2).

3. Firstly a check is performed to see if P1 is a complete subset of P2 or vice versa. These complete subsets can appear provided the frequency they attained in the pattern discovery stage was different (i.e. the smaller pattern appeared in other places as well as being a subsection of the larger pattern). If so, the smaller pattern (say P1) is named as a *macro* and given the next *macro identifier* in succession (e.g. m3).

   The occurrence of this pattern in P2 is replaced by m3. A *macro* is not permitted to contain a step that is known to be a branching step (see step 6). If such a *macro* were permitted and put in place, the method would have to look inside the *macro* to check for branching points or risk not joining known branches together. Looking inside the branches would greatly increase the time taken to perform each iteration and not discovering the branches would be a significant drawback. Losing out on some potential *macros* was seen to be the least important loss. An example of *macro* introduction is shown by figure 5.12.

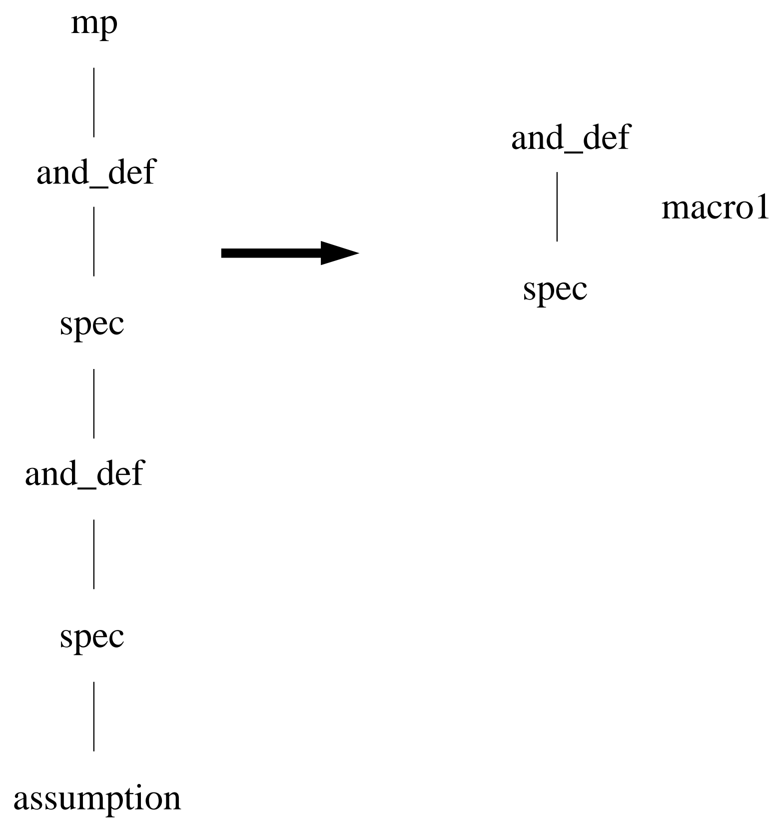

Figure 5.12: Introduction of a macro identifier.

4. If the *macro* step fails, then the potential for a *plus* operator is checked. P1 and P2 are compared. If they only differ by a repeated step (i.e. $[a,b,c,c,d]$ and $[a,b,c,c,c,d]$) then a *plus* is introduced ($[a,b,+(c),d]$). This is better shown by figure 5.13. There are instances where adding a *plus* may be an over-generalisation, but it is common that two sequences which only differ by the repetition of a step are representative of other sequences which have a different number of repetitions of this step.

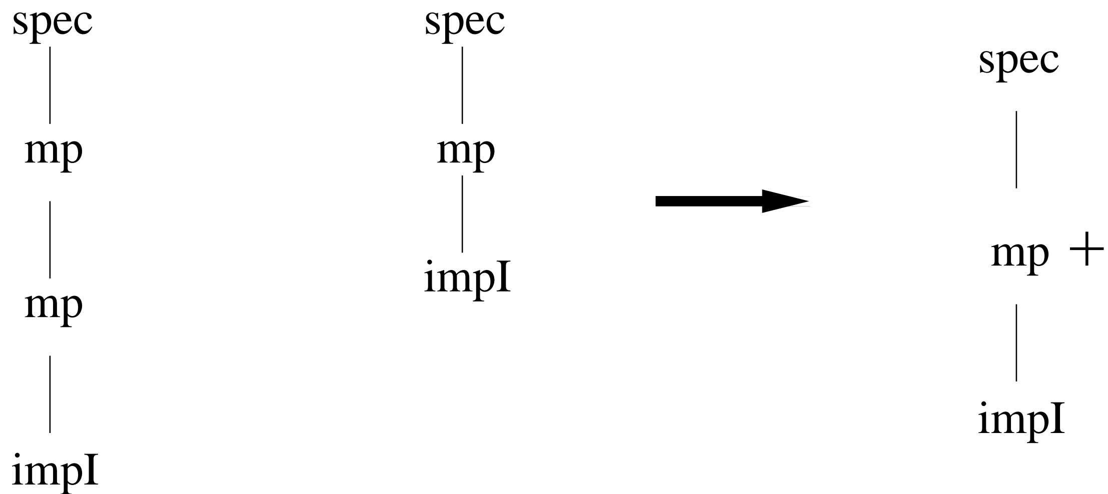

Figure 5.13: Introduction of the plus operator.

5. If P1 and P2 are dissimilar so far then a search is performed for a potential $\vee$ introduction. Both patterns must begin and end the same. If this is true then a filtering process is undertaken until the shortest difference is covered by the $\vee$. For example:

   P1= $[a,b,c,d,e,f]$ P2= $[a,b,g,h,i,e,f]$

   becomes:

   $[a,\vee([b,c,d,e],[b,g,h,i,e]),f]$

   then:

   $[a,b,\vee([c,d,e],[g,h,i,e]),f]$

   and finally:

   $[a,b,\vee([c,d],[g,h,i]),e,f]$

   in which the $\vee$ covers the shortest difference. An example of this is shown in figure 5.14

In order to avoid long sequences with very little in common being joined by a large $\vee$ there is a simple scoring system. A sequence scores 1 point for every original pattern it subsumes and loses 1 for every member of (the longest branch of) an $\vee$. The score must be non-negative for the new sequence to be accepted. This also prevents the disjunction over every possible sequence being returned as a tactic. This would otherwise score well, but would not be very useful!

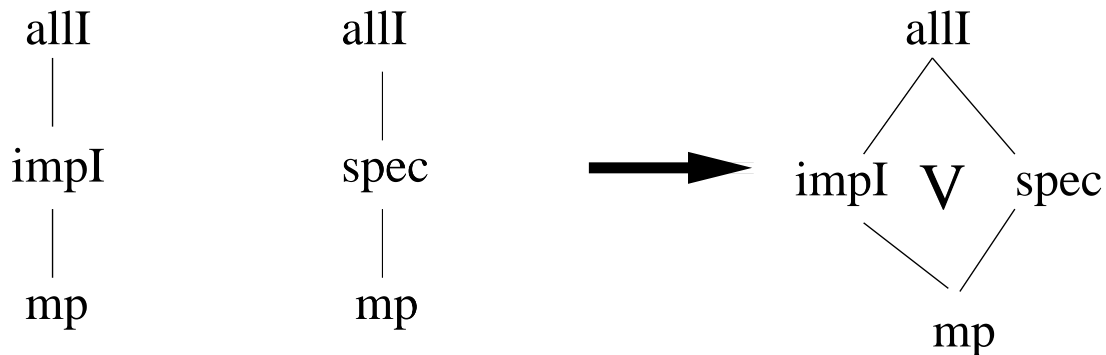

Figure 5.14: Introduction of the $\vee$ operator.

6. If $P1$ and $P2$ are still unchanged a branching point is looked for. All steps which result in branching are known (they are gathered in the weighting stage of the pattern discovery). All steps in $P1$ and $P2$ until a branching step is reached must be matched. If this is possible then the tactic with branching is passed back into the population. This is shown in figure 5.15

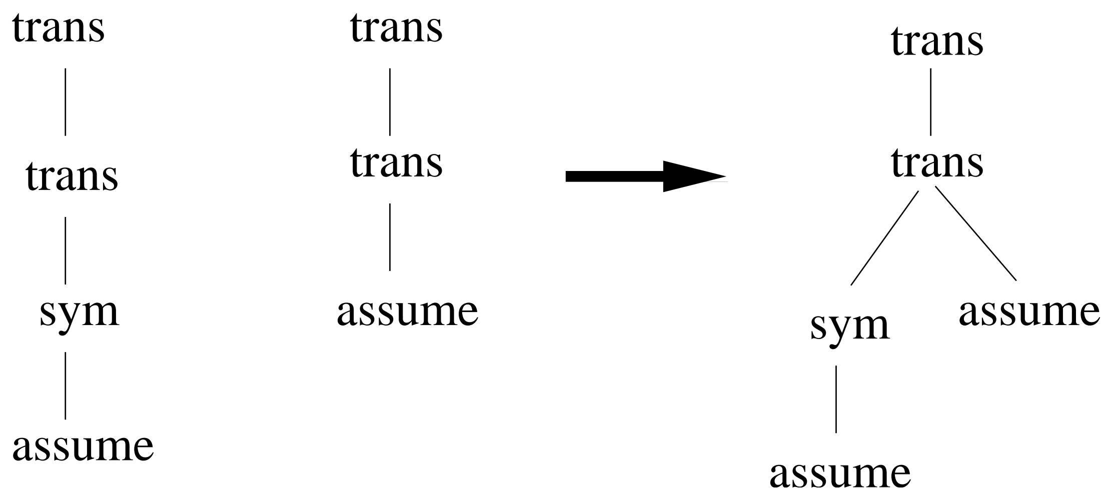

Figure 5.15: Introduction of the $\wedge$ operator.

7. If none of these steps are successful then $P1$ and $P2$ are passed back into the population unchanged.

8. If necessary (i.e. if $P1$ and $P2$ are merged) then the identifiers assigned to the new patterns are updated. Only the new tactic is kept, $P1$ and $P2$ are discarded as they are subsumed by the new tactic. This causes the population to shrink.

The entire population is kept as the current tactic set. Unlike the more traditional GP, there is no 'old' and 'new' population, the population incrementally changes continuously. If a good initial selection of patterns is found and the time limit is sufficient then the patterns set will be significantly reduced.

By choosing the two compared patterns randomly, combinations of different steps are allowed to arise. For example, a *plus* can be introduced over a macro allowing a repetition of more than 1 step. This is shown in figure 5.16.

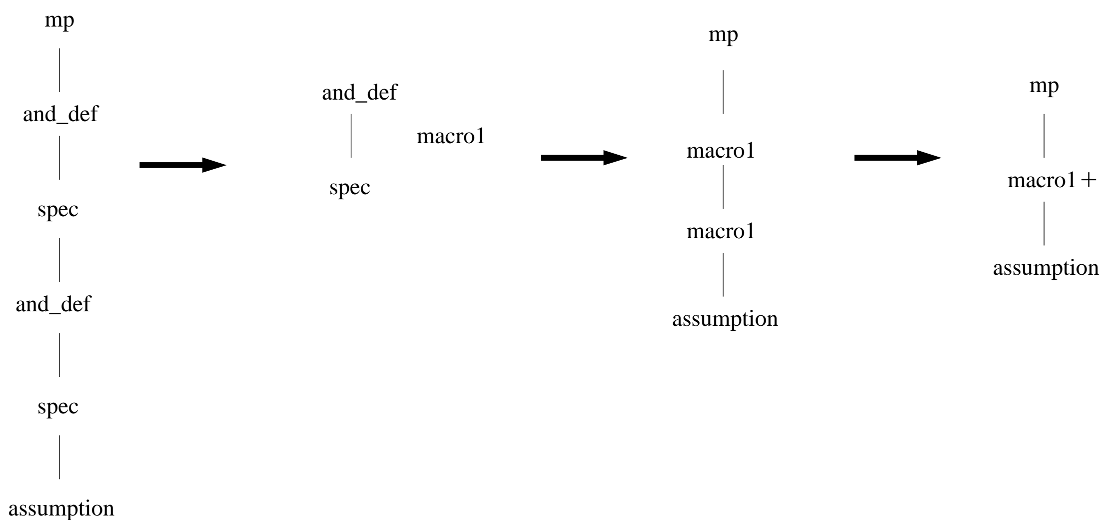

Figure 5.16: How a repetition over more than one step can be found using the macro identifier

As a copy of the initial population is kept when using this technique unchanged tactics can be re-examined. By inspecting both the original score from the pattern discovery process and the final score from this tactic formation step, it can be decided whether they should be eliminated before the application procedure. A tactic which came from a pattern with a low frequency (very close to the threshold) and which also subsumes few, if any, other patterns at this stage (and so attains a score of close to 1), is likely to be a less useful tactic and so can be discarded or reduced in importance when it comes to the application stage.

For the purposes of application, all the tactics have been kept but the tactic set has been arranged so that these ‘weak’ tactics will always be attempted last.

5.6.2 Performance

As with the traditional GP approach, the efficiency of this technique must be measured. The size of the initial population and the time-out value are varied to ensure these are not factors.

The tactics gained from this approach are discussed and a comparison drawn between them with those obtained from traditional GP. As with the pattern discovery stage, only preliminary evaluation is carried out here, with a complete evaluation being carried out in conjunction with the applications.

At this point, of most interest is the comparison between the two techniques.

5.6.2.1 Results

A number of tactics have been evolved using the Pairwise Combination technique. They range from only having 2 steps and no operators to having 16 steps and 7 operators. A fairly typical example with 5 steps and 3 operators is shown in figure 5.17. Also shown are the steps that the macro *macro1* is short for. This tactic represents a sequence of *and_def* followed by *spec* followed by *mp* and then an $\wedge$ split. The *assumption* step should be applied to one subgoal and one or more repetitions of *impI* then *assumption* applied to the other.

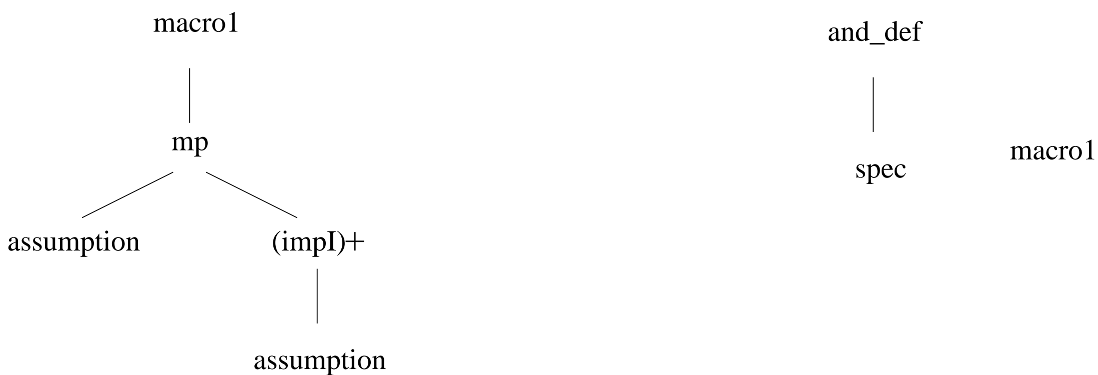

Figure 5.17: Example of a tactic found using Pairwise Combination

This tactic could be applied within the proof of $conjunct1$:

$conjunct1$: “$[| P \wedge Q |] \implies P$”

$apply\ (unfold\ and_def)$

to get:

$\forall R. (P \longrightarrow Q \longrightarrow R) \Longrightarrow P$

then:

$apply\ (drule\ spec)$

to get:

$(P \longrightarrow Q \longrightarrow R) \longrightarrow R \Longrightarrow P$

then:

$apply\ (drule\ mp)$

to get 2 subgoals:

$R \Longrightarrow P$

and: $P \longrightarrow Q \longrightarrow R$

The first of these is solved with *assumption* concluding the first leg of the proof. To the second:

$apply\ (rule\ impI)$

is applied to get:

$P \Longrightarrow Q \longrightarrow P$

The next step, *assumption*, cannot be used at this point so the *plus* operator is utilised:

$apply\ (rule\ impI)$

to get:

$[| P; Q|] \Longrightarrow P$

Which is solved by *assumption* thus completing the proof.

From an input sample of 40 patterns and 2,000,000 iterations, 14 tactics remained at the end of the entire IsaNewT process. Unlike with traditional Genetic Programming, the final population completely describes the initial population. While it would be possible to also measure the increase in score against time for this approach, this cannot be compared directly against the GP approach as the GP approach contains tactics that will be discarded.

As with GP, a measure of the usefulness of these tactics can be gained by comparing them against the test set to see how often they would be applicable. This measure is shown most clearly in figure 5.18.

5.6.2.2 Efficiency

As the Pairwise Combination method is much more directed than traditional GP, it is a much more efficient technique. By measuring the size of population against the

number of iterations, a measure of how many patterns have been combined can be gained. Therefore a measure of the efficiency of the Pairwise Combination technique for IsaNewT's purposes can be gleaned. This is shown in figure 5.19. A full week-long test was run but there were no further changes to the population.

As can be seen from figure 5.19, above 1,000,000 iterations, there is very little (often no) improvement. The initial population for this sample was 48. As with the GP approach, a number of these graphs were compared to obtain an optimal number of iterations. In this case it is roughly proportional to 25,000 for each member of the initial population. This rough guide goes safely past the levelling out point of the that graph is used as a termination criterion. Using the Pairwise Combination technique, increasing the initial population does not result in a direct increase in the number of iterations required as more potential matches exist to be found. However, it does not cost as much to continue with the iterations as time goes on as a falling population means that each iteration is faster.

5.7 Summary

Two methods based on Evolutionary Programming techniques have been described that are used to evolve the discovered patterns into tactics. Some efficiency results

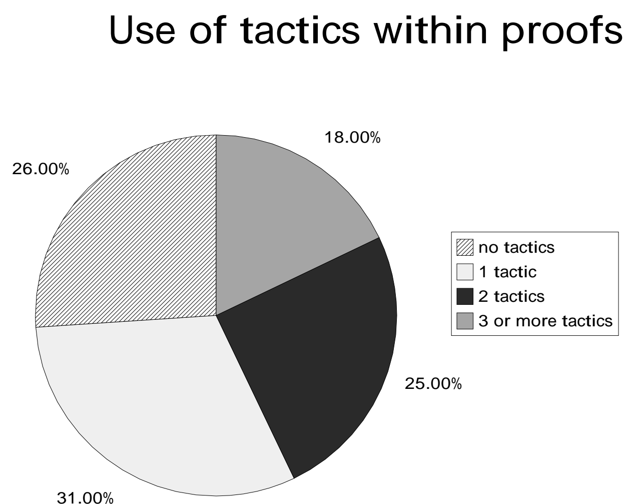

Figure 5.18: No. of tactics applicable within proofs. This shows the percentages of proofs which have: 0, 1, 2 or 3 or more tactics applicable to them.

have been presented on the two techniques and some typical examples described.

Unsurprisingly, a comparison of the results obtained from Genetic Programming and from Pairwise Combination show that PC provides significantly better results. Genetic Programming relies on many random decisions which can be very useful when dealing with a problem where the specification is poor. However, IsaNewT's problem - that of combining patterns to form tactics - has well-defined goals.

Using random choices *can* be very useful; it has been used to choose the candidates for crossover as initially there is no way to tell which selections would make good candidates. In spite of the ranking system implemented, there is no way to know if two (or more) bad patterns might combine to make an excellent tactic. By exploiting this within a directed system, as done in Pairwise Combination, an efficient evolutionary procedure inspired by Koza's GP algorithm has been created.

With a comparison of the worst case estimates of how many iterations a run might need, it is already clear that PC beats GP by a factor of 4. In real time, PC performs even better due to fewer operations in later iterations (this is because of a smaller population size).

In terms of the tactics obtained, both GP and PC provide viable options. Genetic Programming will produce some tactics which could not have been found simply us-

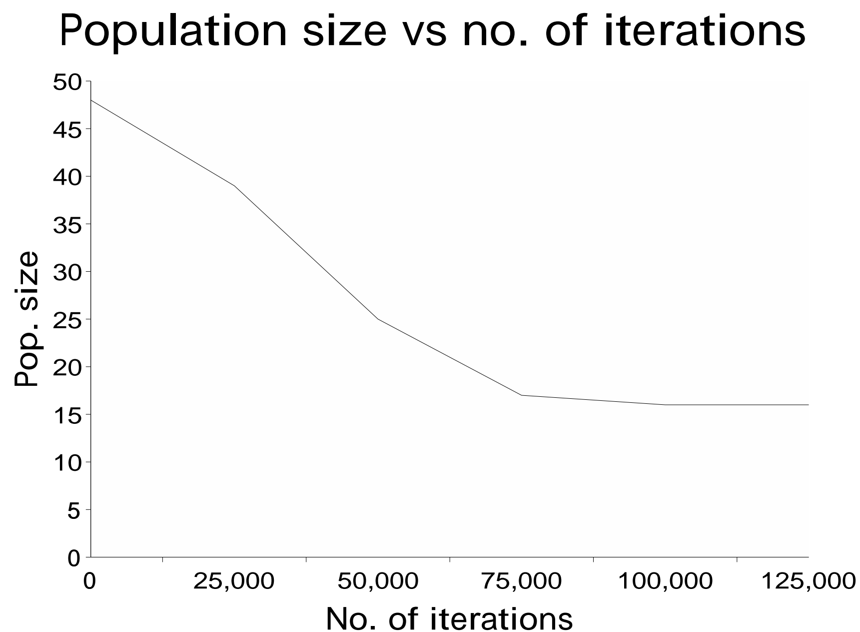

Figure 5.19: Measure of efficiency of Pairwise Combination.

ing the initial population. This means that GP could conceivably produce interesting tactics that could not be discovered using the pattern discovery technique (although this was not observed during any of the tests). As IsaNewT is designed to expressly exploit such patterns to form new tactics, this feature of GP is not necessarily useful (even on the rare occasions it could produce a fantastic tactic by fluke) but may be worth exploring in another context. Most of the tactics produced by GP have at least some part of them which is either redundant or inapplicable. In some cases, GP will produce tactics and parts of tactics which could never be applied, and while most of these will be weeded out by the scoring process it is inevitable that some will linger.

Pairwise Combination produces only tactics which arise from the initial population and because of this it can be guaranteed that all final tactics are applicable in some instances. Indeed, studying the number of proofs which can be partially described by at least 1 tactic, see figure 5.18 an improvement over similar statistics from GP can be seen.

Pairwise Combination shows an improvement both in tactic applicability and in efficiency when compared again to traditional Genetic Programming for IsaNewT's purposes.

Chapter 6

Application

In order to test the new tactics which have been discovered by IsaNewT and to demonstrate whether they are in fact useful tactics, a simple fully-automated theorem prover within the Isabelle interactive prover has been developed (IsaAuto). In this chapter is a description of this prover.

The implementation of IsaAuto is described along with some details about the type of input which will be given for comparison. One of the more complicated steps in implementing the prover was the adaptation for each abstraction in order to ensure that a fair comparison was always maintained.

This chapter is concluded with some results on the performance of IsaAuto. These results do not provide an evaluation of the tactics but rather provide the baseline so that a good selection of theorems can be chosen for testing purposes in the evaluation.

6.1 Designing an Automatic Isabelle Prover

The Isabelle prover described here was designed with the express purpose of evaluating the usefulness of the tactics. To that end, it was decided that the prover would perform a naive search through all the possible rules and tactics in order to find a proof. Evaluation of the discovered tactics would occur by comparing the search described against a search for a proof with these tactics added as heuristics. We argue that an improvement in time and in the number of theorems proved demonstrates that the discovered tactics are indeed useful.

By using Isabelle as a basis for the prover, IsaAuto inherits all the soundness of Isabelle. In addition, IsaAuto has no way to check whether a contradiction has been found (i.e. a theorem is false) and will just report a failure to prove the theorem.

6.1.1 Isabelle Tools

Isabelle has a number of inbuilt tools designed to aid the user in developing an automated prover within Isabelle. These have been utilised in the development of IsaAuto.

6.1.1.1 Isabelle's Inbuilt Tactics

Isabelle uses a different notion of tactics than the one used until now. In order to avoid confusion, the Isabelle tactics will from now on be referred to with the abbreviation *tacs*. Isabelle has a number of *tacs* which are defined as "an abbreviation for functions from theorems to theorem sequences" in the Isabelle reference manual [Paulson (1986)]. In comparison to IsaNewT's tactics, Isabelle's *tacs* contain possible rules to apply, along with the information on how these rules are to be applied. These *tacs* can in fact be thought of as mini-provers.

Some of the more common or relevant *tacs* available within Isabelle are given below, (commands in brackets show abbreviation available to the most common).

resolve_tac *thms i* (rtac) refines the proof state using the rules contained in the list *thms* (normally introduction rules). It resolves a rule's conclusion with subgoal *i* of the proof state

eresolve_tac *thms i* (etac *thm i*) performs elim-resolution with the rules (normally elimination rules). It resolves with a rule, proves its first premise by assumption and *deletes* that assumption from any remaining subgoals

dresolve_tac *thms i* (dtac *thm i*) performs destruct-resolution with the rules (normally destruction rules). This replaces an assumption by the result of applying one of the rules

fresolve_tac *thms i* (ftac *thm i*) like dresolve_tac except the selected assumption is not deleted

assume_tac *i* (atac *i*) attempts to solve subgoal *i* by instantiation

(**eatac** *thm j i*) performs **etac** *thm* then *j* times **atac** on subgoal *i*

(**datac** *thm j i*) performs **etac** *thm* then *j* times **atac** on subgoal *i*

(**fatac** *thm j i*) performs **ftac** *thm* then *j* time **atac** on subgoal *i*

ares_tac *thms i* tries proof by assumption and resolution; it abbreviates **assume_tac** *i* **ORELSE resolve_tac** *thms i*

rewrite_goals_tac *def* (rewtac *def*) unfolds the definitions *defs* throughout the subgoals of the proof state, leaving the main goal unchanged.

rotate_tac *n i* rotates the assumptions of subgoal *i* by *n* positions from right to left (left to right if *n* is negative)

6.1.1.2 Isabelle's Tacticals

Isabelle has an inbuilt notion of tactical which are operations on *tacs*. They can be thought of as high-level control structures.

Some of the more relevant ones are:

tac_1 **THEN** tac_2 is the sequential composition of the two *tacs*. Applied to a proof state, it returns all states reachable in two steps by applying tac_1 followed by tac_2. First, it applies tac_1 to the proof state, getting a sequence of next states; then it applies tac_2 to each of these and concatenates the results.

tac_1 **ORELSE** tac_2 makes a choice between the two tactics. Applied to a state, it tries tac_1 and returns the result if successful; it tac_1 fails then it uses tac_2. This is a deterministic choice; if tac_1 succeeds then tac_2 is excluded.

EVERY $[tac_1, ...tac_n]$ abbreviates tac_1 **THEN** .. **THEN** tac_n. It is useful for writing a series of tactics to be executed in sequence.

FIRST $[tac_1, ..., tac_n]$ abbreviates tac_1 **ORELSE** .. **ORSELSE** tac_n. It is useful for writing a series of tactics to be attempted one after another.

TRY *tac* applies *tac* to the proof state and returns the resulting sequence, if non-empty; otherwise it returns the original state

REPEAT *tac* applies *tac* as many times as possible (including zero), and allows backtracking over each invocation of *tac*.

REPEAT1 *tac* like **REPEAT** *tac* but always applies *tac* at least once.

Isabelle also has a number of search and control tacticals which are adapted in order to provide the automated prover.

DEPTH_FIRST *satp tac* returns the proof state if *satp* returns true. Otherwise it applies *tac*, then recursively searches from each element of the resulting sequence. In effect it applies *tac* **THEN DEPTH_FIRST** *satp tac*.

BEST_FIRST (*satp*,*distf*) *tac* does a heuristic search, using *distf* to estimate the distance from a satisfactory state. It maintains a list of states ordered by distance. It applies *tac* to the head of this list; if the result contains any satisfactory states, then it returns them. Otherwise **BEST_FIRST** adds the new states to the list, and continues. It will find a solution, if one exists

SOLVE *tac* applies *tac* to the proof state and then fails if and only if there are subgoals left

6.1.2 Implementation

A prover of the type required can be implemented in Isabelle with a minimum of effort using the inbuilt Isabelle *tacs* and tacticals. A straightforward implementation of a prover can be achieved using **rtac**, **dtac**, **etac** and **atac** with the argument *thms* containing the names of all the rules used. As described earlier, the entire Isabelle system is based on a few basic axioms, and any rule can be deconstructed into the sequence of rules which made up the proof of its own theorem. This has allowed the restrict of the Isabelle database of rules (which is potentially infinite as more and more lemmas are added with each new theory file) to around a hundred.

However, the implementation described above has a number of problems. Most importantly, allowing rewrite rules to be applied in any direction will leave the problem of looping. Also needed is a method of unfolding definitions. This contains the names of any definitions available. Although this can also be a potentially lengthy list, none will be applicable unless the object appears in the subgoal. If **rtac** is removed then there is an insistence that some part of the rule matches exactly with a part of the subgoal (one of the assumptions for **etac** and the conclusion for **dtac**). Now a new *tac* **PER_SUBGOAL** can be defined:

PER_SUBGOAL *rules* = etac *rules* **ORELSE** dtac *rules* **ORELSE** rewtac *defs*

In Isabelle, subgoals are numbered sequentially. When one subgoal is proved, the rest are moved up (so when subgoal 1 is solved, 2 becomes 1, 3 becomes 2 etc.). By default, a *tac* is applied to the first subgoal unless otherwise stated. Therefore, in order

to search for a proof **per_subgoal** would be applied repeatedly to the first subgoal until a solution is found. Effectively:

IsaAuto *rules* = **SOLVE** (**REPEAT1 per_subgoal** *rules*)

is being performed.

Backtracking occurs when an application of **REPEAT1** fails (i.e. no applicable rule can be found), in such a situation, the last rule application is undone and search continues as if this rule was inapplicable.

Now the discovered tactics are processed into Isabelle-style *tacs* so that they can be used in IsaAuto. Otherwise only the Isabelle *tacs* traditionally used in writing proofs by hand are used as the mechanism for IsaAuto.

erule_tac *thm* works in the same way as **etac** *thms* but only tries to apply one particular rule instead of one from a selection of rules.

drule_tac *thm* works in the same way as **dtac** *thms* but only tries to apply one particular rule instead of one from a selection of rules.

Hence each theorem name can be adapted to a tactics using **erule_tac** , **drule_tac**, **ORELSE**, **REPEAT** and **THEN**. Thus a tactics such as that represented in 6.1 would be represented by:

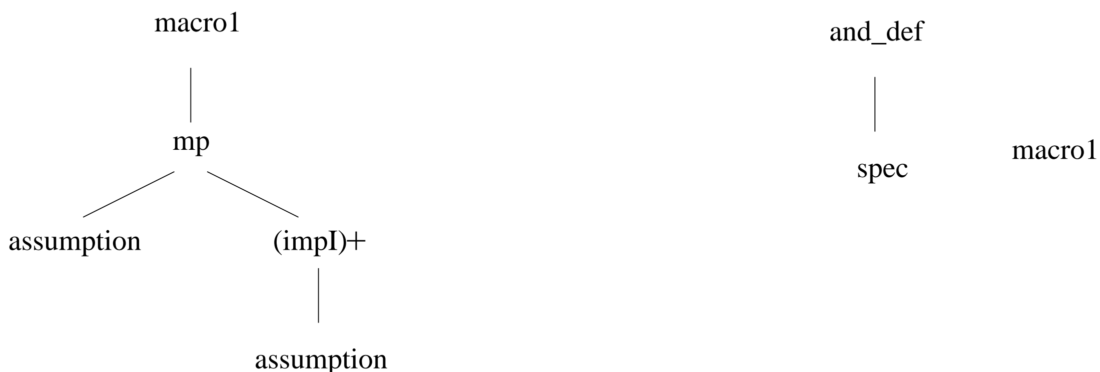

Figure 6.1: Example of a tactic.

branch_tac = ((**REPEAT1 erule_tac** *impI*) **ORELSE** (**REPEAT drule_tac** *impI*)) **THEN** *assumption*

tactic = (**rewtac** *and_def*) **THEN** ((**erule_tac** *spec*) **ORELSE** (**drule_tac** *spec*)) **THEN**
((**erule_tac** *mp*) **ORELSE** (**drule_tac** *mp*))
THEN ((*atac* **THEN branch_tac**) **ORELSE** (**branch_tac THEN** *atac*))

Although this is not a pretty format, it can be generated with a simple parser. This allows an encoding to be generated for each of the tactics as **tactic_n** so that the prover including the tactics would become:

prover_with_tactics =**SOLVE** (**REPEAT1 FIRST** [**tactic_1**,... **tactic_n**,**per_subgoal** *rules*])

After a timeout, search is cancelled and a result of 'no proof found' is returned. This is a very naive prover which will not perform well as a prover in its own right. However, it is suitable for the purposes of being a testing engine for the discovered tactics. Equal inefficiencies occur in both conditions, so the results gained from the tactics are fair.

6.2 Adapting the Prover for Different Abstractions

The description presented of IsaAuto and its implementation deals with the abstraction consisting solely of the rule names at each stage. The alterations needed to adapt IsaAuto for different abstractions must be discussed. As the abstraction chosen represents the level of information from the corpus, it directly affects how much work the prover will still have to do.

1. Rule name with direction

   With tactics: This is almost trivial to adapt. Simply restrict those steps marked forward to the **erule_tac** application, and those marked backwards to **drule_tac**

   Without tactics: In order to keep search as even as possible elimination and destruction rules can be restricted to **etac** and **dtac** respectively

2. Class of rule only

   With tactics: Replace **erule_tac** with **etac** and replace the name of the rule as described above with the list of rules associated with the class. Each of these possibilities must be searched through. Similarly for **drule_tac**

Without tactics: The test without tactics would remain the same.

3. Class of rule with direction

With tactics: This would be a direct combination of the two previous examples.

Without tactics: The test without tactics would remain the same.

4. Main proof operator:

With tactics: This information would be of little help at the application stage. If a rule is applicable then the subgoal *must* contain the significant operator. In fact, no extra theorems were proved and the average time taken to prove the theorems was worse with the tactics in place. This is unsurprising as at every step of the tactic every rule with a matching main operator would have to be searched. Many of these will rarely be used and would not normally be considered in a search until all other possibilities had been discounted.

Without tactics: The test without tactics would remain the same.

5. Rule name with position in proof

With tactics: Restrict any steps marked 'beginning' to the first pass of the loop of IsaAuto and restrict any marked 'end' to only be applied when they solve the subgoal. However, it will not usually be known which step will solve a subgoal until it has been applied. Therefore all steps marked 'end' would have to be attempted at every stage and the result discarded if it did not solve the subgoal (even if it was applicable).

Without tactics: The test without tactics would remain the same.

The only change suggested to the prover without the tactics is the refining of the theorem lists into elimination and destruction rules. As will be shown in the evaluation, both examples of this have been tested. Although this refinement shows an improvement in speed performance, occasionally it results in a theorem not being proved that would have been otherwise. This is due to the rare occasion when a traditional elimination rule (or a traditionally destruction rule) is used in an unusual fashion.

6.2.1 Performance

IsaAuto is not designed to be comparable with other fully automated theorem provers. It does not contain any clever heuristics or techniques to improve performance and search time. It was designed solely to form a basis for comparison to test the discovered tactics.

In this section some results gained from testing IsaAuto without the tactics are provided in order to gain a baseline for performance. This will allow a determination of the complexity of theorems that IsaAuto can reasonably be expected to deal with. This evaluation of performance also allows the time-out level to be set to return a ‘no proof found’ result.

The type of theorems IsaAuto can deal with are examined first. A selection of theorems for testing purposes should be chosen which will contain both theorems that IsaAuto can deal with along with theorems that it cannot.

This section continues by looking at a selection of theorems that are successfully proven and considering the time it takes for these to be proven. By varying the allowed depth of search a level that will provide the best compromise between reasonable search times and number of theorems proved can be found.

6.2.1.1 Types of theorems

This section begins by rating some theorems based on their existing Isabelle proofs. The existing, hand-produced proofs are used as these should represent a good (not wasteful or circuitous) proof. This would not be a reasonable assumption for many hand-produced proofs, especially those written by novices as a certain amount of search will have been done during the proving process and may well still exist in the proof. Nevertheless, the proofs which are held in the Isabelle libraries have been updated many times over the years to keep compatibility to newer versions of Isabelle.

The theorems are rated from a score of 1 (easy to prove) to 10 (very complex to prove), the ratings are calculated on the following criteria:

1. Number of steps in the proof.
    - This reflects the amount of search that will be required to find a solution, as any proof found by the prover is likely to be at least as long. Therefore, this measure provides a measure of complexity.
2. Number of ‘special’ techniques used.

- The term 'special' techniques denotes instances such as situations where the instantiation is specified, or a lemma is inserted with the *cut* technique. These weight heavily towards complexity as the prover with or without the discovered tactics will be unlikely to find a proof for these theorems.
- Use of automatic techniques such as **auto**. These techniques can represent a large area of search space so they have a weight higher than the number of steps. However, any of these tools can automatically be found by Isabelle so they have a complexity weight less than the previous 'special' techniques.

These criteria have allowed us to rank the theorems from simple, such as:

```
theorem:
Q ⟶ P ∨ Q
apply (rule impI)
apply (rule disjI2)
apply assumption
done
```

to complex such as:

```
lemma atleast_free_SucD_lemma:
 !m a. m a = None ⟶ (!c. atleast_free (m(a|→c)) n) ⟶
  (!b d. a $neq$ b ⟶ atleast_free (m(b|→d)) n)

apply (induct_tac "n")
apply  auto
apply (rule_tac x = "a" in exI)
apply  (rule conjI)
apply  (force simp add: fun_upd_apply)
apply (erule_tac V = "m a = None" in thin_rl)
apply clarify
apply (subst fun_upd_twist)
apply  (erule not_sym)
apply (rename_tac "ba")
apply (drule_tac x = "ba" in spec)
apply clarify
apply (erule notE impE)
apply (case_tac "aa = b")
apply fast+
done
```

Figure 6.2 shows the successfulness of IsaAuto without the new tactics in solving these different ranks of theorems.

As can be seen, IsaAuto understandably can only handle the more simple forms of theorem, this is not a reflection on the quality of the tactics but is the result of such a naive prover. Even in these levels, no proofs are found for many of these theorems.

Only one theorem was proved in any section from rank 5 and above, this is probably due to a mis-assignment. In fact examining this proof shows that each step is in fact simple, and that it was ranked due to the high number of applications of **simp** in the proof script.

From this analysis it can be deduced that it would be most useful to use a selection of theorems ranked from 1 to 4 with a very small number ranked 5 in order to create a realistic test set.

6.3 Summary

The inbuilt Isabelle tools (*tacs* and *tacticals*) have proved invaluable in implementing IsaAuto. No use has been made of many of the more complicated tools available such as those allowing more intelligent search techniques and heuristics.

Nevertheless, a simple Isabelle prover which can find proofs for fairly simple theorems has been developed. Although performance is slow and many more complex theorems will not be proven by IsaAuto there now is a good platform for testing the usefulness of the discovered tactics.

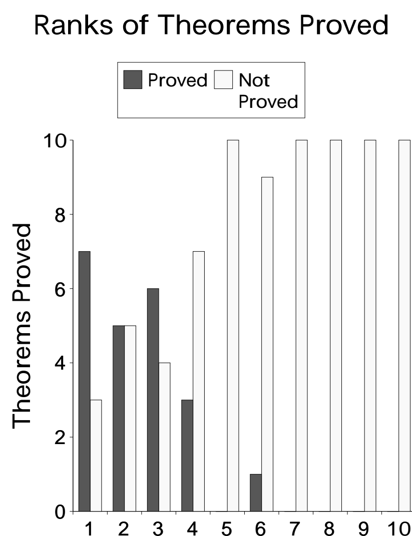

Figure 6.2: Number of theorems proved with increasing complexity. X-axis shows an increasing complexity rank. Y-axis show the number of theorems proved or not proved, respectively.

Chapter 7

Evaluation

This chapter presents an evaluation of each stage of the development of IsaNewT along with an overall evaluation and analysis of the results.

Generally, evaluation of tactics can only truly be carried out through application. It is to this end that the IsaAuto prover has been developed. However, it is possible to carry out some evaluation of the tactics through manual inspection. This is particularly useful when used to compare the evolved tactics to the more basic patterns discovered by the early stages of IsaNewT. The merit of a tactics can be judged by:

1. Its applicability - how broadly can it be used?
2. Its effectiveness - does it advance the state of a proof?
3. Its mathematical interest - how significant a concept does the pattern describe?

The third point listed is best evaluated by means of manual inspection.

A re-examination of some of the patterns discovered earlier is performed. Various measures and examination techniques can be used to indicate how useful later tactics formed from these patterns will be.

Next, results from the tactic formation stage are presented. Some manual evaluation techniques along with some analysis of the measures associated with this stage in the same way as with the patterns is performed. This process particularly rewards complex tactics.

Most importantly the integral part of the evaluation, the results of application are presented. A variety of results from different test sets against equivalent results from IsaAuto without the new tactics are shown. These test sets are completely separate from the training sets used to generate the tactics.

The chapter continues with a discussion of the information gained from different forms of abstraction. In particular, there is a focus on the information learned which could not be exploited at the application stage.

This chapter concludes with a summary of the results.

7.1 Patterns

Already presented in previous chapters were some results on manual evaluation of the patterns within the chapter on pattern discovery. That is expanded on here with some discussion about how manual evaluation can provide both some interesting features and how the measures used at this stage can be adapted to measure the tactics at a later stage.

7.1.1 Manual Evaluation

An important stage in the development of IsaNewT was the rediscovery of an existing theorem. As has been mentioned several times before, some of the more advanced theorems have been deconstructed into their proofs. For this reason, it was postulated that finding a pattern or tactic which described the proof of a known theorem would provide validation that the technique was sensible.

Imagine a theorem *foo* (say) that has been deconstructed to its proof, it can be seen that every time this theorem is used as a rule within any other proof the sequence of steps which make up this theorem would appear. In this way, any theorem that is used as a rule a reasonable number of times should appear as a pattern if the technique is working as intended.

Indeed, an early example of this was discovered in the pattern discovery stage. As the proof trees have been linearised down the branches, it is only to be expected that only one branch of a proof would be found except in the case that a proof has no branches. The perfect example of the latter case is demonstrated by the simple propositional rule:

```
lemma (P ⟶ Q) ∧ (Q ⟶ R)
⟶ (P ⟶ R)
apply (rule impI)
apply (rule impI)
apply (drule conjE)
apply(drule impE)
apply assumption
apply assumption
apply (drule impE)
apply assumption+
done
```

After applying IsaNewT on the theorem set covering basic Higher Order Logic and propositional theorems (in which this rule is used often) this pattern (given in figure 7.1) can be found. As can be seen, this describes the first branch of the proof given above.

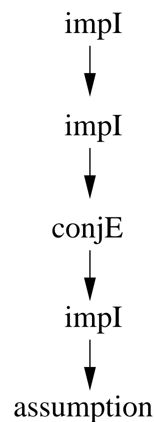

Figure 7.1: Discovered pattern representing one full branch of a proof.

The probabilities assigned to each pattern can be used as a guide to the order of preference the tactics should be given at the application stage. To any tactic is attached the highest probability associated with the patterns that formed it.

That is, if $patterns_1, ..., patterns_n$ are the patterns that were combined to make the tactic tac then the $pattern_j$ such that this pattern had the highest frequency score in the pattern formation stage is located. This frequency now becomes a weight associated with tac, $weight(tac)$. This measure is used to rank the discovered tactics so that ones that would be expected be more applicable from the pattern discovery stage will be attempted first during application.

7.2 Tactics

7.2.1 Manual Evaluation

Many of the tactics discovered represent expected combinations, for example, the stripping away of quantifiers is one set of steps that would be expected to occur together. Others among the tactics discovered, while not expected, are easily understandable when the proof steps and the rules they describe are examined by hand. This is also an invaluable tool for examining the original occurrence of the tactics discovered which turn out to score badly. By looking for the instances within the training set that these patterns were discovered from, it often becomes clear why an unusual combination occurs. This also allows identification of particular combinations which might only be used in a small number of proofs - if this combination always occurs in sequence then a small number may well be enough to identify this set as a pattern.

7.2.2 Usefulness

To evaluate against a test set, some metrics to measure individual tactics have been devised. By measuring the percentage of proofs in a test set that the tactic can be applied to (only one count per proof, even if the tactic could be applied multiple times), an estimate of how useful the tactic will be can be discovered. By using a different set of proofs for the test set than those used for the training set, it is possible to ensure that features peculiar to the training set are not rewarded. Ignoring multiple applications in a single theorem ensures that this usefulness quality reflects the percentage of proofs that a tactic can be applied to. For example, if a tactic could be applied 10 times within one proof but not to any other proofs at all, it would not be a very useful tactic in a general sense. Allowing multiple occurrences within one proof to be counted could give a false reading in the case of a very large but unusual proof. Of course in real applications, multiple applications within one proof are performed.

The usefulness score is given as a percentage.

7.2.3 Quality

In the same way as above, a measure of quality can be taken by weighting usefulness in favour of longer and more complex tactics. This prevents discrimination for two-step combinations which may appear (at least in part) due to chance. This technique prevents longer tactics from being penalised for being less common. It would be expected

that a widely applicable long or complex tactic would be more interesting than a short tactic of the same applicability.

This is measured with a score which describes a trade-off between the complexity and the applicability. This score is calculated with:

$$complexity\ ((no.\ of\ steps\ +\ no.\ of\ operators)/(100 - Usefulness)) * 100 \quad (7.1)$$

This technique does not guarantee a good measure of quality but seems to offer a reasonable solution to an extremely difficult question.

The quality score has a minimum of 2 (2 steps, no operators, and a usefulness score of 1). The maximum is theoretically unlimited as it depends on the number of steps and operators within a tactic. For practical purposes, a tactic can be imagined to have an upper bound complexity of 50 (although it is extremely rare to find any tactic with a complexity above 30). This would give an upper bound to the quality of 5000. This would require an extremely long and complex tactic which is applicable to every proof in the test set. It is not difficult to imagine how unlikely that would be!

Even choosing a score of around 50, it is unlikely that any tactic will come close to this as an upper bound, an ideal tactic would still fall short of this mark. An extremely complex tactic which was applicable to 10% of the proof corpus would only score around 30.

7.3 Some Examples

A large number of discovered tactics are available but only a selected few which demonstrate a range of styles, expressivity and quality will be discussed. The examples given here were generated from an initial training set of 989 theorem which yielded 197 patterns. These patterns were generalised to form 122 tactics. Of the final 122 tactics, 36 were discarded as they had a borderline frequency and had not been improved by the Pairwise Combination stage of IsaNewT.

7.3.1 A Typical Tactic

A fairly typical tactic would have 3-4 steps (rule names) and one operator ($\wedge$, $\vee$, $+$, macros are not considered as operators because although they can make discovered tactics easier to understand and compare, they do not add to the complexity).

7.3.1.1 Scores Gained in Formation Step

From the training set the examples have been chosen from, the average scores would be: 0.063 in the pattern discovery stage, and 7 for the genetic programming stage (this is always Pairwise Combination). The pattern discovery stage has a percentage score ([threshold,100]). The tactic formation stage score reflects the number of other patterns subsumed (including itself). This score can be $[1,n]$ where n is the number of patterns found in the pattern discovery stage (usually 80-110).

7.3.1.2 Manual Evaluation

The average tactic could be fairly easily analysed manually as described with the previous examples. However, the number of patterns and tactics discovered makes this prohibitive, and it is mostly useful to manually examine the few at each end of the scale, along with a couple of random examples.

Usefulness

The average tactic has a usefulness score of around 17%. This means that the average tactic can be applied to around 17% of the proofs in the test set. This reflects well on the tactics, suggesting that the thresholds have helped us to produce some truly significant tactics.

Quality

The average tactic scores around 4.5 for quality. This quality score for an average tactic is excellent as it demonstrates that IsaNewT has not become bogged down with two-step sequences. In finding tactics with 3 or 4 steps and 1 operator which are widely applicable justification of the claim that the tactics discovered are truly useful begins to appear.

7.3.2 A Simple Tactic

A simple tactic could consist of two proof steps with no other operators. For example:

$$[atomize_eq, iffI] \tag{7.2}$$

As in the case above, simple tactics can often be studied by hand to see why certain steps would be likely to occur together. Many of the simple tactics found have been examined, and although many are more obscure than this example, it is often possible to see why such small sets appear. In fact it is normally because such small sets are combined together by an author to form a new rule in some more recent theory file.

7.3.2.1 Scores Gained in Formation Step

Simple tactics such as that in (7.2) do not change from the original pattern discovery stage, therefore the scores assigned in the initial formation provide a good indication of what other evaluation methods find. However, they suffer in the genetic programming stage as they are never improved and therefore never subsume any other tactics. This example had a probability of 0.092 in the original pattern formation and a score of 1 in the genetic programming stage.

7.3.2.2 Manual Evaluation

These simple tactics are ideal candidates for manual evaluation as it is generally easy to spot why small combinations go together.

For example, if:

$$atomize_eq: \ (x \equiv y) \equiv x = y \tag{7.3}$$

and:

$$iffI: \ [P \Longrightarrow Q; Q \Longrightarrow P] \Longrightarrow P = Q \tag{7.4}$$

Then the two can be used in conjunction to reduce an equivalence to two (simpler) subgoals involving implication using backwards reasoning.

Usefulness

These short simple tactics generally score well in the usefulness category, as their simplicity means that they are less likely to be theory-specific and more likely to be applicable in many situations. However, it can be the case that even simple tactics may appear often in the training set and not at all in the test set.

This example could be applied to 11% of theorems in the test set.

Quality

All simple tactics are penalised heavily in the quality measure due to their simplicity. Even a high usefulness rating is not enough to score well here.

This example gets a score of 2.2

7.3.3 A Complicated Tactic

Much more complicated examples exist, such as:

$$[impCE, [m_3] \vee [[(allE)+] \wedge [notE]]] \tag{7.5}$$

where the macro m_3 is

$$[notE, assumption] \tag{7.6}$$

Which is represented pictorially in figure 7.2 for ease of reading. Examples containing

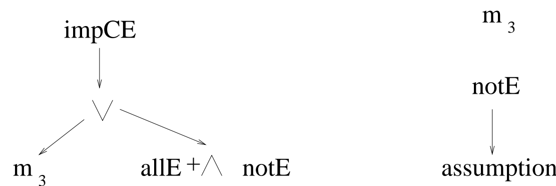

Figure 7.2: Example of Complicated Tactic

all operators, such as this one, are rare, but there are many tactics which contain one or more operator (not including macros).

7.3.3.1 Scores Gained in Formation Step

Complicated tactics such as this are more likely to suffer from a less generous score at the pattern discovery stage. They do not normally score as well here as simpler patterns. However, the changes gained in the genetic programming stage allow this score to contribute.

This example scored 0.006 in the pattern formation (the best-scoring pattern which it subsumes) and 9 in the genetic programming stage.

7.3.3.2 Manual Evaluation

These are the trickiest candidates for manual evaluation as it is not always clear how the steps link together. However, referring back to places in the original proof script where this tactic could be applied is a good way of finding out how sensible these complicated tactics are.

(All the rules associated with each proof step name are given in appendix A.)

Usefulness

These complicated tactics generally don't score very well in the usefulness measure as they are too specific to be widely applicable.

This tactic could be applied to 2% of the theorems in the test set.

Quality

These complicated tactics score very well in the quality test, even when they have a poor usefulness rating.

This example scored 6.1 as a quality rating.

7.3.4 A Good Tactic

Unsurprisingly, some of the best tactics across all the metrics are some of the combinations that originally were expected. Within the domain of Higher Order Logic (i.e. logic-based theorems as opposed to math-based theorems), the best combination found was:

$$[spec, ex1E] \tag{7.7}$$

7.3.4.1 Scores Gained in Formation Step

The best tactics naturally have the most common occurrence from the start and so score very well in the pattern discovery stage. However, they are often not changed by the genetic programming stage so do not receive a good score for this.

This example scored 0.49 in the pattern discovery stage and 1 in the genetic programming stage.

7.3.4.2 Manual Evaluation

The best tactics are often the most obvious or expected. This combination strips quantifiers from a subgoal before other steps are applied. This tactic seems so expected because many people strip away quantifiers when performing proofs. Dale Miller suggested that people liked to transform subgoals into a quasi-normal form before trying to find a proof [Miller and Nadathur (1987)].

In this example, *spec* is a rule which allows you to choose an instantiation for a universal quantifier and $ex1E$ strips off an existential quantifier by automatic instantiation.

Usefulness

These type of tactics are very useful as expected.

This tactic could be applied to 51% of the theorems in the test set.

Quality

These common tactics score reasonably in the quality test. Despite often being short, simple tactics, the high usefulness score can improve the quality score. However, they still often don't score nearly as well as the most complicated examples.

This example scored 4.1 in the quality test.

7.3.5 A Bad Tactic

However, along with the good tactics, also discovered are some bad ones.

$$[least_def, the_equality, conjE, [[allE] \vee [order_antisym]], order_antisym] \quad (7.8)$$

7.3.5.1 Scores Gained in Formation Step

Tactics such as this are a prime example of how patterns specific to a small set can be discovered as commonly occurring patterns although they do not occur often in a wider setting. This example scored 0.5 in the pattern formation score and 1 in the genetic programming stage.

7.3.5.2 Manual Evaluation

This tactic appeared as a commonly occurring pattern from the training set but did not appear once in the test set. The first step "$least_def$" only occurs twice in the set and so this tactic covers all occurrences. However, the pattern finder scores patterns on how often a combination occurs after a step in relation to the number of times this step appears.

This example is only used for proving properties about *Least*, hence the reason the definition is used here and not often elsewhere.

Usefulness

These type of tactics score 0 usefulness.

This tactic could be applied to 0% of the theorems in the test set.

Quality

This pattern can still gain a reasonable quality score. This is not necessarily a mistake, because although it may never appear again, this situation could occur with a mathematically interesting combination of rules.

This example scores 7 in the quality rating. This is due to its complexity and demonstrates the imperfections of the quality scoring system. This system works well in the average case but can be confused by some extremes.

7.3.6 Overall Evaluation

The new tactics can be evaluated as a group using the usefulness and quality measures. From the test set associated with the same domain as that the examples were taken from, 32% of the theorems could have at least one tactic applied to them. In this case

there is no attempt to apply these theorems, simply a comparison between the tactics and the existing proofs of the theorems in the test set. Each proof is examined to see if any complete tactic matches any part of the proof of a theorem. This means that 32% of the theorems could use the tactics in order to find a proof. However, this does not specify if extra information would be needed (such as instantiation information) in order to apply the tactics.

The manual analysis techniques provide a wide range of perspectives. There is a measure for intuitiveness, and the usefulness and quality scores give new ways to rate the newly discovered tactics. The measure that most attention should be paid to must depend on the intention for the tactic (usefulness vs quality) and on the type of tactic it is (simple vs complex).

In particular the manual step can rely on the other measures to some extent. For example, if a 'bad' tactic scores 0 usefulness but a respectable quality, it may well be worth re-investigating the reasons for these steps to be applied.

However, in terms of automation, this kind of tactic (even if mathematically interesting) would probably not be required often enough to justify a heuristic inclusion.

Ultimately, the overall score given to a tactic with these measures is only worthwhile when given some kind of context.

7.4 Application

In this section the details of testing the tactics with IsaAuto is described. The section begins with a description of the choice of test theorems, including the variation of complexity and mathematical similarity.

The section continues with an explanation of the choice of tactics to go into the prover as heuristics. Also described is how this process can be fully automated in order that no human intervention is required at any step of tactic generation.

There have been many questions over the abstraction used and how different abstractions can and will affect the performance of the final tactics. The abstractions that can be applied to the prover are compared and the robustness of the technique with respect to the choice of abstraction is discussed.

Comparisons of IsaAuto's performance with and without the tactics is shown in a number of graphical displays. Explanations are given for these results and how they validate the claim that IsaNewT can automatically formulate useful tactics. The best and worst case examples discovered are discussed along with suggested reasons why

these extremes have been found.

7.4.1 Test Theorems

In the previous chapter IsaAuto was tested to discover what complexities of theorems it would be reasonable to include in the test set. It was demonstrated that the optimal solution appeared to be theorems ranked at difficulty 5 or lower by the ranking system described. The results shown in the previous chapter demonstrated that for a complexity greater than 5, the prover would not be able prove enough theorems to make a comparison possible and lower than 5 would not present a difficult enough challenge.

At the initial stages of this project each Isabelle theory was split in half randomly. This means that the tactics have been trained on theorems both within and exceeding this complexity limit. This will have no bearing on the results, as the capacity for complex theorems is more dependent on IsaAuto than the generated tactics.

The theorems ranked above 5 were removed to bring the test set to a reasonable level. The theorems were left in groups according to the theory they originally occurred in. This allows for a gauge of what type of theorem each is (i.e. propositional logic, natural numbers, set theory).

By separating theorems into groups (some of which overlap), the discovered tactics can be linked to the best set of test theorems.

7.4.2 Choosing the Best Tactics

Tactics trained on a set of theorems perform best when applied to theorems of a similar type. This can be seen in hand-built tactics as these are normally developed with a particular type of problem in mind. Also in human mathematics, techniques learned in a particular discipline are likely to be used to prove similar theorems within the same discipline. Therefore it seems likely that the discovered tactics will demonstrate the best success rate when applied to theorems similar to their test set.

Tactics have been learned from a variety of theory groups. Many of these overlap and so testing has occurred over a range of specificity. For example, some of the tactics have been discovered from all the theorems in the test set from HOL, others only from the group theory subset mentioned previously.

To prove whether or not tactics truly will perform better on a test set of theorems which is mathematically similar to the training set there is a comparison of results taken from a range of theory choices later in this chapter.

7.4.3 Different Abstractions

At previous stages there has been discussion regarding the different options available as to the level of abstraction used. Figures 7.3 and 7.4 show the average performance over a range of test theorem sets of the different abstractions. Each test set comprises 50 theorems; 20 at complexity 1, 10 at 2, 10 at 3, 7 at 4 and 3 at 5. Each test set is taken from a different type of theory, the tactics were trained on different theorems taken from the same type.

The abstractions represented are 'rule name only' *rno*, 'rule name with direction ' *rnwd*, 'class only' *co* and 'class with direction' *cwd*.

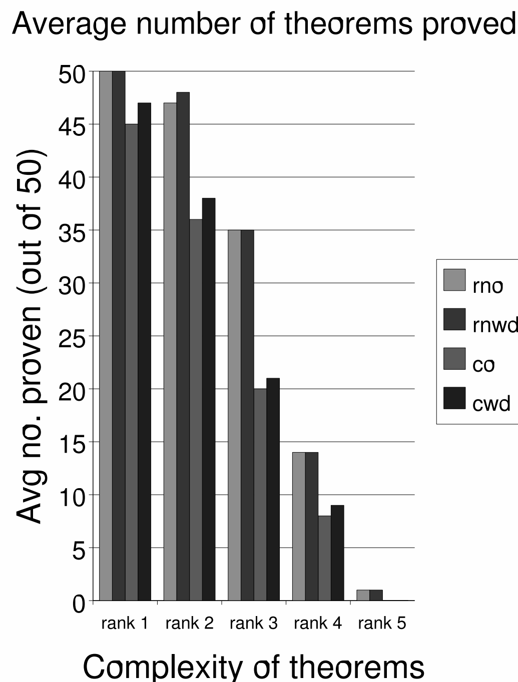

Figure 7.3: Performance of different abstractions. Rule name only: *rno*. Rule name with direction *rnwd*. Class only *co*. Class with direction *cwd*. X-axis shows increasing complexity. Y-axis shows the average number of theorems proved over 20 runs.

As can be seen, using classes instead of rule names results in a significant increase in the time it takes to prove a theorem and a slight drop in the number of theorems proved (mostly due to time-outs). This was expected and does not necessarily mean that the inclusion of classes as a measure does not give us interesting information.

There is a very slight time improvement when the direction is included along with the rule name. Although this is beneficial, it is offset by the extra information which

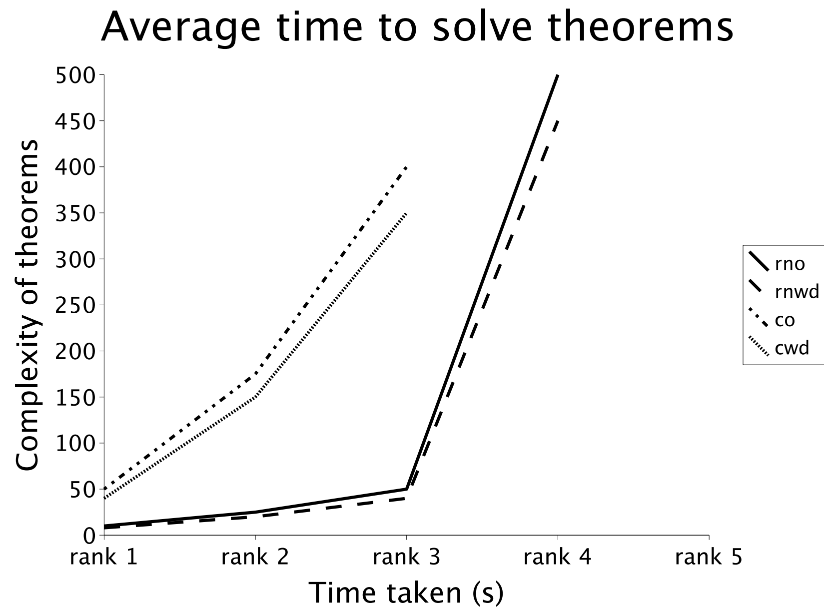

Figure 7.4: Time performance with different abstractions. Rule name only: *rno*. Rule name with direction *rnwd*. Class only *co*. Class with direction *cwd*. X-axis shows increasing complexity, Y-axis shows increasing time.

must be carried at every stage of this project. The time increase is not significant enough to make this a necessary adaptation, but neither is the space requirement onerous enough to make this undesirable. It appears that these two abstractions are comparable in terms of their suitability.

This section has demonstrated that although choice of abstraction does of course play a part in the applicability of the discovered tactics, the technique is robust enough that some flexibility in the choice of abstraction can be tolerated but also that some are consistently better than others.

7.4.4 Tactic Application results

This section describes the most important part of tactic evaluation. The results from the prover with and without tactics are compared when trying to prove theorems taken from the test set. Both the average time taken to prove a theorem and the number of theorems from the test set which the prover successfully proves are compared.

7.4.4.1 Comparison of Domain Specific Training and Test Set

The first comparison shows the average performance of tactics learned from a narrow domain of theorems (such as well-founded recursion), the test theorems are also taken from this narrow domain (from those reserved for the test set). This is represented in figures 7.5 and 7.6

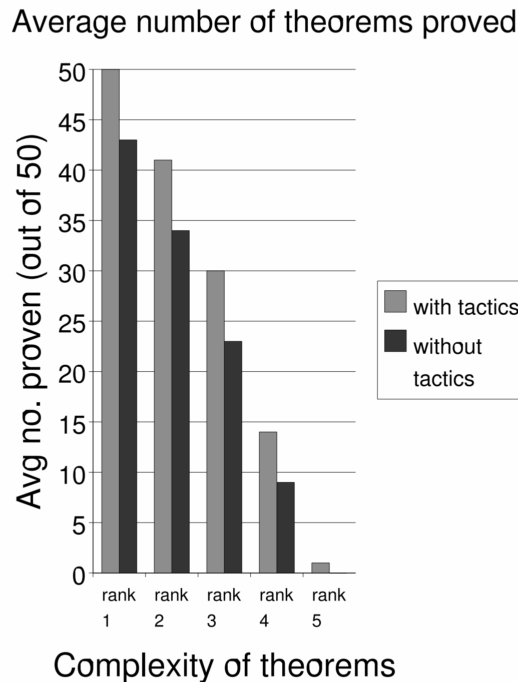

Figure 7.5: Average numbers of theorems proved by Isabelle prover from theorems taken from narrow groups. X-axis shows increasing complexity, Y-axis shows the average number of theorems proved

As can be seen, when the tactics are trained from narrow groups there is a large improvement both in the average time taken to prove a theorem and in the number of theorems proven.

Best Case The best case in this example is shown by a proof found very quickly by the prover with tactics against a proof not found by the prover without. Examination of this shows that the quick proof is almost entirely described by one single tactic (only two steps in this proof are not represented by the tactic).

Worst Case In these test sets, no theorem proved by the prover without tactics is failed to be proved by the prover with the tactics included. This means that the

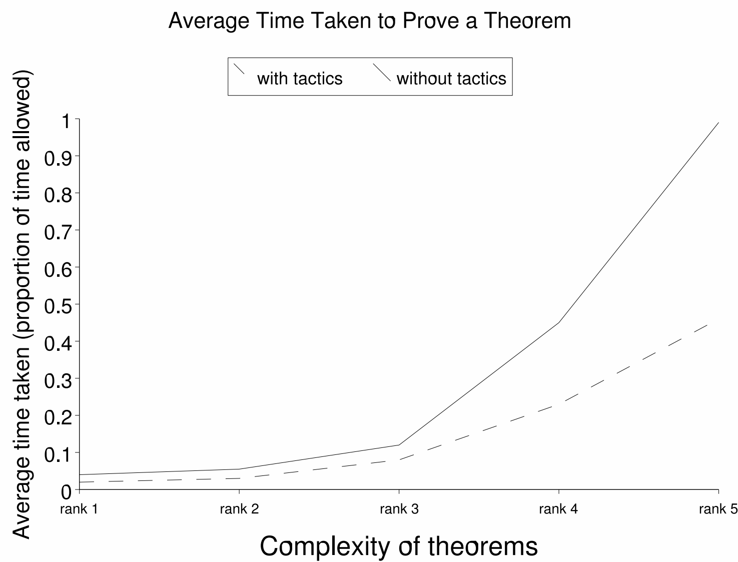

Figure 7.6: Average times taken to prove a theorem by Isabelle prover from theorems taken from narrow groups.

tactics have not slowed the prover down so much that timeouts are called upon. This may not be the case for more complex theorems but there is no realistic way to test this at present. However, there are a few examples where the proof by the prover with tactics takes significantly longer to find. Examination of one of these cases shows that a long proof was found but that no tactic was applicable at any stage, the extra time was taken because each tactic must be tested every time the subgoals change.

7.4.4.2 Comparison of General Training and Test Set

Next to be compared is the average performance of tactics learned from a broad domain of theorems (such as Higher-Order Logic), the test theorems are also taken from this large group (from those removed for testing purposes). This is represented in figures 7.7 and 7.8

As can be seen, when the tactics are trained from broad groups there is a smaller improvement than obtained from the narrow groups. This still represents a noticeable improvement both in time and in number of theorems proved.

Best Case The best case in this example is shown by a proof which contains three separate tactics at different stages.

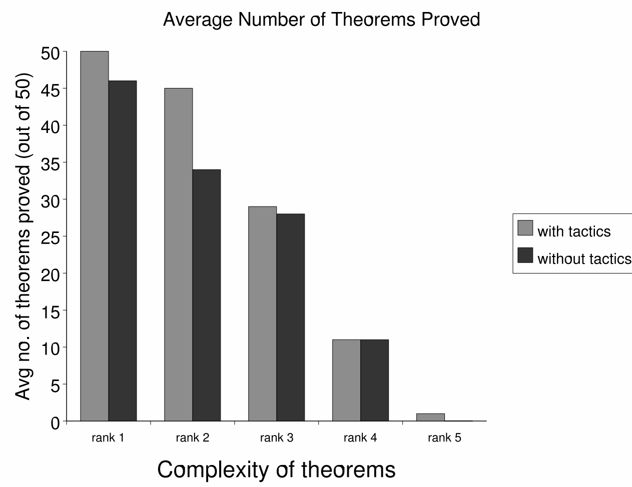

Figure 7.7: Average numbers of theorems proved by Isabelle prover from theorems taken from a broad spectrum of theories.

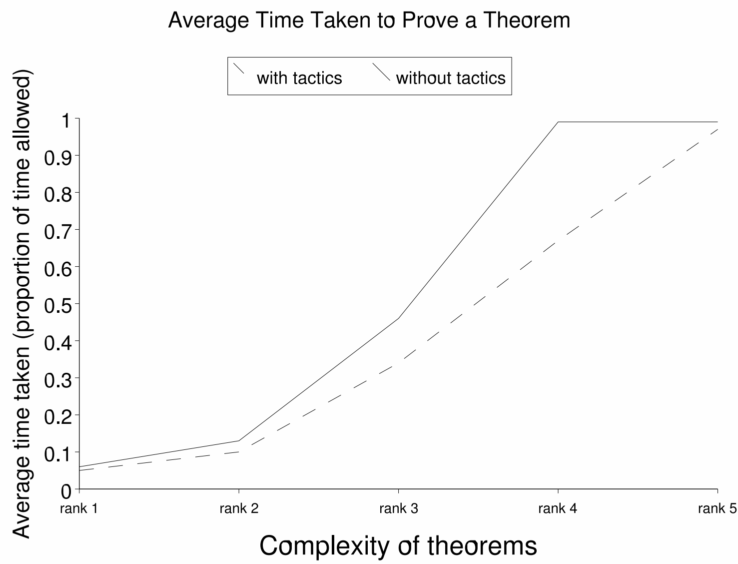

Figure 7.8: Average times taken to prove a theorem by Isabelle prover from theorems taken from a broad spectrum of theories.

Worst Case There is a larger set of discovered tactics in this example and there are the rare occasions where a proof solved by the basic prover fails when the tactics are added. However, this is rare (not more than 1 in 500 theorems).

Defining the thresholds - retrospective These measures were used on tactics discovered from patterns with a range of significance thresholds. It was discovered that the best results were obtained when the threshold was set so as to give around 1 pattern for every 10 theorems. This generally meant that the threshold should be set at around 0.002. However, in order that patterns which would later be combined into tactics are caught, a setup with a lower threshold of 0.001 was chosen. Any tactics which had not been improved (in the tactic formation stage) and had an initial probability (from the pattern discovery stage) of between 0.001 and 0.002. This approach gave a very slight improvement but provided more variety of tactics to work with.

7.4.4.3 Specific Training, General Test

Next to be compared is the average performance of tactics learned from a narrow domain of theorems (such as proofs about hyperreals), the test theorems this time are taken from a large domain. This is represented in figures 7.9 and 7.10

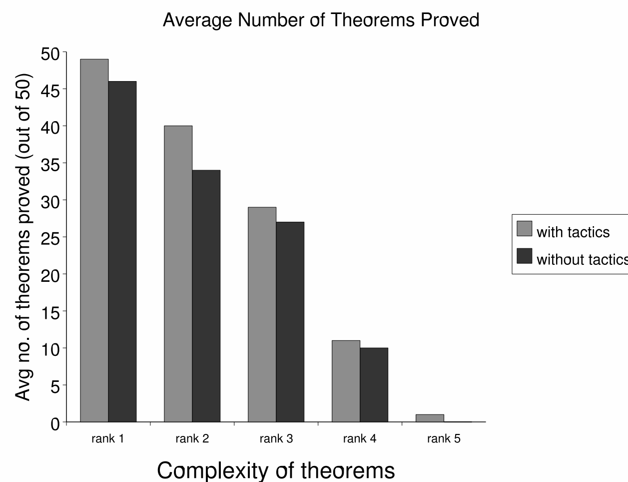

Figure 7.9: Average numbers of theorems proved by Isabelle prover from theorems taken from a broad domain of theories with tactics trained on a narrow domain of theorems.

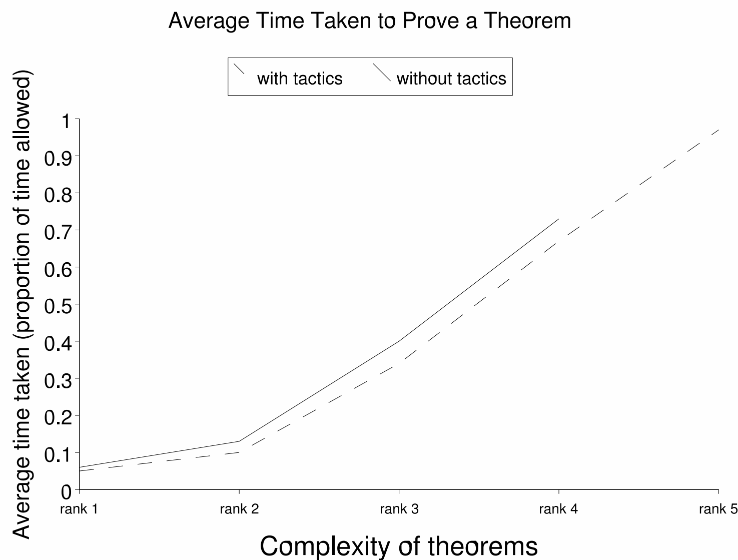

Figure 7.10: Average times taken to prove a theorem by Isabelle prover from theorems taken from a broad spectrum of theories with tactics trained on a narrow set of theorems.

In this example there is still an improvement overall, but it has been reduced to a very slight improvement. It could be imagined that this is because most tactics will only be applicable if the test theorem is also from the narrow group. This setup was not expected to provide promising results but it was examined in order to be thorough and consider every combination.

Best Case The best case in this example is shown by a tactic which is applicable on a number of occasions to theorems which did not come from the narrow group.

Worst Case There are a number of cases where no tactics are applicable. Also, there is at least one example of a tactic which is never used.

7.4.4.4 General Training, Specific Test

Next to be compared are the average performance of tactics learned from a broad domain of theorems, the test theorems this time are taken from a narrow domain. This is represented in figures 7.11 and 7.12

In this example a better improvement can be seen than in the narrow vs broad example. This is because tactics learned from a broad spectrum will have applicability across that spectrum.

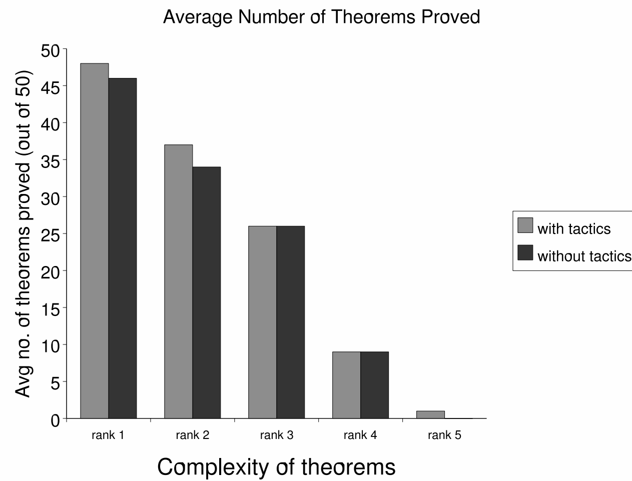

Figure 7.11: Average numbers of theorems proved by Isabelle prover from theorems taken from a narrow group of theories with tactics trained on a broad spectrum of theorems.

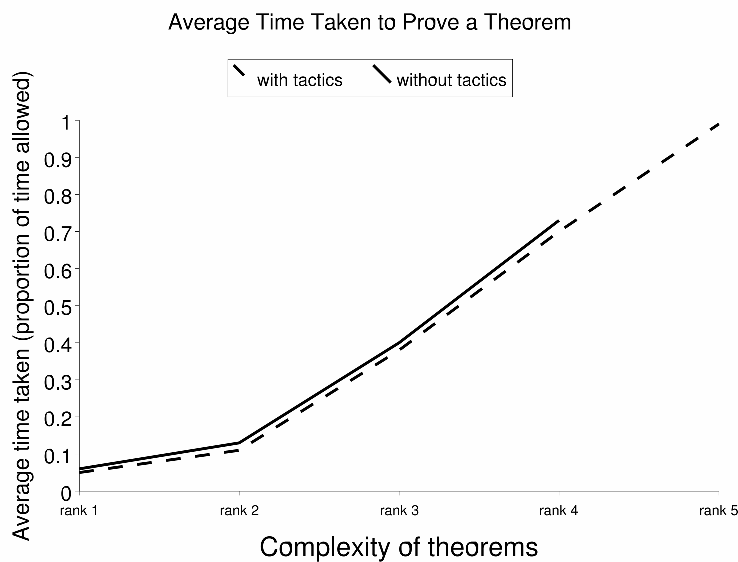

Figure 7.12: Average times taken to prove a theorem by Isabelle prover from theorems taken from a narrow group of theories with tactics trained on a broad spectrum of theorems.

Best Case The best case in this example is shown by a few tactics which are widely applicable (some can be applied in over 60% of theorems)

Worst Case Again, there are a number of theorem where no tactic is applicable throughout the proof. It can be imagined that this is because these theorems require a particular (e.g. mathematical) technique.

7.5 Other Abstractions

In this section some of the information gained that has not been capitalised on is examined. Potential uses of this information is discussed.

Two main pieces of information have been lost (described earlier when discussing different possible abstractions) :

1. Classes

2. Main Rule Operator

While it has not been feasible to apply either of these two pieces of information in the application stage, they still say something interesting about the way that people do proofs.

The abstraction containing class information demonstrated that most classes appear in clumps. In particular, when fed through the pattern discovery stage and then the tactic formation stage tactics of the type

[(rewrite)+,(quantifier elimination)+,(rewrite)+,(simplification)+]

are common. While this has no direct bearing on the IsaNewT methodology it describes a method which could be used as a heuristic in proof search when it is desirable to produce a proof that would be more intuitive to a human.

Similarly, examining the type of patterns found when including the main operator information yields interesting results. In this case a little more examination is necessary in order to spot the patterns. If the operators are grouped together (algebraic, quantifier etc.) then a tendency for operators within these groups to be clustered together is noticeable.

7.6 Summary

This chapter has described the culmination of IsaNewT and the overall performance of the discovered tactics. As a first attempt at providing a method for automatically generating tactics, it was never to be expected that tactics would be produced which could be compared to those produced by humans. Instead, it has been claimed that IsaNewT provides a method for automatically producing useful tactics. The results given in this chapter demonstrate that fact.

Some of the information gained from different stages of the tactic discovery process are discussed along with discussion of the use of different abstractions. Many of the abstractions suggested cannot be directly tested, and others which have been tested did not directly improve the usability of the discovered tactics. Similarly, the manual evaluation of the patterns and tactics provides an interesting intellectual exercise but does not improve the tactics in any way (but it was not intended or expected to). Some of these techniques used to rank the tactics could possibly be adapted but the whole aim is to avoid any kind of necessary human intervention at all.

It is the section on application which really demonstrates the worth of the tactics. Although there are some individual theorems that take longer to prove with the tactics and (rarely) there are cases where this extra time will result in a theorem not being proved, it is important that no test set has resulted in an overall decrease in performance.

Every single test set showed an average increase both in numbers of theorems proved and in the average time taken to prove a theorem. This is unsurprisingly more pronounced when the field is narrower. This undoubtedly proves that the tactics IsaNewT has automatically produced are indeed useful.

Chapter 8

Conclusion

This dissertation set out to provide a method of automatically formulating tactics which would prove to be useful in discovering new proofs. IsaNewT formed new tactics from commonly occurring patterns found in the large corpus of existing Isabelle proofs.

These newly discovered tactics were tested using a number of methods, most importantly a naive automatic prover formed within Isabelle - IsaAuto. This automatic prover allowed an evaluation of the new tactics' usefulness. A comparison of the number of proofs completed and the time taken to complete such proofs provided a measure of usefulness which allowed the tactics to be compared against naive search at the rule level.

The commonly occurring patterns discovered in the Isabelle corpus have provided a good base from which - with Genetic Programming techniques - a good selection of new tactics have been formed. Evaluation has demonstrated that these tactics can certainly be described as useful. Using these new tactics as an aid to search (for a proof) improves results and efficiency in all but a few cases.

8.1 Summary

This project required a number of early decisions such as the choice of prover from which the proof corpus was used, along with the level of abstraction that should be used. A choice was made to use the Isabelle interactive proof system as it satisfied all the necessary criteria:

1. A large proof corpus.

2. The proof corpus stored in electronic form and easily accessible.

3. A variable specificity.

In chapter 3 we discussed in detail a number of possible abstractions such as:

1. Rule name only.
2. Rule name with direction.
3. Class of rule.
4. Class of rule with direction.
5. Main proof operator.
6. Rule name with subgoal information.
7. Main proof operator with direction.
8. Rule name with position in proof.

Evaluation of the different abstractions show that of this selection, ‘rule name only’ or ‘rule name with direction’ are the two most viable options for the purpose of creating new, useful tactics. However, some of the other abstraction options have provided some interesting information. This is typified by the application of the ‘class of rule’ abstraction which clearly demonstrates the preference of proof authors for grouping certain classes of steps (such as simplification) together.

The abstracted proof corpus (mainly the ‘rule name only’ option) was data-mined to find commonly occurring sequences. In searching for a method suitable to use for this process an examination was made of what is meant by *commonly occurring*. For IsaNewT it was appropriate to define a *commonly occurring pattern* to be a sequence of rule steps which occur together with a probability above a specified threshold. Variable Length Markov Models are a probabilistic process which describe the probability of an event occurring after a given sequence of events. This procedure was well suited to IsaNewT.

No pattern discovery technique was found that could cope with the tree structure of the proofs, so linearisation down the branches was performed on the proofs. The loss of connection across the branches was deemed to be minimal, and a method to reconstruct as much information as possible was given in the genetic programming section of the dissertation.

Experimentation with the (mathematical) properties of the proofs used in the training set and with the threshold we used to denote *significance* allowed the process of producing a good pattern set to be refined. Patterns trained on a set of proofs with a similar mathematical background produced better tactics overall. In order to avoid the restrictions caused by requiring proofs to be preselected, and to keep the entire process automated, methods of automating this process were examined. Like most other theorem provers (automated and interactive), Isabelle's existing proofs are stored in grouped theory files, each theory file containing a group of similar theorems. Similar theory files are grouped together in named directories. This hierarchy allows use of the existing grouping as the preselection criteria. Although a better preselection could no doubt be made with careful manual selection, the given method satisfies the criteria of being completely automated.

The third stage of IsaNewT involved the formation of tactics from the discovered patterns. The language used to describe our tactics allowed for four new operators to be introduced at this stage:

1. Macro
2. $\vee$ branching
3. $\wedge$ branching
4. $+$ repetition

As the most cursory examination of any commonly occurring patterns shows, they can be combined in a number of ways, even when there is not necessarily one ideal set. If a pattern ($P1$) could be combined with another $P2$ to form $P3$ or with a different one $P4$ to produce a different new tactics $P5$, but both $P3$ and $P5$ perform equally well in tests, then there is no way to know which would be the best choice for the long term and so there is no ideal solution. In order to formulate the best tactic set it was desirable to use a technique which would allow incremental improvements which could be evaluated at each step. Genetic programming techniques provide exactly this advantage along with the benefit of having a solution at every iteration, removing the need to wait on an ideal solution which may not exist.

In many applications the random elements of GP are one of the main advantages to this technique. For the purposes of forming new tactics from the existing patterns, which has a well defined goal, it was more efficient to have a directed approach. To

this end, the Pairwise Combination technique was developed. As described in chapter 5, this technique has many similarities to the traditional GP. However, results show it produces patterns which are directly connected to the initially discovered patterns - this is sometimes lost in the random mutations of more traditional GP. The PC technique also allows for a concrete aim towards reforming the branches lost in the abstractions. All significant links are rediscovered to some extent.

PC, the directed genetic technique is much more efficient and successfully combines the discovered patterns into compound tactics.

The final stage of the IsaNewT methodology involves testing the new tactics to discover if they are in fact useful and applicable in a number of cases. To this end IsaAuto, an automatic prover within the Isabelle system, was developed. This prover was developed to search exhaustively through the available steps to search for a proof. No sophisticated heuristics or proof techniques were used within this prover, but the intention was not to develop a good automated prover, but to provide a platform for testing the tactics.

The tactics performed well when added as a heuristic to the automated prover. This is demonstrated fully in chapter 6. This testing demonstrated that using tactics learned from a preselected set provides better results. In addition to producing patterns which have a higher probability in the patterns discovery stage, the tactics performed much better when used against a test set of similar theorems. Using the hierarchy within Isabelle as described earlier, test sets can be formed of theorems similar to the ones the tactics were generated from.

Using such a selection of tactics and test set, the discovered tactics perform well across the board. On average, the automated prover with the tactics added outperforms the basic prover in both time taken to prove theorems and the number of theorems proven. Although search time takes longer when the tactics fail to be useful, this is outweighed overall because a successful application of a tactic removes the need for the search for a number of individual steps.

This application demonstrates that IsaNewT's newly discovered tactics can indeed be described as useful. They are applicable in a variety of situations and do not require a prohibitive level of search.

8.2 Critique

Many of the decisions made during the course of this project were only one of a number of available alternatives. For example, the choice of Isabelle as the foundation prover was made in spite of a number of alternatives which would have been suitable for NewT. There is no reason why the techniques and principles within this project could not be used with any other prover. Indeed, any prover which satisfies the initial constraints would be suitable.

In this case, all that would be required to adapt our system to another prover would be a formatter to parse the existing corpus into a suitable format for the pattern discovery process. The tactic applicator here is presented for evaluation purposes, a different system could use the tactics in any way it wished. This process is not dependent on the rule names specific to Isabelle, nor does it depend on the rule names being of the Isabelle structure. Each "rule step" is read in as a string at this point. Therefore, if the corpus of a new proof was abstracted to a suitable proof step format, it could be applied directly to the tactic formation system.

It would have been desirable to apply NewT to a number of other provers in order to check the robustness and ensure that the successful results shown with the Isabelle automatic prover would be reflected with other provers but time did not allow, this is planned as future work.

A variety of abstractions were tested, and one was selected which produced the best results for tactic discovery. However, NewT's technique does not allow for any subgoal information (such as instantiation information). Inclusion of subgoal information would allow learning of patterns that are used in specific situations as opposed to just those which are used frequently. Subgoal information could also be used to indicate when the tactics should be applied, this would remove some unnecessary search when attempting to apply our tactics.

In discovering patterns, a technique of linearisation has been devised and used to circumvent the problem of pattern discovery within tree structures. Ideally a technique which learns directly from the tree structures would be used. Although no such techniques had been found, some suggestions have since been made and are planned to be carried out in future. However, during the tactic formation stage, any patterns which match up to a branching point will be joined together. Although this means that a pattern may reach a point where several branches must be chosen from, it also means that any significant links across branches will be regrouped together. As such, these

connections may be applied together during the application stage.

In applying the tactics, they have been tested with a naive search method. This type of method is extremely inefficient and so although the new tactics perform well in this setting it is doubtful that they would perform well against any automatic prover which uses more sophisticated techniques. It would be interesting to examine their performance when added as an extra heuristic to a more sophisticated prover.

8.3 Related Work

Before this project the LearnΩmatic project was the most similar attempt to automatically formulate tactics. In chapter 2 we described how Kerber, Jamnik, Pollet and Benzmüller utilised the technique of least general generalisation to learn new proof methods for various domains.

Their project requires that a family of similar proof be carefully hand chosen. Although we have previously described a method to automatically group proofs together in order that the resulting tactics will be more successful, this grouping is still far more general than that used by LearnΩmatic. Unlike IsaNewT, the LearnΩmatic approach learns proof *methods* which encompasses preconditions, postconditions and a tactic in order to construct proof plans. Their higher-level approach increases complexity resulting in the requirement that every proof to be learnt from must be an instance of the pattern.

The IsaNewT approach learns patterns from a lower-level within the proof, so no additional information is required and learning can be performed from any diversity of proof corpus. Restricting the proof scripts to specific domains is not necessary, it simply provides a better quality of final tactic. In any case, assigning proofs to domains as suggested can be fully automated.

Both Silver (1984) and Desimone's (1987) work with precondition analysis learn new proof steps which can be equated to learning new tactics. The reuse of existing proofs in both cases has a direct relation to the work presented here. However both Silver and Desimone generalised single successful proofs in order to develop an new method. IsaNewT differs significantly from this approach in that a broad range of proofs are examined in order to find similarities which can then be reused.

Recently, Alison Mercer has written an extension to IsaNewT which uses the patterns discovered in the initial stages of IsaNewT as a primer for a recommender system within Isabelle. Her work (PGTips) is integrated into the Proof General [Aspinall

(2000)] scripting system to provides users with a 'recommend' option while they write a proof. At the beginning of an Isabelle theory file, the user must input any dependencies on existing theory files. This dependency is used to select the set of patterns which should be used (the patterns are grouped according to the sets of theories they were trained on).

When a user reaches a point within a proof where they require a recommendation, the click Mercer's 'recommend' button in the Proof General window. This button prompts PGTips to match the commands previously used in the proof to any patterns existing in the pattern set. Up to three patterns with the highest probability are chosen and their subsequent steps are returned as a recommendation.

Having Mercer's project developed with the patterns discovered here is an ideal usage of IsaNewT. It would be more complicated but interesting to see a similar system which used the final tactics discovered in our project as an initial input.

8.4 Further Work

The IsaNewT methodology provides an original way to automatically produce tactics which can be useful in a number of situations. There are a number of improvements and extensions which would be interesting.

Firstly, IsaNewT uses only proof step information. There is currently no method to include information from the subgoal. This could be a useful inclusion to the tactics. It would allow more specificity about the occasions when these tactics should be applied. Inclusion of subgoal information would also allow learning of patterns of the form 'when the subgoal contains x, then apply y' rather than the current 'if a then b are applied, then apply c'. It may be possible for techniques which allow relationships to be quantified to be utilised to this affect.

Inclusion of subgoal information would have the further benefit of allowing terminating conditions for the final tactics. This would allow an enrichment of the current grammar, permitting $if...then$ and *while* statements. In particular, this would allow an enrichment of the *plus* operator by giving it termination conditions.

Discussed in the critique was the desire to extend NewT to encompass other theorem provers. Provers such as COQ, NUPRL, PVS, Mizar and LEGO would be ideal candidates for this extension. Correctly formatting the corpus from any one of these would allow direct application of NewT.

In applying the IsaNewT techniques to other provers the door would be opened to

extending Mercer's recommender system to be used with these provers also. Using the finished tactics as a basis for recommendation instead of the initial patterns would allow a more sophisticated and robust recommendation. Indeed, Mercer's system currently provides 3 recommendations when a request is made. Often, the patterns that form these recommendations would be combined into one tactic at the tactic formation stage. A good extension to both projects would be to extend this recommender to utilise the finished tactics in other provers along with Isabelle.

Many sophisticated automated provers exist. It would be beneficial to examine the possibilities for incorporating tactics discovered using NewT into a sophisticated hierarchy. This would allow further testing of the quality of the tactics along with, hopefully, providing another concrete usage.

A final extension to IsaNewT would be to learn from more complicated tactics. At the current stage, it is reasonable to test the tactics against a basic prover setup as the tactics were learned from such low-level proof steps. It would be hoped that if NewT's techniques were applied to a corpus consisting of complex tactics that it would be possible to learn in turn even more complicated tactics. For example, it must be considered that if NewT is applied as it stands, then every available position in a proof is replaced with the applicable tactic in place of the rule name sequences, there would be a good basis for reapplying the tactics discovery process. In this way it could be possible to incrementally increase the complexity of the corpus, and hence the tactics being discovered.

Appendix A

Glossary

$\vee$ **branch** An $\vee$ branch in a tactic can be read as "do x OR do y". Described fully in chapter 5.

$\wedge$ **branch** An $\wedge$ branch in a tactic can be read as "do x to subgoal 1 AND THEN do y to subgoal 2". These are designed to reflect the original $\wedge$ branches in a proof structure. Described fully in chapter 5.

crossover Creation of two new programs (or tactics) by combining randomly chosen parts of two existing programs. Described fully in chapter 2.

$+$ **repetition** The $+$ operator reflects a "1 or more" repetition. Described fully in chapter 5.

abstraction We have produced several viable abstractions of the Isabelle proof corpus. Each abstraction contains the information from the corpus that we use as an input to our pattern discovery process. Each abstraction is described in detail in chapter 3.

Automatic theorem prover A fully automated theorem prover is one which requires no human intervention to find a proof of a mathematical theorem.

class For the purposes of abstraction, we have in some cases grouped the rules into classes. These describe the type of rule used, such as simplification rule, definition, rewrite etc.

direction Proofs in Isabelle can be formed either forwards or backwards. Direction denotes which way the rule should be applied.

drule, rule, erule, frule These are the directional instructions used in Isabelle proof steps. They are described full in chapter 3.

Genetic Programming (GP) Genetic Programming is an evolutionary technique pioneered by John Koza which is used to incrementally adapt a population to satisfy a specified criteria. It is described in detail in chapter 2.

Interactive theorem prover An interactive prover is a theorem prover which is often designed to be closer to a proof assistant. At each stage a user must input the next proof step(s) and the prover then ensures that a correct proof has been generated.

macro A macro (m_x) is used to denote a common subtactic as shown in figure 5.12. Described fully in chapter 5.

mathematically similar We define the notion of mathematically similar to describe theorems which prove facts in a similar domain. These can be a group of theorems such as theorems on geometry to the basic theorems of Higher Order Logic.

Pairwise Combination (PC) Pairwise Combination is our own GP process inspired by evolutionary programming and GP in particular. It is described in detail in chapter 5.

pattern Patterns in this project are commonly occurring sequences of proof steps. These can also be thought of as simple tactics.

proof step A proof step for our purposes is based on our abstraction. It contains the information in our abstraction which describes the transition from one subgoal to the next. This is usually a rule name or a rule name with the direction it should be applied, although for some abstractions it may just be the class a rule has been assigned to.

reproduction Copying of an existing program into new population. Described fully in chapter 2.

rule name The rule name is simply the name of the rule applied within a proof step. In Isabelle each rule name is the name of the theorem, definition or axiom that is applied at that point.

sequence A sequence is a sequence of proof steps. These are potential patterns but we do not yet know how frequently they occur.

significance threshold The significance threshold is probability above which a pattern in the pattern discovery phase is deemed to be significant. Described fully in chapter 4.

split token The name given to a proof step which results in a branch.

tactic A tactic is a function which furthers the state of a proof. We use tactics to mean a combination of rule steps. Our tactics can be anything from a simple sequence of proof steps to a more complicated arrangement of proof steps which contain operators such as $\wedge$ and $\vee$ branching.

Theory We use 'theory' in the Isabelle sense to mean a (usually small) collection of mathematically similar theorems. In Isabelle, every time a user inputs new theorems, he must group them in a new theory file.

Variable Length Markov Model (VLMM) A Variable Length Markov Model is a probabilistic technique which models sequences or varying length and assigns them a probability.

Appendix B

Isabelle rules and theorems

B.1 Rule definitions

and_def: ``P ∧ Q ≡ ∀ R. (P ⟶ Q ⟶ R) ⟶ R''

all_dupE: `` [| ∀ x. P x; [| P x; ∀ x. P x |] ⟹ R |] ⟹ R

allE: `` [| ∀ x. P x; P x ⟹ R |] ⟹ R''

allI: ``(!!x. P x) ⟹ ∀ x. P x''

atomize_eq: ``x ≡ y ≡ x = y''

atomize_all: ``(!!x. P x) ≡ ∀ x. P x''

assumption *unifies the subgoal with an assumption*

box_equals: ``[| a = b; a = c; b = d|] ⟹ c = d''

ccontr: ``(¬ P ⟹ False) ⟹ P''

conjE: ``[| P ∧ Q; [| P; Q|] ⟹ R |] ⟹ R''

conjI: ``[| P; Q |] ⟹ P ∧ Q''

contrapos_nn: ``[| ¬ Q; P ⟹ Q |] ⟹ ¬ P''

disjCI: ``(¬ Q ⟹ P) ⟹ P ∨ Q''

disjE: ``[| P ∨ Q; P ⟹ R; Q ⟹ R|] ⟹ R''

disjI1: ``P ⟹ P ∨ Q''

disjI2 ``Q ⟹ P ∨ Q''

ex1E: ``[| ∃! x. P x; !!x. [| P x; ∀ y. P y ⟶ y = x |] ⟹ R |] ⟹ R''

Ex_def: ``∃ P ≡ ∀ Q. (∀ x. p x ⟶ Q) ⟶ Q''

exE: ``[| ∃ x. P x; !!x. P x ⟹ Q |] ⟹ Q''

exI: ``P x ⟹ ∃ x. P x''

```
iffI: ``[| P ⟹ Q; Q ⟹ P |] ⟹ P = Q''

impI: ``(P ⟹ Q) ⟹ P ⟶ Q''

impCE: ``[| P ⟶ Q; ¬ P ⟹ R; Q ⟹ R |] ⟹ R''

Least_def: ``Least P ≡ THE x. P x ∧ (∀ y. P y ⟶ x ≤ y)''

mp: ``[| P ⟶ Q; P|] ⟹ Q''

notE: ``[| ¬ P; P |] ⟹ R''

notI: ``(P ⟹ False) ⟹ ¬ P''

order_antisym: ``[| x ≤ y; y ≤ x|] ⟹ x = y''

some1_equality: ``[| ∃! x. P x; P a |] ⟹ (SOME x. P x) = a''

someI: ``P x ⟹ P (SOME x. P x)''

spec: ``∀ x. P x ⟹ P x''

ssubst: ``[| t = s; P s|] ⟹ P t''

swap: ``[| ¬ $P_2$; ¬ $P_1$ ⟹ $P_2$ |] ⟹ P''

sym: ``s = t ⟹ t = s''

the_equality: ``[| p a; !!x. P x ⟹ x = a|] ⟹ (THE x. P x) = a''

trans: ``[|r = s; s = t|] ⟹ r = t''
```

B.2 Some complete proof scripts

```
lemma box_equals:"[| a = b; a = c; b = d |] ⟹ c = d"
apply(drule trans)
apply(assumption)
apply(rule trans)
apply(rule sym)
apply(assumption)
apply(assumption)
done
```

```
lemma conjI:"[| P; Q |] ⟹ P ∧ Q"
apply(unfold and_def)
apply(rule allI)
apply(rule impI)
apply(drule mp)
apply(assumption)
apply(drule mp)
apply(assumption)
apply(assumption)
done
```

```
lemma contrapos_nn: ``[| Q; P ⟹ Q |] ⟹ P''
apply (rule notI)
apply (rule_tac P=Q in notE)
apply simp
apply assumption
done

lemma disjI1:"P ⟹ P ∨ Q"
apply(unfold or_def)
apply(rule allI)
apply(rule impI)
apply(rule impI)
apply(erule mp)
apply(assumption)
done

lemma exI: `` P x ⟹ ∃ x. P x.''
apply (unfold Ex_def)
apply (rule allI)
apply (rule impI)
apply (erule allE)
apply (erule mp)
apply assumption
done
```

Bibliography

Aspinall, D. (2000). Proof General: A generic tool for proof development. In 1985, editor, *Proceedings of TACAS 2000: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer-Verlag.

Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, K., Meier, A., Melis, E., Schaarschmidt, W., Siekmann, J., and Sorge, V. (1997). Ωmega: Towards a mathematical assistant. In McCune, W., editor, *14th International Conference on Automated Deduction*, pages 252–255. Springer-Verlag.

Brazma, A. and Cerans, K. (1994). Efficient learning of regular expressions from good examples. In Arikawa, S. and Jantke, K. P., editors, *Algorithmic Learning Theory: Proc. of the 4th International Workshop on Analogical and Inductive Inference, AII'94*, pages 76–90, Berlin, Heidelberg. Springer.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567.

Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E., editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Constable, R. L., Allen, S. F., Bromley, H. M., et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

DeJong, G. (1988). *Exploring Artificial Intelligent*. Morgan Kaufmann, San Fransisco, CA.

Della Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing feature of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393.

Denzinger, J. and Schulz, S. (1996). Learning Domain Knowledge to Improve Theorem Proving. In McRobbie, M. and Slaney, J., editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 62–76. Springer.

Desimone, R. V. (1987). Learning control knowledge within an explanation-based learning framework. In Bratko, I. and Lavrač, N., editors, *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87, Bled, Yugoslavia*. Sigma Press. Also available from Edinburgh as DAI Research Paper 321.

Dixon, L. and Fleuriot, J. D. (2003). IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, LNCS, pages 279–283.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Paulin, C., and Werner, B. (1991). The Coq proof assistant user's guide, version 5.6. Technical Report 134, INRIA.

Eisner, J. M. (1996). An empirical comparison of probability models for dependency grammar. Technical report, University of Pennsylvania.

Eleazar Eskin, W. N. G. and Singer, Y. (2000). Protein family classification using sparse markov transducers. *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, August 20-23.

Fuchs, M. and Fuchs, M. (1998). Feature-based learning of search-guiding heuristics for theorem proving. *AI Communications*, (11):175–189.

Furse, E. (1995). *Learning university mathematics. In: Mellish, C. (Ed), Proceedings of the 14th IJCAI. Vol. 2. International Joint Conference on Artificial Intelligence, Morgan Kaufmann, pp.2057-2058.*

Giesl, J., Walther, C., and Brauburger, J. (1998). Termination analysis for functional programs. In Bibel, W. and Schmitt, P., editors, *Automated Deduction – A Basis for Applications, Vol III: Applications*, volume 10 of *Applied Logic Series*, chapter 6, pages 135–164. Kluwer Academic.

Gordon, M. J. (1985). Hol: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge. revised version.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Horn, C. and Smaill, A. (1990). Theorem proving and program synthesis with Oyster. In *Proceedings of the IMA Unified Computation Laboratory*, Stirling.

Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111. Also available from Edinburgh as DAI Research Paper No 716.

Jamnik, M., Kerber, M., and Pollet, M. (2002). Automatic learning in proof planning. In van Harmelen, F., editor, *Proceedings of 15th ECAI*. European Conference on Artificial Intelligence.

Kohlhase, M. (2000). OMDOC: Towards an OPENMATH representation of mathematical documents. SEKI-Report SR-00-02, Universität des Saarlandes. `http://www.mathweb.org/ilo/omdoc`.

Kolbe, T. and Walther, C. (1998). Proof analysis, generalization and reuse. In Bibel, W. and Schmitt, P. H., editors, *Automated Deduction - A Basis for Applications, Vol. II Systems and Implementation Techniques*, Applied Logic Series, vol. 9, pages 189–219. Kluwer Academic Publishers, Dordrecht, Boston, London.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press.

Levine, J. and Humphreys, D. (2003). Learning action strategies for planning domains using genetic programming. Technical report, University of Edinburgh.

Luo, Z. and Pollack, R. (1992). Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh. See also `http://www.dcs.ed.ac.uk/home/lego`.

Melis, E. and Whittle, J. (1998). Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2).

Mercer, A. (1996). Pgtips: A recommender system for isabelle. Undergraduate project dissertation, School of Informatics, University of Edinburgh.

Miller, D. and Nadathur, G. (1987). A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*. IEEE Press.

Minton, S., Knoblock, C., Koukka, D., Gil, Y., Joseph, R., and Carbonell, J. (1989). Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Muggleton, S. (1990). Inductive acquisition of expert knowledge. In *Addison-Wesley, Reading, MA*.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS : An integrated approach to specification and verification. Tech report, SRI International.

Paulson, L. (1986). Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258.

Paulson, L. (1994). *Isabelle: A generic theorem prover*. Springer-Verlag.

Pereira, L. M., Pereira, F. C., and Warren, D. H. (1979). User's guide to DECsystem-10 PROLOG. Occasional Paper 15, Dept. of Artificial Intelligence, University of Edinburgh.

Richardson, J. D. C., Smaill, A., and Green, I. (1998). System description: proof planning in higher-order logic with Lambda-Clam. In Kirchner, C. and Kirchner, H., editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, Lindau, Germany.

Rigoutsos, I. and Floratos, A. (1998). Combinatorial pattern discovery in biological sequences: the teiresias algorithm. *Bioinformatics*, January(14(1)).

Ron, D., Singer, Y., and Tishby, N. (1996). The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25.

Rosenbloom, P., Laird, J., and A., N. (1993). The soar papers: Readings on integrated intelligence. *MIT Press*.

Rudnicki, P. (1992). An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, Bastad. Chalmers University of Technology. See http://mizar.org for up-to-date information on Mizar and the Journal of Formalized Mathematics.

Schulz, S. (2001). Learning Search Control Knowledge for Equational Theorem Proving. In Baader, F., Brewka, G., and Eiter, T., editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer.

Silver, B. (1984). *Using Meta-Level Inference To Constrain Search And To Learn Strategies In Equation Solving*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh. Published as a book by North Holland.

Simon, H. A. and Newell, A. (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1).

Sun, R. and Giles, L. E. (2000). Sequence learning: Paradigms, algorithms and applications. *No. 1828 in Lecture Notes in Artificial Intelligence*.